



Antonio Brogi Stefano Forti

Advanced Software Engineering

Lecture Notes



Copyright © 2018

UNIVERSITY OF PISA

These teaching notes are meant to be used only by the students of the *Advanced Software Engineering* course of the *Master's Degree in Computer Science* and of the *Master's Degree in Computer Science and Networking* at the University of Pisa. Please note that the notes are still in draft form and they may contain inaccuracies. Comments and suggestions are welcome, please email them to `antonio.brogi@di.unipi.it` and `stefano.forti@di.unipi.it`.



Contents

1	Core Interoperability Standards	5
1.1	XML	5
1.2	REST	7
1.3	SOAP	10
1.4	WSDL	11
2	Business Process Modelling with Workflow Nets	15
2.1	Background: Petri Nets	15
2.2	Workflow Nets	16
	Bibliography	19



1. Core Interoperability Standards

1.1 XML

The Extensible Markup Language [6] (XML) is a language used for the description and delivery of marked up electronic text over the Web. It comprehends the concept of document type, having XML's constituent parts and their structure formally define the type of a document, and the concept of portability over different computing environments (same character encoding). An XML document is composed of named containers (tags) and their contained data values. Containers are represented as:

- a declaration of the XML version,
- a set of elements, textual units that may contain data, other elements or nothing at all,
- attributes added to start tags to better specify a single property for an element using a name/value pair (e.g. <BillingInformation customer-type="manufacturer">).

XML documents are often referred to as instances. Figure 1.1 shows an example of an instance. XML enforces a hierarchical structure of elements that can be nested one into another, starting from a common root element.

```
<?xml version="1.0" encoding="UTF-8"?>
<BillingInformation customer-type="manufacturer">
  <Name> Right Plastic Products </Name>
  <BillingDate> 2002-09-15 </BillingDate>
  <Address>
    <Street> 158 Edward st. </Street>
    <City> Brisbane </City>
    <State> QLD </State>
    <PostalCode> 4000 </PostalCode>
  </Address>
</BillingInformation>
```

Figure 1.1: An XML instance.

XML allows designers to choose their own names and, consequently, clashes may occur. Hence, the language provides the mechanism of **namespaces** as a way of distinguishing between elements that use the same local name but are actually different. The only rule: tag names within a namespace must be unique. By namespaces it is possible to associate elements or attributes in all or part of a document with a particular schema. A namespace declaration is in scope for the element on which it is declared and for all of its children elements. Furthermore, to avoid any possible ambiguities, names within a namespace can be qualified by writing them as a prefix of the namespace and the local name, separated by a column (e.g. `addr:Address`). A namespace in XML is declared through URI specification as follows: `xmlns:<Namespace Prefix> = <some URI>`. An example with both qualified names is reported in Figure 1.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<BillingInformation customer-type="manufacturer"
  xmlns:bi="http://www.plastics_supply.com/BillInfo"
  xmlns:addr="http://www.plastics_supply.com/Addr">
  <bi:Name> Right Plastic Products </bi:Name>
  <addr:Address>
    <addr:Street> 158 Edward st. </addr:Street>
    <addr:City> Brisbane </addr:City>
    <addr:State> QLD </addr:State>
    <addr:PostalCode> 4000 </PostalCode>
  </addr:Address>
  <bi:BillingDate> 2002-09-15 </bi:BillingDate>
</bi:BillingInformation>
```

Figure 1.2: Example of qualified names.

A way of defining XML tags and structure is with **schemas**: documents that clearly define the content of, and structure of, a class of, XML documents. The XML Schema Definition Language (XSD) provides a type system for XML processing environments. An XML schema is made out of components that can be used in turn to assess the validity of well-formed element and attribute information items; they include data types that embrace simple/complex/extensible data types, element types and attribute declarations, constraints, relationships between elements, namespaces.

XSD differentiates between:

- complex types, including nested elements that can consist of further elements and attributes,
- simple types, including elements that can only contain data (possibly with constraints)

and:

- definitions, creating new types,
- declarations, enabling elements and attributes with specific names and types to appear in document instances.

The `<xsd:element>` construct defines the element name, content model and allowable attributes and data types for each element type. A construct that declares an element may use compositors to aggregate existing types into a structure. Compositors are `sequence` (a sequence of individual elements defined within a complex type or group must be followed by the corresponding XML document), `choice` (requiring that the document designer makes a choice among different options), `all` (requiring that the elements contained in a complex type or group appear only once or not at all, in any order), `minOccur` and `maxOccur` (imposing constraints to the number of times an element can be repeated). For attributes, we must specify types.

Schemas can be built out from reusable components, i.e. polymorphically. Complex types can be either extended or restricted via apposite tags (`<xsd:extension>`, `<xsd:restriction>`). Finally, schemas can be imported (different namespace) or included (same namespace) into other ones.

xPath is a query language that permits to create expressions to find specific information in a XML document. The XPath data model views documents as trees as shown in Figure 1.3. **XSLT** (eXtensible Stylesheet Language Transform) permits converting XML documents into other formats.

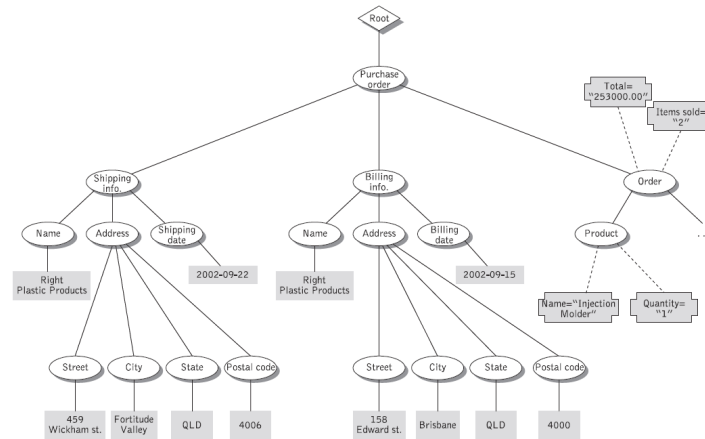


Figure 1.3: Tree structuring of XPath [11].

1.2 REST

Representational State Transfer (REST) [9] was originally proposed as an architectural style, emerging as a lighter alternative to SOAP for message exchange. It was developed by R.T. Fielding in the early '00s as an abstract model of the Web architecture, needed to guide the redesign and definition of HyperText Transfer Protocol (HTTP) [4] and Uniform Resource Identifiers (URIs) [5]. Fielding explained the choice of the name REST as follows:

“[...] to evoke an image of how a well-designed Web application behaves: a network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user.”

In contrast with the complexity of WS* standards, RESTful services constitute a simpler approach to build service-oriented architectures. REST principles can be summarised as:

- (1) *Resource identification through URIs* - A RESTful service exposes a set of resources which identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery.
- (2) *Uniform interface* - Resources are manipulated using a fixed set of operations:
 - PUT creates a new resource,
 - GET retrieves the current state of a resource,
 - POST transfers a new state onto a resource,
 - DELETE deletes an existing resource.
- (3) *Self-descriptive messages* - The RESTful approach is *message-centric*, since all services operate on data by means of a request/response mechanism based on the primitives listed in (2) and requests always contain all information needed to process them. Furthermore, resources are decoupled from their representation so that their content can be accessed in

a variety of formats¹, such as JSON, XML, YAML or HTML. Finally, metadata about the resource is available and used, for instance, to control caching or to negotiate representation format.

- (4) *Stateful interactions through URIs* - When a RESTful service requires a state to perform some computation, the client should send it to the server that offers the service, having the server respond with a new state if needed for later interaction.

Figure 1.4 shows a first example of RESTful service, using XML. Fred would like to update his order to the `barbera.com` food service, to purchase 50 Brontoburgers instead of only one. First, he retrieves his data by accessing the resource that represents him as a customer (i.e., `/customers/fred`) via a GET request to the `barbera.com` server. Then, he exploits the URI that he got to GET back information about his last order (i.e., `/orders/1122/`). Finally, he POSTs an update to the same URI, by changing the quantity of ordered Brontoburgers from 1 to 50.



Figure 1.4: A RESTful service using XML.

Figure 1.5 shows another example of RESTful web service, using on JSON. Now, Fred wishes to organise his Friday evening with Barney, deciding together what to do. Hence, he creates a new poll via a PUT request to the `barbera.com` service. The poll has a title and lists some options. Barney retrieves the poll by exploiting the identifier that Fred got back from `barbera.com` and

¹Currently JSON represents the *de facto* standard for RESTful APIs, since JSON objects can be processed more efficiently, directly as JavaScript objects in modern browsers, or by using off-the-shelf libraries available for all different languages and frameworks. However, with respect to XML, JSON lacks a well-defined, widely supported and clearly structured way to represent data as well as mechanisms that permit extensibility and avoidance of namespace clashes. In this regards, many available commercial services (e.g., Google Maps), still feature both JSON and XML output, being XML more suitable to handle complex situations.

forwarded to him (i.e., GET /polls/112233). He then PUTs his vote into the poll by creating a new vote resource and getting back the URI that he could exploit to update his preference in case he changes his mind. Fred can then retrieve (the result of) the poll as previously done by Barney (i.e., GET /polls/112233) and delete the resource (exploiting DELETE polls/112233) just before his wife Wilma finds out that he has no scheduled work meeting on Friday night.



Figure 1.5: A RESTful service using JSON.

On one hand, the RESTful approach to service-oriented engineering offers some important advantages, being simpler, more scalable and more efficient with respect to traditional SOA. Simplicity is due to the availability of well-established tools to quickly transform existing software into RESTful services by exposing HTTP APIs (e.g., Spring [13] for Java, Django [7] for Python), with a very low learning curve. Scalability and efficiency are guaranteed by the exploitation of the same application protocols that run the Internet and of lightweight formats, ensuring that stateless RESTful services can serve a very large number of clients. Furthermore, deployment and testing of such services is analogous to building dynamic Web sites and they can be performed on a local Web browser or with simple tooling (e.g., Postman [12]).

On the other hand, the success of RESTful architectures also shows shortcomings to this approach. Particularly, (i) complex interfaces or more articulated resources are not easy to be represented as URLs, (ii) no prescriptive best practices exist to design RESTful APIs (ending up in services that mostly expose only GET methods with suitable workarounds to include other functionalities), (iii) management and monitoring of QoS (e.g., security, reliability, transactions) for RESTful services must be implemented "by hand", and (iv) decisions that look very easy to take for RESTful services may lead to significant development wading and technical risks e.g., the design of the exact specification of the resources and their URI addressing scheme. All of this makes it

difficult to extend RESTful Web services to support advanced functionalities in an interoperable manner.

To conclude, RESTful technology is definitely more adequate for human consumption (programmers that read documentation), whilst WSDL/BPEL described services are easily handled and interpreted by the machines of a middleware infrastructure. On one hand REST architectural style is easy and convenient when one wants to get (simple) services running in a short time span, what makes it a good ally when performing *ad hoc* integration over the Web (e.g., services mash-ups). On the other hand, WS-* is still preferable for professional enterprise application integration scenarios with longer lifespan and advanced QoS requirements.

1.3 SOAP

Problem: How to enable the communication among separate systems running on heterogeneous infrastructures? Web services employ SOAP (Simple Object Access Protocol) [2] to address this problem: an XML based standard that relies on HTTP as transport protocol. Communication happens regardless of operating system, programming environment or object model framework differences. The primary use of SOAP is for inter-application communication. SOAP messages are carried as HTTP requests and responses where endpoints are HTTP based URLs. SOAP is a stateless, one-way network application protocol that is used to transfer messages between service instances described by WSDL interfaces.

Each SOAP message consists of an Envelope (Figure 1.6) containing an optional Header and a mandatory Body. The SOAP envelope wraps the XML document interchange and provides a mechanism to augment the payload with additional information useful for routing purposes.

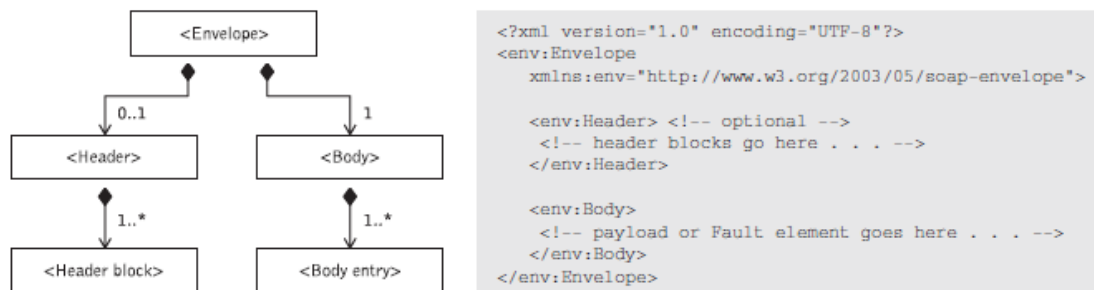


Figure 1.6: SOAP envelope structure.

A SOAP header contains information relevant to how the message must be processed. It may contain information about where the document must be sent, where it originated and may even carry digital signatures (extensibility). SOAP headers have been designed in anticipation of participation of other SOAP processing nodes – called SOAP intermediaries – along a message’s path from an initial SOAP sender to a final SOAP recipient. Different message parts can be processed differently, exploiting many code libraries and different namespaces. When a node receives a message, it determines what role it will assume by inspecting the `role` attribute, which identifies the intended target for that block by means of a URI. The additional attribute `mustUnderstand` specifies whether or not the node processing the header must absolutely process it in a manner consistent with the specifications, or else not process the message at all and report fault.

The SOAP body is mandatory, as an immediate child of the envelope. It contains either an application specific payload (data, parameters to a method call) or a fault message but not both.

SOAP supports two possible communication styles: Remote Procedure Invocation (RPC) or document-style. An RPC-style web service appears as a remote object to a client application: clients express their request as a method call with its parameters and get back a response envelope, enclosing the requested result (or a fault message, `<env:Fault>`).

To conclude SOAP is simple, portable, firewall-friendly, interoperative, open but, on the other hand, it is bound to HTTP (request/response, slow protocol) and stateless.

1.4 WSDL

As we have seen, SOAP defines how to format and process communication messages. However, to send a message to a service *S*, a sender must know the names and parameters of the operations that *S* features, its address and the transport protocol it uses.

Hence, service description is an integral part of designing and developing SOA based applications. The Web Service Description Language [3] (WSDL) is an XML based language that provides a model for achieving this purpose. To develop service-based applications, Web services need to be described in a consistent manner so that they can be published by service providers, and discovered and invoked by clients and developers.

WSDL has become a *de facto* standard language for describing the interface of a Web service, since it recognises the need for rich type systems for describing message formats and supports the XML Schema specification (XSD) as its canonical type system. It provides the means to group messages into operations and operations into interfaces, being nowadays broadly implemented and (supported-by-industry) standards.

Essentially, WSDL is used to describe precisely three things:

1. what a service *S* does (operations featured by *S*),
2. where it resides (the URI of *S*), and
3. how to invoke it (the transport protocol to be used, the parameters of the operations).

A WSDL interface represents a service contract between the service requester and the service provider, in much the same way object oriented programming interfaces work. WSDL is platform and language independent and is used primarily to describe SOAP enabled services, allowing interoperation of heterogeneous applications. The WSDL description consists of two distinct parts:

1. an *abstract service description* with the operations supported by the service, their parameters and abstract data types, and
2. a *concrete endpoint implementation* which binds the abstract interface to an actual network address, to a specific protocol, and to concrete data structures.

This approach leads to modularity and reuse of the interfaces. Particularly, a service can support many bindings to the same abstract interface, accessible at distinct endpoints. The data exchanged is specified as part of a message, every kind of activity allowed at an endpoint is an operation, and collections of permissible operations at an endpoint are grouped into port types. Figure 1.7 shows an actual example of WSDL abstract interface and concrete implementation, highlighting the elements of such document.

A message exchange pattern is defined as a possible order of the input and/or the output elements supported by a given operation. WSDL supports four operation types and any combinations of those four, as sketched in Figure 1.8.

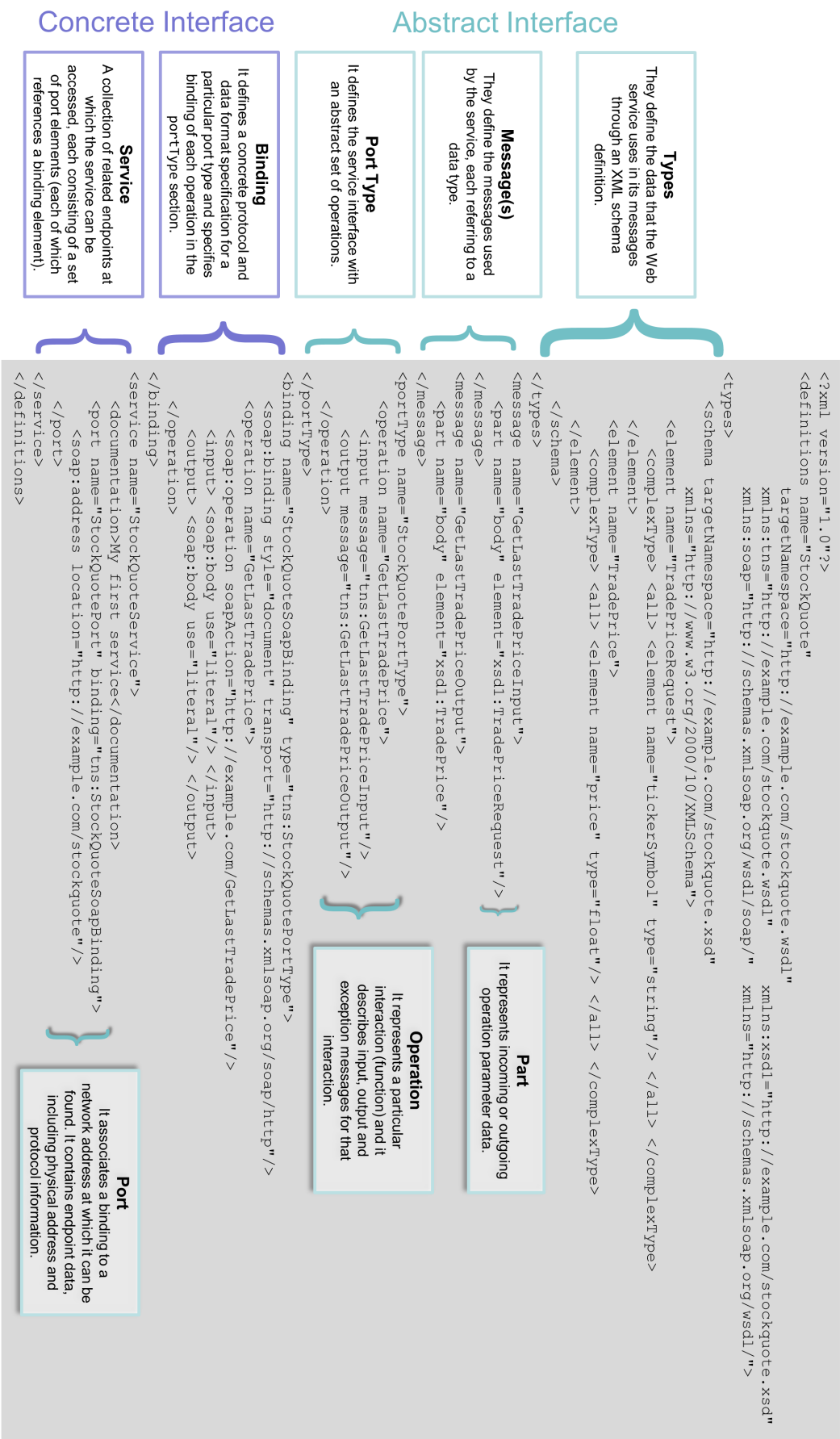


Figure 1.7: WSDL interface elements.



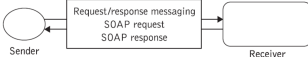
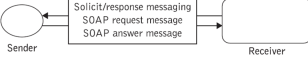
Single Message Passing	One-Way Message		<i>The operation can receive a message but will not return a response.</i>
	Notification		<i>The operation can send a message but will not wait for a response.</i>
Two-way Message Exchange	Request/Response		<i>The operation can receive a request and will return a response.</i>
	Solicit/Reply		<i>The operation can send a request and will wait for a response.</i>

Figure 1.8: Message exchange patterns [11].



2. Business Process Modelling with Workflow Nets

Nowadays, information systems need to support businesses at hand by permitting to control, monitor and support the logistical aspects of their business processes. In other words, the information system also has to manage the flow of work through the organisation, what has been termed *workflow management* [10]. The main purpose of a workflow management system is the support of the definition, execution, registration and control of processes. Since processes are a dominant factor in workflow management, it is important to use an established framework for modelling and analysing business (workflow) processes.

In this chapter we present a framework based on an extended version of Petri nets (PNs), which are known as *workflow nets* [1] (WNs). WNs are a well-established formalism that features a formal semantics, a simple graphical representation, enough expressiveness to model business processes, and a set of well-founded analysis techniques which can be performed automatically by tools like WoPeD [8].

In what follows, after briefly recapitulating on PNs (Section 2.1), we introduce WNs and some of their properties (Section 2.2).

2.1 Background: Petri Nets

Petri nets are directed bipartite graphs with two node types: places and transitions. Nodes are connected via directed arcs and connections between two nodes of the same type are not allowed. Places are represented by circles and transitions by rectangles, as shown in Figure 2.1.

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

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

A place $p \in P$ is called an *input place* of a transition $t \in T$ if and only if there exists a directed arc from p to t . A place $p \in P$ is called an *output place* of a transition $t \in T$ if and only if there exists a directed arc from t to p . We denote by $\bullet t$ the set of input places of t , and by $t \bullet$ the set of output

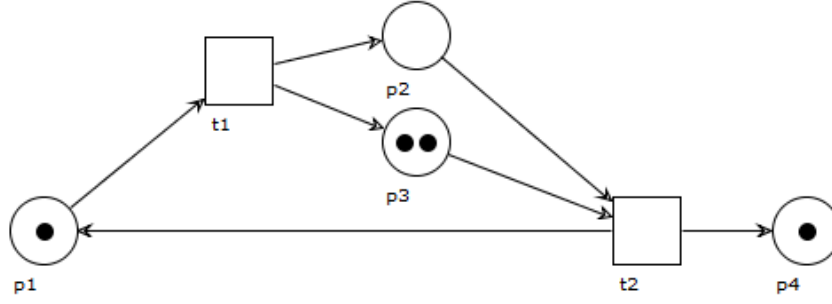


Figure 2.1: An example of Petri net.

places of t . Similarly, we denote by $\bullet p$ and $p\bullet$ the set of transitions for which p is input and output place, respectively.

At any time a place can contain zero or more *tokens*. A *state* M , also referred to as *marking*, of a Petri net is a distribution of tokens over places, i.e., $M \in \{f \mid f : P \rightarrow \mathbb{N}\}$. Given two states M_1 and M_2 , we write $M_1 \geq M_2$ iff $\forall p \in P : M_1(p) \geq M_2(p)$.

A transition t is *enabled* if and only if each input place p of t contains at least one token.

An enabled transition may *fire*. If transition t fires, then t consumes one token from each input place p of t and produces one token in each output place p of t . The notation $M_1 \xrightarrow{t} M_2$ is used to denote that transition t is enabled in state M_1 and that firing t in M_1 results in M_2 .

A state M_n is *reachable* from a state M_1 ($M_1 \xrightarrow{*} M_n$) if and only if there is a firing sequence t_1, t_2, \dots, t_{n-1} such that $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \dots \xrightarrow{t_{n-1}} M_n$.

We use (PN, M) to denote a Petri net PN with initial state M . A state M' is a *reachable state* of (PN, M) if and only if M' is reachable from M .

A Petri net (PN, M) is *live* if and only if for every reachable state M' and every transition t , there is a state M'' reachable from M' which enables t .

A Petri net (PN, M) is *bounded* if and only if for each place p there is a $n \in \mathbb{N}$ such that for every reachable state the number of tokens in p is less than n .

2.2 Workflow Nets

In this section, we briefly introduce workflow nets. A thorough introduction to workflow nets can be found in [1].

A Petri net is a *workflow net* if and only if:

- there is a source place with no incoming edge, and there is a sink place with no outgoing edge, and
- all places and transitions are located on some path from the source place to the sink place.

Definition 2.2.1 A Petri net $PN = (P, T, F)$ is a *workflow net* if and only if:

1. PN has a special place source $i \in P$ such that $\bullet i = \emptyset$ and a special sink place $o \in P$ such that $o\bullet = \emptyset$, and
2. If we add a transition \bar{t} which connects place o with i (viz., $\bullet \bar{t} = \{i\}$ and $\bar{t}\bullet = \{o\}$), then the resulting Petri net is strongly connected.

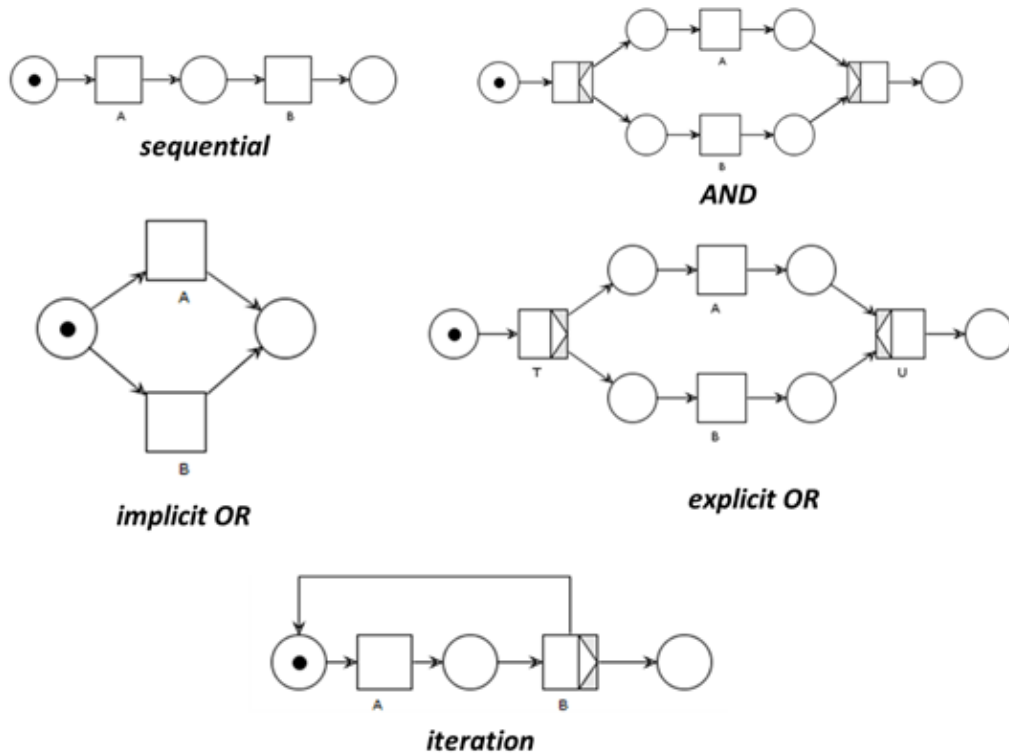


Figure 2.2: Composition patterns in workflow nets.



Figure 2.3: Triggers in workflow nets.

Workflow nets support various composition patterns, as illustrated in Figure 2.2. Note that in the explicit OR pattern – differently from Petri nets – T places only one token in one of its output places, while U can fire if and only if (at least) one of its input places contains a token.

Workflow net transitions can be annotated with triggers, as illustrated in Figure 2.3, to denote who/what is responsible for an enabled transition to fire.

A workflow net is sound if and only if

- every net execution starting from the initial marking (one token in the source place, no tokens elsewhere) eventually leads to the final marking (one token in the sink place, no tokens elsewhere), and
- every transition occurs in at least one net execution.

The formal definition of sound workflow net follows¹.

Definition 2.2.2 A workflow net $WN = (P, T, F)$ is *sound* if and only if:

1. $\forall M : (\{i\} \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} \{o\})$, and
2. $\forall M : (\{i\} \xrightarrow{*} M \wedge M \geq \{o\}) \Rightarrow (M = \{o\})$, and

¹ $\{i\}$ and $\{o\}$ denote the states containing only one token in i and o , respectively.

3. $\forall t \in T \exists M, M' : \{i\} \xrightarrow{*} M \xrightarrow{t} M'$.

The following theorem comes in our aid giving a necessary and sufficient condition to prove soundness of any given workflow net.

Theorem 2.2.1 A workflow net N is *sound* if and only if $(\check{N}, \{i\})$ is live and bounded, where \check{N} is N extended with a transition from the sink place o to the source place i .



Bibliography

- [1] Wil MP Van der Aalst. “The application of Petri nets to workflow management”. In: *Journal of circuits, systems, and computers* 8.01 (1998), pages 21–66 (cited on pages 15, 16).
- [2] M. Gudgin et al. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. Technical report. 2007. URL: <https://www.w3.org/TR/soap12-part1/> (cited on page 10).
- [3] R. Chinnici et al. *Web Services Description Language (WSDL) Version 2.0*. Technical report. 2007. URL: <https://www.w3.org/TR/2007/REC-wsd120-20070626/> (cited on page 11).
- [4] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. 1999, pages 1–176. URL: <https://tools.ietf.org/html/rfc2616> (cited on page 7).
- [5] T. Berners-Lee et al. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. 2005, pages 1–61. URL: <https://tools.ietf.org/html/rfc3986> (cited on page 7).
- [6] T. Bray et al. *Extensible Markup Language (XML)*. Technical report. 2008. URL: <https://www.w3.org/TR/2008/REC-xml-20081126/> (cited on page 5).
- [7] *Django*. <https://www.djangoproject.com> (cited on page 9).
- [8] A Eckleder and T Freytag. “WoPeD A tool for teaching, analyzing and visualizing workflow nets”. In: *Petri Net Newsletter* 75 (2008), pages 3–8 (cited on page 15).
- [9] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000 (cited on page 7).
- [10] Keith Hales and Mandy Lavery. *Workflow management software: the business opportunity*. Ovum Ltd., 1991 (cited on page 15).
- [11] Michael Papazoglou. *Web services & SOA: principles and technology*. Second Edition. Pearson Education, 2012 (cited on pages 7, 13).
- [12] *Postman*. <https://www.getpostman.com> (cited on page 9).
- [13] *Spring.io*. <https://spring.io> (cited on page 9).