Verification of XRL: An XML-based Workflow Language

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.
[2] Database Systems Research Department, Bell Laboratories,
600 Mountain Ave., 2A-406, Murray Hill, NJ 07974, USA.
E-mail: w.m.p.v.d.aalst@tm.tue.nl, h.m.w.verbeek@tm.tue.nl, akhil@acm.org

Abstract

XRL (eXchangeable Routing Language) is an instance-based workflow language that 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, 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. In this paper, we show soundness properties of XRL constructs by using a novel, constructive approach. We also describe a software tool based on XML and Petri-net technologies for verifying XRL workflows.

1. Introduction

Recent years have seen the proliferation of workflow management systems developed for different types of workflows and based on different paradigms [9]. Despite the abundance of such tools, the critical issue of workflow verification is virtually neglected [1]. Few tools provide any form of verification support. The tools Woflan [11] and Flowmake [10] are two noteworthy exceptions.

To complicate matters, more and more workflow management systems are used to support inter-organizational business processes, e.g., in the context of Business-To-Business (B2B) E-commerce. Especially for *open* E-commerce (i.e., 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. One of the prerequisites for this is the guarantee that the workflow process definitions do not contain any logical errors.

Another requirement for *open* E-commerce is that one cannot make any arbitrary assumptions about the workflow processes implemented in the participating organizations. For instance, in the context of inter-organizational workflow it is unrealistic to assume that the different organizations share a common process model. Therefore, we developed the *eXchangeable Routing Language* (XRL), which describes processes at an instance level [4]. Traditional workflow modeling languages describe processes at a class or type level [7]. 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 (workflow modeling languages typically have problems handling a variable number of parallel or alternative branches) [4].

The semantics of XRL are expressed in terms of Petri nets [4]. Such formal semantics allow for powerful analysis techniques, an efficient and compact implementation, interfaces to many existing tools, and, last but not least, an unambiguous understanding of XRL.

We have developed a workflow management system, named XRL/flower [4], 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. Last, but not least, the Petri net

representation can be analyzed using state-of-the-art analysis techniques and tools.

In this paper, we present a verification tool that can analyze workflows specified in terms of XRL. The tool is built on top of Woflan [11] and detects design errors resulting in deadlocks, livelocks, etc. We also give some analytical results which show that for a subset of XRL correctness is guaranteed if the design is consistent (i.e., a well-formed and valid XML file) with the XRL Document Type Definition (DTD).

2. XRL and XRL/flower

The focus of this paper is on verification. Therefore, we limit ourselves to only a brief introduction to XRL, the translation of XRL to Petri nets, and the workflow management system XRL/flower. The syntax of XRL is completely specified by the DTD given in the appendix. A routing element is an important building block of XRL and it can be any one of the following: *task* (a step to be performed), *sequence* (a set of tasks to be done in a specific order), *any_sequence* (a set of tasks to be done in any order), *choice* (any one task out of a set of tasks), *condition* (test a condition and determine next step based on result of the test), *parallel_sync* (create multiple parallel routing elements and later join them), *parallel_no_sync* (create multiple parallel routing elements which do not have to join), *parallel_part_sync* (create multiple parallel routing elements, some of which must join), *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 task while a condition is true), *stop* (end the execution of this particular path of the workflow instance), *terminate* (end this workflow instance).

An example of a consistent XRL file is shown in Figure 1. This corresponds to the Petri net shown in Figure 2. It describes a workflow where a customer complaint is *registered* and then a questionnaire is sent to the customer in parallel with the complaint being sent to a manager for *evaluation*. If the customer does not fill the form within 3 days the event *receipt* occurs any way and puts a token in the *receipt* place. This event and the *evaluate* activity enable the loop in which *process* and *check* steps occur. After the result of *check* step is okay, the *archive* step is enabled.

This translation from XRL to Petri net is done according to rules given in [4]. For the sake of readability, we removed superfluous transitions and places that were generated by the rules. Note that task transitions are represented by squares and routing transitions by bars.

```
<?xml version="1.0" encoding="UTF-8"?>
<route name="example">
  <sequence>
    <task name="register"/>
    <parallel_sync>
      <sequence>
        <task name="send"/>
        <wait_all>
          <event_ref name="receipt"/>
          <timeout time="3 days">
            <task name="timeout">
              <event name="receipt"/>
            </task>
          </timeout>
        </wait_all>
      </sequence>
      <sequence>
        <task name="evaluate"/>
        <condition>
          <true>
            <sequence>
              <wait_all>
                <event_ref name="receipt"/>
              </wait_all>
              <while_do condition="okay">
                <sequence>
                  <task name="process"/>
                  <task name="check"/>
                </sequence>
              </while_do>
            </sequence>
          </true>
        </condition>
      </sequence>
    </parallel_sync>
    <task name="archive"/>
  </sequence>
</route>
```

Figure 1. An example XRL file

In the remainder, we will use the term *XRL route* to refer to a consistent XRL routing schema such as the one shown in Figure 1. A consistent sub-workflow made from these constructs is also called a routing element, e.g., `re_1, re_2,` etc. 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.

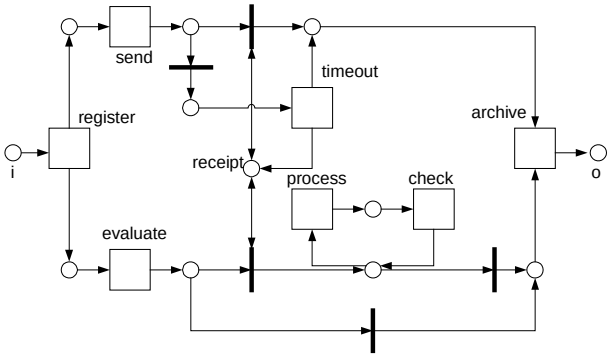

Figure 2. Petri net of example XRL file

Based on XRL, we have developed a workflow management system named XRL/flower. XRL/flower. 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.

3. Verification

XRL is aimed towards application domains where workflows cross organizational boundaries and change over time. In these applications domains, the correctness issues are particularly relevant because the distributed and dynamic nature of the workflow is a potential source of errors. Unfortunately, today's workflow management systems do not support advanced techniques to verify the correctness of workflow process definitions. These systems typically restrict themselves to a number of simple syntactic checks. Therefore, erroneous conditions such as deadlocks and livelocks may remain undetected. This means that an erroneous workflow may go into production, thus causing dramatic problems for the organization. An erroneous workflow may lead to extra work, legal problems, angry customers, managerial problems, and depressed employees. Therefore, it is important to verify the correctness of a workflow process definition before it becomes operational. In fact, for inter-organizational workflows, the costs of putting an erroneous workflow process definition into production are enormous because of the efforts required to repair errors crossing organizational boundaries.

The *soundness property*, defined in [1], relates to the dynamics of the workflow process definition expressed in terms of a so-called *workflow net*. A workflow net is a Petri net with one unique source place (for the initial state) and one unique sink place (for the final state). A workflow net is *sound* if the following requirements are satisfied: (1) termination is guaranteed, (2) upon termination, no dangling references (tokens) are left behind, and (3) there are no dead tasks, i.e., it should be possible to execute an arbitrary task by following the appropriate route. Soundness is the minimal property any workflow net should satisfy. Note that soundness implies the absence of livelocks and deadlocks. Soundness can be verified using Petri net techniques. In fact, we have developed a workflow verification tool, named *Woflan*, to decide soundness [12]. For a given workflow net, Woflan is able to decide whether it is sound. For this purpose, Woflan uses an interesting relation between soundness on the one hand, and liveness and boundedness on the other. A workflow net is sound, if and only if, the net obtained by connecting the sink place to the source place via an additional transition t^* is live and bounded. This relationship enables the use of efficient analysis techniques and the deployment of powerful software packages.

In a workflow net there is one unique source place and one unique sink place. For a straightforward translation of XRL to Petri nets, this requirement is too restrictive. Therefore, we introduce *extended workflow nets*. An extended workflow net may have multiple source places and multiple sink places. Furthermore, the requirement that only source places can indicate the arrival of a case and only sink places can indicate completion of a case is too restrictive. For this reason, we introduce so-called start- and end places. A workflow instance (i.e., case) is started by marking *one* of the start places, and it is completed if only end places are marked. Typically, a start place is a source place and an end place is a sink place. However, we also allow for instance the situation with end places that are no sink places (i.e., with outgoing arcs). We define start and end places as follows. A place is a *start place* if and only if it is a source place, i.e., a place without any ingoing arcs. A place p is an *end place* if every transition that consumes tokens from p also produces at least one token for p. Formally, a place is an end place if and only if it is a trap [6]. Note that any sink node is a trap, i.e., an end place. Any node of the extended workflow net should be on a path from some start place to some end place. Finally, we drop the assumption that every transition corresponds to some task. Transitions that do correspond to tasks are called task transitions.

Definition I. Extended workflow net
A Petri net is called an extended workflow net if and only if (1) it contains at least one start place, (2) it contains at least one end place, and (3) every task transition is on a path from some start place to some end place.

The Petri net shown in Figure 2 is an extended workflow net, There is one start place (i.e., place *i*) and two end places: *receipt* and *o*. Note that *receipt* is an end place but not a sink place. A state is called a *start state* if precisely one of the start places is marked. The set of *reachable states* of an extended workflow net are those states that are reachable from any start state. A reachable state is called a *final state* if and only if only end places are marked.

Definition II. Soundness
An extended workflow net is *sound* if and only if (1) from any reachable state it is possible to reach a final state, (2) no transitions are enabled in any reachable final state, and (3) there are no dead task transitions, i.e., it should

be possible to execute an arbitrary task by following the appropriate route.

Recall that the soundness property is important because it implies absence of deadlocks and livelocks. It is easy to verify that the extended workflow net shown in Figure 2 is indeed sound.

4. Modeling XRL constructs

Although a (non-extended) workflow net is too restrictive in general, it is sufficient if we limit ourselves to use only certain XRL constructs. These constructs typically can be modeled as a workflow net and preserve soundness. We show this in a constructive manner. Afterwards, we discuss constructs for which workflow nets are too restrictive.

The Petri net corresponding to a *task* is a workflow net, provided it does not generate any *events*. It is straightforward to show that this net is sound.

For all constructs, we assume that each embedded routing element corresponds to a Petri net that is a sound workflow net.

For the *sequence* construct, it is straightforward to see that its corresponding Petri net is a workflow net. This workflow net is sound under the assumption mentioned above, which is easy to verify.

In a similar way, the constructs *any_sequence*, *parallel_sync*, *choice*, *condition*, and *while_do* have corresponding Petri nets that are workflow nets and are sound under the assumption as mentioned above. For sake of completeness, we mention that the semantics as given in [4] for the *condition* construct assumes that exactly two routing elements are embedded: one if the condition evaluates to true and one if it evaluates to false. In general, the semantics of the *condition* construct can be described as in Figure 3.

Result 1: If an XRL route is restricted to the constructs *sequence*, *any_sequence*, *choice*, *condition*, *while_do*, and *parallel_sync,* and if no *events* are produced by the *tasks*, then the corresponding Petri net representation is a sound workflow net.

The use of *tasks* producing *events* and the constructs *parallel_no_sync*, *parallel_part_sync*, *wait_all*, and *wait_any* results in additional end places. For the *parallel_no_sync*, an output place is added to each embedded routing element. Such an output place is an end place. For the *parallel_part_sync*, a rather complex network is used to make sure that the remaining control threads are detached, i.e., superfluous tokens are removed. This network contains an additional end place to signal completion. Furthermore, each event place is also an end place. As a result, if the constructs *stop* and *terminate* are not used, the Petri net corresponding to an XRL route is an *extended* workflow net. This net is sound, which is straightforward to prove, if the following conditions hold: (1) for each *parallel_part_sync* the number of branches *k* to synchronize is valid, i.e., at least 1 and at most the total number of branches *n*, (2) every *wait_all* and *wait_any* construct has a *timeout* defined. From now on we will assume that all *parallel_part_sync* constructs are valid, which seems reasonable.

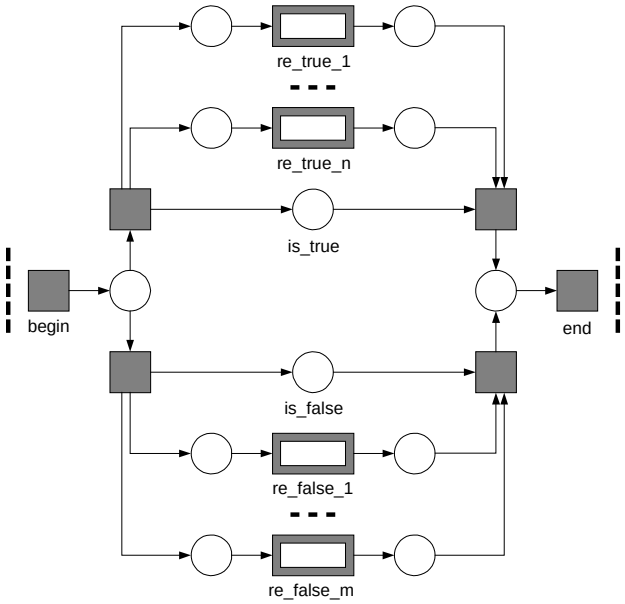

Figure 3. Construct *condition*

Result 2: If an XRL route does not contain *stop* or *terminate* constructs, and if every *wait_all* and *wait_any* construct contains a *timeout*, then the corresponding Petri net representation is a sound extended workflow net.

This result can be applied in many situations and requires only checks at the syntactical level. However, whenever *stop* and *terminate* constructs are used and/or *wait_all* and *wait_any* constructs without a *timeout* are used, a more detailed analysis is required.

Assume the *stop* construct is put in the middle of a sequence, e.g., in-between tasks *t1* and *t2*. Task *t2* can never be executed, thus violating the third requirement of soundness. Therefore, the *stop* construct should only be used at the end of a sequence. Moreover, when a sequence is embedded in, for instance, a *parallel_sync* construct, then no *stop* construct is allowed in it.

Similarly, the *terminate* construct can invalidate soundness. Moreover, the *terminate* construct can also prevent tasks executed in parallel from being executed.

By enforcing syntactical requirements it is possible to extend Result 2 to nets using *stop* and *terminate* constructs. For example, in any sequence a *stop* should not be followed by a task. However, if some *wait_all* or

wait_any does not contain a *timeout*, it is not possible to decide whether or not the net is sound using only syntactical criteria. To deal with this situation, we have developed a link between XRL and our workflow verification tool Woflan [11].

5. Verification tool: XRL/Woflan

In this section we describe an automatic translation from XRL to Woflan. This way any workflow management based on XRL can benefit from state-of-the-art verification software.

Woflan (http://www.tm.tue.nl/it/woflan) is designed as a WFMS-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).

Woflan can read Petri Net Markup Language (PNML) files. PNML is a Petri net file format based on XML [8]. Therefore, it is obvious to use XSL to automatically translate an XRL route into a PNML representation that can be diagnosed using Woflan. This translation can be done in a rather straightforward way; the only difficulty is the generation of unique names for the transitions and places used in the various constructs. Figure 4 shows a snippet of the PNML file corresponding to the example XRL file:

```
<?xml version="1.0" encoding="UTF-8"?>
<net id="example">
  <name>
    <value>example</value>
  </name>
  <transition id="register_begin">
    <name>
      <value>"register begin"</value>
    </name>
  <place id="register_executing">
    <name>
      <value>"register executing"</value>
    </name>
  </place>
  <arc id="register_begin_executing"
       source="register_begin"
       target="register_executing"/>
  …
</net>
```

Figure 4. The resulting PNML file (snippet)

Note that it is only necessary to use Woflan if Result 2 does not apply, i.e., for verifying *wait_all* and *wait_any* constructs that do not contain *timeouts* sub-elements or nasty *stop* and *terminate* constructs. We tested Woflan on the example XRL file of Figure 1. In this case there is one *wait_all* without *timeout*. Therefore, we use Woflan to check whether this *wait_all* can deadlock. At first, Woflan concludes that a transition corresponding to the other *wait_all* is dead. But for this *wait_all* a *timeout* is defined, so this situation is not problematic. After removing the dead transition from the net, Woflan concludes that the net is a sound extended workflow net.

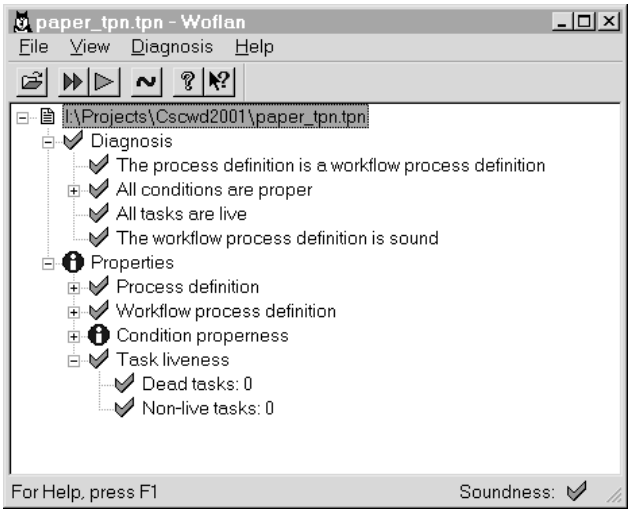

Figure 5. Woflan's results

6. Conclusion

We presented a way to verify XRL routes. Depending on the constructs used in an XRL route, we can (1) claim soundness, (2) decide soundness by checking the XRL route's structure, or (3) decide soundness by using Woflan as a verification tool. We only need to use Woflan if *wait_all* or *wait_any* construct are used that do not contain a *timeout*. If those constructs are not used, soundness depends only on the positioning of the *stop* and *terminate* constructs.

Because Woflan only could check soundness on workflow nets, we have extended Woflan. Woflan can now also check soundness on extended workflow.

For translating XRL route to Woflan, we have build an XSL file that translates the XRL route to a PNML file. Woflan can read PNML files.

References

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

[2] W.M.P. van der Aalst. Process-oriented Architectures for Electronic Commerce and Interorganizational Workflow. *Information Systems*, 25(1):43-69, 2000.

[3] W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of

Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2000.
[4] W.M.P. van der Aalst and A. Kumar. XML Based Schema Definition for Support of Inter-organizational Workflow. Technical report Bell-Labs. 2001.
[5] 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, World Wide Web Consortium (W3C), October 2000.
[6] J. Desel and J. Esparza. *Free choice Petri nets*, volume 40 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1995.
[7] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation* International Thomson Computer Press, 1996.
[8] M. Jüngel, E. Kindler and M. Weber. The Petri Net Markup Language. In S. Philippi (ed.), AWPN, Koblenz, 2000.
[9] P. Lawrence, editor. *Workflow Handbook 1997, Workflow Management Coalition*. John Wiley and Sons, New York, 1997.
[10] W. Sadiq and M.E. Orlowska. Analyzing Process Models using Graph Reduction Techniques. *Information Systems*, 25(2): 117-134, 2000.
[11] 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.
[12] H.M.W. Verbeek and T. Basten and W.M.P. van der Aalst. Diagnosing Workflow Processes Using Woflan, BETA Working Paper Series, WP 48. Eindhoven University of Technology, Eindhoven, 2000.
[13] Workflow Management Coalition, Workflow Standard – Interoperability Wf-XML Binding, Document Number WFMC-TC-1023, Version 1.0, http://www.wfmc.org, May 2000.

Appendix: XRL Document Type Definition

For the semantics of the following DTD, we refer to [4].

```
<!ENTITY % routing_element
"task|sequence|any_sequence|choice|
condition|parallel_sync|parallel_no_sync|
parallel_part_sync|wait_all|wait_any|
while_do|stop|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 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 stop EMPTY>
<!ELEMENT terminate EMPTY>
<!ELEMENT state (event+)>
```