Information Technology and Management 5, 65–110, 2004

XRL/Woflan: Verification and Extensibility of an XML/Petri-Net-Based Language for Inter-Organizational Workflows

H.M.W. VERBEEK * and W.M.P. VAN DER AALST {h.m.w.verbeek;w.m.p.v.d.aalst}@tm.tue.nl
Department of Information and Technology, Faculty of Technology Management, Eindhoven University of Technology, 5600 MB, Eindhoven, The Netherlands

AKHIL KUMAR akhil@computer.org
Smeal College of Business, Pennsylvania State University, University Park, PA 16802, USA

Abstract. In this paper, we present XRL/Woflan. XRL/Woflan is a software tool using state-of-the-art Petri-net analysis techniques for verifying XRL workflows. The workflow language XRL (eXchangeable Routing Language) supports cross-organizational processes. XRL uses XML for the representation of process definitions and Petri nets for its semantics. XRL is instance-based, therefore, workflow definitions can be changed on the fly and sent across organizational boundaries. These features are vital for today's dynamic and networked economy. However, the features also enable subtle, but highly disruptive, cross-organizational errors. On-the-fly changes and one-of-a-kind processes are destined to result in errors. Moreover, errors of a cross-organizational nature are difficult to repair. XRL/Woflan uses *eXtensible Stylesheet Language Transformations* (XSLT) to transform XRL specifications to a specific class of Petri nets, and to allow users to design new routing constructs, thus making XRL extensibe. The Petri-net representation is used to determine whether the workflow is correct. If the workflow is not correct, anomalies such as deadlocks and livelocks are reported.

Keywords: workflow, inter-organizational, verification, extensibility, XML, WF-net

Today's corporations often must operate across organizational boundaries. Phenomena such as E-commerce, extended enterprises, and the Internet stimulate cooperation between organizations. Therefore, the importance of workflows distributed over a number of organizations is increasing [3,4,21,28]. Inter-organizational workflow offers companies the opportunity to re-shape business processes beyond the boundaries of their own organizations. However, inter-organizational workflows are typically subject to conflicting constraints. On the one hand, there is a strong need for coordination to optimize the flow of work in and between the different organizations. On the other hand, the organizations involved are essentially autonomous and have the freedom to create or modify workflows at any point in time. These conflicting constraints complicate the development of languages and tools for cross-organizational workflow support.

* Corresponding author.

Recent development in Internet technology, and the emergence of the "electronic market makers", such as ChemConnect, Ariba, CommerceOne, Clarus, staples.com, Granger.com, VerticalNet, and mySAP.com have resulted in many XML-based standards for electronic commerce. The XML Common Business Library (xCBL) by CommerceOne, the Partner Interface Process (PIP) blueprints by RosettaNet, the Universal Description, Discovery and Integration (UDDI), the Electronic Business XML (ebXML) initiative by UN/CEFACT and OASIS, the Open Buying on the Internet (OBI) specification, the Open Application Group Integration Specification (OAGIS), and the BizTalk Framework are just some examples of the emerging standards based on XML. *These standards primarily focus on the exchange of data and not on the control flow among organizations.* Most of the standards provide standard *Document Type Definitions* (DTDs) or XML schemas for specific application domains (such as procurement). One of the few initiatives that also address the control flow is RosettaNet. The Partner Interface Process (PIP) blueprints by RosettaNet do specify interactions using UML activity diagrams for the Business Operational View (BOV) and UML sequence diagrams for the Functional Service View (FSV) in addition to DTDs for data exchange. However, the PIP blueprints are not executable and need to be predefined. Moreover, like most of the standards, RosettaNet is primarily focusing on electronic markets with long-lasting pre-specified relationships between parties with one party (such as the market maker) imposing rigid business rules.

Looking at existing initiatives, it can be noted that (until recently):

(1) process support for cross-organizational workflow has been neglected since lion's share of attention has gone to data and

(2) mainly pre-specified standardized processes have been considered (such as, market places, procurement, and so on).

Based on these observations, we developed the *eXchangeable Routing Language* (XRL). The idea to develop a language like XRL was raised in [26] and the definition of the language was given in [9]. XRL uses the syntax of XML, but contains constructs that embed the semantics of control flow. Moreover, XRL supports highly dynamic one-of-a-kind workflow processes. For example, we consider the "first trade problem", that is, the situation where parties have no prior trading relationship [29]. Clearly, the "first-trade problem" is the extreme case of highly dynamic one-of-a-kind workflow processes and therefore also the most difficult. To support highly dynamic one-of-a-kind workflow processes, XRL describes processes at the instance level. Traditional workflow modeling languages describe processes at the class or type level [23,27]. An XRL routing schema describes the partial ordering of tasks for one specific instance. The advantages of doing so are that:

(1) the workflow schema can be exchanged more easily,

(2) the schema can be changed without causing any problems for other instances, and

(3) the expressive power is increased.

The other side of the picture is that we have additional overhead, and management information is harder to obtain. The fact that the schema can be exchanged more easily outweighs, in our opinion, the additional overhead. The management information disadvantage can be lessened for a great deal when we restrict ourselves to changes that preserve a number of inheritance relations [6].

Workflow-modeling languages typically have problems handling a variable number of parallel or alternative branches [7]. In our research on workflow patterns [7], we compared the expressive power of many contemporary workflow management systems including COSA, HP Changengine, Forté Conductor, I-Flow, InConcert, MQ Series Workflow, R/3 Workflow, Staffware, Verve, and Visual WorkFlo using a set of workflow patterns (see *http://www.tm.tue.nl/it/research/patterns/*). Based on the workflow patterns supported by these systems, and their relative use in practice, we carefully selected the most relevant constructs for XRL. As a result, many of the workflow management systems mentioned above can be covered by XRL, which makes XRL a kind of least common multiple of these systems.

As was shown in [9], the semantics of XRL can be expressed in terms of Petri nets [34,35]. Unfortunately, this semantics did not allow for a direct use of these theoretical results and tools. This limitation was recognized in [10]. In this paper, we present a direct transformation from XRL to so-called *WorkFlow nets* (WF-nets), that is, the semantics of XRL is given in terms of WF-nets. WF-nets are a special subclass of Petri nets which possess an appealing correctness notion (the soundness property [1]), are based on strong theoretical results (such as, the link between soundness, liveness, and boundedness [1]), and are supported by powerful software (such as, the tool Woflan [40]). The transformation has been implemented in XSLT (eXtensible Stylesheet Language Transformations) and resulted in the tool XRL/Woflan.

XRL/Woflan builds on the workflow verification tool Woflan [39,40]. Developers of contemporary workflow management systems have virtually neglected correctness issues. As a result, in most workflow management systems, it is possible to design workflows which suffer from anomalies such as deadlocks and livelocks without any form of warning. Few tools provide any form of workflow verification support. The tools Woflan [40] and Flowmake [38] are two noteworthy exceptions. To complicate matters, more and more workflow management systems are used to support inter-organizational business processes, for example, in the context of Business-To-Business (B2B) E-commerce. Especially for open E-commerce (that is, doing business among parties having no prior trading relationship), the workflow support should be trustworthy in the sense that trading partners who do not know each other, and may even come from different countries and cultures, may conduct business with the assurance that their interests will be protected in the event that "things go wrong", whether by accident, negligence, or intentional fraud [29]. One of the prerequisites for this is the guarantee that the workflow process definitions do not contain any logical errors. Therefore, XRL/Woflan, the verification tool presented in this paper, is highly relevant for developers of inter-organizational workflows.

The remainder of this paper is organized as follows. Section 1 introduces XRL and gives an example of how a workflow can be represented in XRL. Section 2 introduces WF-nets. Then section 3 provides the formal semantics of XRL in terms of WF-nets. Based on these semantics, section 4 discusses the soundness of XRL routes, proposes a verification procedure that exploits the structural properties of certain XRL constructs and Petri-net-based reduction rules [34], and presents our tool XRL/Woflan. Section 5 demonstrates the extensibility of XRL by showing how new constructs may be added to XRL. Section 6 relates this paper to known research. Section 7 concludes the paper. Appendix A shows the DTD of XRL after the extensions from section 5 have been added. Appendix B shows the XRL route for processing a customer order that is introduced in section 1.

1. XRL: An XML based routing language

The focus of this paper is on verification and extensibility. Therefore, we limit ourselves to only a brief introduction to XRL and the workflow management system XRL/Flower.

1.1. Syntax of XRL

The syntax of XRL is completely specified by the DTD [16] shown in figure 1. *An XRL route* is a consistent XML document, that is, a well-formed and valid XML file with top element *route* (see figure 1).

The structure of any XML document forms a tree. In case of XRL, the root element of that tree is the route. A route contains exactly one *routing element*. A routing element (RE) is an important building block of XRL. It can either be simple (no child routing elements) or complex (one or more child routing elements). A complex routing element specifies *whether*, *when* and in *which order* the child routing elements are done.

XRL provides the following routing elements:

Task: Offer the given task to some resource and wait until the task has been performed. Afterwards, set all associated events.

Sequence: Start the child routing elements in the given order and wait until all have been performed.

```
<!ENTITY % routing_element
"task|sequence|any_sequence|choice|condition|parallel_sync|parall
el_no_sync|parallel_part_sync|parallel_part_sync_cancel|wait_all
wait_any|while_do|terminate">
<!ELEMENT route ((%routing_element;), event*)>
<!ATTLIST route
  name ID #REQUIRED
  created_by CDATA #IMPLIED
  date CDATA #IMPLIED>
```

Figure 1. The DTD of XRL.

```
<!ELEMENT task (event*)>
<!ATTLIST task
  name ID #REQUIRED
  address CDATA #REQUIRED
  role CDATA #IMPLIED
  doc_read NMTOKENS #IMPLIED
  doc_update NMTOKENS #IMPLIED
  doc_create NMTOKENS #IMPLIED
  result CDATA #IMPLIED
  status (ready|running|enabled|disabled|aborted|null) #IMPLIED
  start_time NMTOKENS #IMPLIED
  end_time NMTOKENS #IMPLIED
  notify CDATA #IMPLIED>
<!ELEMENT event EMPTY>
<!ATTLIST event
  name ID #REQUIRED>
<!ELEMENT sequence ((%routing_element;|state)+)>
<!ELEMENT any_sequence ((%routing_element;)+)>
<!ELEMENT choice ((%routing_element;)+)>
<!ELEMENT condition ((true|false)*)>
<!ATTLIST condition
  condition CDATA #REQUIRED>
<!ELEMENT true (%routing_element;)>
<!ELEMENT false (%routing_element;)>
<!ELEMENT parallel_sync ((%routing_element;)+)>
<!ELEMENT parallel_no_sync ((%routing_element;)+)>
<!ELEMENT parallel_part_sync ((%routing_element;)+)>
<!ATTLIST parallel_part_sync
  number NMTOKEN #REQUIRED>
<!ELEMENT parallel_part_sync_cancel ((%routing_element;)+)>
<!ATTLIST parallel_part_sync_cancel
  number NMTOKEN #REQUIRED>
<!ELEMENT wait_all ((event_ref|timeout)+)>
<!ELEMENT wait_any ((event_ref|timeout)+)>
<!ELEMENT event_ref EMPTY>
<!ATTLIST event_ref
  name IDREF #REQUIRED>
<!ELEMENT timeout ((%routing_element;)?)>
<!ATTLIST timeout
  time CDATA #REQUIRED
  type (relative|s_relative|absolute) "absolute">
<!ELEMENT while_do (%routing_element;)>
<!ATTLIST while_do
  condition CDATA #REQUIRED>
<!ELEMENT terminate EMPTY>
<!ELEMENT state EMPTY>
```

Figure 1. (Continued.)

Any_sequence: Start the child routing elements in any order and wait until all have been performed.

Choice: Start one of the child routing elements and wait until it has been performed.

Condition: If the given condition holds, start the child routing elements of all *true* child elements in parallel and wait until all have been performed. Otherwise, start the child routing elements of all *false* child elements in parallel and wait until all have been performed. A condition may have any number (even none) of *true* and *false* child elements.

Parallel_sync: Start the child routing elements in parallel and wait until all have been performed.

Parallel_no_sync: Start the child routing elements in parallel but do not wait for any of them.

Parallel_part_sync: Start the child routing elements in parallel and wait until the given number of child routing elements has been performed.

Parallel_part_sync_cancel: Start the child routing elements in parallel, wait until the given number of child routing elements has been performed and cancel the remaining child routing elements if possible.

Wait_all: Wait until either all associated events are set, or wait until the given deadline of some child timeout element has expired. If this timeout element has a child routing element, start it and wait until is has been performed.

Wait_any: Wait until either at least one of the associated events is set, or wait until the given deadline of some child timeout element has expired. If this timeout element has a child routing element, start it and wait until is has been performed.

While_do: As long as the given condition holds, start the child routing element and wait until it has been performed.

Terminate: End this workflow instance.

As mentioned before, the routing elements of XRL are based on a thorough analysis of the workflow patterns supported by leading workflow management systems.

1.2. Example: An electronic bookstore

We illustrate XRL using an example inspired by electronic bookstores, such as Amazon [13] and Barnes and Noble [14]. The activity diagram in figure 2 shows a typical order flow. This figure gives the four parties or organizations involved (that is, customer, bookstore, publisher and shipper), and the steps performed by each one. The arrows show the sequence in which these steps are carried out. Some of the details of the real-world process are omitted from this diagram for clarity.

The workflow represented by the sequence diagram is described in XRL in appendix B. The XRL rendition covers the typical order flow of figure 2, and also some more details. First, the customer places an order (task *place_c_order*). This customer order is sent to and handled by the bookstore (task *handle_c_order*). The electronic bookstore is a virtual company that has no books in stock. Therefore, the bookstore transfers the order for the desired book to the first appropriate publisher (task *place_b_order*). We use the

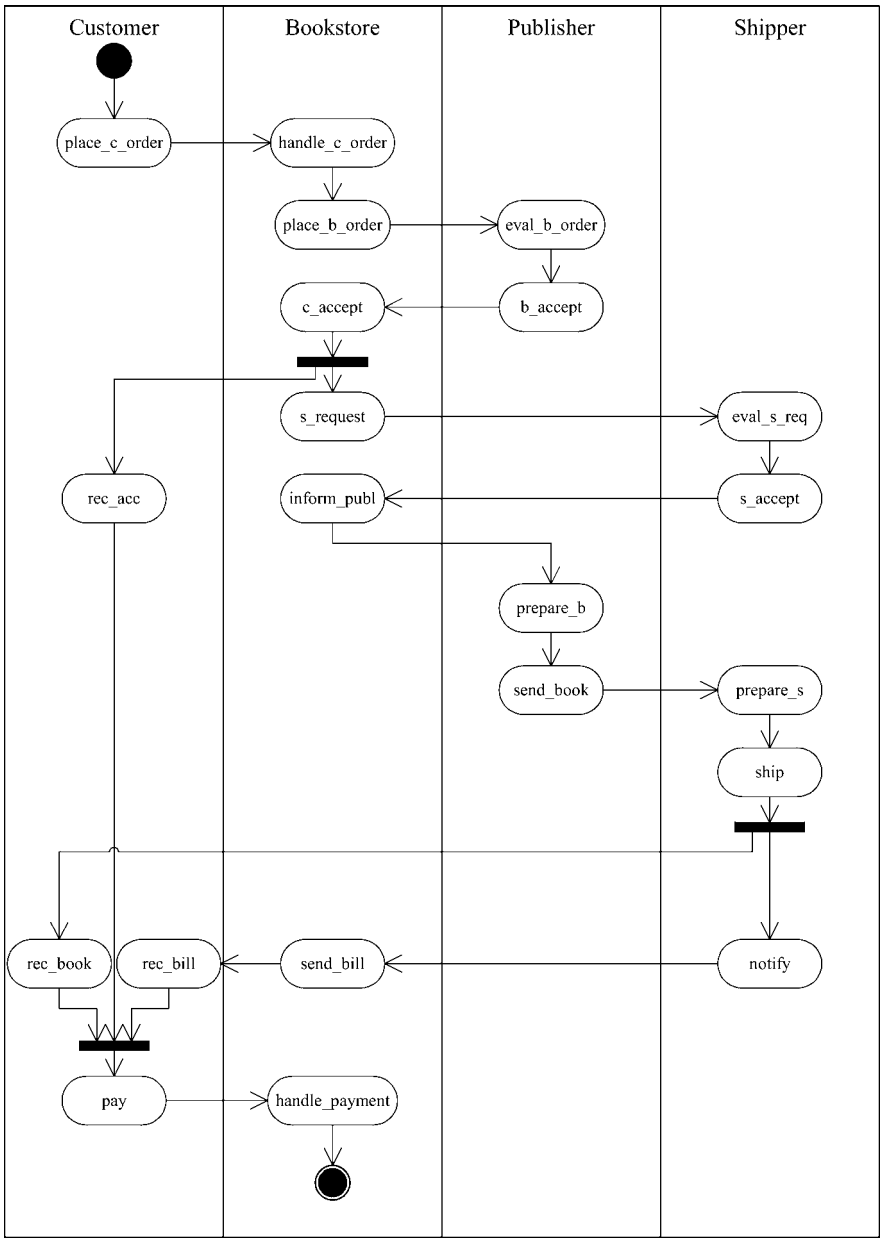

Figure 2. Typical order flow for an electronic bookstore.

term “bookstore order” for the transferred order. The publisher evaluates the bookstore order (task *eval_b_order*). By shipping the XRL route back to the bookstore, the publisher informs the bookstore about the availability of the book. If the book is not available, the bookstore decides (task *decide)* to either search for an alternative publisher (task *alt_publ*) or to reject the customer (task *c_reject*). If the customer receives a negative

answer (task *rec_decl*), the workflow terminates. If the book is available (task *c_accept*), the customer is informed (task *rec_acc*) and the bookstore continues processing the customer order. The bookstore sends a request to the shipper (task *s_request*), the shipper evaluates the request (task *eval_s_req*) and either accepts (task *s_accept*) or rejects (task *s_reject*). If the bookstore receives a negative answer, it searches for another shipper.

After a shipper has been found, the publisher is informed (task *inform_publ*), the publisher prepares the book for shipment (task *prepare_b*), and the book is sent from the publisher to the shipper (task *send_book*). The shipper prepares the shipment to the customer (task *prepare_s*) and actually ships the book to the customer (task *ship*). The customer receives the book (task *rec_book*) and the shipper notifies the bookstore (task *notify*). The bookstore sends the bill to the customer (task *send_bill*). After receiving both the book and the bill (task *rec_bill*), the customer makes a payment (task *pay*). Then the bookstore processes the payment (task *handle_payment*) and the inter-organizational workflow terminates.

The XRL route shown in appendix B just illustrates some of the XRL routing elements. The description is far from complete, for example, the detailed descriptions of tasks and conditions have not been included. Please note that, since an XRL route specifies the life cycle of a particular workflow instance (that is, work case), any instance can be modified without reference to some underlying workflow schema type.

1.3. XRL/Flower

Based on the XRL semantics, we developed a workflow management system, named *XRL/Flower*, to support XRL. XRL/Flower benefits from the fact that it is based on both XML and Petri nets. Standard XML tools can be deployed to parse, check, and handle XRL documents. The Petri-net representation allows for a straightforward and succinct implementation of the workflow engine. XRL constructs are automatically transformed into Petri-net constructs. On the one hand, this allows for an efficient implementation. On the other hand, the system is easy to extend:

> For supporting a new routing primitive, only the transformation to the Petri-net format needs to be added and the engine itself does not need to change.

Figure 3 shows the architecture of the toolset involving XRL/Flower and XRL/Woflan. Using both the control flow data for the workflow case and the case specific data, the Petri-net engine computes the set of enabled tasks, that is, the set of *work items* that are *ready*. The engine sends this set to the work distribution module. Based on information on the organizational roles and users, the work distribution module sends e-mails offering the work item to certain users who are qualified to work on it. A user would receive an e-mail notification with a URL pointing to new work item(s) waiting for her. By clicking on the URL, the user accepts the work item; thus, the *work item* becomes an *activity* for a specific user, and other users to whom the work item was also offered are notified that it has already been accepted and is no longer available to them. A user who has accepted an activity may perform work on it either at acceptance time or

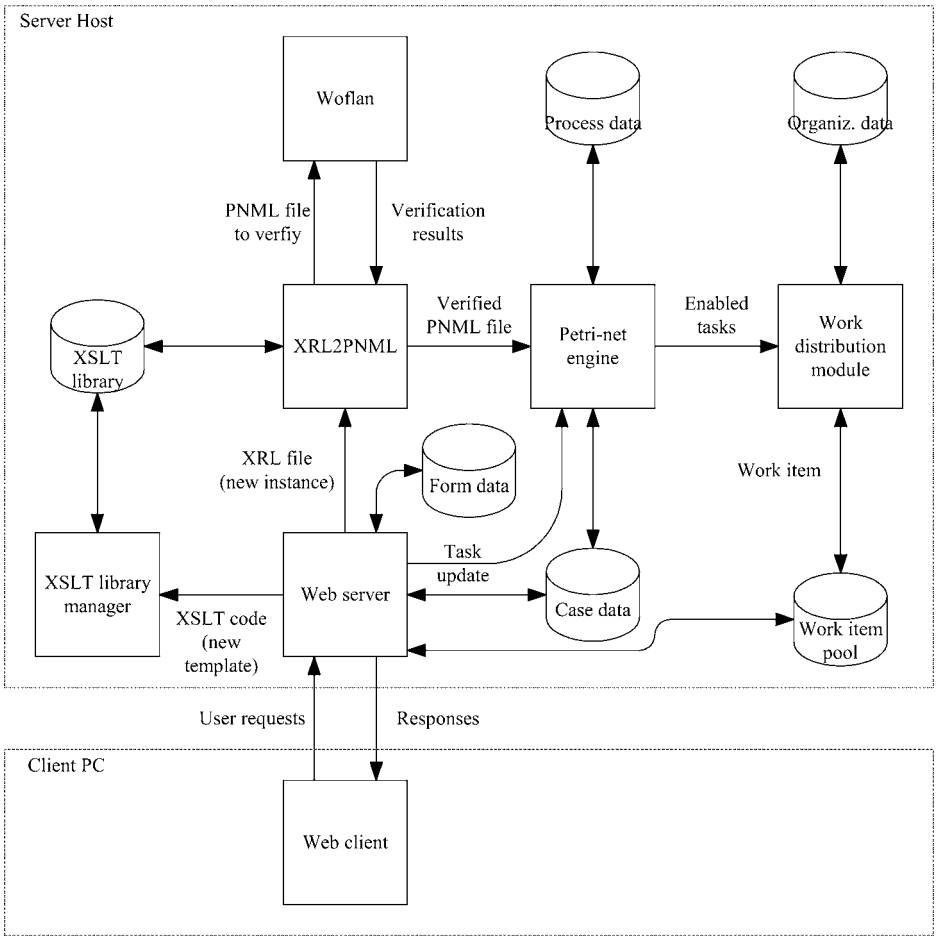

Figure 3. A detailed architecture for implementing inter-organizational workflows using XRL/Woflan.

later. In order to enable a user to perform an activity, the web server fills the appropriate form template with the case specific data for the activity. The user indicates completion of an activity by, say, pressing a submit button. The web server stores the updated case data and signals the Petri-net engine that the activity has been completed. The Petri-net engine then recomputes a new set of work items that are ready. The user can also start an XRL instance by sending the corresponding XRL file to the web server. The web server forwards the XRL file to the XRL2PNML module that transforms XRL to PNML (Petri-Net Markup Language), which is a standard representation language for a Petri net in XML format [24].

Figure 4 shows how the XRL2PNML module makes the transformation from XRL to PNML. First, it transforms the XRL file to two PNML files: one for verification and one for enactment. The first PNML file (for verification) can be considerably smaller in size than the second (for enactment), but their soundness characteristics are the same.

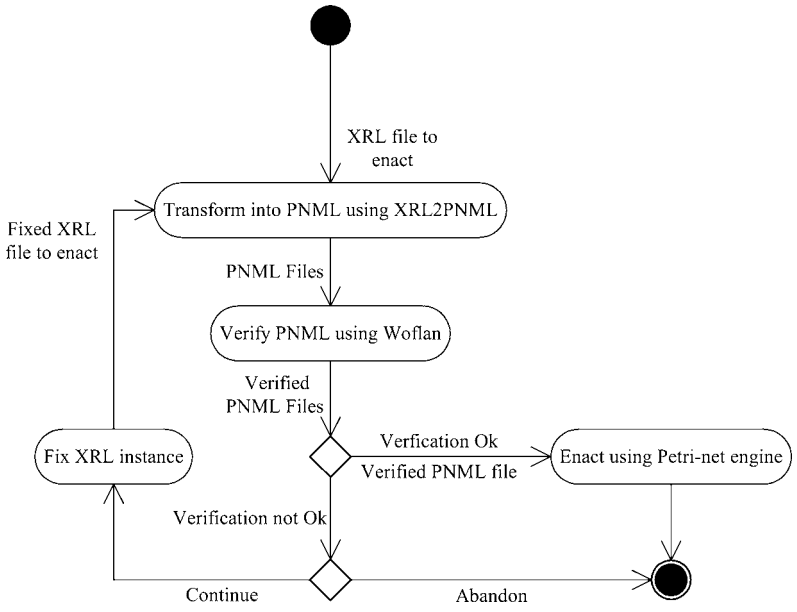

Figure 4. Transformation from XRL to PNML.

The first PNML file is verified using the XRL/Woflan tool. Based on the result of the verification, either the second PNML file is sent to the Petri-net engine for enactment, or the user is informed that the XRL instance contains flaws. In the latter case, the user may either abandon the new instance, or modify it to fix the errors. Of course, the fixed instance is also verified before it is enacted. If the expressive power of the current XSLT library does not satisfy the user's needs, she may decide to extend this library by adding a new template to it. Figure 5 shows how this is done. First, the user describes the DTD of the new pattern. Second, she writes the XSLT code that will transform the new template to the appropriate PNML code. After the new XSLT code is verified, it is incorporated into the XSLT library.

2. WorkFlow nets

Before we present the new transformation, we briefly introduce some of the concepts related to WF-nets. We assume some basic knowledge of Petri nets [34,35].

A Petri net that models the control-flow dimension of a workflow, is called a *WF-net*. Recall that a WF-net specifies the dynamic behavior of a single case in isolation.

Definition 1 (WF-net). A Petri-net $PN = (P, T, F)$ is a WorkFlow net (WF-net) if and only if:

(1) There is one source place $i \in P$, that is, one place without any predecessors.

(2) There is one sink place $o \in P$, that is, one place without any successors.

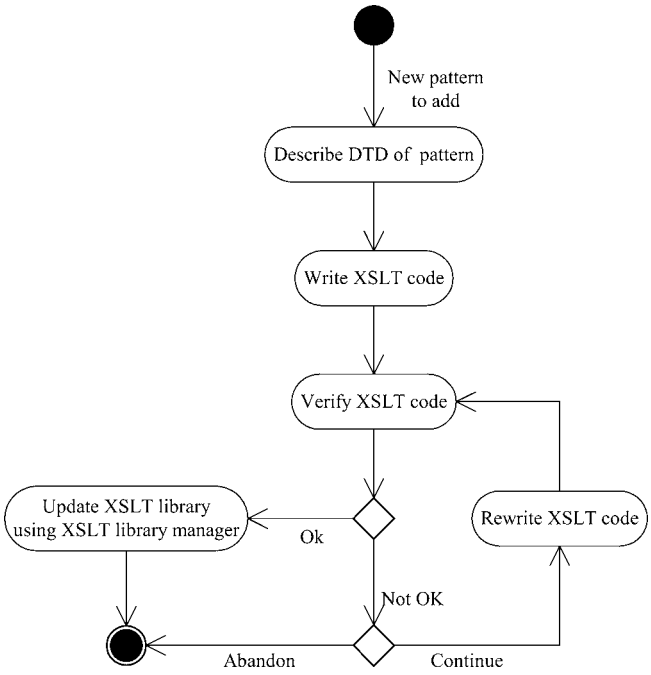

Figure 5. Adding a new template to the library.

(3) Every node $x \in P \cup T$ is on a path from i to o.

A WF-net has one input place (i) and one output place (o) because any case handled by the procedure represented by the WF-net is created when it enters the workflow management system and is deleted once it is completely handled by the workflow management system, that is, the WF-net specifies the life-cycle of a case. The third requirement in definition 1 has been added to avoid 'dangling tasks', that is, tasks that do not contribute to the processing of cases.

For sake of completeness, we mention that the original definition of WF-nets did not include arc weights (sometimes also called multiple arcs). However, as mentioned in [40], it is straightforward to extend WF-nets by allowing arc weights. For the remainder of this paper, we assume that arc weights are allowed in WF-nets.

The three requirements stated in definition 1 can be verified statically, that is, they only relate to the structure of the Petri net. However, there is another requirement that should be satisfied:

> For any case, the procedure will terminate eventually and upon termination there is a token in place o and all the other places are empty.

Moreover, there should be no dead tasks, that is, it should be possible to execute an arbitrary task by following the appropriate route through the WF-net. These two additional requirements correspond to the so-called *soundness property*.

Definition 2 (Soundness). A procedure modeled by a WF-net $PN = (P, T, F)$ is sound if and only if:

(1) For every state M reachable from state i, there exists a firing sequence leading from state M to state o.

(2) State o is the only state reachable from state i with a token in place o.

(3) There are no dead transitions when starting in state i.

Note that the soundness property relates to the dynamics of a WF-net. The first requirement in definition 2 states that starting from the initial state (state i), it is always possible to reach the state with one token in place o (state o). The second requirement states that the moment a token is put in place o, all the other places should be empty. The last requirement states that there are no dead transitions (tasks) in the initial state i.

In [1], it is shown that there is an interesting relation between soundness and well-known Petri-net properties such as liveness and boundedness. A WF-net is sound if and only if the short-circuited net (that is, the net obtained by linking the sink place to the source place) is live and bounded. This result illustrates that standard Petri-net-based analysis techniques can be used to verify soundness.

3. Semantics of XRL in terms of WF-nets

The DTD shown in figure 1 only describes the syntax of XRL and does not specify the semantics. To provide operational semantics of the routing elements we transform each routing element mentioned in the DTD into a Petri net. Such a transformation was given in [9]. However, as indicated earlier, this transformation does not necessarily yield WF-nets. Therefore, we have modified the transformation given in [9] such that XRL routes are transformed into WF-nets. First, we discuss the problems we encountered when trying to transform XRL routes into WF-nets. Second, we present for the route and for each routing element its Petri-net semantics. Third, we transform the example from section 1 (see appendix B for the XRL specification) into a WF-net.

3.1. Discussion of the semantics

In section 1 we already observed that the structure of an XRL document forms a tree, with the route element as root. Many routing elements interface only with their parent element and their child elements. For this reason, we propose to 'copy' this tree structure to the resulting WF-net: Every routing element is replaced by some Petri-net fragment that interfaces with the Petri nets associated with its parent child elements nets. The exceptions to this rule are the terminate routing elements and the task, wait_all and wait_any routing elements (when events are involved). We deal with these exceptions later on; first we focus on the interface between a parent and a child element.

At first glance, only two places seem to be necessary for the communication between a parent and child: one from parent to child indicating that the child can be started,

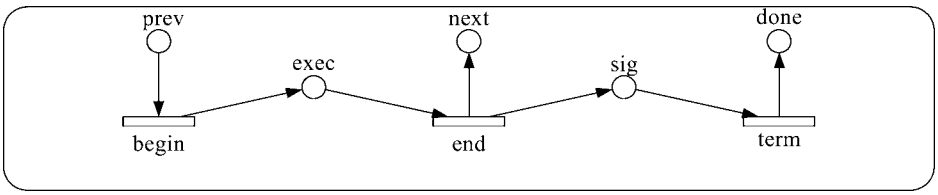

Figure 6. Basic routing element.

and one from child to parent indicating that the child has been performed. However, according to the descriptions given in section 1, there are three routing elements that do not wait until all child elements have been performed: parallel_no_sync, parallel_part_sync, and parallel_part_sync_cancel. As a result, an instance might reach the point of completion while still, somewhere deep inside some subtree, elements still can be performed. Take for instance the simple example where the route contains a parallel_no_sync containing only one task. Because the parallel_no_sync does not wait until the task has been performed, the entire instance might reach the point of completion before the task has been accepted and started. Recall that soundness requires that the remainder of the entire Petri net is empty when a token is put into the sink place, that is, a Petri-net fragment associated with any routing element has to be empty at that point. Therefore, before actually reaching completion, we have to wait until all these fragments are empty. For this reason, we introduce a third communication place: from child to parent indicating that the Petri-net fragment associated with the entire subtree of the child is now empty of tokens.

Figure 6 shows the basic routing element. A token in place *prev* indicates that the routing element can be started. A token in place *exec* indicates that the routing has started. However, in many routing elements this *exec* place is redundant, and therefore omitted. When the routing element has been performed, it puts a token in places *next* and *sig*. The token in place *next* informs the parent that this element has been performed, while the token in place *sig* indicates that the routing element is waiting until all descending routing elements have been performed too, that is, until all subtrees are empty. If all are empty, it puts a token in place *done*, indicating that it is now empty of tokens except for the token in the *done* place.

When a terminate occurs, the entire instance, that is, the route itself, is to be completed. This clearly has an effect on the instance level. In Petri nets, it is hard to foresee all possible reachable states, and to add transitions such that from every reachable state we can reach state o. A simple observation alleviates this problem: If we bypass every task, wait_all, and wait_any, the instance automatically reaches completion! The tasks need to be bypassed because we cannot allow that a task is started after a terminate occurred. Both wait routing elements need to be bypassed because they are not allowed to wait any longer after a terminate occurred. Note that this solution assumes that running tasks are not preempted when a terminate occurs. As a result, we can treat a terminate almost in a similar way as we treat an event. Because all tasks need to have both places *terminate* and *nonterminate* present, and all waits need to

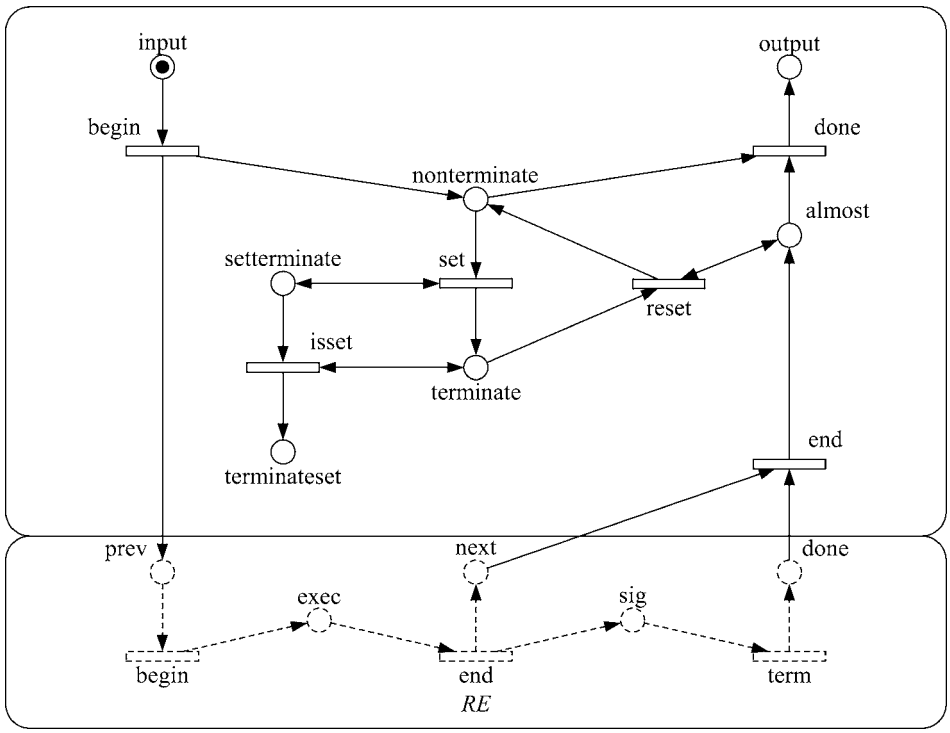

Figure 7. Semantics of route.

have the place *terminate* present, we incorporate the global part of the terminate in the route.

Figure 7 shows the semantics of the route element, containing the terminate at the instance level. For sake of clarity, a stub (drawn dotted) replaces the top routing element (the only child routing element of the route element). Initially, place *input* (which corresponds to place i in definition 1) contains one token, indicating that the instance has not started yet. Transition *begin* starts the instance and

(1) starts the top child routing element,

(2) enables the terminate,

(3) enables all events, and

(4) enables every parallel_part_sync or parallel_part_sync_cancel.

Note that items (3) and (4) are explained later on and not shown in this net. Item (3) is shown in figure 8 and item (4) is shown in figures 16 and 17.

After the top routing element and all its descendants have completed, transition *end* initiates the completion phase of the instance. First, all events and the terminate are reset (which happens while place *almost* contains a token). Second and last, transition *done* completes the instance and

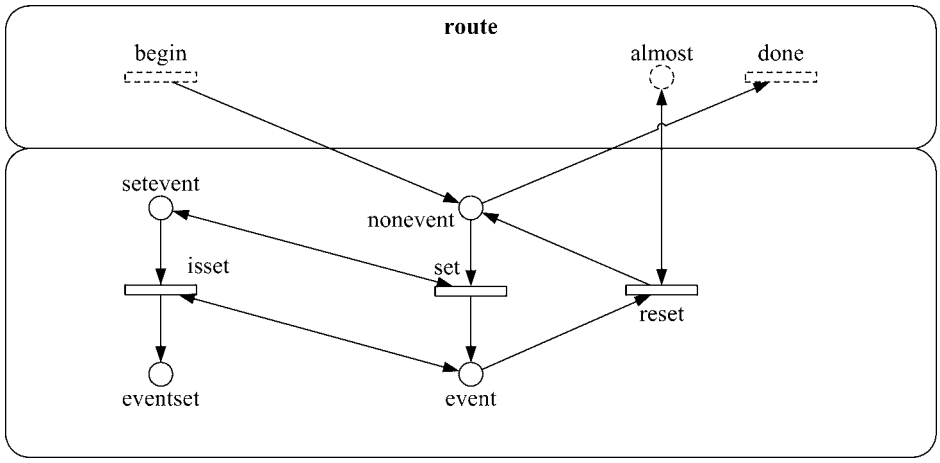

Figure 8. Event on global level.

(1) removes the token from place *almost*,

(2) disables the terminate,

(3) disables all events, and

(4) disables every parallel_part_sync and parallel_part_sync_cancel.

Again, items (3) and (4) are explained later on and not shown in this net.

Of course, if no terminates are present in the XRL route, the part concerning the terminate is discarded from the semantics of the route.

Like terminate, events are defined on the top level of the instance, that is, the route level, not on some local level deeply nested in some subtree: If some task in some subtree sets a certain event, then some wait in some other subtree might be affected. For this reason, *events are handled on the instance level*: For every event, we add a Petri-net fragment that manages the event. Such a fragment interfaces only with the Petri nets associated with the route itself (for enabling and disabling the event), tasks (for setting the event), and both waits (for testing the event). On the instance level, two places are introduced for every event: *event* and *nonevent.* Only one of them may contain a token: Either the event has occurred (*event* contains a token) or not (*nonevent* contains a token). When the instance is started, each event is enabled by putting a token into place *nonevent,* and when the instance completes, it is disabled by removing the token from that place. When the event has been set, transition *reset* can move the token from *event* to *nonevent* when the instance is completing. A task sets the event by putting a token in place *setevent* and waiting until a token is put into place *eventset*. The transitions *set* and *isset* take care that the event is set when this place contains a token. They also take care that this token is moved to place *eventset* when the event has been set. Figure 8 shows an event on the instance level.

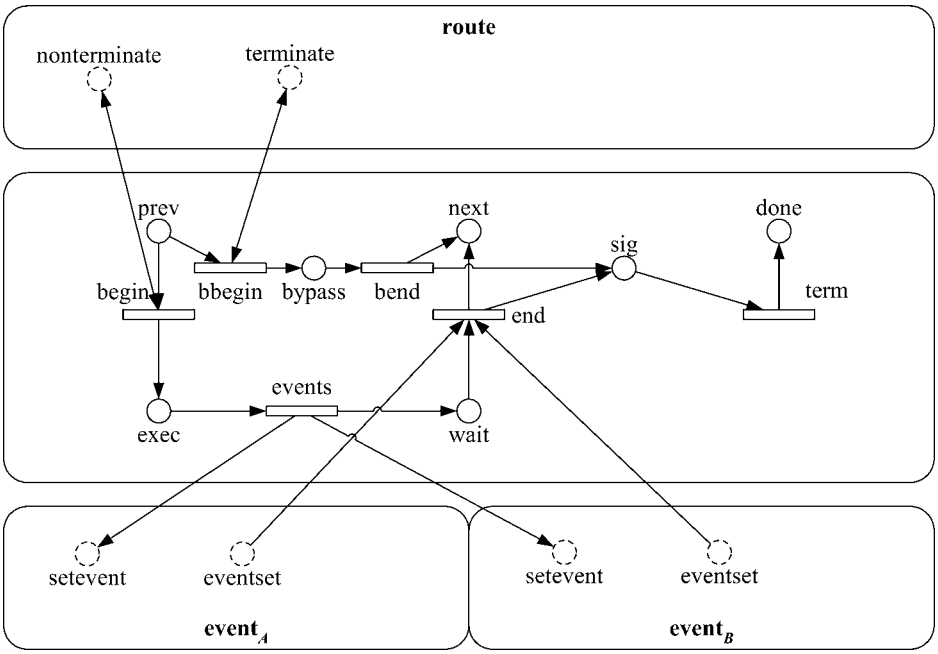

Figure 9. Semantics of task.

3.2. The Petri-net semantics

Figure 9 shows the semantics of the task routing element. When a task is started, what happens depends on whether or not a terminate has occurred. If a terminate has occurred, the transitions *bbegin* and *bend* bypass the normal execution of the task. Otherwise, transition *begin* starts the execution by offering the task to some resources. After the task has been performed by some resource, transition *events* has the appropriate events set (could be none), and the task waits until these events are set. After all have been set, transition *end* signals that the task has been performed. Because there are no child routing elements, this automatically results in an empty subtree, which is signaled too.

Figure 10 shows the semantics of the sequence routing element. Transition *begin* starts the sequence, and puts a token in the place *prev* of the first child routing element. Doing so, this routing element can be started. When this routing element has been performed, it puts a token in its place *next*. Transition $next_1$ moves this token to the place *prev* of the next child routing element, and so on. When the last child routing element has been performed, transition *end* puts a token in place *next*, signaling that the entire sequence has been performed. After firing *end*, the sequence waits until all descending routing elements have been performed. If all have been performed and hence their subtrees are empty, transition *term* puts a token in place *done*.

Figure 11 shows the semantics of the any_sequence routing element. Transition *begin* starts the any_sequence, and puts tokens in the place *prev* for every child routing element, and a token in the place *exec*. The token in place *exec* guarantees the mutual

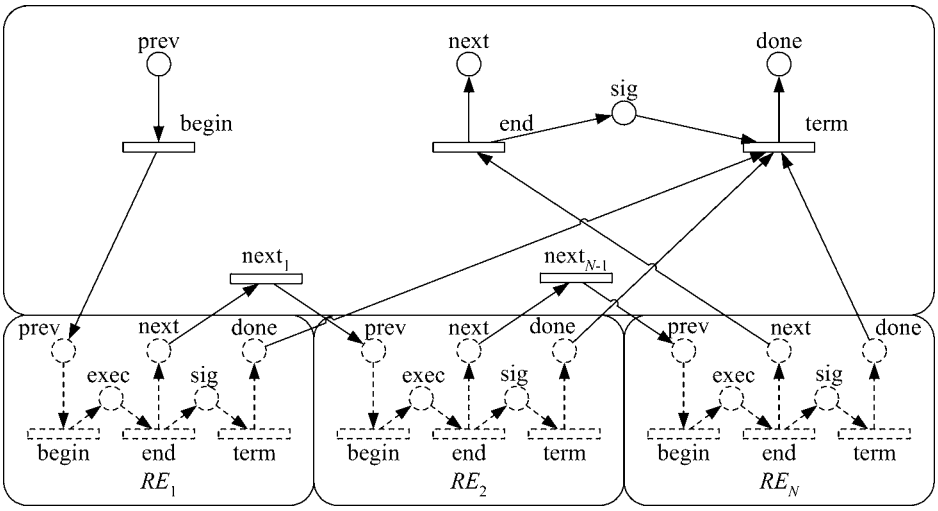

Figure 10. Semantics of sequence.

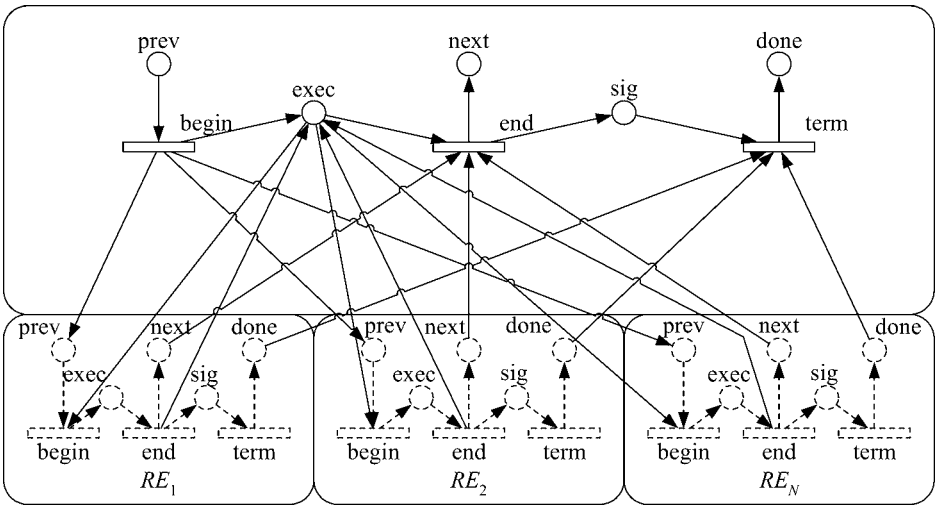

Figure 11. Semantics of any_sequence.

exclusion of the child routing elements. A child routing element can only start if place *exec* contains a token, and when it starts, it removes that token. When the child routing element has been performed and transition *end* fires, the token is put back. After all child routing elements have been performed, the transition *end* puts a token in place *next*, signaling that the entire any_sequence has been performed. After firing *end*, the any_sequence waits until all descending routing elements have performed. If all have been performed and hence heir subtrees are empty, it puts a token in place *done*.

Figure 12 shows the semantics of the choice routing element. Transition *begin* starts the *choice*, and puts a token in every child's *prev* place. The first child routing

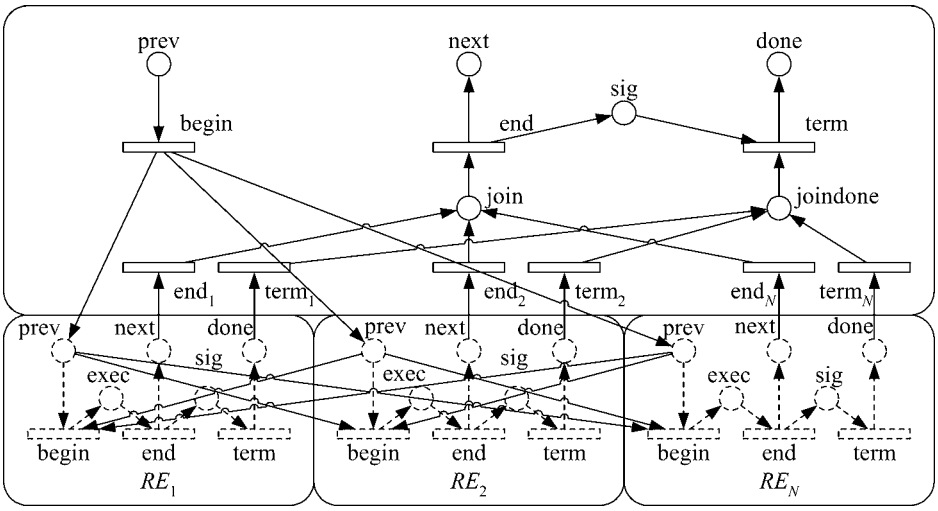

Figure 12. Semantics of choice.

element that fires its transition *begin*, disables the other child routing elements. After this first child routing element has been performed, its transition *end* fires, which puts a token in place *join*. As a result, transition *end* fires, signaling that the choice has been performed. After firing *end*, the choice waits until the first child routing element and all its descending routing elements have been performed. If all have been performed and hence their subtrees are empty, the child's transition *term* fires, followed by the choice's transition *term,* which puts a token in its place *done*.

Figure 13 shows the semantics of the condition routing element. Transition *begin* starts the condition, and puts a token in place *split*. At this point, the condition is evaluated. If the condition is evaluated to true, transition *tbegin* fires, otherwise transition *fbegin* fires. Assume without loss of generality that transition *tbegin* fires. This transition puts a token in place *texec* and enables the child routing element of every *true* child element. After all these 'grandchild' routing elements have been performed, transition *tend* fires, which puts tokens in places *join* and *tsig*. At this point, transition *end* fires, signaling that the condition has been performed. After all descending routing elements of the *true* child elements have also have been performed, transition *tterm* fires, followed by transition *term*, which puts a token in place *done*.

Figure 14 shows the semantics of the parallel_sync routing element. Transition *begin* starts the parallel_sync, and puts a token in the place *prev* for every child routing element. After all child routing elements have been performed, transition *end* fires, which puts a token in place *next*. After firing *end*, the parallel_sync waits until all descending routing elements have been performed. If all have been performed, transition *term* fires, which puts a token in place *done*.

Figure 15 shows the semantics of the parallel_no_sync routing element. Transition *begin* starts the parallel_no_sync, and puts a token in place *prev* for every child routing element and a token in place *exec*. As a result of the token in place *exec*, transition

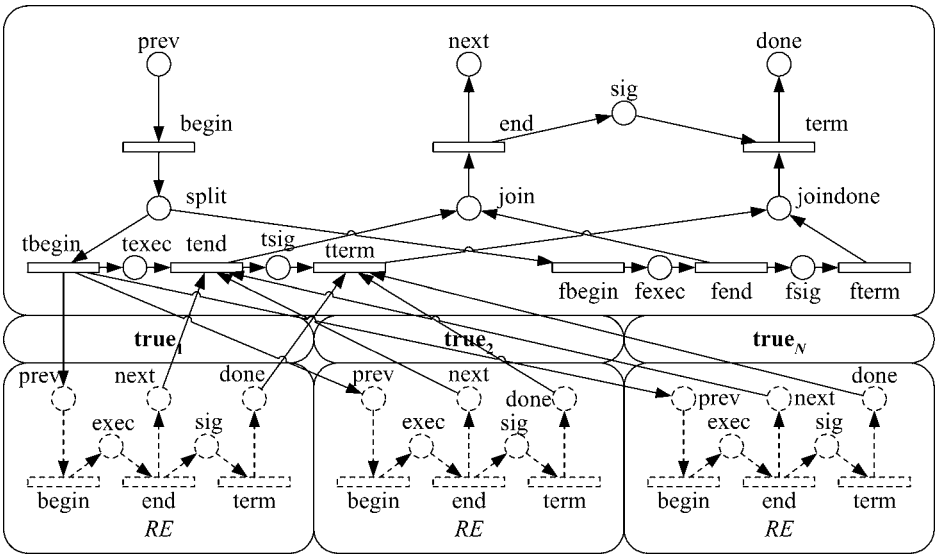

Figure 13. Semantics of condition.

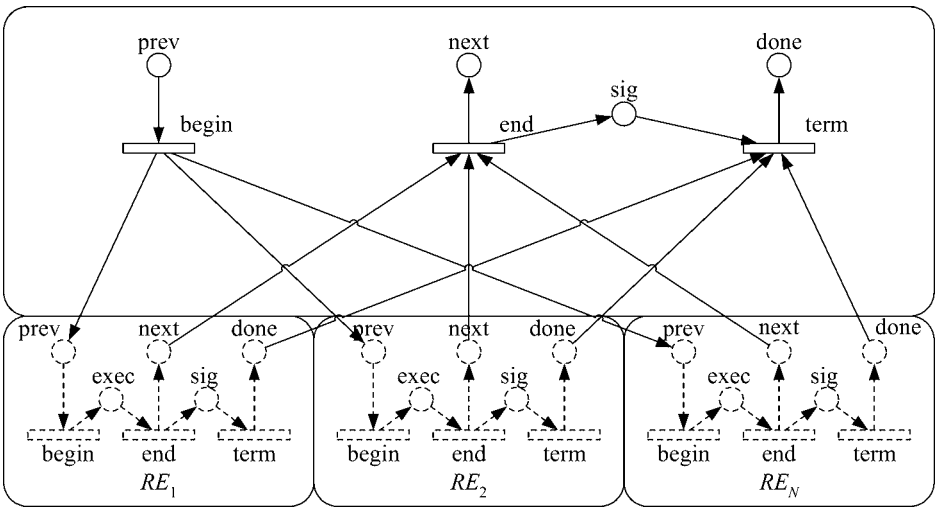

Figure 14. Semantics of parallel_sync.

end can fire immediately, signaling that the parallel_no_sync has been performed. After firing *end*, the parallel_no_sync waits until all descending child routing elements have been performed. If all have been performed, transition *term* fires, which puts a token in place *done*.

Figure 16 shows the semantics of the parallel_part_sync routing element. Transition *begin* starts the parallel_part_sync, and puts a token in place *prev* for every child routing element and a token in place *exec*. After any child routing element has been per-

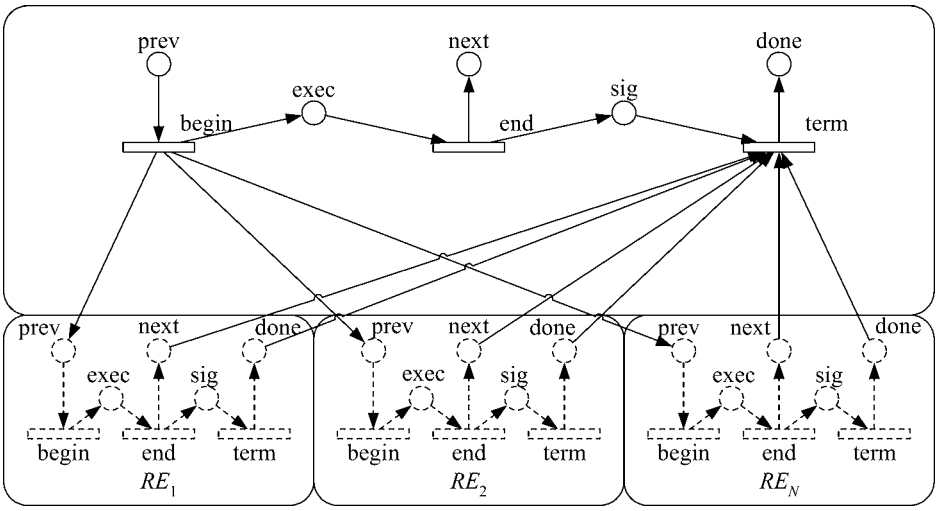

Figure 15. Semantics of parallel_no_sync.

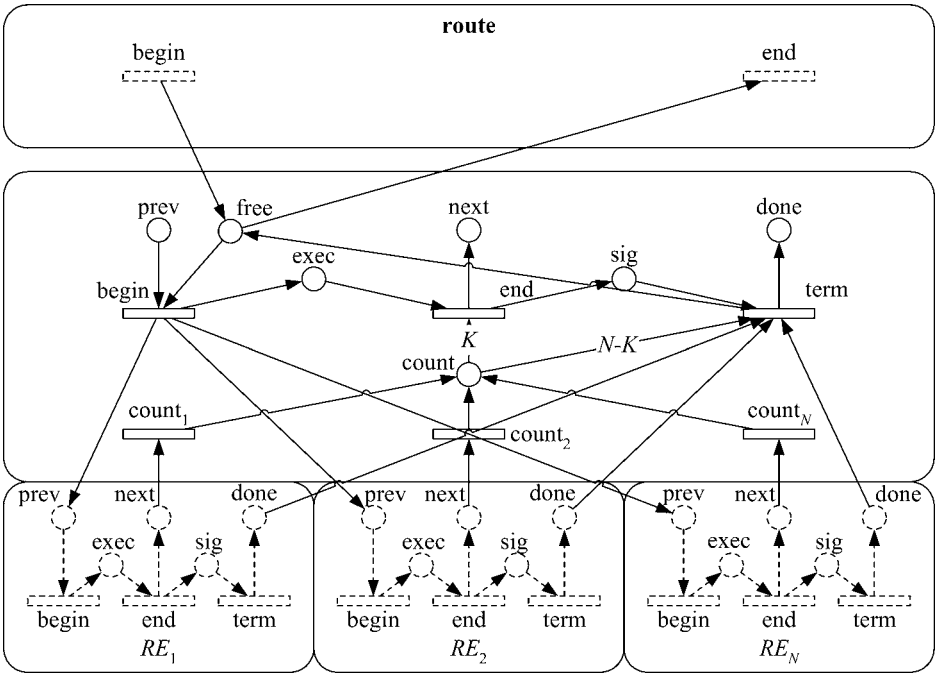

Figure 16. Semantics of parallel_part_sync.

formed, the appropriate transition $count_i$ fires, which moves the token to the place *count*. After K tokens have been moved this way, that is, after at least K child routing elements have been performed, transition *end* fires, signaling that the parallel_part_sync has been performed, and puts a token in place *next*. Note that the place *exec* is not redundant for

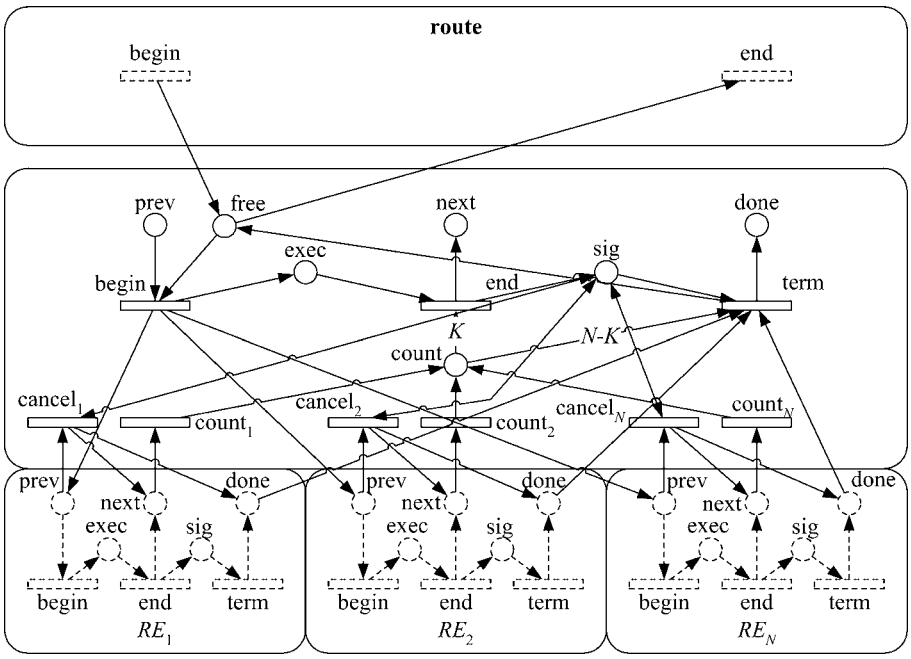

Figure 17. Semantics of parallel_part_sync_cancel.

this routing element: if we had removed the place *exec*, transition *end* could fire $\lfloor N/K \rfloor$ times. For instance, if there were 10 child routing elements ($N = 10$) and the given number were 3 ($K = 3$), transition *end* could fire 3 times, signaling 3 times that the parallel_part_sync has been performed, which is clearly an error. After firing *end*, the parallel_part_sync waits until the remaining child routing elements and all descending routing elements have been performed. If all have been performed, transition *term* fires, which puts a token in place *done*.

To avoid problems with recurrent parallel_part_syncs, that is, if a parallel_part_sync is embedded in a while_do, we prevent the parallel_part_sync to be recurrent by introducing the place *free*. As indicated in figure 16, transition *begin* of the route puts a token in place *free*, and transition *end* of the route removes the token. Note that transition *begin* has to fire before any routing element is started, and that *end* can only fire after all routing elements have been performed.

Figure 17 shows the semantics of the parallel_part_sync_cancel routing element. This semantics is an extension of the semantics of the parallel_part_sync: transitions $cancel_i$ have been added. After it has been performed, any child routing element that has not started yet can be cancelled by firing the appropriate transition $cancel_i$. When transition $cancel_i$ fires, it removes the token from the place *prev* of the ith child routing element, and puts tokens in its places *next* and *done*.

Figure 18 shows the semantics of the wait_all routing element. Transition *begin* starts the wait_all, and puts a token in place *wprev*. After firing *begin*, the wait_all waits until

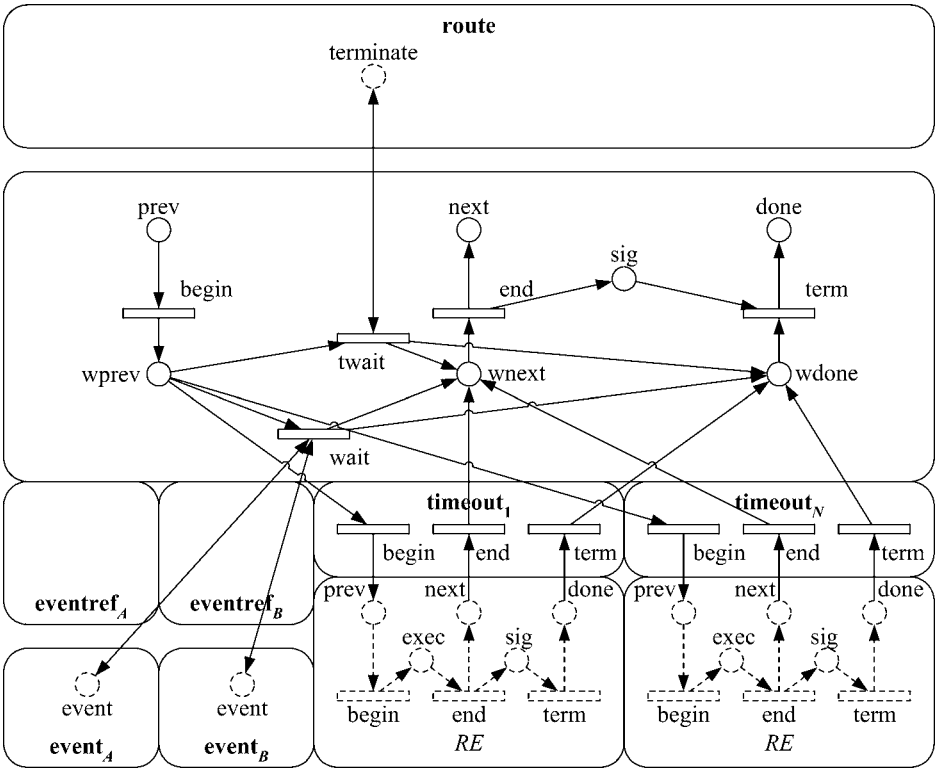

Figure 18. Semantics of wait_all.

(1) all events referred to have occurred,

(2) a terminate has occurred, or

(3) any of deadlines associated with the timeouts are exceeded.

If all events referred to have occurred, transition *wait* fires, which puts tokens in places *wnext* and *wdone*. If a terminate has occurred, transition *twait* fires, which also puts tokens in places *wnext* and *wdone*. If a timeout occurs, the child routing element of this timeout is started. After this child routing element has been performed, a token is put in place *wnext*. After all descending routing elements of the timeout have been performed, a token is put in place *wdone*. When place *wnext* contains a token, transition *end* fires, signaling that the wait_all has been performed, and puts a token in place *next*. After firing *end*, the wait_all waits until place *wdone* contains a token. If this place contains a token, transition *term* fires and puts a token in place *done*.

Figure 19 shows the semantics of the wait_any routing element. Transition *begin* starts the wait_any, and puts a token in place *wprev*. After firing *begin*, the wait_any waits until either

(1) any event referred to has occurred,

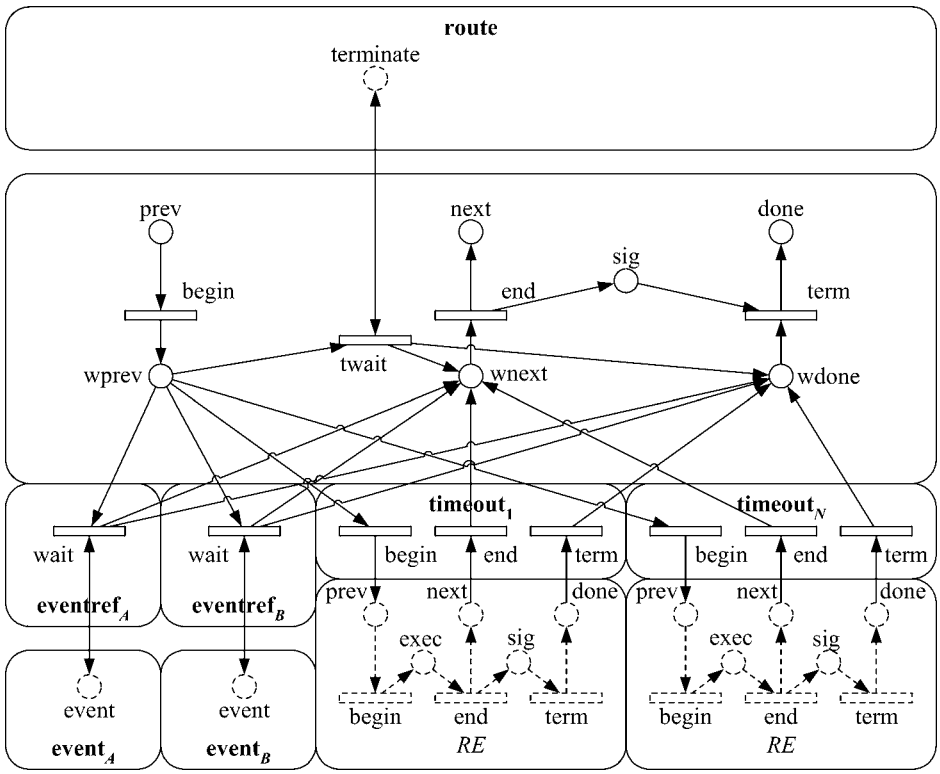

Figure 19. Semantics of wait_any.

(2) a terminate has occurred, or

(3) any of the timeouts occurs.

If any event referred to has occurred, its transition *wait* fires, which puts tokens in places *wnext* and *wdone.* If a terminate has occurred, transition *twait* fires, which also puts tokens in places *wnext* and *wdone*. If a timeout occurs, the child routing element of this timeout is started. After this child routing element has been performed, a token is put in place *wnext*. After all descending routing elements of the timeout have been performed, a token is put in place *wdone*. When place *wnext* contains a token, transition *end* fires, signaling that the wait_any has been performed, and puts a token in place *next*. After firing *end*, the wait_any waits until place *wdone* contains a token. If this place contains a token, transition *term* fires and puts a token in place *done*.

Figure 20 shows the semantics of the while_do routing element. Transition *begin* starts the while_do, and puts a token in place *next* of its child routing element, and X tokens in the place *done*, where X is a positive number that serves as a parameter to limit the number of concurrent instances. After firing *begin*, the condition is evaluated (by the enactment server). If the condition evaluates to true, the transition *true* fires, which starts another iteration of the child routing element. If the condition evaluates to false, transition *end* fires, signaling that the while_do has been performed and puts

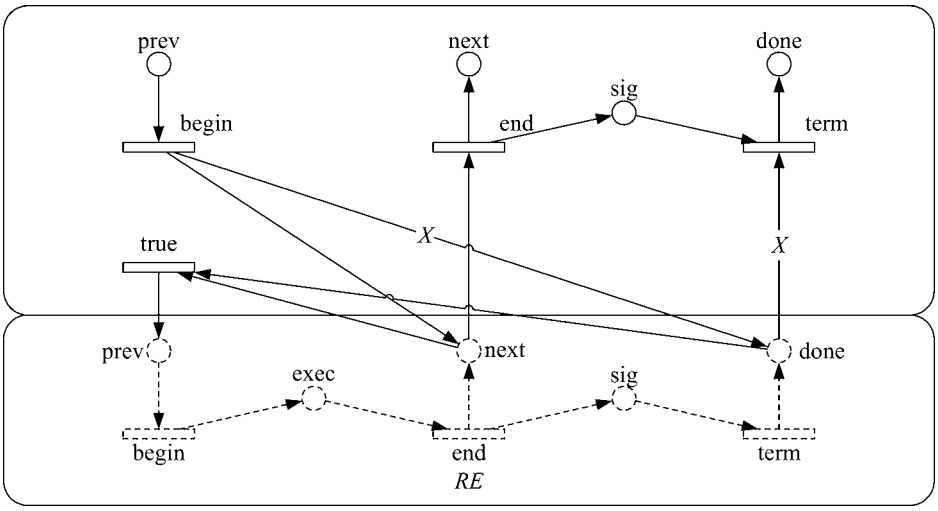

Figure 20. Semantics of while_do.

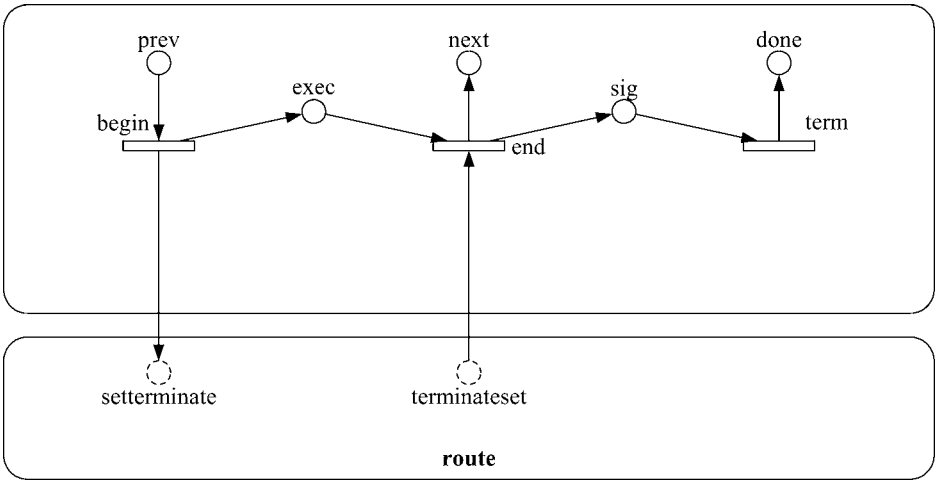

Figure 21. Semantics of terminate.

a token in place *next*. After firing *end*, the while_do waits until all descending routing elements corresponding to all iterations have been performed. If all have, transition *term* fires, which puts a token in place *done*. Note that a new iteration can only start, when at most $X - 1$ previously started iterations are still running. Also note that X can be set to a very large value when enacting the entire XRL route, but for verification purposes X should be as small as possible: The larger X is, the larger the state space of the entire resulting Petri net becomes.

Figure 21 shows the semantics of the terminate routing element. Recall that the instance level part is incorporated in the semantics of the route element, cf. figure 7.

Transition *begin* starts the terminate, and has the terminate set. Transition *end* fires if the terminate has been set, signaling that the terminate has been performed, and puts a token in place *next*. After firing *end*, transition *term* puts a token in place *done*.

At this point, all XRL routing elements can be transformed into Petri nets. By starting with the XRL route and recursively transforming each child routing element into its corresponding Petri-net semantics, one obtains a WF-net. As a result, we can check the important *soundness property* introduced in the previous section. This will be discussed in the next section.

The example shown in appendix B results in a WF-net containing 303 places and 275 transitions when applying the rules presented in this section. These numbers indicate that the dynamic behavior of the XRL route presented in section 1 is complex. However, this complexity is hidden from both the designer and the user.

4. Verification of XRL

This section discusses the soundness of an XRL route, that is, the soundness of the WF-net it is transformed into. Recall that for soundness three requirements should hold. The first requirement states that the final state o should be reachable, that is, proper completion is possible. The second requirement states that completion is always proper, that is, no tokens are left after completion. The third requirement states that for every transition there is a way to fire it from the initial state i.

4.1. Structural analysis

By using structural analysis, we show that completion of an XRL route is always proper.

Lemma 1 (Coverage of a single routing element). Let *RE* be a routing element and let *PN* be its Petri-net semantics as described in section 3. Then all places in *PN* can be covered by two place invariants P_{RE} and Q_{RE}, where P_{RE} contains the places *prev* and *next* with identical weights and Q_{RE} contains the places *prev* and *done* with identical weights.

Proof. By induction. Most of this proof is straightforward. For this reason, we will restrict ourselves to two routing elements that serve as examples: *task* and *parallel_part_sync_cancel*. For sake of simplicity, we assume that the invariants of the child routing elements are recalibrated such that the places *next* and *done* have weight 1. As a result, some weights might be fractions instead of natural numbers, but for the proof this is irrelevant. For the task (*tk* for short), which serves as example for the induction base, the invariants are as follows:

$$P_{\mathrm{tk}} = \mathrm{prev} + \mathrm{exec} + \mathrm{wait} + \mathrm{bypass} + \mathrm{next},$$
$$Q_{\mathrm{tk}} = \mathrm{prev} + \mathrm{exec} + \mathrm{wait} + \mathrm{bypass} + \mathrm{sig} + \mathrm{done}.$$

For the parallel_part_sync (*ppsc* for short), which serves as example for the induction step, the invariants are as follows:

$$P_{\text{ppsc}} = \text{prev} + \text{exec} + \text{next},$$

$$Q_{\text{ppsc}} = 2N \times (\text{prev} + \text{done}) + \text{free} + \left(\sum_{i=1}^{N} P_{\text{RE}_i} + Q_{\text{RE}_i} \right) + \text{exec} + \text{count} + (K+1) \times \text{sig}.$$

□

Lemma 2 (Coverage of top routing element). Let R be an XRL route, let *RE* be the top routing element in R, and let PN_R and PN_{RE} be the Petri-net semantics of R and *RE* as described in section 3. Then there exists a place invariant P_1 containing (i) the places *input* and *output* with identical weight, and (ii) all places in PN_{RE}.

Proof. By construction. $P_1 = \text{input} + \text{almost} + \text{output} + P_{\text{RE}} + Q_{\text{RE}}$. □

Lemma 3 (Coverage of a single event). Let R be an XRL route, let E be an event and let PN_R and PN_E be their Petri-net semantics as described in section 3. Then there exists a place invariant P_E containing (i) the place *input* and *output* with identical weight and (ii) all places of PN_E.

Proof. By construction. The places *event* and *nonevent* are easily covered. However, to cover *setevent* and *eventset* we need a variant of P_1. Let P_1^E be equal to P_1, except for the tasks where event E is set. For these tasks, the occurrence of place *wait* is replaced by the expression $E_{\text{setevent}} + E_{\text{setevent}}$. It is straightforward to check that P_1^E is a place invariant containing places *input* and *output* with identical weights. Then $P_E = P_1^E + \text{input} + \text{almost} + \text{output} + E_{\text{event}} + E_{\text{nonevent}}$. □

Lemma 4 (Coverage of all events). Let R be an XRL route and let S be the set of events in R. Then there exists a place invariant P_2 containing (i) the places *input* and *output* with identical weight, and (ii) all places corresponding to the events in S.

Proof. By construction. $P_2 = \sum_{E \in S} P_E$. □

Lemma 5 (Coverage of complete XRL route). Let R be an XRL route and let PN_R be its Petri-net semantics as described in section 3. Then there exists a place invariant P containing (i) the places *input* and *output* with identical weight, and (ii) all places in PN_R.

Proof. By construction. To cover the places *terminate* and *nonterminate* we introduce another variant of P_1: P_1^T. In P_1^T all occurrences of any terminate's *exec* place is replaced by the expression terminate + nonterminate. It is straightforward to check that P_1^T is a place invariant containing places *input* and *output* with identical weights. Then $P = P_1 + P_1^T + P_2$. □

Theorem 6. Completion of an XRL route is always proper.

Proof. By lemma 5, place invariant P contains (i) places *input* and *output* with identical weights, and (ii) all places of the Petri-net semantics of the XRL route. Because initially only *input* is marked with one token, no other place can be marked when *output* is marked. □

4.2. Behavioral analysis

By using behavioral analysis, we show that only certain XRL routing elements can invalidate the other two soundness requirements.

Theorem 7. If every wait_all and every wait_any in an XRL route contains a timeout, then completion of the XRL route is guaranteed.

Proof. By induction. It is straightforward to check that only a wait_all or wait_any that does not contain a timeout can get stuck. Note that we assume that tasks are completed eventually. □

Theorem 8. If every wait_all and every wait_any in an XRL route contains a timeout, and if the XRL route contains no terminate, then no transition in the Petri-net semantics of the route is dead.

Proof. It is straightforward to check that only a wait_all or wait_any that does not contain a timeout can get stuck. It is also straightforward to check that some transitions can only get by-passed by a terminate construct. □

Corollary 1. There are two possible causes for violation of soundness: (i) a wait_all or wait_any gets stuck, or (ii) a terminate occurs.

In case a terminate occurs, the short-circuited transition in the short-circuited net will be live: a terminate guarantees completion. In case a wait_all or wait_any gets stuck, the short-circuiting transition (that is, the transition linking the place *output* to *input* in the short-circuited net) will not be live.

4.3. Reduction techniques

With the semantics specified in terms of WF-nets, described in section 3, the theory and tools for WF-nets can be deployed in a straightforward manner. This allows us to use Woflan for verifying the correctness of an XRL route using criteria such as the soundness property. Unfortunately, XRL routes with a lot of parallelism tend to have a large state space, thus complicating verification from a computational point of view. Therefore, we propose a verification procedure that consists of two optimization steps. In the first step, the XSL transformation, which transforms the XRL route into a WF-net, reduces

the WF-net by using structural properties of XRL. In the second step, Woflan reduces the WF-net by applying the well-known liveness and boundedness preserving reduction rules for Petri nets [34].

The first step is the reduction by the XSL transformation based on structural properties of XRL. Figures 9–21 show a place named *done* to accommodate the situation where completion of a routing element does not automatically yield completion of its descendants. This situation can only occur if the routing element ascends some parallel_no_sync, parallel_part_sync, or parallel_part_sync_cancel routing element. In all other cases, there is no need to model things related to these done places. For instance, assuming the routing element in figure 20 has no done place allows us to remove the upper half of the Petri net (that is, RE_{done}, *sig*, *term*, and *done*). Similar simplifications are possible if no events are used. Moreover, we can apply the result presented in [10]: a routing element without any event, wait_all, wait_any, or terminate is sound and can therefore be replaced by the basic routing element shown in figure 6. Note that this is consistent with the results in section 4. When these reduction rules are applied, the XRL route shown in appendix B is transformed into a WF-net that contains only 108 places and 105 transitions. Compared to the original WF-net, the reduced WF-net is considerably smaller and less complex. Note that several routing elements can be abstracted from, and that the resulting WF-net need not contain any done places.

The second step is the reduction of the resulting WF-net by Woflan based on liveness and boundedness preserving reduction rules. Fragments of various routing elements are connected by transitions. This introduces a lot of transitions that are not relevant for the verification but introduce transient states. These and other parts of the WF-net can be reduced enormously without losing information relevant for the verification. In section 2, it was pointed out that soundness corresponds to liveness and boundedness [1]. This allows us to apply the well-known liveness and boundedness preserving reduction rules for Petri nets [34]. These rules are shown in figure 22. Note that not all rules are relevant for reducing a WF-net derived from an XRL route: for instance the fifth rule will not be applied, because the only marked place in the WF net has no input arcs. After these reduction rules are applied, the reduced WF-net mentioned under step 1 contains only 21 places and 18 transitions and is shown in figure 23. Table 1 shows how the reductions affect the size (in number of places and transitions) of the example WF-net.

After making the appropriate changes, the soundness results from section 4 still hold upon applying these reduction rules. It is straightforward to check that the *wait* transition in a wait_all or wait_any will not be reduced. Nor will be the terminate, if present. Note that we do not apply the six WF-net-based reduction rules on the short-circuited net, but on the original WF-net. As a result, the short-circuited transition will still be present after the WF-net has been reduced. As mentioned in section 4, this transition can be very useful when diagnosing the net.

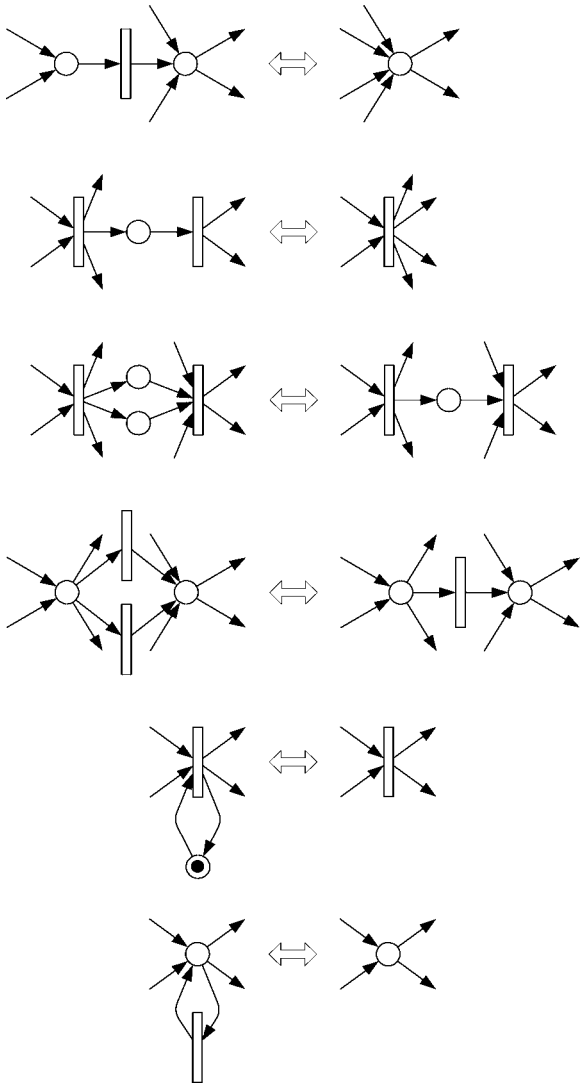

Figure 22. Liveness and boundedness preserving transformation rules.

4.4. Verification procedure

We propose a verification procedure that consists of three steps. In the first step, the XRL route is transformed into a WF-net, taking into account the three reduction rules based on the structure of XRL. In the second step, the resulting WF-net is reduced even further using the six reduction rules based on the structure of the WF-net. In the third step, we use Woflan to verify and diagnose the reduced WF-net.

Using standard Petri-net-based analysis tools, or dedicated tools such as Woflan, it is easy to show that figure 23 is sound. Therefore, the XRL route shown in section 2 is

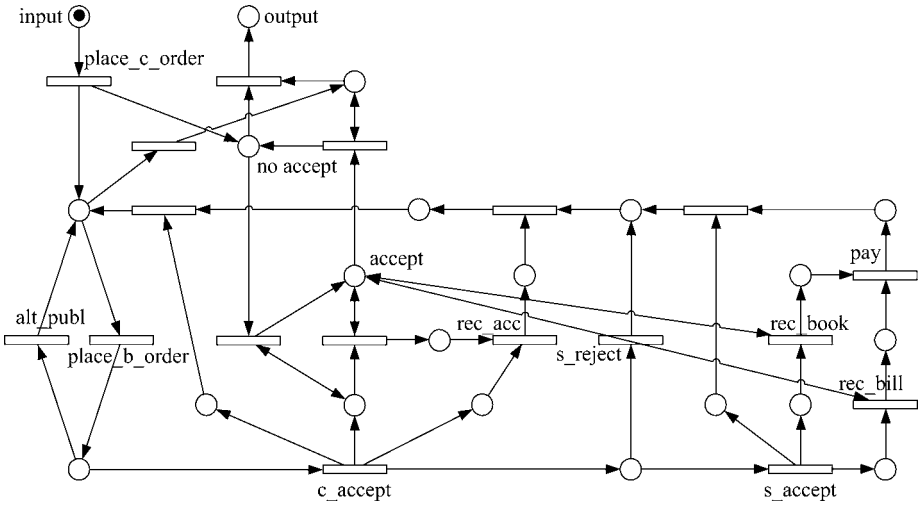

Figure 23. The WF-net corresponding to the example XRL route in section 2 after both reductions.

Table 1
The effects of both reductions on the size of the example WF-net.

WF-net	Number of places	Number of transitions
Original	303	275
After XRL-based reduction	108	105
After both XRL-based and Petri-net-based reduction	21	18

correct, that is, free of deadlocks, livelocks and other anomalies. Note that figure 23 is obtained after applying both types of reduction.

A final note on the terminate. If terminates are present, there might be tasks and waits that have to complete before any terminate can occur. In such a task or wait, the bypassing transitions are dead. However, these dead transitions do not correspond to an error in the XRL route, they only befoul the diagnostic information. To get rid of these dead transitions, we can add two additional routing elements to the route: A parallel_sync and a terminate. The parallel_sync will be the new top routing element and will have the old top routing element and the additional terminate as child routing elements. As a result, a terminate can occur immediately after the instance has started, so the set of tasks and waits that have to complete before any terminate can occur will be empty.

4.5. The tool

XRL/Woflan is based on our workflow verification tool Woflan [39,40]. Woflan (*http://www.tm.tue.nl/it/woflan*) is designed as a workflow-management-system-independent analysis tool. In principle, it can interface with many workflow management systems. At present, Woflan can also interface with the workflow products COSA (Thiel Logis-

Table 2
Abbreviations.

Routing element	Abbreviation	Routing element	Abbreviation
Any_sequence	as	Parallel_sync	ps
Choice	ce	Restricted_parallel_sync	rps
Condition	cn	Sequence	s
Event	e	True	t
Event_ref	er	Task	tk
False	f	Timeout	to
Fast_sequence	fs	Terminate	tt
Parallel_no_sync	pns	Wait_all	wl
Parallel_part sync	pps	Wait_any	wy
Parallel_part_sync_cancel	ppsc	While_do	wd

```
<xsl:template match="sequence" mode="id">
  <xsl:apply-templates select=".." mode="id"/>/s<xsl:number/>
</xsl:template>
```

Figure 24. Identifying a sequence.

tics AG/Software Ley), METEOR (LSDIS), and Staffware (Staffware), and the BPR tool Protos (Pallas Athena).

We have implemented the transformation from an XRL route into a WF-net using an XSLT specification. This XSLT specification produces a *Petri-Net Markup Language* (PNML) file. PNML is the standard Petri-net file format based on XML [24]. A second XSLT specification has been implemented that enables Woflan to read these PNML files.

The reduction rules based on the XRL structure have been implemented as an option in the first XSLT specification. The reduction rules based on the structure of the WF-net have been implemented as an option in Woflan.

PNML requires that every place, transition, and arc has a unique id. For this reason, the identification of these objects is an important implementation issue. For diagnostic purposes, it is vital that the names of the places and transitions are meaningful to the developer of the XRL route. We use the fact that a route is structured as a tree. For instance, we could identify a transition as transition *begin* of the second sequence of the first parallel_sync of the third true of the condition of the second while_do of the sequence of the route named *route*. To avoid long names as much as possible, we use abbreviations to identify the different routing elements. Table 2 lists these abbreviations. Figure 24 shows how a sequence is identified. The mode *id* is used to obtain the identification of any routing element. First, the sequence requests the id of its parent. Second, it adds "/s". Third, it adds its rank among the sibling sequences. The example transition mentioned is now identified by "route/sl/wd2/cnl/t3/psl/s2/begin". For most routing elements, this is fine, but we make an exception for tasks and events. These elements have explicit XRL names, and we want to use these names to identify them. Suppose the transition mentioned happens to be the transition *begin* of the task named *task*. Then

```
<xsl:template match="task" mode="id">
  <xsl:apply-templates select="/" mode="id"/>/<xsl:value-of
select="@name"/>
</xsl:template>
```

Figure 25. Identifying a task.

Figure 26. Woflan screendump.

it is identified by “route/task/begin”. Figure 25 shows how this is done. First, the task requests the name of the top element, that is, of the route. Second, it adds “/”. Third, it adds the value of its attribute *name*. Likewise, the transition *set* of the event named *event* is identified by ”route/event/set”.

Figure 26 shows a screendump of Woflan after it diagnosed the example XRL route with all reductions applied. Clearly, after the transformation and both reductions, the example XRL route corresponds to a sound WF-net. As a result, we may conclude that the XRL route itself meets the soundness requirements. Therefore, we can take it into production safely.

5. Extensibility of XRL

Extensibility is an important feature of our approach. Therefore, the architecture of figure 3 has been designed for extensibility. This means that an end-user can add his or her own new routing elements to the DTD and implement them using this toolset. This gives the end-user a powerful capability. We first describe the general approach and then illustrate it with three examples.

The basic approach is as follows:

(1) Add a new routing element to the DTD of XRL.

(2) Specify the semantics of the new routing element in XSLT.

(3) Verify the XSLT specification of the new routing element.

(4) Add the XSLT specification to the library used in the transformation step.

Thus, for any new routing element it suffices to add it to the DTD and specify its corresponding Petri-net semantics in XSLT. However, XSLT is a very verbose format, and editing XSLT specifications directly is cumbersome. For this reason, we use a set of macros while editing XSLT specifications.

Now, consider the architecture shown in figure 3. Note that, in the context of this architecture, the engine of XRL/Flower does not change after extending XRL in this manner. To demonstrate the extensibility of XRL we discuss the effort it takes to add two new routing elements.

The first new routing element we want to extend XRL with is a generalization of both the any_sequence as the parallel_sync and is called *restricted_parallel_sync*. Basically, the restricted_parallel_sync is a parallel_sync restricted by a maximum number of branches that can execute in parallel. If this number equals the number of branches, the restricted_parallel_sync resembles a parallel_sync, if the number equals 1, it resembles an any_sequence. The restricted_parallel_sync can be used when the parallel branches share a limited number of identical resources. For example, a database might be involved in all branches, and only a number of connections can be made to that database. First, we add the following lines to the DTD shown in figure 1:

```
<!ELEMENT restricted_parallel_sync ((%routing_element;)+)>
<!ATTLIST restricted_parallel_sync number NMTOKEN #REQUIRED>.
```

Note that the second line is added to specify the restrictive number of this element. The first line of the DTD is also changed to add this element to the list of routing elements. Second, the XSLT code of this routing element in terms of Petri nets is specified. Figure 27 shows the equivalent Petri net of the restricted_parallel_sync. The following macro snippet shows the core of the restricted_parallel_sync:

```
rememberNumber()
forEveryChildRE()
  ...
  switch()
```

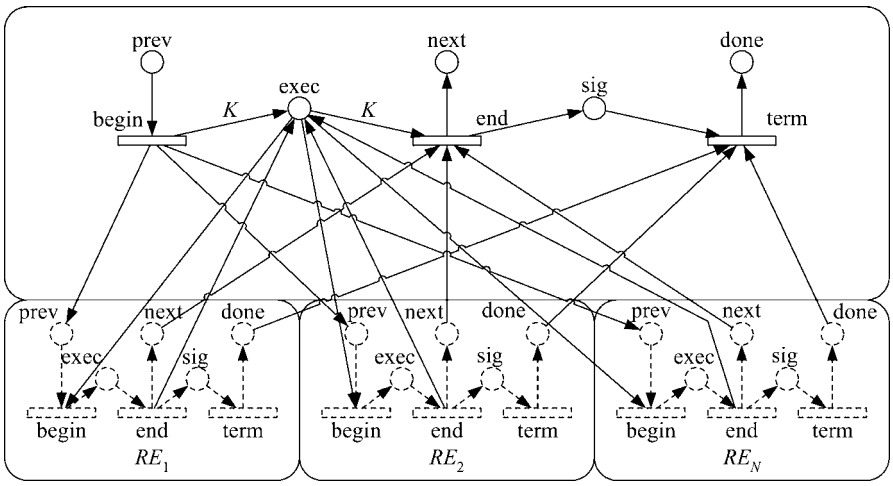

Figure 27. Semantics of restricted_parallel_sync.

```
    caseNumber()
      arc("..",arc2_<xsl:number value="position()"/>,"..",
      begin,"..",exec)
      arc("..",arc3_<xsl:number value="position()"/>,"..",
      exec,"..",end)
    endCase()
  endSwitch()
endFor()
```

If K is the value of the number attribute, then the first K child routing elements add K arcs from the parent's *begin* transition to the parent's *exec* place, and K arcs from that *exec* place to the parent's *end* transition. Third, we verify the restricted_parallel_sync. This can be done in numerous ways. For example, we could take a representative set of existing sound XRL instances and replace every parallel_sync by a restricted_ parallel_sync. After transforming these instances to PNML, we can verify them using Woflan. If all are sound, verification is complete. Fourth and last, we add the restricted_ parallel_sync to the XSLT transformation library.

The second new routing element we want to extend XRL with is called *fast_sequence*, and demonstrates that tasks need not be atomic in XRL. This routing element executes a sequence of tasks such that when two tasks are in a fast_sequence, the second task can start before the preceding task is completed, provided the preceding task has started. Moreover, the second task can only complete if the preceding task is completed. The fast_sequence is frequently seen for long-lived activities that can span days or months. For instance, let the first task in the fast_sequence represent an approval process, while the second represents the preparation for the actual construction. We may want to specify that the preparation for the construction can start any time after the approval process has been started and that it cannot be completed before the approval

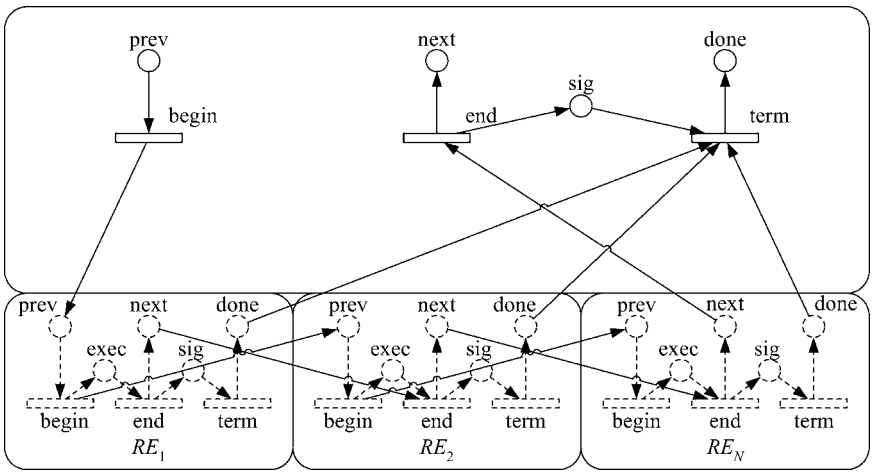

Figure 28. Semantics of fast_sequence.

process has completed, because only then all things are fixed. To add this routing element, we follow the steps as described above. First, we first add the following line to the DTD shown in figure 1:

```
<!ELEMENT fast_sequence ((%routing_element;)+)>.
```

The first line of the DTD is also changed to add this element to the list of routing elements. Second, the XSLT code for this routing element is specified in terms of Petri nets. Figure 28 shows the equivalent Petri net of the fast_sequence. The following macro snippet shows the core of the fast_sequence:

```
forEveryNonLastChildRE()
  arc("..",arc5_<xsl:number value="position()"/>,".",begin,
  nextRE(),prev)
  arc("..",arc6_<xsl:number value="position()"/>,".", next,
  nextRE(),end)
  ...
endFor()
```

For every child routing element except the last, arcs are added that connect its *begin* transition to the *prev* place of the next routing element, and its *next* place to the *end* transition of that next routing element. After verifying the fast_sequence, we add it to the XSLT transformation library.

As a third and final example, we extend XRL in such a way that an event can also be *reset*. Note that, although this extends the functionality of XRL, this is not really a new routing element. First, we extend the DTD with a *type* attribute for events:

```
<!ATTLIST event name ID #REQUIRED type(set|reset)"set">.
```

Possible values for the type attribute are *set* and *reset*, with *set* being the default value. This ensures backward compatibility: if unspecified, *set* is assumed. Second, we

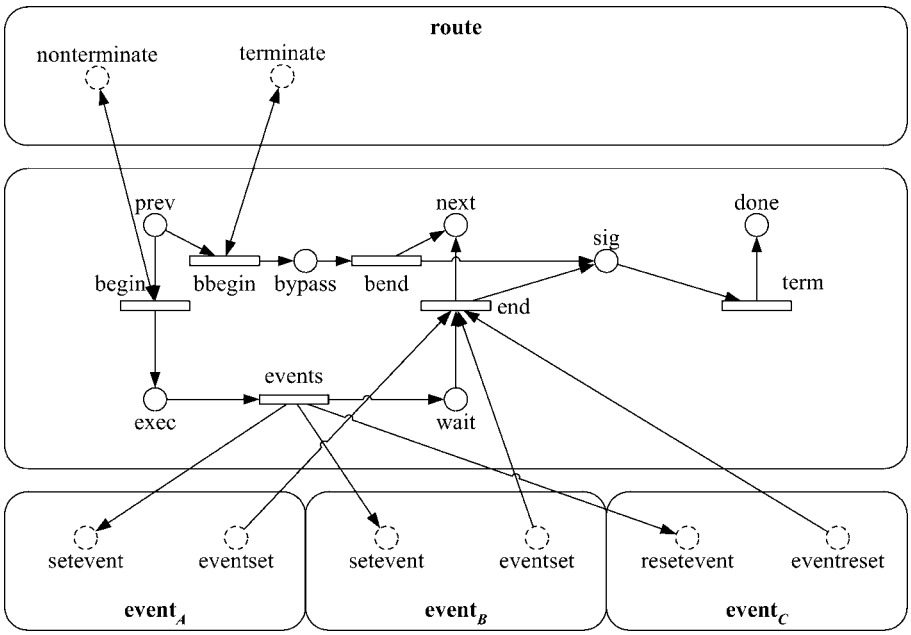

Figure 29. Semantics of task after extending event.

update the appropriate semantics. Figure 29 shows the updated semantics of the task. Depending on the type attribute, either the places *setevent* and *eventset* are connected to transition *events*, or the places *resetevent* and *eventreset*. Figure 30 shows the updated event on the global level. When the place *setevent* contains a token, the event is set, and the token is moved to *eventset*. When the place *resetevent* contains a token, the event is reset and the token is moved to *eventreset*. Note that it is possible to set and reset an event simultaneously, in which case it is impossible to tell whether the event is set or reset at the end. After verifying that the updates were specified correctly, the new specifications replace the old ones in the library. Note that, after adding this extension, the transformation form XRL to Petri nets does not necessarily result in a WF-net: if a certain event is only *reset* in a route, then the corresponding *event* place will be an additional source place. Because a route containing an event that can never be set is evidently erroneous, we do not regard this as a problem. It even helps verifying the XRL route: If some *event* place happens to be an additional source place, then that event is only reset in the route.

After adding the XSLT specifications for both new XRL routing elements to the XSLT transformation library, and replacing the old specifications by the new ones for the third extension, both the enactment service (XRL/Flower) and the verification tool (XRL/Woflan) support the new or updated routing elements.

Similarly, other new application-specific routing elements may be added on-the-fly in this manner. Therefore, it is possible to create new application-specific workflow patterns by writing XSLT routines that describe the semantics of the pattern. The patterns

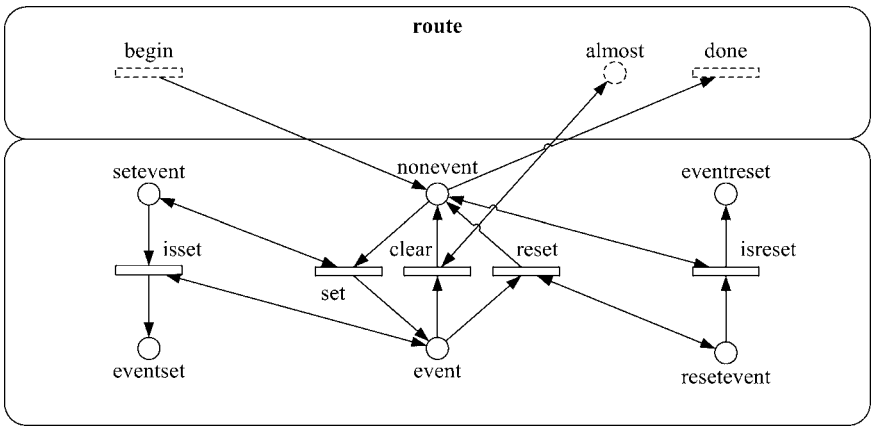

Figure 30. Semantics of event on global level after extending event.

can then be incorporated into the DTD of XRL after they have been tested and verified with Woflan. We are currently developing a complete methodology for workflow extensibility.

6. Related work

The following discussion of related research is organized according to various topics that are covered in the paper.

6.1. Petri nets

The semantics of XRL is expressed in terms of Petri nets. Petri nets have been proposed for modeling workflow process definitions long before the term "workflow management" was coined and workflow management systems became readily available. Consider for example the work on Information Control Nets, a variant of the classical Petri nets, in the late seventies [17,18]. The readers interested in the application of Petri nets to workflow management may refer to the two recent workshops on workflow management held in conjunction with the annual International Conference on Application and Theory of Petri Nets [8,33] and an elaborate paper on workflow modeling using Petri nets [1].

6.2. Workflow verification

Only a few papers in the literature focus on the verification of workflow process definitions. In [22] some verification issues have been examined and the complexity of selected correctness issues has been identified, but no concrete verification procedures have been suggested. In [1,12] concrete verification procedures based on Petri nets have been proposed. This paper builds upon the work presented in [1] where the concept of a sound WF-net was introduced (see section 2). The technique presented in [12] has

been developed for checking the consistency of transactional workflows including temporal constraints. However, the technique is restricted to acyclic workflows and only gives necessary conditions (that is, not sufficient conditions) for consistency. In [38] a reduction technique has been proposed. This reduction technique uses a correctness criterion which corresponds to soundness and the class of workflow processes considered are in essence acyclic free-choice Petri nets. Based on this reduction technique the analysis tool *FlowMake* [37] has been developed. Flowmake can interface with the IBM MQSeries Workflow product. Note that, although completely different techniques are used, Flowmake is very similar to the tool Woflan used in this paper [40].

This paper differs from the above approaches because the focus is on interorganizational workflows. Only a few papers explicitly focus on the problem of verifying the correctness of interorganizational workflows [2,25]. In [2] the interaction between domains is specified in terms of message sequence charts and the actual overall workflow is checked with respect to these message sequence charts. A similar, but more formal and complete, approach is presented by Kindler et al. in [25]. The authors give local criteria, using the concept of scenarios (similar to runs or basic message sequence charts), to guarantee the absence of certain anomalies at the global level.

6.3. Workflow standards

Clearly, the work presented in this paper is related to the standards developed by the Workflow Management Coalition (WfMC, [27]). XPDL (XML Process Definition Language [41], is the XML version of WfMC's language to exchange workflow process definitions (cf. Interface 1 of the reference architecture). Wf-XML [42] is an XML-based language to support interoperability between multiple enactment services (cf. Interface 4 of the reference architecture). The scope of XRL can be compared to the combination of XPDL and Wf-XML. However, there are some striking differences:

- XRL supports an abundance of routing constructs while XPDL supports only the very basic ones (AND/XOR-split/join and loops),
- XRL is extensible with new routing primitives while XPDL only allows for additional attributes,
- XRL is instance based, and
- XRL has formal semantics.

6.4. Cross-organizational workflows

Much work has been done on workflow transactions in the context of cross-organizational workflows, for example [20,21,36]. However, this work typically considers correctness issues at the task level rather than the process level. For example, the coordination model and the service model presented in [20] are not explicitly addressing control flow problems resulting from causal relations (or the absence of such relations). The work conducted in projects such as CrossFlow [21], WISE [28], OSM [31], and COSMOS [32]

is highly relevant for the enactment of interorganizational workflows. However, these projects do not consider the correctness issues tackled in this paper. Consider for example the Common Open Service Market (COSM) infrastructure proposed in [31,32]. This infrastructure proposes mobile agents. The control-flow within each agent is managed by a Petri-net-based workflow engine. Unfortunately, this work does not address correctness issues at the process level.

6.5. Electronic commerce

Recent development in Internet technology, and the emergence of the "electronic market makers", such as ChemConnect, Ariba, CommerceOne, Clarus, staples.com, Granger.com, VerticalNet, and mySAP.com have resulted in many XML-based standards for electronic commerce. The XML Common Business Library (xCBL) by CommerceOne, the Partner Interface Process (PIP) blueprints by RosettaNet, the Universal Description, Discovery and Integration (UDDI), the Electronic Business XML (ebXML) initiative by UN/CEFACT and OASIS, the Open Buying on the Internet (OBI) specification, the Open Application Group Integration Specification (OAGIS), and the BizTalk Framework are just some examples of the emerging standards based on XML. *These standards primarily focus on the exchange of data and not on the control flow among organizations.* Most of the standards provide standard Document Type Definitions (DTDs) or XML schemas for specific application domains (such as procurement). Initiatives that also address the control flow are RosettaNet and ebXML:

RosettaNet. The Partner Interface Process (PIP) blueprints by RosettaNet do specify interactions using UML activity diagrams for the Business Operational View (BOV) and UML sequence diagrams for the Functional Service View (FSV) in addition to DTDs for data exchange. However, the PIP blueprints are not executable and need to be predefined. Moreover, like most of the standards, RosettaNet is primarily focusing on electronic markets with long-lasting pre-specified relationships between parties with one party (such as the market maker) imposing rigid business rules.

ebXML. Electronic Business XML (ebXML) is an interesting new framework for the conduct of business between different enterprises through the exchange of XML based documents. It consists of a set of specifications that together enable a modular, yet complete electronic business framework. Among other things, the ebXML architecture provides a way to define business processes and their associated messages and content [19].

The ebXML initiative is designed for electronic interoperability, allowing businesses to find each other, agree to become trading partners and conduct business. It is a joint initiative of the United Nations (UN/CEFACT) and OASIS, developed with global participation for global usage. Another important feature of ebXML is the systematic representation of company capabilities to conduct e-business in the form of a Collaboration Protocol Profile (CPP). CPPs give companies a common XML format to describe the industries, business processes, messages, and data-exchange technologies that they

support in a structured way. With CPPs companies can agree on the business processes, messages and technologies used to exchange business messages for a specific trading need. These are expressed in a Collaborative Protocol Agreement (CPA), which is itself an ebXML document. Thus, the CPA provides the technical features of the agreement in automated form. The ebXML messages use the SOAP (Simple Object Access Protocol) specification. SOAP is an XML application that defines a message format with headers to indicate sender, receiver, and routing and security details.

The ebXML proposal looks promising; however, it lacks many of the richer routing constructs that are present in XRL. Moreover, routing is modeled somewhat indirectly by means of a Document envelope sent by one role and received by another. Nevertheless, it appears that ebXML can solve the first-trade problem mentioned in the introduction.

6.6. Interorganizational workflows based on Petri nets

Finally, we would like to refer to two existing approaches toward interorganizational workflows based on Petri nets. The first approach uses Documentary Petri Nets (DPN's), that is, a variant of high-level Petri nets with designated places for message exchange, to model and enact trade procedures [15,29,30]. The Interprocs system is based on these nets. The main difference between the Interprocs language and XRL is that XRL is instance based and supports less structured and more dynamic processes. Another approach combining Petri nets and interorganizational workflows is the P2P approach described in [11]. This approach uses inheritance to align local workflows. In [11] this approach is used to design an interorganizational workflow for a fictitious electronic bookstore similar to amazon.com or bn.com. A predecessor of the P2P approach has also been applied to an interorganizational workflow in the Telecom industry [5]. An interesting topic for future research is to see how the inheritance concepts used in [5,11] translate to XRL. We would also like to develop a more formal methodology for verification of new workflow patterns like the ones we introduced in section 5.

7. Conclusion

XRL is an XML-based language for describing workflow enactments. Woflan is a tool for the verification of Petri-net workflows. In this paper, we show how these two technologies can be combined together to create a powerful toolset for designing, verifying and implementing extensible workflows.

We have presented a novel way to verify the correctness of XRL routes. XRL routes are automatically transformed into WF-nets using XSLT. As a result, Woflan can be used to verify the correctness of the XRL route. The analysis procedure is optimized by exploiting dynamic properties of XRL routing elements and by using standard reduction rules at the Petri-net level [34]. We consider these verification capabilities essential for inter-organizational workflows. As was argued in the introduction, contemporary workflows need to be changed on the fly and sent across organizational boundaries. Unfortunately, the features also enable subtle, but highly disruptive, cross-organizational

errors. On-the-fly changes and one-of-a-kind processes are destined to result in errors. Moreover, errors of a cross-organizational nature are difficult to repair. Therefore, a language such as XRL (that is, a language with formal semantics) and verification tools such as XRL/Woflan are highly relevant for today's dynamic and networked economy.

Appendix A. The extended DTD of XRL

This appendix shows the DTD of XRL *after* all extensions have been added. For sake of clarity, the differences with the original DTD (see figure 1) are emphasized.

```
<!ENTITY % routing_element
"task|sequence|any_sequence|choice|condition|fast_sequence|
parallel_sync|parallel_no_sync|parallel_part_sync|
parallel_part_sync_cancel|restricted_parallel_sync|wait_all|
wait_any|while_do|terminate">
<!ELEMENT route ((%routing_element;), event*)>
<!ATTLIST route
  name ID #REQUIRED
  created_by CDATA #IMPLIED
  date CDATA #IMPLIED>
<!ELEMENT task (event*)>
<!ATTLIST task
  name ID #REQUIRED
  address CDATA #REQUIRED
  role CDATA #IMPLIED
  doc_read NMTOKENS #IMPLIED
  doc_update NMTOKENS #IMPLIED
  doc_create NMTOKENS #IMPLIED
  result CDATA #IMPLIED
  status (ready|running|enabled|disabled|aborted|null)  #IMPLIED
  start_time NMTOKENS #IMPLIED
  end_time NMTOKENS #IMPLIED
  notify CDATA #IMPLIED>
<!ELEMENT event EMPTY>
<!ATTLIST event
  name ID #REQUIRED
  type (set/reset) "set">
<!ELEMENT sequence ((%routing_element;|state)+)>
<!ELEMENT any_sequence ((%routing_element;)+)>
<!ELEMENT choice ((%routing_element;)+)>
<!ELEMENT condition ((true|false)*)>
<!ATTLIST condition
  condition CDATA #REQUIRED>
<!ELEMENT true (%routing_element;)>
<!ELEMENT false (%routing_element;)>
<!ELEMENT fast_sequence ((%routing_element;)+)>
<!ELEMENT parallel_sync ((%routing_element;)+)>
<!ELEMENT parallel_no_sync ((%routing_element;)+)>
<!ELEMENT parallel_part_sync ((%routing_element;)+)>
```

```
<!ATTLIST parallel_part_sync
  number NMTOKEN #REQUIRED>
<!ELEMENT parallel_part_sync_cancel ((%routing_element;)+)>
<!ATTLIST parallel_part_sync_cancel
  number NMTOKEN #REQUIRED>
<!ELEMENT restricted_parallel_sync ((%routing_element;)+)>
<!ATTLIST restricted_parallel_sync
  number NMTOKEN #REQUIRED>
<!ELEMENT wait_all ((event_ref|timeout)+)>
<!ELEMENT wait_any ((event_ref|timeout)+)>
<!ELEMENT event_ref EMPTY>
<!ATTLIST event_ref
  name IDREF #REQUIRED>
<!ELEMENT timeout ((%routing_element;)?)>
<!ATTLIST timeout
  time CDATA #REQUIRED
  type (relative|s_relative|absolute) "absolute">
<!ELEMENT while_do (%routing_element;)>
<!ATTLIST while_do
  condition CDATA #REQUIRED>
<!ELEMENT terminate EMPTY>
<!ELEMENT state EMPTY>
```

Appendix B. The e-bookstore example

This appendix shows the XRL route for processing a customer order by the e-bookstore. Note that most attributes have been omitted for brevity.

```
<!DOCTYPE route SYSTEM "xrl.dtd">
<route name="e-bookstore" created_by="H.M.W. Verbeek"
date="June 11, 2001">
  <sequence>
    <task name="place_c_order" address="customer"/>
    <task name="handle_c_order" address="bookstore"/>
    <while_do condition="No publisher found yet">
      <sequence>
        <task name="place_b_order" address="bookstore"/>
        <task name="eval_b_order" address="publisher"/>
        <condition condition="No publisher found yet">
          <true>
            <sequence>
              <task name="decide" address="publisher"/>
              <condition condition="Try alternative publisher">
                <true>
                  <task name="alt_publ" address="publisher"/>
                </true>
                <false>
                  <sequence>
                    <task name="b_reject" address="publisher"/>
```

```
            <task name="c_reject" address="bookstore"/>
            <task name="rec_decl" address="customer"/>
          </sequence>
        </false>
      </condition>
    </sequence>
  </true>
  <false>
    <sequence>
      <task name="b_accept" address="publisher"/>
      <task name="c_accept" address="bookstore"/>
      <parallel_sync>
        <task name="rec_acc" address="customer">
          <event name="accept"/>
        </task>
        <sequence>
          <while_do condition="No shipper found yet">
            <sequence>
              <task name="s_request" address="bookstore"/>
              <task name="eval_s_req" address="shipper"/>
            </sequence>
          </while_do>
          <condition condition="Shipper found">
            <true>
              <sequence>
                <task name="s_accept" address="shipper"/>
                <task name="inform_publ" address="bookstore"/>
                <task name="prepare_b" address="publisher"/>
                <task name="send_book" address="publisher"/>
                <task name="prepare_s" address="shipper"/>
                <task name="ship" address="shipper"/>
                <parallel_sync>
                  <sequence>
                    <task name="notify" address="shipper"/>
                    <task name="send_bill" address="bookstore"/>
                    <wait_all>
                      <event_ref name="accept"/>
                    </wait_all>
                    <task name="rec_bill" address="customer"/>
                  </sequence>
                  <sequence>
                    <wait_all>
                      <event_ref name="accept"/>
                    </wait_all>
                    <task name="rec_book" address="customer"/>
                  </sequence>
                </parallel_sync>
                <task name="pay" address="customer"/>
                <task name="handle_payment" address="bookstore"/>
              </sequence>
```

```
                </true>
                <false>
                  <task name="s_reject" address="shipper"/>
                </false>
              </condition>
            </sequence>
          </parallel_sync>
        </sequence>
      </false>
    </condition>
  </sequence>
 </while_do>
</sequence>
</route>
```

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) (1998) 21–66.

[2] W.M.P. van der Aalst, Interorganizational workflows: An approach based on message sequence charts and Petri nets, Systems Analysis – Modelling – Simulation 34(3) (1999) 335–367.

[3] W.M.P. van der Aalst, Loosely coupled interorganizational workflows: Modeling and analyzing workflows crossing organizational boundaries, Information and Management 37(2) (March 2000) 67–75.

[4] W.M.P. van der Aalst, Process-oriented architectures for electronic commerce and interorganizational workflow, Information Systems 24(8) (2000) 639–671.

[5] W.M.P. van der Aalst and K. Anyanwu, Inheritance of interorganizational workflows to enable business-to-business e-commerce, in: *Proceedings of the 2nd International Conference on Telecommunications and Electronic Commerce (ICTEC'99)*, eds. A. Dognac, E. van Heck, T. Saarinnen et al., Nashville, Tennessee (October 1999) pp. 141–157.

[6] W.M.P. van der Aalst and T. Basten, Inheritance of workflows: An approach to tackling problems related to change, Theoretical Computer Science 270(1–2) (2002) 125–200.

[7] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski and A.P. Barros, Advanced workflow patterns, in: *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, eds. O. Etzion and P. Scheuermann, Lecture Notes in Computer Science, Vol. 1901 (Springer, Berlin, 2000) pp. 18–29.

[8] W.M.P. van der Aalst, G. De Michelis and C.A. Ellis, eds., *Proceedings of Workflow Management: Net-Based Concepts, Models, Techniques and Tools (WFM'98)*, Lisbon, Portugal, June 1998 (UNINOVA, Lisbon, 1998).

[9] W.M.P. van der Aalst and A. Kumar, XML based schema definition for support of interorganizational workflow, Information Systems Research 14(1) (2003) 23–46.

[10] W.M.P. van der Aalst, H.M.W. Verbeek and A. Kumar, Verification of XRL: An XML-based workflow language, in: *Proceedings of the 6th International Conference on CSCW in Design (CSCWD 2001)*, eds. W. Shen, Z. Lin, J.-P. Barthès and M. Kamel, London, Ontario, Canada (July 2001) pp. 427–432.

[11] W.M.P. van der Aalst and M. Weske, The P2P approach to interorganizational workflows, in: *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, eds. K.R. Dittrich, A. Geppert and M.C. Norrie, Lecture Notes in Computer Science, Vol. 2068 (Springer, Berlin, 2001) pp. 140–156.

[12] N.R. Adam, V. Atluri and W. Huang, Modeling and analysis of workflows using Petri nets, Journal of Intelligent Information Systems 10(2) (1998) 131–158.

[13] Amazon.com, Inc. Amazon.com, http://www.amazon.com (1999).
[14] Barnes and Noble, bn.com. http://www.bn.com (1999).
[15] R.W.H. Bons, R.M. Lee and R.W. Wagenaar, Designing trustworthy interorganizational trade procedures for open electronic commerce, International Journal of Electronic Commerce 2(3) (1998) 61–83.
[16] T. Bray, J. Paoli, C.M. Sperberg-McQueen and E. Maler, *eXtensible Markup Language (XML) 1.0*, 2nd edn., http://www.w3.org/TR/REC-xml (2000).
[17] C.A. Ellis, Information control nets: A mathematical model of office information flow, in: *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, Boulder, Colorado (ACM Press, 1979) pp. 225–240.
[18] C.A. Ellis and G.J. Nutt, Modelling and enactment of workflow systems, in: *Application and Theory of Petri Nets 1993*, ed. M. Ajmone Marsan, Lecture Notes in Computer Science, Vol. 691 (Springer, Berlin, 1993) pp. 1–16.
[19] Enabling Electronic Business with ebXML, White Paper, http://www.ebxml.org/white_papers/whitepaper.htm.
[20] D. Georgakopoulos, H. Schuster, A. Cichocki and D. Baker, Managing process and service fusion in virtual enterprises, Information Systems 24(6) (1999) 429–456.
[21] P. Grefen, K. Aberer, Y. Hoffner and H. Ludwig, CrossFlow: Cross-organizational workflow management in dynamic virtual enterprises, International Journal of Computer Systems, Science, and Engineering 15(5) (2001) 277–290.
[22] A.H.M. ter Hofstede, M.E. Orlowska and J. Rajapakse, Verification problems in conceptual workflow specifications, Data and Knowledge Engineering 24(3) (1998) 239–256.
[23] S. Jablonski and C. Bussler, *Workflow Management: Modeling Concepts, Architecture, and Implementation* (International Thomson Computer Press, London, UK, 1996).
[24] M. Jungel, E. Kindler and M. Weber, The Petri Net Markup Language, in: *Proceedings of AWPN 2000 – 7th Workshop Algorithmen und Werkzeuge für Petrinetze*, ed. S. Philippi, Research Report 7/2000 (Institute for Computer Science, University of Koblenz, Germany, 2000) pp. 47–52.
[25] E. Kindler, A. Martens and W. Reisig, Inter-operability of workflow applications: Local criteria for global soundness, in: *Business Process Management: Models, Techniques, and Empirical Studies*, eds. W.M.P. van der Aalst, J. Desel and A. Oberweis, Lecture Notes in Computer Science, Vol. 1806 (Springer, Berlin, 2000) pp. 235–253.
[26] A. Kumar and J.L. Zhao, Workflow support for electronic commerce applications, Decision Support Systems 32(3) (2000) 265–278.
[27] P. Lawrence, ed., *Workflow Handbook 1997, Workflow Management Coalition* (Wiley, New York, 1997).
[28] A. Lazcano, G. Alonso, H. Schuldt and C. Schuler, The WISE approach to electronic commerce, International Journal of Computer Systems, Science, and Engineering 15(5) (2001) 345–357.
[29] R.M. Lee, Distributed electronic trade scenarios: Representation, design, prototyping, International Journal of Electronic Commerce 3(2) (1999) 105–120.
[30] R.M. Lee and R.W.H. Bons, Soft-coded trade procedures for open-edi, International Journal of Electronic Commerce 1(1) (1996) 27–49.
[31] M. Merz, B. Liberman and W. Lamersdorf, Using mobile agents to support interorganizational workflow-management, International Journal on Applied Artificial Intelligence 11(6) (1997) 551–572.
[32] M. Merz, B. Liberman and W. Lamersdorf, Crossing organisational boundaries with mobile agents in electronic service markets, Integrated Computer-Aided Engineering 6(2) (1999) 91–104.
[33] G. De Michelis, C. Ellis and G. Memmi, eds., *Proceedings of the 2nd Workshop on Computer-Supported Cooperative Work, Petri Nets and Related Formalisms*, Zaragoza, Spain (June 1994).
[34] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77(4) (1989) 541–580.

[35] W. Reisig and G. Rozenberg, eds., *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science, Vol. 1491 (Springer, Berlin, 1998).
[36] A. Reuter and F. Schwenkreis, Contracts – a low-level mechanism for building general-purpose workflow management-systems, Data Engineering Bulletin 18(1) (1995) 4–10.
[37] W. Sadiq and M.E. Orlowska, FlowMake Product Information, Distributed Systems Technology Centre, Queensland, Australia. http://www.dstc.edu.au/Research/Projects/FlowMake/productinfo/index.html.
[38] W. Sadiq and M.E. Orlowska, Analyzing process models using graph reduction techniques, Information Systems 25(2) (2000) 117–134.
[39] H.M.W. Verbeek and W.M.P. van der Aalst, Woflan 2.0: A Petri-net-based workflow diagnosis tool, in: *Application and Theory of Petri Nets 2000*, eds. M. Nielsen and D. Simpson, Lecture Notes in Computer Science, Vol. 1825 (Springer, Berlin, 2000) pp. 475–484.
[40] H.M.W. Verbeek, T. Basten and W.M.P. van der Aalst, Diagnosing workflow processes using Woflan, The Computer Journal (British Computer Society) 44(4) (2001) 246–279.
[41] Workflow Management Coalition Workflow Standard – Workflow Process Definition Interface – XML Process Definition Language, WFMC-TC-1025, Draft 0.03a (22 May 2001).
[42] Workflow Management Coalition Workflow Standard – Interoperability Wf-XML Binding, WFMC-TC-1023, Version 1.1 (14 November 2001).