XRL/Woflan: Verification and Extensibility of an XML/Petri-net based language for inter-organizational workflows

W.M.P. van der Aalst[1], H.M.W. Verbeek[1], and A. Kumar[2]
[1]Faculty of Technology and Management,
Eindhoven University of Technology,
PO Box 513, NL-5600 MB, Eindhoven, The Netherlands
{w.m.p.v.d.aalst,h.m.w.verbeek}@tm.tue.nl
[2]College of Business, CB 419
University of Colorado
Boulder, Co 80309, USA
akhil@acm.org

Abstract

Internet-based technology, E-commerce, and the rise of networked virtual enterprises have fueled the need for inter-organizational workflows. Although XML allows trading partners to exchange information, it cannot be used to coordinate activities in different organizational entities. Therefore, we developed a workflow language named XRL (eXchangeable Routing Language) for supporting cross-organizational processes. XRL uses XML for the representation of process definitions and Petri nets for its semantics. Since XRL is instance-based, 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, these features make cross-organizational workflows susceptible to errors. In this paper, we present XRL/Woflan, a software tool using state-of-the-art Petri-net analysis techniques for verifying XRL workflows. The tool uses XSL (Extensible Style Language) Transformations (called XSLT) to translate XRL specifications to a specific class of Petri nets called workflow nets. 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. This approach also makes XRL extensible. Therefore, new, application-specific workflow patterns can be created and incorporated into XRL by expressing their semantics in XSLT.

1 Introduction

Today's corporations must often 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 [2, 9, 14]. Inter-organizational workflows offer companies the opportunity to re-shape business processes beyond the boundaries of their own organizations. However, the design and deployment of inter-organizational workflows must address the following issues. 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. Therefore, these conflicting constraints complicate the development of languages and tools for cross-organizational workflow support.

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 has resulted in many XML-based standards for electronic commerce such as xCBL, UDDI, ebXML, OBI, OAGIS, BizTalk, etc. *These standards primarily focus on the exchange of data and not on the flow of control among organizations*. Most of the standards provide standard DTDs or XML schemas for specific application domains (e.g. procurement). One of the few initiatives which also addresses the control flow is RosettaNet. The Partner Interface Process (PIP) blueprints by RosettaNet do specify interactions. However, the PIP blueprints are not executable and need to be predefined. Moreover, like most of the standards, RosettaNet primarily focuses on electronic markets with long-lasting pre-specified relationships with one party (e.g., the market maker) imposing rigid business rules.

Looking at existing initiatives two things can be noted: (1) process support for cross-organizational workflow has been neglected since the lion's share of attention has gone to data and (2) only pre-specified standardized processes have been considered (e.g., market places, procurement, etc.). Based on these observations, we developed the *eXchangeable Routing Language* (XRL). The idea to develop a language like XRL was raised in [12] and the definition of the language was given in [4]. XRL uses the syntax of XML, but contains constructs which 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”, i.e., the situation where parties have no prior trading relationship [15]. To support such 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 [10, 13]. On the other hand, 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. In our research on workflow patterns [3], 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[1]. Based on the workflow patterns supported by these systems, and their relative use in practice, we carefully selected the most relevant constructs for XRL. Note that the expressive power of XRL far exceeds that of each of the workflow management systems mentioned above.

As shown in [4], the semantics of XRL can be expressed in terms of Petri nets [16, 17]. Based on these 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 translated 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 translation to the Petri-net engine needs to be added and the engine itself does not need to change.

Unfortunately, the Petri-net based semantics of XRL given in [4] results in Petri nets which do not fit into the class of *WorkFlow nets* (WF-nets). *WF-nets are a special subclass of Petri nets*, which possess an appealing correctness notion (the soundness property [1], strong theoretical

1. See *http://www.tm.tue.nl/it/research/patterns/* for a list of workflow patterns.

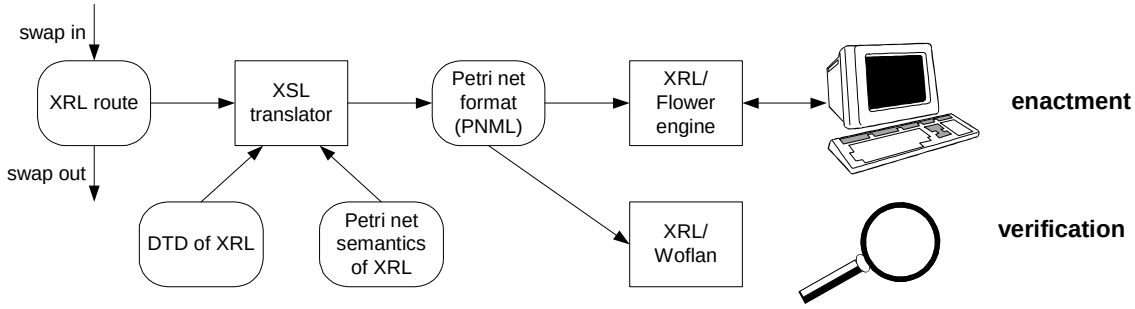

Figure 1: Toolset for enactment and verification of XRL workflows.

results (e.g., the link between soundness, liveness, and boundedness [1]), and powerful software (e.g., the tool Woflan [19]). The semantics given in [4] does not allow these theoretical results and tools to be used directly. This limitation was recognized in [5]. In this paper, we present a direct translation from XRL to WF-nets, i.e., the semantics of XRL is given in terms of WF-nets. The translation has been implemented using XSL and resulted in a tool XRL/Woflan.

XRL/Woflan builds on the workflow verification tool Woflan [19, 20]. 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 [19] and Flowmake [18] are two noteworthy exceptions. To complicate matters, more and more workflow management systems are being used to support inter-organizational business processes, e.g., in the context of Business-To-Business (B2B) E-commerce, where it is vital to guarantee that 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.

Figure 1 gives an overview of the toolset involving XRL/Flower and XRL/Woflan. XRL routes, representing workflow instances, can be swapped in (e.g., received from another organizations or created) or swapped out (e.g., shipped to another organization after modification). An XSL (Extensible Stylesheet Language) translator which is driven by the DTD (describing the syntax of each XRL construct) and the Petri-net semantics (i.e., a graphical description of the behavior of each construct) converts each XRL route onto a generic Petri-net based format. This format can be used by both the enactment service, i.e., the workflow engine of XRL/Flower, and by the verification tool XRL/Woflan.

The remainder of this paper is organized as follows. In Sections 2 and 3, we introduce XRL and give an example of how a workflow can be represented in XRL. Section 4 introduces WF-nets, a special sub-class of Petri nets, and gives their main properties. Then Section 5 shows how the formal semantics of XRL can be expressed in terms of WF-nets. Based on these semantics, in Section 6, we propose a verification procedure which exploits the structural properties of certain XRL constructs and Petri-net based reduction rules [16]. Then we present our verification tool XRL/Woflan. Finally, Section 7 concludes the paper.

2 XRL: An XML based routing language

The focus of this paper is on the verification and extensibility aspects of XRL [4]. Therefore, we limit ourselves to only a brief introduction to XRL. XRL essentially gives a syntax for describing

```
<!ENTITY % routing_element "task|sequence|any_sequence|choice|
condition|parallel_sync|parallel_no_sync|parallel_part_sync|
parallel_part_sync_cancel|wait_all|wait_any|while_do|terminate">
<!ELEMENT route (%routing_element;)>
<!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 NMTOKEN #IMPLIED
  end_time NMTOKEN #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 (event+)>
```

Figure 2: The DTD of XRL.

workflows in XML. The syntax of XML documents is completely specified by the *Document Type Definition* (DTD) [8] of the document. The DTD for XRL is shown in Figure 2. An *XRL route* describing a workflow instance is a consistent XML document, i.e., a well-formed and valid XML file with top element *route* (see Figure 2). XRL provides a rich set of constructs for modeling real workflows in XML as we discuss next.

A *routing element* is an important building block of XRL for constructing workflow enactments. It can be any one of the following: *task* (a step to be performed), *sequence* (a set of routing elements to be done in a specific order), *any_sequence* (a set of routing elements to be done in any

order), *choice* (any one routing element out of a set of routing elements), *condition* (test a condition and determine next routing element based on result of the test), *parallel_sync* (start multiple parallel routing elements and later join them), *parallel_no_sync* (start multiple parallel routing elements that do not have to join), *parallel_part_sync* (start multiple parallel routing elements, some of which must join), *parallel_part_sync_cancel* (start multiple parallel routing elements, some of which must join, while the remaining ones are withdrawn if possible), *wait_all* (insert a wait step to wait for the completion of a group of events), *wait_any* (insert a wait step to wait for the completion of any one of a group of events), *while_do* (enable repetition of a routing element while a condition is true), and *terminate* (end this workflow instance). The DTD also describes attributes associated with each element. The attributes describe various properties or aspects of an element. For example, there are several attributes associated with the task element. The address attribute gives a URL of the location where the task is to be performed. The doc_read, doc_update and doc_create attributes give the names of documents which may be read, updated and created, respectively during the performance of the task. The other attributes are self-explanatory.

It is important to note that the constructs of XRL are based on a thorough analysis of the workflow patterns supported by leading workflow management systems. In the next section, we show how the various constructs of XRL can be used to design a real workflow, which is consistent with the DTD.

3 Example: An electronic bookstore

In this section we illustrate XRL using an example inspired by electronic bookstores, such as Amazon [6] and Barnes and Noble [7]. A typical order flow is shown by the activity diagram in Figure 3. This figure gives the four parties or organizations involved (i.e., 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 are omitted from this diagram to prevent clutter.

The workflow represented by the activity diagram is described in XRL in Figure 4. The XRL rendition covers the typical order flow of Figure 3, 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 of the desired book to the first appropriate publisher (task *place_b_order*). We will use the term “bookstore order” for the transferred order. The bookstore order is next evaluated by the publisher (task *eval_b_order*). The publisher, in turn, informs the bookstore about the availability of the book. If the book was 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 *b_reject*). If the bookstore receives a negative answer, it searches for another shipper.

After a shipper is 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 (task *prepare_s*) and ships the book to the customer (task *ship*). The customer receives the book (task *rec_book*) and the shipper notifies the

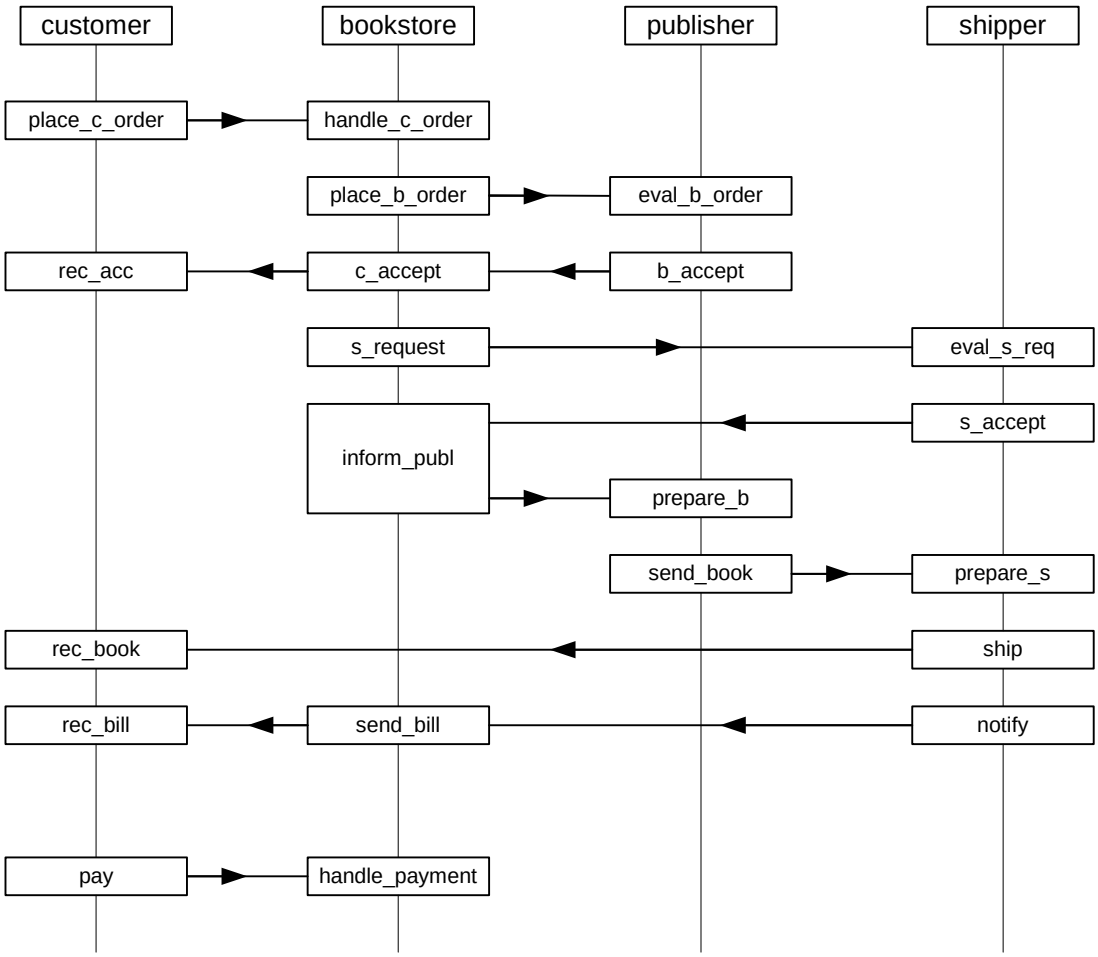

Figure 3: Typical order flow of the electronic bookstore.

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 Figure 4 just illustrates some of the XRL routing constructs. The description is far from complete, e.g., the detailed description of tasks and conditions have not been added. Please note that, since an XRL route specifies the life cycle of a particular workflow instance (i.e., work case), any instance can be modified without reference to some underlying workflow schema type.

Based on XRL, we have developed a prototype named *XRL/Flower* [4]. XRL/Flower can handle XRL files arriving through e-mail or ftp. An incoming XRL file, i.e., workflow instance, is parsed and translated into a Petri net. The Petri-net description drives the workflow engine, which calculates enabled tasks. The enabled tasks are offered to the proper workers through role-based worklists. Whenever a task is executed, the engine calculates newly enabled tasks. The engine or an authorized user can also decide to migrate a running instance to another workflow engine. For migration, an XRL file is created with entries for the current workflow state and shipped through e-mail or ftp.

```
<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>
```

Figure 4: The XRL route for processing a customer order
(many attributes were omitted for brevity).

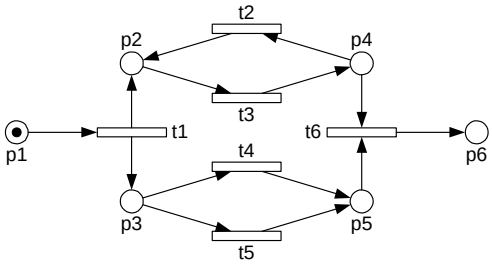

Figure 5: An example Petri net.

4 Workflow nets

As shown in [4], the semantics of XRL can easily be expressed in terms of Petri nets. XRL/Flower is a prototype that uses a Petri-net engine to interpret and execute XRL routes. However, it was observed in [5] that the translation given in [4] does not result in *WorkFlow nets* (WF-nets), a special class of Petri nets. Consequently, the strong theoretical results for WF-nets cannot be used. Moreover, it is difficult to deploy our workflow verification tool Woflan [19] which was developed independently from XRL. In this paper we present a new approach that allows us to integrate the XRL and Woflan technologies. In order to do so, the semantics presented in [4] has been modified and the resulting Petri nets are WF-nets which allows us to use the results presented in [1, 20]. Moreover, the robustness and extensibility of XRL is increased by the new semantics. Before we present this new mapping, we briefly introduce some of the concepts related to WF-nets. We give a brief introduction to Petri nets next and refer the reader to [16, 17] for more details.

A *Petri net* (or place-transition net) is represented graphically by rectangles and circles. The rectangles are called transitions and the circles representing the state are called places. The arrows between places and transitions are used to specify causal relations. Figure 5 shows a Petri net composed of six places and six transitions[1]. A place *p* is called an input place of a transition *t* if and only if there exists a directed arc from *p* to *t*. Place *p* is called an output place of transition *t* if and only if there exists a directed arc from *t* to *p*. At any time a place contains zero or more tokens, drawn as black dots. The state of the net, often referred to as marking, is the distribution of tokens over places. In Figure 5, only place *p1* contains a token. The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: They change the state (or marking) of the net according to the following firing rule:

- A transition is said to be enabled if and only if each input place contains at least one token.
- An enabled transition may fire. If a transition fires, then it consumes one token from each input place and produces one token for each output place.

In Figure 5, when transition *t1* fires, it will remove the token from place *p1* and put one token each in places *p2* and *p3*, which are the output places of *t1*. A token in place *p3* may cause either transition *t3* or *t4* to fire but not both. When there are tokens in places *p4* and *p5*, one of two transitions may occur: either *t6* may fire (consuming the tokens in *p4* and *p5*) and put a token in place *p6*, or *t2* may fire and place a token in place *p2*. In this way, it is possible to model complex coor-

1. In general, the number of places and number of transitions are not the same.

dination constraints using Petri nets. A Petri net like the one in this figure may be represented as follows:

$PN = (P, T, F)$
$P = \{p1, p2, p3, p4, p5, p6\}$
$T = \{t1, t2, t3, t4, t5, t6\}$
$F = \{(p1, t1), (t1, p2), (t1, p3), (p2, t3), (p3, t4), (p3, t5), (t2, p2), \ldots, (t6, p6)\}$

In this form, P is the set of places, T is the set of transitions and F is the flow relation or the set of connections between places and transitions. In the context of workflows, the transitions correspond to various tasks to be executed, such as place an order, process an order, ship order, etc. *A Petri net which models the control-flow aspect of a workflow is called a WF-net*. A *WF-net* specifies the dynamic behavior of a single case in isolation, and it may be defined formally as follows.

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

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

(ii) There is one sink place $o \in P$, i.e., one place without any successors.

(iii) 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, i.e., the WF-net specifies the life-cycle of a case. The third requirement in Definition 1 has been added to avoid 'dangling tasks and/or conditions', i.e., tasks and conditions that do not contribute to the processing of cases.

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

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

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

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

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

(ii) State o is the only state reachable from state i with at least one token in place o.

(iii) There are no dead transitions 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 properties such as liveness and boundedness. A Petri net is bounded if and only if the number of reachable states is finite. A Petri net is live if and only if, no matter what happens, every transition can be enabled again. A WF-net is sound if and only if the short-circuited net (i.e., 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.

5 Semantics of XRL in terms of WF-nets

The DTD shown in Figure 2 only describes the syntax of XRL and does not discuss the semantics pertaining to each construct. In this section, we show that each XRL construct has a corresponding representation in terms of Petri nets. Moreover, *replacing each construct by its Petri-net equivalent produces a workflow net (WF-net)* as discussed above. Our goal is to show that the Petri-net representations we give produce a WF-net which is sound. Although each construct of Section 2 has Petri-net representation, for brevity, here we restrict ourselves[1] to two of the most complex constructs: *parallel_part_sync_cancel* and *while_do*.

Before discussing the constructs, it should be noted that an XRL route has a tree-like structure. A *route* consists of routing elements such as *task*, *sequence*, *any_sequence*, *choice*, etc. Most of these routing elements contain other routing elements. Only *task* and *terminate* are atomic. It is also important to note, that, although a *route* is mapped onto a WF-net, it is not feasible to map individual routing elements onto WF-nets.

5.1 Petri-net representation of parallel_part_sync_cancel construct

Figure 6 shows the semantics of the *parallel_part_sync_cancel*. This routing element starts a number (*N*) of child routing elements (RE_1 to RE_N) and synchronizes after *K* child routing elements have completed. Note that this *K* corresponds to the obligatory *number* attribute. After synchronizing, the $N - K$ remaining routing elements may be cancelled. Note that this construct is similar to the *K-out-of-N* construct proposed in [10].

Place *prev* is the input place of any routing element. The parent routing element can put a token in this place, which corresponds to activating the element. Place *next* is the output place. A token put into this place indicates completion and can be removed by the parent (so the parent can continue). Note that, due to the possible presence of the constructs *parallel_no_sync*, *parallel_part_sync* and *parallel_part_sync_cancel*, this does not necessarily mean that the *descendant* routing elements have completed. There may still be ongoing work in one of the *N* child elements. Therefore, in addition to place *next*, the semantics of every routing element contains a place *done* indicating (to the parent) that all descendants have completed too. Place *free* ensures that the *parallel_part_sync_cancel* and its descendants have completed before it can start again, say, if it were nested somewhere within a *while_do* construct. Both transitions *begin* and *end* are part of the semantics of the top element *route*. This *route* is defined in such a way that fir-

1. See *http://www.tm.tue.nl/it/staff/wvdaalst/workflow/xrl* for the Petri-net mappings of all the other constructs.

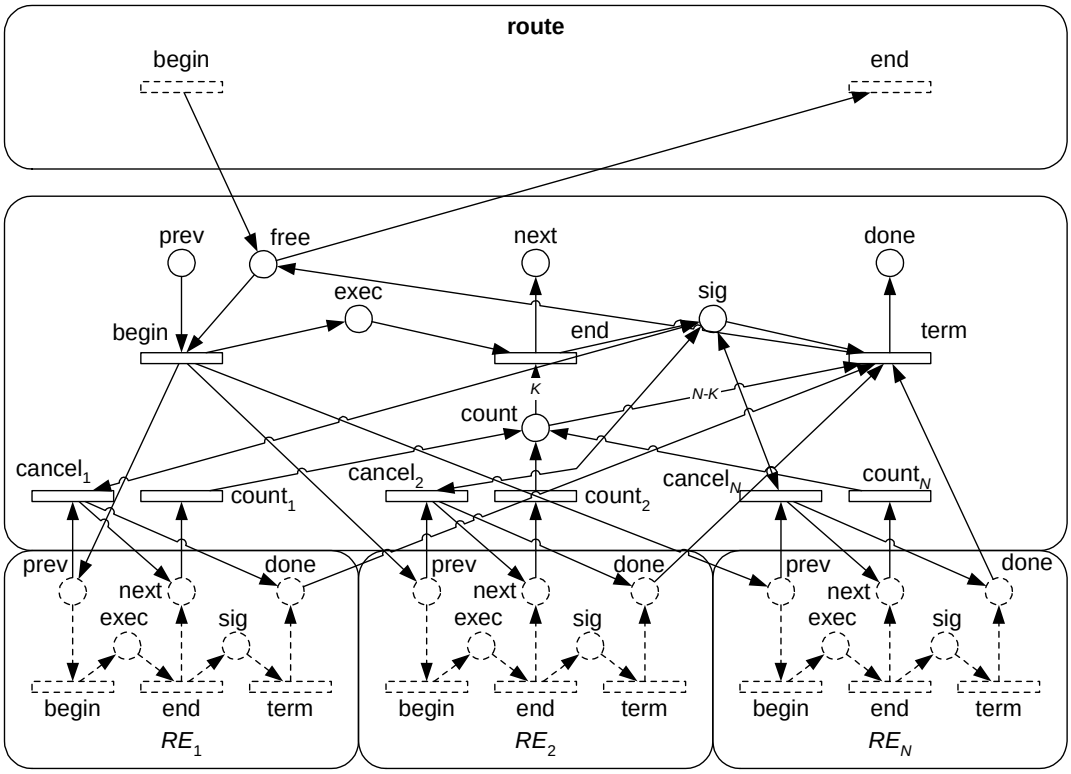

Figure 6: The Petri-net semantics of the routing element *parallel_part_sync_cancel*.

ing the *begin* transition starts the (only) child of the *route*, and *only* when this element and its descendants have completed, i.e., *only* when its *next* and *done* place are marked, the *end* transition fires and puts a token in *next*. When *K* children have completed, place *count* contains *K* tokens and transition *end* can fire, i.e., the *parallel_part_sync_cancel* can complete. The remaining $N - K$ children are either cancelled or ignored, after which the construct signals that all descendants have completed too by firing *term* and putting a token in *done*.

5.2 Petri-net representation of while_do construct

Figure 7 shows the semantics of the *while_do*. Initially, *X* tokens are put into place *RE/done* and one token is put into place *RE/next*. Note that these places are in fact the places *next* and *done* of the child *RE*, i.e., they signal completion of *RE* and its descendants. Every time there is a token in *RE/next*, a choice is made. If the condition evaluates to "true", *true* fires and another iteration takes place; if it evaluates to "false", *end* fires, i.e., the *while_do* completes. Because a *while_do* can complete before its descendants have completed, several iterations can be active at the same time. The number *X* is an upper bound for the number of iterations active at the same time: If *X* iterations are active, then *RE/done* is empty and no new iteration can start. When verifying the workflow, it is necessary to restrict the number of active iterations by some upper bound *X* because otherwise the state space could potentially be infinite.

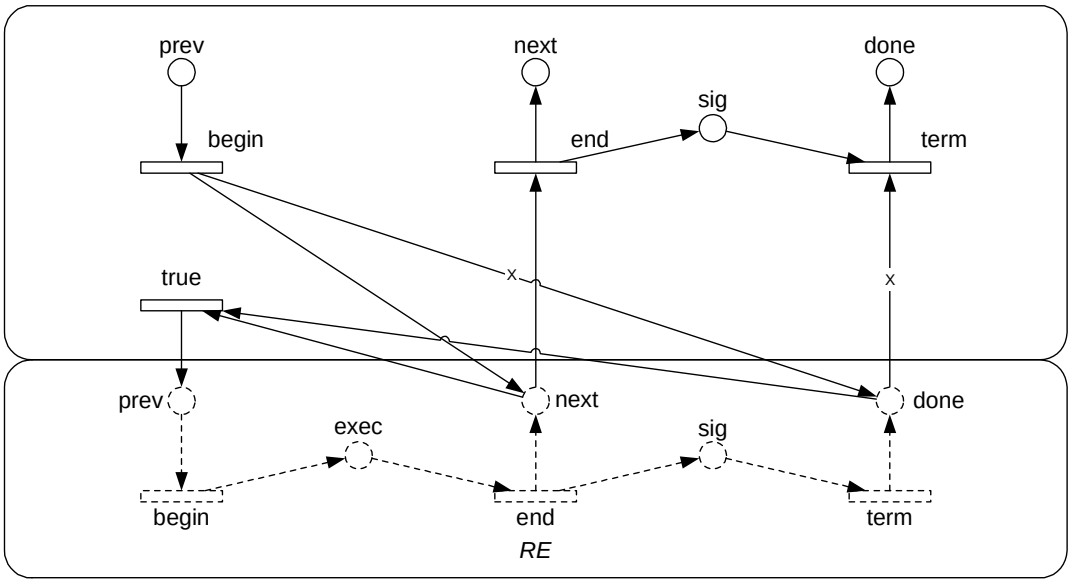

Figure 7: The Petri-net semantics of the routing element *while_do*.

5.3 Discussion

Figures 6 and 7 specify the semantics of the *parallel_part_sync_cancel* and *while_do* constructs. These two constructs were chosen because they are more complex and, therefore, more interesting for the reader than the others. The other constructs of XRL also have equivalent Petri-net representations that have been specified in a similar fashion. By starting with the XRL route and recursively replacing each child routing element by its corresponding Petri-net semantics, one obtains a WF-net. As a result, we can check the important *soundness property* discussed in the previous section.

As an example of this mapping process, the XRL route shown in Figure 4 is mapped to a WF-net containing 303 places and 275 transitions. These numbers indicate that the dynamic behavior of the XRL route presented in Section 3 is complex. However, this complexity is hidden from both the designer and the user.

6 Verification of XRL

With the semantics specified in terms of WF-nets, described in the previous section, 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 translator, which translates the XRL route to 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* [16].

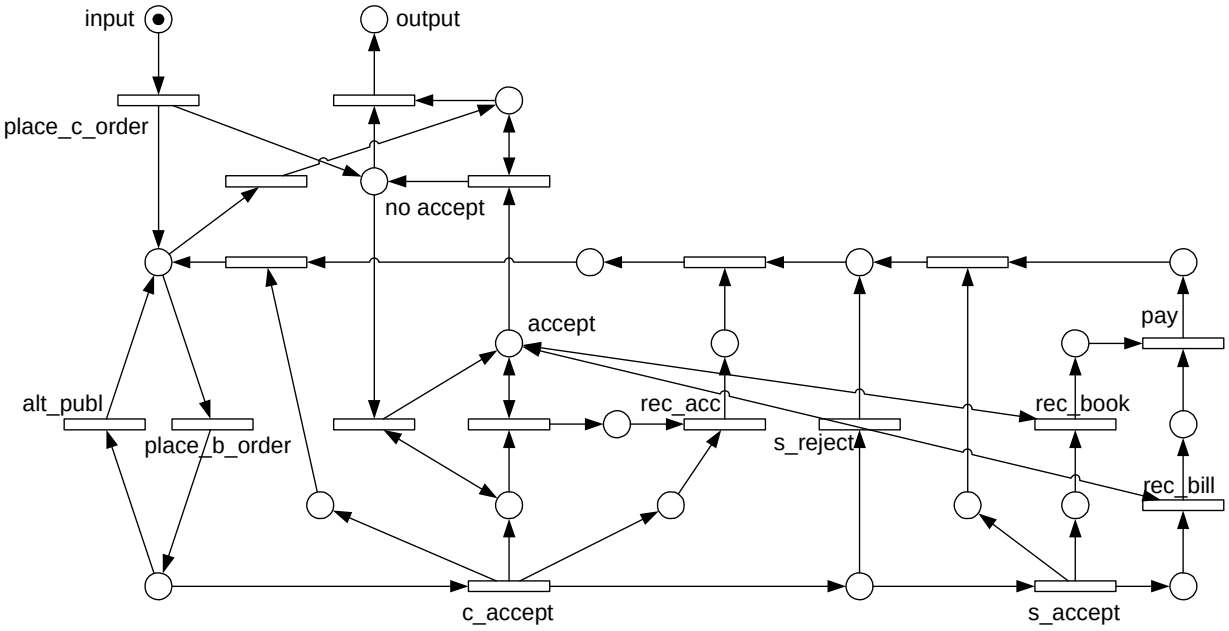

Figure 8: The WF-net of the bookstore example after a reduction based on liveness and boundedness preserving reduction rules.

6.1 Two-Step Reduction

Step 1: Reduction by the XSL translator based on structural properties of XRL. Figures 6 and 7 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 contains 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. Assuming *RE* in Figure 7 has no *done* place allows us to remove almost half of the Petri net (i.e., *sig*, *term*, and *done*). Similar simplifications are possible if no events are used. Moreover, we can apply the result presented in [5]: A routing element without any *event*, *wait_all*, *wait_any*, or *terminate* is sound and can therefore be considered to be atomic. Therefore, there is no need to model the internal structure of these routing elements. When these reduction rules are applied, the XRL route shown in Figure 4 is mapped to 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 WF-net need not contain any *done* places.

Step 2: Reduction 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 4, 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 [16]. 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 8.

6.2 XRL/Woflan Verification Tool

Using standard Petri-net based analysis tools, or dedicated tools such as Woflan, it is easy to show that Figure 8 is sound. Therefore, the XRL route shown in Figure 4 is correct, i.e., free of deadlocks, livelocks and other anomalies. Note that Figure 8 is obtained after applying both types of reduction.

XRL/Woflan is based on our workflow verification tool Woflan [19, 20]. Woflan[1] is designed as a workflow-management-system-independent analysis tool. In principle, it can interface with many workflow management systems. At present, Woflan can interface with the workflow products COSA (Thiel Logistic AG/Software Ley), METEOR (LSDIS), and Staffware (Staffware), and the BPR-tool Protos (Pallas Athena). Furthermore, Woflan can read *Petri Net Markup Language* (PNML) files. PNML is a Petri-net file format based on XML [11]. Therefore, it is natural to use XSL to automatically translate an XRL route into a PNML representation that can be diagnosed using Woflan. We have implemented this translation using XSL and the two types of reduction rules presented in this section. These two types of reduction rules allow us to verify large XRL routes containing hundreds of tasks.

7 Conclusion

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

We presented a novel way to verify the correctness of XRL routes by automatically translating them into WF-nets using XSL (Extensible Style Language) Transformations. 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 constructs and by using standard reduction rules at the Petri-net level [16]. 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, cross-organizational workflows are more susceptible to errors than intra-organizational workflows. Moreover, one-of-a-kind processes and on-the-fly changes further exacerbate the problem. Finally, errors of a cross-organizational nature are also very difficult to repair. Therefore, a language such as XRL (i.e., a language with formal semantics) and verification tools such as XRL/Woflan are highly relevant for today's dynamic and networked economy.

The approach presented in this paper also makes XRL extensible. Therefore, it is possible to create new application-specific workflow patterns by writing XSLT routines that describe the semantics of the pattern. The patterns can then be incorporated into the DTD of XRL after they have been tested and verified with Woflan. We are currently working on developing a complete methodology for extensibility.

References

[1] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21-66, 1998.

1. See *http://www.tm.tue.nl/it/woflan* for information on Woflan.

[2] W.M.P. van der Aalst. Process-oriented Architectures for Electronic Commerce and Interorganizational Workflow. *Information Systems*, 24(8):639-671, 2000.

[3] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced Workflow Patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18-29. Springer-Verlag, Berlin, 2000.

[4] W.M.P. van der Aalst and A. Kumar. XML Based Schema Definition for Support of Interorganizational Workflow. (Technical report, accepted by ISR), 2000.

[5] W.M.P. van der Aalst, H.M.W. Verbeek, and A. Kumar. Verification of XRL: An XML-based Workflow Language. In W. Shen, Z. Lin, J.-P. Barthès, and M. Kamel, editors, *Proceedings of the Sixth International Conference on CSCW in Design* (CSCWD 2001), pages 427-432, London, Ontario, Canada, July 2001.

[6] Amazon.com, Inc. Amazon.com. http://www.amazon.com, 1999.

[7] Barnes and Noble. bn.com. http://www.bn.com, 1999.

[8] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. eXtensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml, 2000.

[9] 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):277-290, 2001.

[10] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.

[11] M. Jungel, E. Kindler, and M. Weber. The Petri Net Markup Language. In S. Philippi, editor, *Proceedings of AWPN 2000 - 7thWorkshop Algorithmen und Werkzeuge für Petrinetze*, pages 47-52. Research Report 7/2000, Institute for Computer Science, University of Koblenz, Germany, 2000.

[12] A. Kumar and J.L. Zhao. Workflow Support for Electronic Commerce Applications. (http://spot.colorado.edu/~akhil/, to appear in DSS), 1999.

[13] P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.

[14] 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):345-357, 2001.

[15] R.M. Lee. Distributed Electronic Trade Scenarios: Representation, Design, Prototyping. *International Journal of Electronic Commerce*, 3(2):105-120, 1999.

[16] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541-580, 1989.

[17] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*}, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[18] W. Sadiq and M.E. Orlowska. Analyzing Process Models using Graph Reduction Techniques. *Information Systems*, 25(2):117-134, 2000.

[19] H.M.W. Verbeek and W.M.P. van der Aalst. Woflan 2.0: A Petri-net-based Workflow Diagnosis Tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 475-484. Springer-Verlag, Berlin, 2000.

[20] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes Using Woflan. The Computer Journal, 44(4):246-279. British Computer Society, 2001