2

Specifying and Monitoring Service Flows: Making Web Services Process-Aware

W.M.P. van der Aalst and M. Pesic

Department of Technology Management, Eindhoven University of Technology,
P.O.Box 513, NL-5600 MB, Eindhoven, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl,m.pesic@tm.tue.nl

Abstract. BPEL has emerged as the de-facto standard for implementing processes based on web services while formal languages like Petri nets have been proposed as an "academic response" allowing for all kinds of analysis. Although languages such as BPEL and Petri nets can be used to describe service flows, they both tend to "overspecify" the process and this does not fit well with the autonomous nature of services. Therefore, we propose *DecSerFlow* as a *Declarative Service Flow Language*. By using a more declarative style, there is no need to overspecify service flows. The declarative style also makes DecSerFlow an ideal language for monitoring web services, i.e., using process mining techniques it is possible to check the conformance of service flows by comparing the DecSerFlow specification with reality. This can be used to expose services that do not follow the rules of the game. This is highly relevant given the autonomous nature of services.

2.1 Introduction

Web services, an emerging paradigm for architecting and implementing business collaborations within and across organizational boundaries, are currently of interest to both software vendors and scientists [4]. In this paradigm, the functionality provided by business applications is encapsulated within web services: software components described at a semantic level, which can be invoked by application programs or by other services through a stack of Internet standards including HTTP, XML, SOAP [23], WSDL [24], and UDDI [22]. Once deployed, web services provided by various organizations can be interconnected in order to implement business collaborations, leading to *composite web services*.

Today, workflow management systems are readily available [7, 58, 68] and workflow technology is hidden in many applications, e.g., ERP, CRM, and PDM systems. However, their application is still limited to specific industries such as banking and insurance. Since 2000, there has been a growing interest in web services. This resulted in a stack of Internet standards (HTTP,

XML, SOAP, WSDL, and UDDI) which needed to be complemented by a process layer. Several vendors proposed competing languages, e.g., IBM proposed WSFL (Web Services Flow Language) [57] building on FlowMark/MQSeries and Microsoft proposed XLANG (Web Services for Business Process Design) [84] building on Biztalk. BPEL [18] emerged as a compromise between both languages.

The *Business Process Execution Language for Web Services* (BPEL4WS, or BPEL for short) has become the de-facto standard for implementing processes based on web services [18]. Systems such as Oracle BPEL Process Manager, IBM WebSphere Application Server Enterprise, IBM WebSphere Studio Application Developer Integration Edition, and Microsoft BizTalk Server 2004 support BPEL, thus illustrating the practical relevance of this language. Although intended as a language for connecting web services, its application is not limited to cross-organizational processes. It is expected that in the near future a wide variety of process-aware information systems [30] will be realized using BPEL. Whilst being a powerful language, BPEL is difficult to use. Its XML representation is very verbose and readable only to the trained eye. It offers many constructs and typically things can be implemented in many ways, e.g., using links and the flow construct or using sequences and switches. As a result, only experienced users are able to select the right construct. Several vendors offer a graphical interface that generates BPEL code. However, the graphical representations are a direct reflection of the BPEL code and are not intuitive to end-users. Therefore, BPEL is closer to classical programming languages than, e.g., the more user-friendly workflow management systems available today.

In discussions, Petri nets [78] and Pi calculus [67] are often mentioned as two possible formal languages that could serve as a basis for languages such as BPEL. Some vendors claim that their systems are based on Petri nets or Pi calculus and other vendors suggest that they do not need a formal language to base their system on. In essence, there are three "camps" in these discussions: the "Petri net camp," the "Pi calculus" (or process algebra) camp, and the "Practitioners camp" (also known as the "No formalism camp"). This was the reason for starting the "Petri nets and Pi calculus for business processes" working group [76] in June 2004. More than two years later the debate is still ongoing and it seems unrealistic that consensus on a single language will be reached.

This chapter will *discuss the relation between Petri nets and BPEL and show that today it is possible to use formal methods in the presence of languages like BPEL*. However, this will *only be the starting point* for the results presented in this chapter. First of all, we introduce a new language *DecSerFlow*. Second, we show that *process mining* techniques can be very useful when monitoring web services.

The language *DecSerFlow* is a *Declarative Service Flow Language*, i.e., it is intended to describe processes in the context of web services. The main motivation is that languages like BPEL and Petri nets are procedural by

nature, i.e., rather than specifying "what" needs to happen these languages describe "how" things need to be done. For example, it is not easy to specify that anything is allowed as long as the receipt of a particular message is never followed by the sending of another message of a particular type. DecSerFlow allows for the specification of the "what" without having to state the "how." This is similar to the difference between a program and its specification. (For example, one can specify what an ordered sequence is without specifying an algorithm to do so.)

In a service-oriented architecture, a variety of events (e.g., messages being sent and received) are being logged [6, 73]. This information can be used for *process mining* purposes, i.e., based on some event log it is possible to *discover* processes or to *check conformance* [14, 13]. The goal of process discovery is to build models without a priori knowledge, i.e., based on sequences of events one can look for the presence or absence of certain patterns and deduce some process model from it. For conformance checking, there has to be an initial model. One can think of this model as a "contract" or "specification" and it is interesting to see whether the parties involved stick to this model. Using conformance checking it is possible to quantify the fit (fewer deviations result in a better fit) and to locate "problem areas" where a lot of deviations take place.

In this chapter we will show that there is a clear *link between more declarative languages such as DecSerFlow and process mining*. In order to do so, it is important to look at the roles that process specifications can play in the context of web services [94, 95]:

- DecSerFlow can be used as a *global model*, i.e., interactions are described from the viewpoint of an external observer who oversees all interactions between all services. Such a model is also called a *choreography model*. Note that such a global model does not need to be executable. However, the model is still valuable as it allows for conformance checking, i.e., by observing interactions it is possible to detect deviations from the agreed upon choreography model. Here DecSerFlow is competing with languages such as the *Web Services Choreography Description Language* (WS-CDL) [54].
- DecSerFlow can be used as a *local model*, i.e., the model that is used to specify, implement, or configure a particular service. Here DecSerFlow is competing with languages such as BPEL [18].

As discussed in [94, 95], it is interesting to link global and local models. Relating global models (that are produced by analysts to agree on interaction scenarios from a global perspective) to local models (that are produced during system design and handed on to implementers) is a powerful way of ensuring that services can work together. Although DecSerFlow can be used at both levels, we will argue that it is particularly useful at the global level. Moreover, we will show that global models can be used to check conformance using process mining techniques.

The remainder of this chapter is organized as follows. Section 2.2 describes the "classical approach" to processes in web services, i.e., Petri nets and BPEL are introduced and pointers are given to state-of-the-art mappings between them. Section 2.3 first discusses the need for a more declarative language and then introduces the DecSerFlow language. In Sect. 2.4 the focus shifts from languages to the monitoring of services. Finally, there is a section on related work (Sect. 2.5) and a conclusion (Sect. 2.6).

2.2 Classical Approaches: BPEL and Petri Nets

Before we introduce the DecSerFlow, we focus on two more traditional languages for the modeling of service flows, i.e., Petri nets and BPEL. Petri nets are more at the conceptual level and can serve only as a theoretical basis for the modeling and analysis of service flows. BPEL is emerging as the de-facto standard for implementing processes based on web services. In this section, we also discuss the link between Petri nets and BPEL and present two tools: one to map Petri nets onto BPEL and another to map BPEL onto Petri nets.

2.2.1 Petri Nets

Petri nets [78] were among the first formalisms to capture the notion of concurrency. They combine an intuitive graphical notation with formal semantics and a wide range of analysis techniques. In recent years, they have been applied in the context of process-aware information systems [30], workflow management [7, 9], and web services [64].

To illustrate the concept of Petri nets we use an example that will be used in the remainder of this chapter. This example is inspired by electronic bookstores such as Amazon and Barnes and Noble and taken from [16]. Figure 2.1 shows a Petri-net that will be partitioned over four partners: (1) the *customer*, (2) the *bookstore* (e.g., Amazon or Barnes and Noble), (3) the *publisher*, and (4) the *shipper*. As discussed in Sect. 2.1, Fig. 2.1 can be considered as a *global model*, i.e., interactions are described from the viewpoint of an external observer who oversees all interactions between all services.

The circles in Fig. 2.1 represent *places* and the squares represent *transitions*. Initially, there is one token in place *start* and all other places are empty (we consider one book order in isolation [7]). Transitions are *enabled* if there is a token on each of input places. Enabled transitions can *fire* by removing one token from each input place and producing one token for each output place. In Fig. 2.1, transition *place_c_order* is enabled. When it fires one token is consumed and two tokens are produced. In the subsequent state (also called marking) transition *handle_c_order* is enabled. Note that transitions *rec_acc* and *rec_decl* are not enabled because only one of their input places is marked with a token.

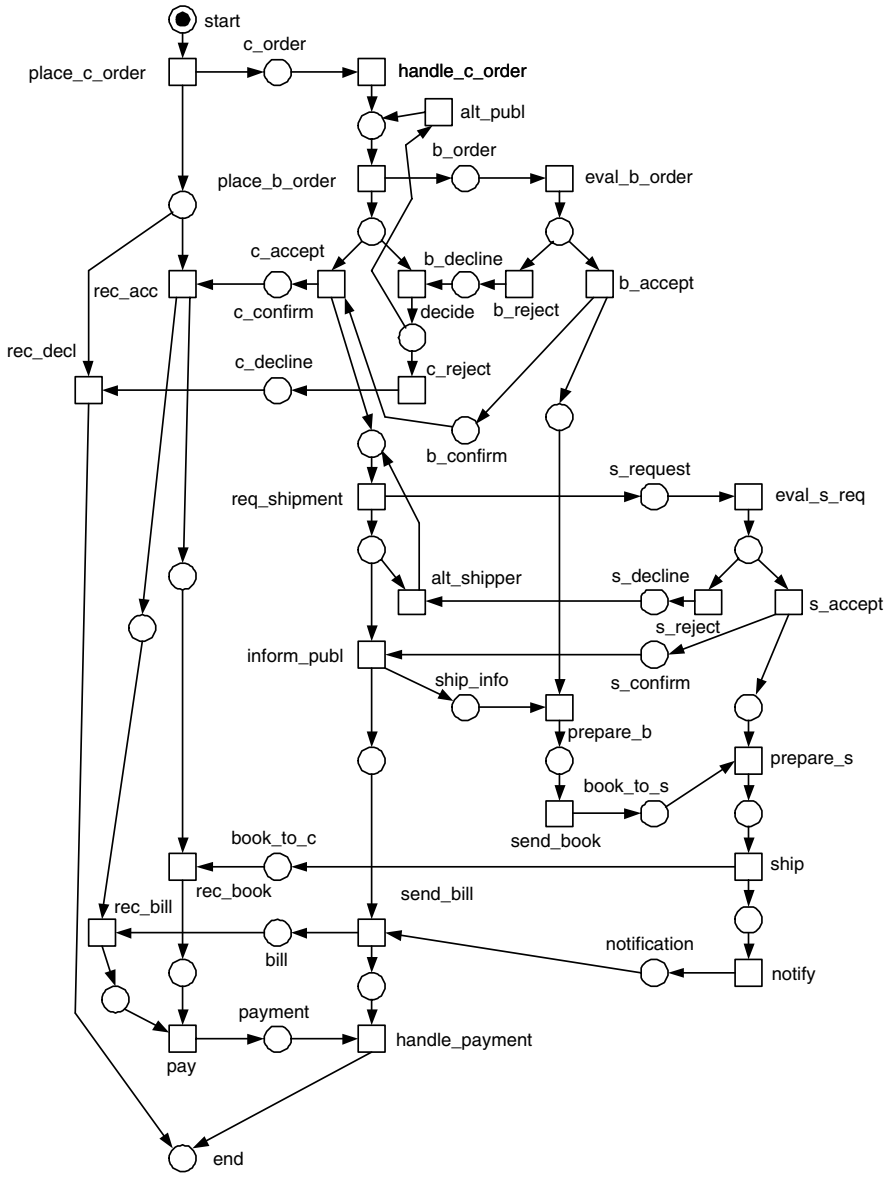


Fig. 2.1. A Petri net describing the process as agreed upon by all four parties (i.e., the global model)

Figure 2.1 represents an inter-organizational workflow that is initiated by a customer placing an order (activity *place_c_order*). This customer order is sent to and handled by the bookstore (activity *handle_c_order*). The electronic bookstore is a virtual company which has no books in stock. Therefore, the bookstore transfers the order of the desired book to a publisher

(activity *place_b_order*). We will use the term "bookstore order" to refer to the transferred order. The bookstore order is evaluated by the publisher (activity *eval_b_order*) and either accepted (activity *b_accept*) or rejected (activity *b_reject*). In both cases an appropriate signal is sent to the bookstore. If the bookstore receives a negative answer, it decides (activity *decide*) to either search for an alternative publisher (activity *alt_publ*) or to reject the customer order (activity *c_reject*). If the bookstore searches for an alternative publisher, a new bookstore order is sent to another publisher, etc. If the customer receives a negative answer (activity *rec_decl*), then the workflow terminates. If the bookstore receives a positive answer (activity *c_accept*), the customer is informed (activity *rec_acc*) and the bookstore continues processing the customer order. The bookstore sends a request to a shipper (activity *req_shipment*), the shipper evaluates the request (activity *eval_s_req*) and either accepts (activity *s_accept*) or rejects (activity *b_reject*) the request. If the bookstore receives a negative answer, it searches for another shipper. This process is repeated until a shipper accepts. Note that, unlike the unavailability of the book, the unavailability of a shipper cannot lead to a cancellation of the order. After a shipper is found, the publisher is informed (activity *inform_publ*), the publisher prepares the book for shipment (activity *prepare_b*), and the book is sent from the publisher to the shipper (activity *send_book*). The shipper prepares the shipment to the customer (activity *prepare_s*) and actually ships the book to the customer (activity *ship*). The customer receives the book (activity *rec_book*) and the shipper notifies the bookstore (activity *notify*). The bookstore sends the bill to the customer (activity *send_bill*). After receiving both the book and the bill (activity *rec_bill*), the customer makes a payment (activity *pay*). Then the bookstore processes the payment (activity *handle_payment*) and the inter-organizational workflow terminates.

The Petri net shown in Fig. 2.1 is the so-called "WF-net" (WorkFlow-net) because it has one input place (*start*) and one output place (*end*) and all places' transitions are on a path from *start* to *end*. Using tools such as Woflan [88] or ProM [29], we can show that the process is *sound* [2, 7]. Figure 2.2 shows a screenshot of the Woflan plug-in of ProM. Soundness means that each process instance can terminate without any problems and that all parts of the net can potentially be activated. Given a state reachable from the marking with just a token in place *start*, it is always possible to reach the marking with one token place *end*. Moreover, from the initial state it is possible to enable any transition and to mark any place. Using ProM it is possible to prove that the Petri net shown in Fig. 2.1 is sound, cf. Fig. 2.2.

One can think of the Petri net shown in Fig. 2.1 as the contract between the customer, the bookstore, the publisher, and the shipper (i.e., global model). Clearly, there are many customers, publishers, and shippers. Therefore, the Petri net should be considered as the contract between all customers, publishers, and shippers. However, since we model the processing of an order for a single book, we can assume, without loss of generality, that only one customer, one publisher, and at most one shipper (at any time) are involved. Note that

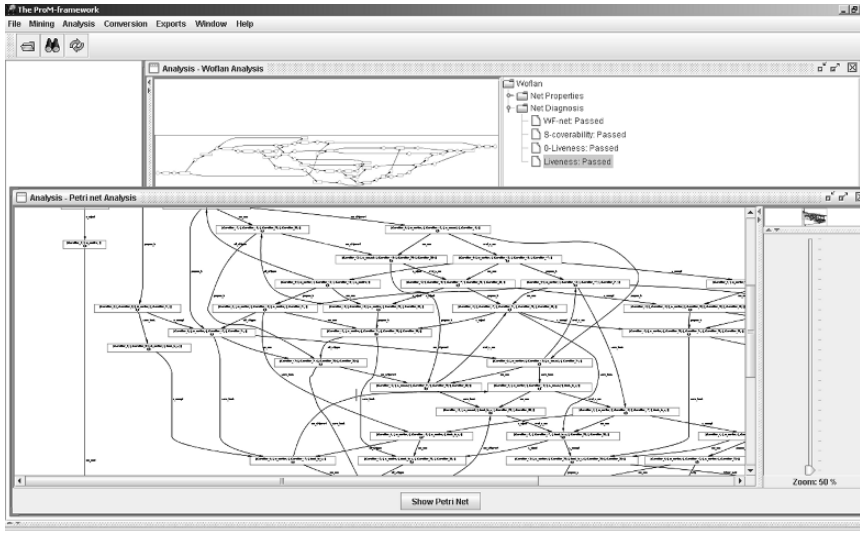


Fig. 2.2. Two analyses plug-in of ProM indicate that the Petri net shown in Fig. 2.1 is indeed sound. The top window shows some diagnostics related to soundness. The bottom window shows part of the state space

Fig. 2.1 abstracts from a lot of relevant things. However, given the purpose of this chapter, we do not add more details.

Figure 2.3 shows the same process but now all activities are partitioned over the four parties involved in the ordering of a book. It shows that each of the parties is responsible for a part of the process. In terms of web services, we can think of each of the four large-shaded rectangles as a service. The Petri-net fragments inside these rectangles can be seen as specifications of the corresponding services (i.e., local models).

It is interesting to point out that in principle multiple shippers could be involved, i.e., the first shipper may decline and then another shipper is contacted, etc. However, at any point in time, at most one shipper is involved in each process instance. Another interesting aspect is the *correlation* between the various processes of the partners. There may be many instances of the process shown in area labeled *bookstore* in Fig. 2.3. However, each instance is unique and messages passed over the places connecting the bookstore to the other partners refer to a particular process instance. In general, it is a non-trivial problem to correlate messages to process instances. See [6, 73] for a more detailed discussion on correlation.

We will refer to whole diagram shown in Fig. 2.3 as the *choreography* or *orchestration* model of the four services.

2.2.2 BPEL

BPEL [18] supports the modeling of two types of processes: executable and abstract processes. An *abstract* (not executable) *process* is a business protocol,

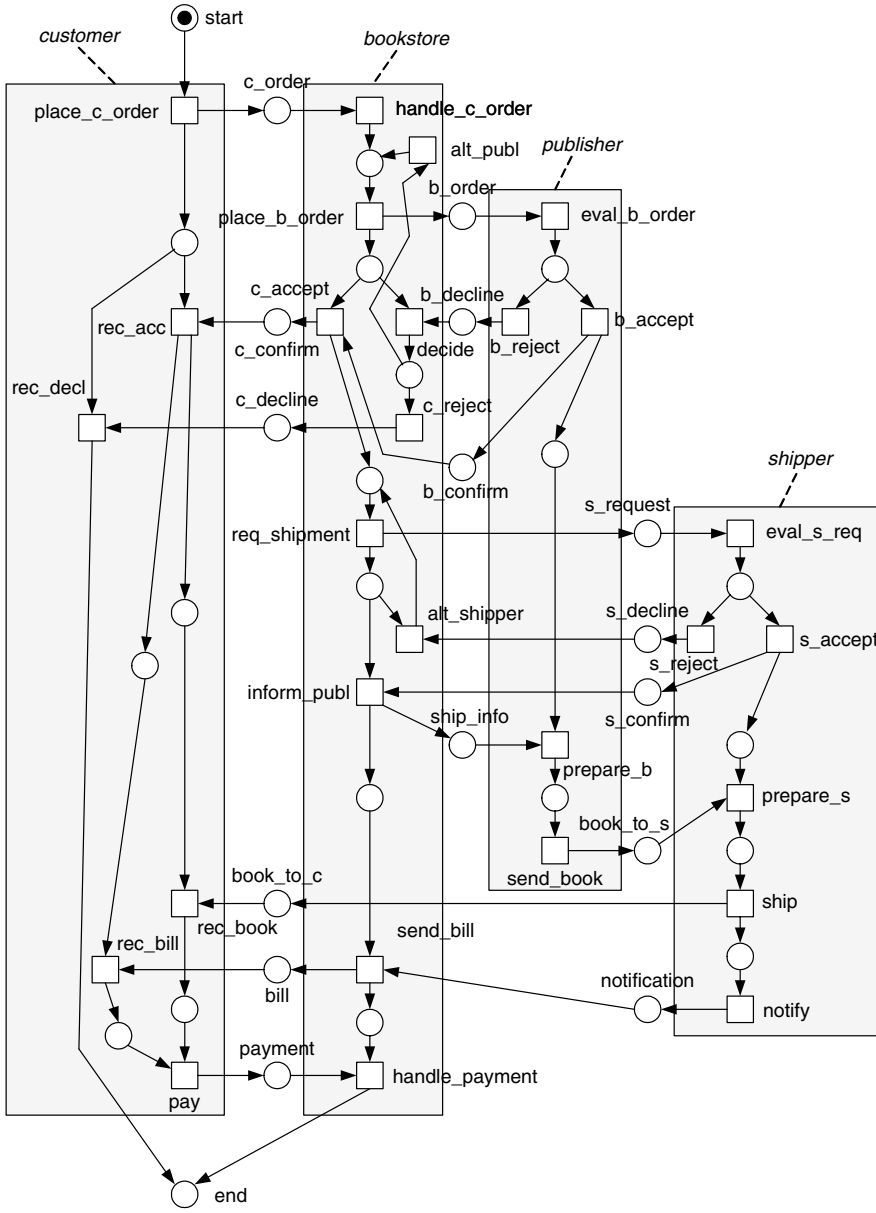


Fig. 2.3. The process as partitioned over (1) the *customer*, (2) the *bookstore*, (3) the *publisher*, and (4) the *shipper* (i.e., four local models and their interconnections)

specifying the message exchange behavior between different parties without revealing the internal behavior of any one of them. This abstract process views the outside world from the perspective of a single organization or (composite) service. An *executable process* views the world in a similar manner; however, things are specified in more detail such that the process becomes executable, i.e., an executable BPEL process specifies the execution order of a number of *activities* constituting the process, the *partners* involved in the process, the *messages* exchanged between these partners, and the *fault* and *exception handling* required in cases of errors and exceptions.

In terms of Fig. 2.3, we can think of *abstract BPEL* as the language to specify one service, i.e., describing the desired behavior of a single Petri-net fragment (e.g., *shipper*). *Executable BPEL* on the other hand can be used as the means to implement the desired behavior.

A BPEL process itself is a kind of flow-chart, where each element in the process is called an *activity*. An activity is either a primitive or a structured activity. The set of *primitive activities* contains *invoke*, invoking an operation on a web service; *receive*, waiting for a message from an external source; *reply*, replying to an external source; *wait*, pausing for a specified time; *assign*, copying data from one place to another; *throw*, indicating errors in the execution; *terminate*, terminating the entire service instance; and *empty*, doing nothing.

To enable the presentation of complex structures the following *structured activities* are defined: *sequence*, for defining an execution order; *switch*, for conditional routing; *while*, for looping; *pick*, for race conditions based on timing or external triggers; *flow*, for parallel routing; and *scope*, for grouping activities to be treated by the same fault-handler. Structured activities can be nested and combined in arbitrary ways. Within activities executed in parallel the execution order can further be controlled by the usage of *links* (sometimes also called control links, or guarded links), which allows the definition of directed graphs. The graphs too can be nested but must be acyclic.

As indicated in Sect. 2.1, BPEL builds on IBM's WSFL (Web Services Flow Language) [57] and Microsoft's XLANG (Web Services for Business Process Design) [84] and combines the features of a block-structured language inherited from XLANG with those for directed graphs originating from WSFL. As a result, simple things can be implemented in two ways. For example, a sequence can be realized using the *sequence* or *flow* elements (in the latter case links are used to enforce a particular order on the parallel elements), a choice based on certain data values can be realized using the *switch* or *flow* elements, etc. However, for certain constructs one is forced to use the block-structured part of the language, e.g., a *deferred choice* [8] can only be modeled using the *pick* construct. For other constructs one is forced to use links, i.e., the more graph-oriented part of the language, e.g., two parallel processes with a one-way synchronization require a *link* inside a *flow*. In addition, there are very subtle restrictions on the use of links: "A link MUST NOT cross the boundary of a while activity, a serializable scope, an event handler or a compensation handler... In addition, a link that crosses a fault-handler boundary MUST

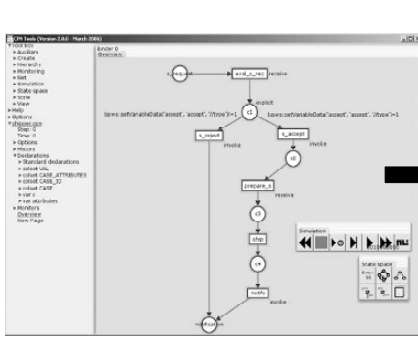
be outbound, i.e., it MUST have its source activity within the fault handler and its target activity within a scope that encloses the scope associated with the fault handler. Finally, a link MUST NOT create a control cycle, i.e., the source activity must not have the target activity as a logically preceding activity, where an activity A logically precedes an activity B if the initiation of B semantically requires the completion of A. Therefore, directed graphs created by links are always acyclic" (see p. 64 in [18]). All of this makes the language complex for end-users. A detailed or complete description of BPEL is beyond the scope of this chapter. For more details, the reader is referred to [18] and various web sites such as the web site of the OASIS technical committee on WS-BPEL [70].

2.2.3 BPEL2PN and PN2BPEL

As shown, both BPEL and Petri nets can be used to describe the process-aspect of web services. There are several process engines supporting Petri nets (e.g., COSA, YAWL, etc.) or BPEL (e.g., Oracle BPEL, IBM WebSphere, etc.). BPEL currently has strong industry support while Petri nets offer a graphical language and a wide variety of analysis tools (cf. Fig. 2.2). Therefore, it is interesting to look at the relation between the two. First of all, it is possible to map BPEL onto Petri nets for the purpose of analysis. Second, it is possible to generate BPEL on the basis of Petri nets, i.e., mapping a graphical, more conceptual, language onto a textual language for execution purposes.

Several tools have been developed to map BPEL onto Petri nets (see Sect. 2.5). As a example, we briefly describe the combination formed by BPEL2PNML and WofBPEL developed in close collaboration with QUT [72]. BPEL2PNML translates BPEL process definitions into Petri nets represented in the Petri Net Markup Language (PNML). WofBPEL, built using Woflan [88], applies static analysis and transformation techniques to the output produced by BPEL2PNML. WofBPEL can be used (1) to simplify the Petri net produced by BPEL2PNML by removing unnecessary silent transitions and (2) to convert the Petri net into the so-called "WorkFlow net" (WF-net) which has certain properties that simplify the analysis phase. Although primarily developed for verification purposes, BPEL2PNML and WofBPEL have also been used for conformance checking using abstract BPEL processes [6].

Few people have been working on the translation from Petri nets to BPEL. In fact, [9] is the only work we are aware of that tries to go from (colored) Petri nets to BPEL. Using our ProM tool [29] we can export a wide variety of languages to CPN Tools. For example, we can load Petri-net models coming from tools such as Protos, Yasper, and WoPeD, EPCs coming from tools such as ARIS, ARIS PPM, and EPC Tools, and workflow models coming from tools such as Staffware and YAWL, and automatically convert the control-flow in these models to Petri nets. Using our ProM tool this can then be exported to CPN Tools where it is possible to do further analysis (state space analysis, simulation, etc.). Moreover, WF-nets in CPN Tools can be converted into BPEL using *WorkflowNet2BPEL4WS* [9, 55]. To illustrate this, consider the



```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
name="shipper" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://schemas.xmlsoap.org/ws/2003/03/business-
process/ http://schemas.xmlsoap.org/ws/2003/03/business-process/"
targetNamespace="http://shipper">
 <sequence name="Sequence_F3">
  <receive name="eval_s_req" />
  <switch name="Switch_F2">
   <case condition="bpws:getVariableData('accept', 'accept', '//type')!=1">
    <invoke name="s_reject" />
   </case>
   <case condition="bpws:getVariableData('accept', 'accept', '//type')=1">
    <sequence name="Sequence_F1">
     <invoke name="s_accept" />
     <receive name="prepare_s" />
     <invoke name="ship" />
     <invoke name="notify" />
    </sequence>
   </case>
  </switch>
 </sequence>
</process>
```

Fig. 2.4. The Petri net describing the service offered by the shipper is mapped onto BPEL code using WorkflowNet2BPEL4WS, a tool to automatically translate colored Petri nets into BPEL template code

shipper service shown in Fig. 2.3. The WF-net corresponding to the shipper process was modeled using the graphical editor of the COSA workflow management system. This was automatically converted by Woflan to ProM. Using ProM the process was automatically exported to CPN Tools. Then using WorkflowNet2BPEL4WS the annotated WF-net was translated into BPEL template code. Figure 2.4 shows both the annotated WF-net in CPN Tools (left) and the automatically generated BPEL template code (right).

The availability of the tools and systems mentioned in this section makes it possible to support *service flows*, i.e., the process-aspect of web services, at the design, analysis, and enactment levels. For many applications, BPEL, Petri nets, or a mixture of both provide a good basis for making web services "process-aware." However, as indicated in Sect. 2.1, the focus of this chapter is on DecSerFlow. Section 2.3 introduces DecSerFlow and shows that it is a truly declarative language which addresses the problem of overspecification typically resulting from the procedural languages described in this section. After introducing the language we focus on the monitoring of service flows (Sect. 2.4) specified in terms of DecSerFlow.

2.3 DecSerFlow

The goal of this section is to provide a "fresh view" on process support in the context of web services. We first argue why a more declarative approach is needed and then introduce a concrete language: DecSerFlow.

2.3.1 The Need for More Declarative Languages

Petri nets and BPEL have in common that they are highly procedural, i.e., after the execution of a given activity the next activities are scheduled.[1] Seen

[1] Note that both BPEL and Petri nets support the deferred choice pattern [8], i.e., it is possible to put the system in a state where several alternative activities are

from the viewpoint of an execution language the procedural nature of Petri nets and BPEL is not a problem. However, unlike the modules inside a classical system, web services tend to be rather autonomous and an important challenge is that all parties involved need to agree on an overall global process. Currently, terms like *choreography* and *orchestration* are used to refer to the problem of agreeing on a common process. Some researchers distinguish between choreography and orchestration, e.g., "In orchestration, there's someone—the conductor—who tells everybody in the orchestra what to do and makes sure they all play in sync. In choreography, every dancer follows a pre-defined plan—everyone independently of the others." We will not make this distinction and simply assume that *choreographies define collaborations between interacting parties*, i.e., the coordination process of interconnected web services all partners need to agree on. Note that Fig. 2.3 can be seen as an example of a choreography.

Within the Web Services Choreography Working Group of the W3C, a working draft defining version 1.0 of the *Web Services Choreography Description Language* (WS-CDL) has been developed [54]. The scope of WS-CDL is defined as follows: "Using the Web Services Choreography specification, a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged is produced that describes, from a global viewpoint, the common and complementary observable behavior of all the parties involved. Each party can then use the global definition to build and test solutions that conform to it. The global specification is in turn realized by a combination of the resulting local systems, on the basis of appropriate infrastructure support. The advantage of a contract based on a global viewpoint as opposed to any one endpoint is that it separates the overall global process being followed by an individual business or system within a domain of control (an endpoint) from the definition of the sequences in which each business or system exchanges information with others. This means that, as long as the observable sequences do not change, the rules and logic followed within a domain of control (endpoint) can change at will and interoperability is therefore guaranteed" [54]. This definition is consistent with the definition of choreography just given. Unfortunately, like most standards in the web services stack, CDL is verbose and complex. Somehow the essence as shown in Fig. 2.3 is lost. Moreover, the language again defines concepts such as "sequence," "choice," and "parallel" in some ad hoc notation with unclear semantics. This suggests that some parts of the language are an alternative to BPEL while they are not.

However, the main problem is that WS-CDL, like Petri nets and BPEL, is *not declarative*. A choreography should allow for the specification of the "what" without having to state the "how". This is similar to the difference

enabled but the selection is made by the environment (cf. the *pick* construct in BPEL). This allows for more flexibility. However, it does not change the fact that in essence both Petri nets and BPEL are procedural.

between the implementation of a program and its specification. For example, it is close to impossible to describe that within a choreography two messages exclude one another. Note that such an exclusion constraint is not the same as making a choice! To illustrate this, assume that there are two actions A and B. These actions can correspond to exchange of messages or some other type of activity which is relevant for the choreography. The constraint that "A and B exclude one another" is different from making a choice between A or B. First of all, A and B may be executed multiple times, e.g., the constraint is still satisfied if A is executed five times while B is not executed at all. Second, the moment of choice is irrelevant to the constraint. Note that the modeling of choices in a procedural language forces the designer to indicate explicit decision points which are evaluated at explicit decision times. Therefore, there is a tendency to overspecify things.

Therefore, we propose a more declarative approach based on *temporal logic* [61, 74] as described in the following subsection.

2.3.2 DecSerFlow: A Declarative Service Flow Language

Languages such as *Linear Temporal Logic* (LTL) [41, 45, 46] allow for more declarative style of modeling. These languages include temporal operators such as nexttime ($\bigcirc F$), eventually ($\Diamond F$), always ($\Box F$), and until ($F \sqcup G$), cf. Table 2.1. However, such languages are difficult to read. Therefore, we define a graphical syntax for the typical constraints encountered in service flows. The combination of this graphical language and the mapping of this graphical language to LTL forms the *Declarative Service Flow (DecSerFlow) Language*. We propose DecSerFlow for the *specification of a single service, simple service compositions, and more complex choreographies*.

Developing a model in DecSerFlow starts with creating activities. The notion of an activity is like in any other workflow-like language, i.e., an activity is atomic and corresponds to a logical unit of work. However, the nature of the *relations between activities* in DecSerFlow can be quite different than in

Table 2.1. Brief explanation of the basic LTL temporal operators

name	notation	explanation
nexttime	$\bigcirc F$	F has to hold at the next state, e.g., *[A,F,B,C,D,E]*, *[A,F,F,F,F,F,B,C,D,E]*, *[F,F,F,F,A,B,C,D,E]*, etc.
eventually	$\Diamond F$	F has to hold eventually, e.g., *[F,A,B,C,D,E]*, *[A,B,C,F,D,E]*, *[ABFCDFEF]*, etc.
always	$\Box F$	F has to always hold, e.g., *[F,F,F,F,F,F]*.
until	$F \sqcup G$	G holds at the current state or at some future state, and F has to hold until G holds. When G holds F does not have to hold any more. Examples are *[G,A,B,C,D,E]*, *[F,G,A,B,C,D,E]*, *[F,F,F,F,G,A,B,C,D,E]*, *[F,F,F,F,G,A,B,G,F,C,D,E,F,G]*, etc.

traditional procedural workflow languages (like Petri nets and BPEL). For example, places between activities in a Petri net describe causal dependencies and can be used to specify sequential, parallel, alternative, and iterative routing. By using such mechanisms, it is both possible and necessary to strictly define *how* the flow will be executed. We refer to the relations between activities in DecSerFlow as *constraints*. Each of the constraints represents a policy (or a business rule). At any point in time during the execution of a service, each constraint evaluates to *true* or *false*. This value can change during the execution. If a constraint has the value *true*, the referring policy is fulfilled. If a constraint has the value *false*, the policy is violated. The execution of a service is *correct* (according to the DecSerFlow model) at some point in time if all constraints (from the DecSerFlow model) evaluate to *true*. Similarly, a service has *completed correctly* if at the end of the execution all constraints evaluate to *true*. The goal of the execution of any DecSerFlow model is not to keep the values of all constraints *true* at all times during the execution. A constraint which has the value *false* during the execution is not considered an error. Consider, e.g., the LTL expression $\Box(A \longrightarrow \Diamond B)$ where A and B are activities, i.e., each execution of A is eventually followed by B. Initially (before any activity is executed), this LTL expression evaluates to *true*. After executing A the LTL expression evaluates to *false* and this value remains *false* until B is executed. This illustrates that a constraint may be temporarily violated. However, the goal is to end the service execution in a state where all constraints evaluate to *true*.

To create constraints in DecSerFlow, we use *constraint templates*. Each constraint template consists of a formula written in LTL and a graphical representation of the formula. An example is the "response constraint" which is denoted by a special arc connecting two activities A and B. The semantics of such an arc connecting A and B are given by the LTL expression $\Box(A \longrightarrow \Diamond B)$, i.e., any execution of A is eventually followed by B. We have developed a starting set of constraint templates and we will use these templates to create a DecSerFlow model for the electronic bookstore example. This set of templates is inspired by a collection of specification patterns for model checking and other finite-state verification tools [32]. Constraint templates define various types of dependencies between activities at an abstract level. Once defined, a template can be reused to specify constraints between activities in various DecSerFlow models. It is fairly easy to change, remove, and add templates, which makes DecSerFlow an "open language" that can evolve and be extended according to the demands from different domains. There are three groups of templates: (1) "existence," (2) "relation," and (3) "negation" templates. Because a template assigns a graphical representation to an LTL formula, we will refer to such a template as a formula.

Before giving an overview of the initial set of formulas and their notation, we give a small example explaining the basic idea. Figure 2.5 shows a DecSerFlow model consisting of four activities: A, B, C, and D. Each activity is tagged with a constraint describing the number of times the activity should

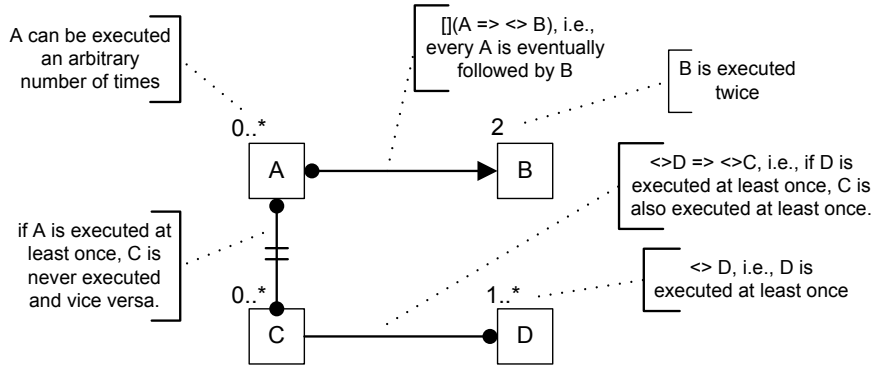


Fig. 2.5. A DecSerFlow model showing some example notations. (Note that the temporal operators $\Diamond$ and $\Box$ are denoted as <> and [ ])

be executed, these are the so-called "existence formulas." The arc between A and B is an example of a "relation formula" and corresponds to the LTL expression discussed before: $\Box(A \longrightarrow \Diamond B)$. The connection between C and D denotes another relation formula: $\Diamond D \longrightarrow \Diamond C$, i.e., if D is executed at least once, C is also executed at least once. The connection between A and C denotes a "negation formula" (the LTL expression $\Diamond(A) \Leftrightarrow \neg(\Diamond(B))$ is not shown in diagram but will be explained later). Note that it is not easy to provide a classical procedural model (e.g., a Petri net) that allows for all behavior modeled in Fig. 2.5.

Existence Formulas

Figure 2.6 shows the so-called "existence formulas". These formulas define the cardinality of an activity. For example, the first formula is called *existence* and its visualization is shown (i.e., the annotation "1..$*$" above the activity). This indicates that A is executed at least once. Formulas *existence2*, *existence3*, and $existence_N$ all specify a lower bound for the number of occurrences of A. It is also possible to specify an upper bound for the number of occurrences of A. Formulas *absence*, *absence2*, *absence3*, and $absence_N$ are also visualized by showing the range, e.g., "0...N" for the requirement $absence_{N+1}$. Similarly, it is possible to specify the exact number of occurrences as shown in Fig. 2.6, e.g., constraint $exactly_N(A : activity)$ is denoted by an N above the activity and specifies that A should be executed exactly N times.

Table 2.2 provides the semantics for each of the notations shown in Fig. 2.6, i.e., each formula is expressed in terms of an LTL expression. Formula $existence(A : activity)$ is defined as $\Diamond(A)$, i.e., A has to hold eventually which implies that in any full execution of the process A occurs at least once. Formula $existence_N(A : activity)$ shows how it is possible to express a lower bound N for the number of occurrences of A in a recursive manner, i.e., $existence_N(A) = \Diamond(A \wedge \bigcirc(existence_{N-1}(A)))$. Formula $absence_N(A : activity)$ can be defined

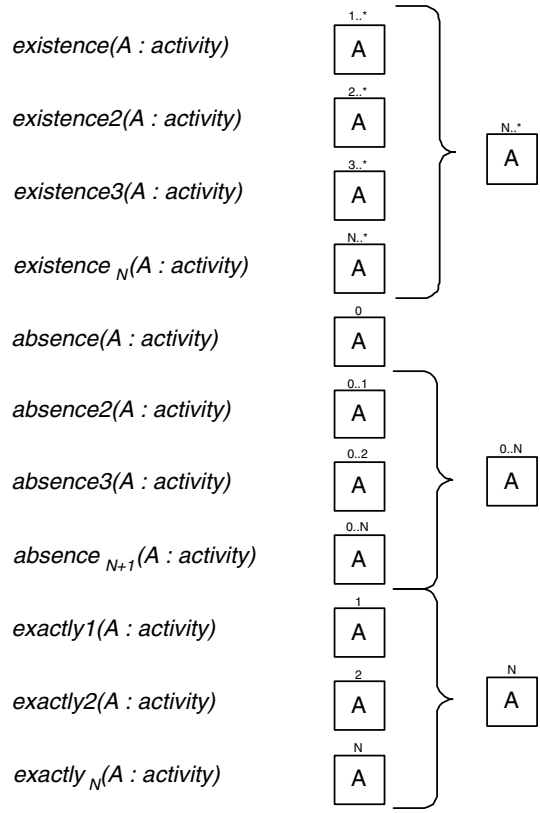


Fig. 2.6. Notations for the "existence formulas"

as the inverse of $existence_N(A)$. Together they can be combined to express that for any full execution, A should be executed a prespecified number N, i.e., $exactly_N(A) = existence_N(A) \ \wedge \ absence_{N+1}(A)$.

Relation Formulas

Figure 2.7 shows the so-called "relations formulas." While an "existence formula" describes the cardinality of one activity, a "relation formula" defines relation(s) (dependencies) between multiple activities. Figure 2.7 shows only binary relationships (i.e., between two activities); however, in DecSerFlow there are also notations involving generalizations to three or more activities, e.g., to model an OR-split. For simplicity, however, we first focus on the binary relationships shown in Fig. 2.7. All relation formulas have activities A and B as parameters and these activities are also shown in the graphical representation. The line between the two activities in the graphical representation is unique for the formula, and reflects the semantics of the relation. The *existence_response* formula specifies that if activity A is executed, activity

Table 2.2. Existence formulas

name of formula	LTL expression
$existence(A : activity)$	$\Diamond(A)$
$existence2(A : activity)$	$\Diamond(A \wedge \bigcirc(existence(A)))$
$existence3(A : activity)$	$\Diamond(A \wedge \bigcirc(existence2(A)))$
...	...
$existence_N(A : activity)$	$\Diamond(A \wedge \bigcirc(existence_{N-1}(A)))$
$absence(A : activity)$	$\Box(\neg A)$
$absence2(A : activity)$	$\neg existence2(A)$
$absence3(A : activity)$	$\neg existence3(A)$
...	...
$absence_N(A : activity)$	$\neg existence_N(A)$
$exactly1(A : activity)$	$existence(A) \wedge absence2(A)$
$exactly2(A : activity)$	$existence2(A) \wedge absence3(A)$
...	...
$exactly_N(A : activity)$	$existence_N(A) \wedge absence_{N+1}(A)$

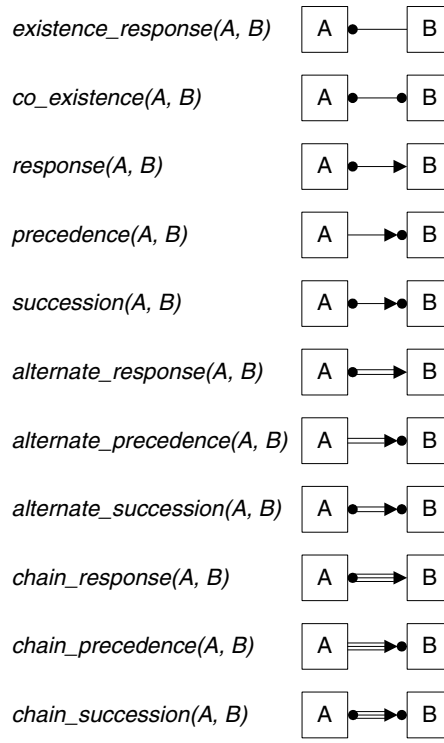


Fig. 2.7. Notations for the "relation formulas"

B also has to be executed (at any time, i.e., either before or after activity *A* is executed). According to the *co-existence* formula, if one of the activities *A* or *B* is executed, the other one has to be executed also. While the first two formulas do not consider the order of activities, formulas *response*, *precedence*, and *succession* do consider the ordering of activities. Formula *response* requires that every time activity *A* executes, activity *B* has to be executed after it. Note that this is a very relaxed relation of response, because *B* does not have to execute straight after *A*, and another *A* can be executed between the first *A* and the subsequent *B*. For example, the execution sequence *[B,A,A,A,C,B]* satisfies the formula *response*. The formula *precedence* requires that activity *B* is preceded by activity *A*, i.e., it specifies that if activity *B* was executed, it could not have been executed until activity *A* was executed. According to this formula, the execution sequence *[A,C,B,B,A]* is correct. The combination of the *response* and *precedence* formulas defines a bi-directional execution order of two activities and is called *succession*. In this formula, both *response* and *precedence* relations have to hold between the activities *A* and *B*. Thus, this formula specifies that every activity *A* has to be followed by an activity *B* and there has to be an activity *A* before every activity *B*. For example, the execution sequence *[A,C,A,B,B]* satisfies the *succession* formula.

Formulas *alternate_response*, *alternate_precedence*, and *alternate_succession* strengthen the *response*, *precedence*, and *succession* formulas, respectively. If activity *B* is *alternate response* of activity *A*, then after the execution of an activity *A* activity *B* has to be executed and between the execution of each two activities *A* at least one activity *B* has to be executed. In other words, after activity *A* there must be an activity *B*, and before that activity *B* there cannot be another activity *A*. The execution sequence *[B,A,C,B,A,B]* satisfies the *alternate response*. Similarly, in the *alternate precedence* every instance of activity *B* has to be preceded by an instance of activity *A* and the next instance of activity *B* cannot be executed before the next instance of activity *A* is executed. According to the *alternate_precedence*, the execution sequence *[A,C,B,A,B,A]* is correct. The *alternate_succession* is a combination of the *alternate_response* and *alternate_precedence* and the sequence *[A,C,B,A,B,A,B]* would satisfy this formula.

Even more strict ordering relations are specified by the last three constraints shown in Fig. 2.7: *chain_response*, *chain_precedence*, and *chain_succession*. These require that the executions of the two activities (*A* and *B*) are next to each other. According to the *chain_response* constraint the first activity after activity *A* has to be activity *B* and the execution *[B,A,B,C,A,B]* would be correct. The *chain_precedence* formula requires that the activity *A* is the activity directly preceding any *B* and, hence, the sequence *[A,B,C,A,B,A]* is correct. Since the *chain_succession* formula is the combination of the *chain_response* and *chain_precedence* formulas, it requires that activities *A* and *B* are always executed next to each other. The execution sequence *[A,B,C,A,B,A,B]* is correct with respect to this formula.

Table 2.3. Relation formulas

name of formula	LTL expression
$existence\_response(A : activity, B : activity)$	$\Diamond(A) \Rightarrow \Diamond(B)$
$co\_existence(A : activity, B : activity)$	$\Diamond(A) \Leftrightarrow \Diamond(B)$
$response(A : activity, B : activity)$	$\Box(A \Rightarrow \Diamond(B))$
$precedence(A : activity, B : activity)$	$\Diamond(B) \Rightarrow ((\neg B) \sqcup A)$
$succession(A : activity, B : activity)$	$response(A, B) \wedge precedence(A, B)$
$alternate\_response(A : activity, B : activity)$	$\Box(A \Rightarrow \bigcirc((\neg A) \sqcup B))$
$alternate\_precedence(A : activity, B : activity)$	$precedence(A, B) \wedge \Box(B \Rightarrow \bigcirc(precedence(A, B)))$
$alternate\_succession(A : activity, B : activity)$	$alternate\_response(A, B) \wedge alternate\_precedence(A, B)$
$chain\_response(A : activity, B : activity)$	$\Box(A \Rightarrow \bigcirc(B))$
$chain\_precedence(A : activity, B : activity)$	$\Box(\bigcirc(B) \Rightarrow A)$
$chain\_succession(A : activity, B : activity)$	$\Box(A \Leftrightarrow \bigcirc(B))$

Table 2.3 shows the formalization of the "relations formulas" depicted in Fig. 2.7. $existence\_response(A, B)$ is specified by $\Diamond(A) \Rightarrow \Diamond(B)$ indicating that some occurrence of A should always imply an occurrence of B either before or after A. $co\_existence(A, B)$ means that the existence of one implies the existence of the other and vice versa, i.e., $\Diamond(A) \Leftrightarrow \Diamond(B)$. $response(A, B)$ is defined as $\Box(A \Rightarrow \Diamond(B))$. This means that at any point in time where activity A occurs there should eventually be an occurrence of B. $precedence(A, B)$ is similar to response but now looking backward, i.e., if B occurs at all, then there should be no occurrence of B before the first occurrence of A. This is formalized as $\Diamond(B) \Rightarrow ((\neg B) \sqcup A)$. Note that we use the $\sqcup$ (until) operator here: $(\neg B) \sqcup A$ means that A holds (i.e., occurs) at the current state or at some future state, and $\neg B$ has to hold until A holds. When A holds $\neg B$ does not have to hold any more (i.e., B may occur). $succession(A, B)$ is defined by combining both into $response(A, B) \wedge precedence(A, B)$. $alternate\_response(A, B)$ is defined as $\Box(A \Rightarrow \bigcirc((\neg A) \sqcup B))$, i.e., any occurrence of A implies that in the next state and onward no A may occur until a B occurs. In other words, after activity A there must be an activity B, and before that activity B occurs there cannot be another activity A. $alternate\_precedence(A, B)$ is a bit more complicated: $\Box((B \wedge \bigcirc(\Diamond(B))) \Rightarrow \bigcirc(A \sqcup B))$. This implies that at any point in time where B occurs and at least one other occurrence of B follows, an A should occur before the second occurrence of B. $alternate\_succession(A, B)$ combines both into $alternate\_response(A, B) \wedge alternate\_precedence(A, B)$. $chain\_response(A, B)$ is defined as $\Box(A \Rightarrow \bigcirc(B))$ indicating that any occurrence of A should be directly followed by B. $chain\_precedence(A, B)$ is the logical counterpart: $\Box(\bigcirc(B) \Rightarrow A)$. $chain\_succession(A, B)$ is defined as $\Box(A \Leftrightarrow \bigcirc(B))$ and specifies that any occurrence of A should be directly followed by B and any occurrence of B should be directly preceded by A.

Negation Formulas

Figure 2.8 shows the "negation formulas," which are the negated versions of the "relation formulas." (Ignore the grouping of constraints on the right-hand side of Fig. 2.8 for the moment. Later, we will show that the eight constraints can be reduced to three equivalence classes.) The first two formulas negate the *existence_response* and *co_existence* formulas. The *neg_existence_response* formula specifies that if activity A is executed activity then B must never be executed (not before nor after activity A). The *neg_co_existence* formula applies *neg_existence_response* from A to B and from B to A. *It is important to note that the term* "negation" *should not be interpreted as the* "logical negation," e.g., if activity A never occurs, then both *existence_response(A,B)* and *neg_existence_response(A,B)* hold (i.e., one does not exclude the other). The *neg_response* formula specifies that after the execution of activity A, activity B cannot be executed any more. According to the formula *neg_precedence*, activity B cannot be preceded by activity A. The last three formulas are negations of formulas *chain_response*, *chain_precedence*, and *chain_succession*. *neg_chain_response* specifies that A should never be followed directly by B. *neg_chain_precedence* specifies that B should never be preceded directly by A. *neg_chain_succession* combines both *neg_chain_response* and *neg_chain_precedence*. Note that Fig. 2.8 does not show "negation formulas" for the alternating variants of response, precedence, and succession. The reason is that there is no straightforward and intuitive interpretation of the converse of an alternating response, precedence, or succession.

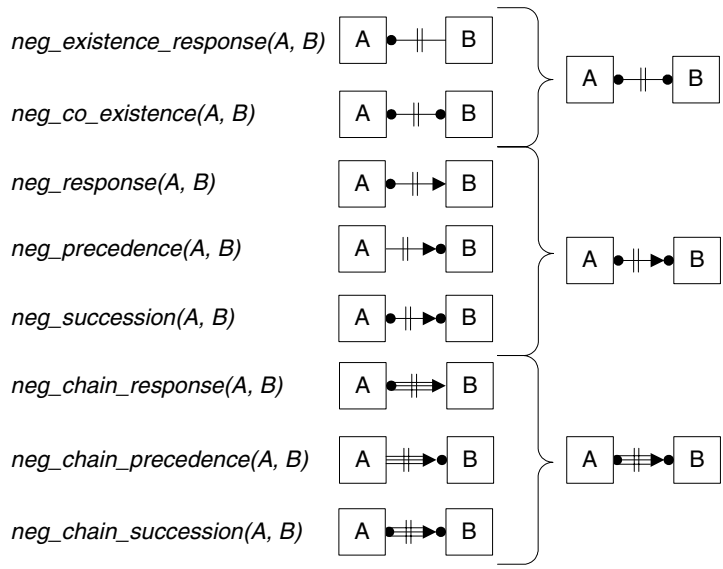


Fig. 2.8. Notations for the "negations formulas"

Table 2.4. Negation formulas (formulas grouped together are equivalent)

name of formula	LTL expression
$neg\_existence\_response(A : activity, B : activity)$	$\Diamond(A) \Rightarrow \neg(\Diamond(B))$
$neg\_co\_existence(A : activity, B : activity)$	$neg\_existence\_response(A, B) \wedge neg\_existence\_response(B, A)$
$neg\_response(A : activity, B : activity)$	$\Box(A \Rightarrow \neg(\Diamond(B)))$
$neg\_precedence(A : activity, B : activity)$	$\Box(\Diamond(B) \Rightarrow (\neg A))$
$neg\_succession(A : activity, B : activity)$	$neg\_response(A, B) \wedge neg\_precedence(A, B)$
$neg\_chain\_response(A : activity, B : activity)$	$\Box(A \Rightarrow \bigcirc(\neg(B)))$
$neg\_chain\_precedence(A : activity, B : activity)$	$\Box(\bigcirc(B) \Rightarrow \neg(A))$
$neg\_chain\_succession(A : activity, B : activity)$	$neg\_chain\_response(A, B) \wedge neg\_chain\_precedence(A, B)$

Table 2.4 shows the LTL expressions of the notations shown in Fig. 2.8. Table 2.4 also shows that some of the notions are equivalent, i.e., *neg_co_existence* and *neg_existence_response* are equivalent and similarly the next two pairs of three formulas are equivalent. Note that a similar grouping is shown in Fig. 2.8 where a single representation for each group is suggested. $neg\_existence\_response(A, B)$ is defined as $\Diamond(A) \Rightarrow \neg(\Diamond(B))$. However, since the ordering does not matter, $neg\_existence\_response(A, B) = neg\_existence\_response(A, B)$ and hence coincides with $neg\_co\_existence(A, B)$. $neg\_response(A, B)$ is defined as $\Box(A \Rightarrow \neg(\Diamond(B)))$, i.e., after any occurrence of A, B may never happen (or formulated alternatively: any occurrence of B should take place before the first A). $neg\_precedence(A, B)$ is defined as $\Box(\Diamond(B) \Rightarrow (\neg A))$, i.e., if B occurs in some future state, then A cannot occur in the current state. It is easy to see that $neg\_precedence(A, B) = neg\_response(A, B)$ because both state that no B should take place after the first A (if any). Since $neg\_succession(A, B)$ combines both, also $neg\_succession(A, B) = neg\_response(A, B)$. The last three formulas are negations of formulas *chain_response*, *chain_precedence*, and *chain_succession*. It is easy to see that they are equivalent, $neg\_chain\_response(A, B) = neg\_chain\_precedence(A,B) = neg\_chain\_succession(A, B)$.

Figures 2.7 and 2.8 and the corresponding formalizations show only binary relationships. However, these can easily be extended to deal with more activities. Consider, e.g., the *response* relationship, i.e., $response(A, B) = \Box(A \Rightarrow \Diamond(B))$. This can easily be extended to $response(A, B, C) = \Box(A \Rightarrow (\Diamond(B) \vee \Diamond(C))$, i.e., every occurrence of A is eventually followed by an occurrence of B or C. This can also be extended to a choice following A of N alternatives, i.e., $response(A, A_1, A_2, \ldots, A_N) = \Box(A \Rightarrow (\Diamond(A_1) \vee \Diamond(A_2) \vee \ldots \vee \Diamond(A_N)))$. Many of the other formulas can be generalized in a similar fashion and represented graphically in an intuitive manner. For example, $response(A, B, C)$, i.e., A is eventually followed by an occurrence of B or C, is depicted by multiple

arcs that start from the same dot. Similarly, it is possible to have a precedence constraint where different arrows end in the same dot indicating that at least one of the preceding activities should occur before the subsequent activity is executed.

DecSerFlow is an extendible language, i.e., designers can add their own graphical notations and provide their semantics in terms of LTL. For example, one can add constraints similar to the control-flow dependencies in classical languages such as Petri nets, EPCs, etc. and draw diagrams similar to the diagrams provided by these languages. However, the aim is to have a relatively small set of intuitive notations. In this chapter we show only a core set. Figure 2.9 assists in reading diagrams using this core notation. When extending the language with new constraints, it is important to use a set of drawing conventions as shown in Fig. 2.9. For example, a dot connected to some activity A means that "A occurs" and is always associated to some kind of connection, a line without some arrow means "occurs at some point in time," an arrow implies some ordering relation, two short vertical lines depict a negation, etc. Note that Fig. 2.9 also shows the $response(A, A_1, A_2, \ldots, A_N)$ constraint described earlier, i.e., A is followed by at least one of its successors.

2.3.3 The amazon.com Example in DecSerFlow

In this subsection, we revisit the amazon.com example to show how DecSerFlow language can be used to model services. For this purpose, we will model

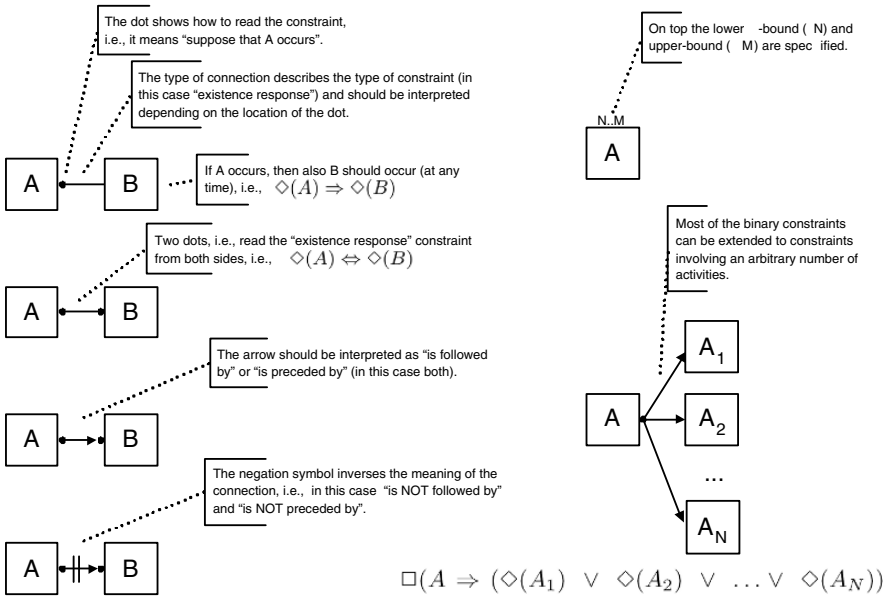


Fig. 2.9. Explanation of the graphical notation

the customer service using existence, relation, and negation formulas. In this way, we will use the defined templates for formulas, apply them to activities from our example and thus create real constraints in our DecSerFlow model. In addition to this model of a single service, we will also show how the communication between services can be presented with DecSerFlow by modeling the communication of the customer service with other services. We start by removing all arcs and places from the example model. This results in an initial DecSerFlow model populated only by unconnected activities. Next, we create necessary constraints for the *customer*. Adding constraints to the rest of the model is straightforward and easy but not necessary for illustrating the DecSerFlow language.

Figure 2.10 shows the new model with DecSerFlow constraints for the *customer*. We added *existence* constraints for all activities which can be seen as cardinality specifications above activities. Activity *place_c_order* has to be executed exactly one time. Activities *rec_acc* and *rec_decl* can be executed zero or one time, depending on the reply of the bookstore. Similarly, activities *rec_book*, *rec_bill*, and *pay* can be executed at most one time.

Every occurrence of *place_c_order* is eventually followed by at least one occurrence of *rec_acc* or *rec_decl*, as indicated by the non-binary relationship also shown in Fig. 2.9. However, it is possible that both activities are executed, and to prevent this we add the *neg_co_existence* constraint between activities *rec_acc* and *rec_decl*. So far, we have managed to make sure that after activity *place_c_order* one of the activities *rec_acc* and *rec_decl* will execute in the service. One problem remains to be solved – we have to specify that none of the activities *rec_acc* and *rec_decl* can be executed before activity *place_c_order*. We achieve this by creating two *precedence* constraints: (1) the one between the activities *place_c_order* and *rec_acc*, making sure that activity *rec_acc* can be executed only after activity *place_c_order* was executed and (2) the one between activities *place_c_order* and *rec_decl*, making sure that activity *rec_decl* can be executed only after activity *place_c_order* was executed. It is important to note that the constraints related to *place_c_order*, *rec_acc*, and *rec_decl* together form a "classical choice". It may seem rather clumsy that four constraints are needed to model a simple choice. However, (1) the four constraints can be merged into a single notation and LTL formula that can be re-used in other diagrams and (2) it is a nice illustration of how procedural languages like Petri nets and BPEL tend to overspecify things. In fact, in a classical language one would not only implicitly specify four elementary constraints but would typically need to specify the data conditions. In DecSerFlow one can add these conditions, but one does not need to do so, i.e., one can drop any of the four constraints involving *place_c_order*, *rec_acc*, and *rec_decl* and still interpret the resulting set of constraints in a meaningful way.

The next decision to be made is the dependency between the activities *rec_acc* and *rec_book*. In the old model, we had a clear sequence between these two activities. However, due to some problems or errors in the bookstore it might happen that, although the order was accepted (activity *rec_acc* is

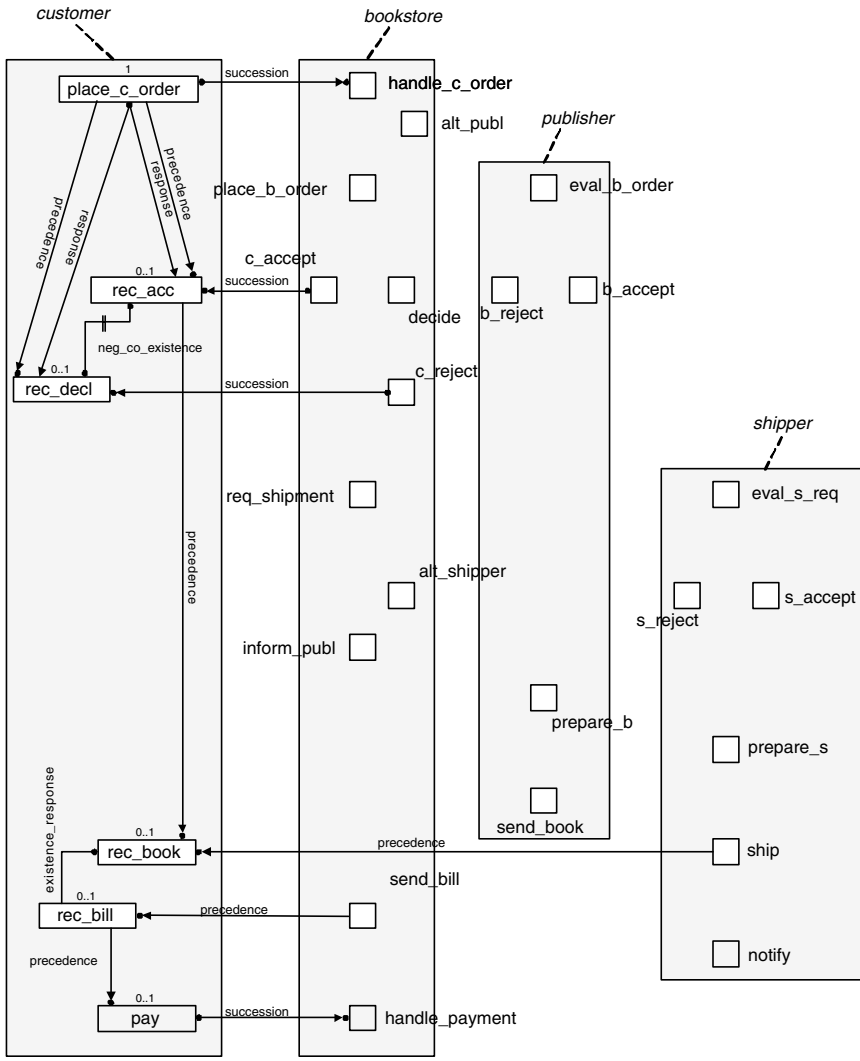


Fig. 2.10. DecSerFlow model

executed), the book does not arrive (activity *rec_book* is not executed). However, we assume that the book will not arrive before the order was accepted. The constraint *precedence* between the activities *rec_acc* and *rec_book* specifies that activity *rec_book* can be executed only after activity *rec_acc* is executed. The old model specified that the bill arrives after the book. This may not be always true. Since the bill and the book are shipped by different services through different channels, the order of their arrival might vary. For example, it might happen that the shipper who sends the book is closer to the location

of the customer and the bookstore is on another continent, or the other way around. In the first scenario the book will arrive before the bill, and in the second one the bill will arrive before the book. Therefore, we choose not to create an *ordering* constraint between the activities *rec_book* and *rec_bill*. Even more, our DecSerFlow model accepts the error when the bill arrives even without the book being sent. This could happen in the case of an error in the *bookstore* when a declined order was archived as accepted, and the bill was sent without the shipment of the book. However, we assume that every bookstore that delivers a book, also sends a bill for the book. We specify this with the *existence_response* constraint between the *rec_book* activity and the *rec_bill* activity. This constraint forces that if activity *rec_book* is executed, then activity *rec_bill* must have been executed before or will be executed after activity *rec_book*. Thus, if the execution of activity *rec_book* exists, then the execution of activity *rec_bill* also exists. The constraint *precedence* between the activities *rec_bill* and *pay* means that the customer will pay only after the bill is received. However, after the bill is received the customer does not necessarily pay, like in the old model. It might happen that the received book was not the one that was ordered or it was damaged. In these cases, the customer can decide not to pay the bill.

Besides for the modeling of a single service, DecSerFlow language can as well be used to model the communication between services. In Fig. 2.10, we can see how constraints specify the communication of the customer with the bookstore and the shipper. First, the *succession* constraint between activities *place_c_order* and *handle_c_order* specifies that after activity *place_c_order* activity *handle_c_order* has to be executed, and that activity *handle_c_order* can be executed only after activity *place_c_order*. This means that every order of a customer will be handled in the bookstore, but the bookstore will handle the order only after it is placed. The same holds (constraint *succession*) for the pairs of activities (*c_accept*, *rec_acc*), (*c_reject*, *rec_decl*), and (*pay*, *handle_payment*). The relations between the pairs of activities (*ship*, *rec_book*) and (*send_bill*, *rec_bill*) are more *relaxed* than the previous relations. These two relations are not *succession*, but *precedence*. We can only specify that the book will be received after it is sent, but we cannot claim that the book that was sent will indeed be received. It might happen that the shipment is lost or destroyed before the customer receives the book. The same holds for the bill. Because of this, we create the two *precedence* constraints. The first precedence constraint is between activity *ship* and *rec_book* to specify that activity *rec_book* can be executed only after activity *ship* was executed. The second one is between the activities *send_bill* and *rec_bill*, according to which activity *rec_bill* can be executed only after activity *send_bill* is executed.

Figure 2.10 shows how DecSerFlow language can be used to specify services. While the old Petri-net model specified the strict sequential relations between activities, with DecSerFlow we were able to create many different relations between the activities in a more natural way. For the illustration, we developed constraints only for the *customer* service and its communication

with other services, but developing of the rest of the model is as easy and straightforward.

2.3.4 Mapping DecSerFlow Onto Automata

DecSerFlow can be used in many different ways. Like abstract BPEL it can be used to specify services but now in a more declarative manner. However, like executable BPEL we can also use it as an execution language. The DecSerFlow language can be used as an execution language because it is based on LTL expressions. Every constraint in a DecSerFlow model has both a graphical representation and a corresponding parameterized LTL formula. The graphical notation enables a user-friendly interface and masks the underlying formula. The formula, written in LTL, captures the semantics of the constraint. The core of a DecSerFlow model consists of a set of activities and a number of LTL expressions that should all evaluate to *true* at the end of the model execution.

Every LTL formula can be translated into an automaton [26]. Algorithms for translating LTL expressions into automata are given in [40, 92]. The possibility to translate an LTL expression into an automaton and the algorithms to do so have been extensively used in the field of *model checking* [26]. Moreover, the initial purpose for developing such algorithms comes from the need to, given a model, check if certain properties hold in the model. The SPIN tool [50] can be used for the simulation and exhaustive formal verification of systems, and as a proof approximation system. SPIN uses an automata theoretic approach for the automatic verification of systems [86]. To use SPIN, the system first has to be specified in the verification modeling language Promela (PROcess MEta LAnguage) [50]. SPIN can verify the correctness of requirements, which are written as LTL formulas, in a Promela model using the algorithms presented in [40, 48, 49, 51, 52, 86, 77, 91]. When checking the correctness of an LTL formula, SPIN first creates an automaton for the *negation* of the formula. If the intersection of this automaton and the system model automaton is empty, the model is correct with respect to the requirement described in LTL. When the system model does not satisfy the LTL formula, the intersection of the model and the automaton for the negated formula will not be empty, i.e., this intersection is a *counterexample* that shows how the formula is violated. The approach based on the negation of the formula is quicker, because the SPIN runs verification until the first counterexample is found. In the case of the formula itself, the verifier would have to check all possible scenarios to prove that a counterexample does not exist.

Unlike SPIN, which generates an automaton for the negation of the formula, we can execute a DecSerFlow model by constructing an automaton for the *formula itself*. We will use a simple DecSerFlow model to show how processes can be executed by translating LTL formulas into automata. Figure 2.11 shows a DecSerFlow model with three activities: *curse*, *pray*, and *bless*. The only constraint in the model is the *response* constraint between activity *curse* and activity *pray*, i.e., $response(curse, pray) = \Box(curse \Rightarrow \Diamond(pray))$. This

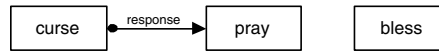


Fig. 2.11. A simple model in DecSerFlow

constraint specifies that if a person curses, she/he should eventually pray after this. Note that there is no restriction on the execution of the activities *pray* and *bless*. There are no existence constraints in this model, because all three activities can be executed an arbitrary number of times.

Using the example depicted in Fig. 2.11, we briefly show the mapping of LTL formulas onto automata [40], which is used for execution of DecSerFlow models. Automata consists of states and transitions. By executing activities of DecSerFlow model, we fire transitions and thus change state of the related automaton. Automaton can be in an accepting or not-accepting state. If the automaton is in an accepting state after executing a certain trace (of DecSerFlow activities), the trace fulfills the related LTL formula. If the automaton is not in an accepting state after executing a certain trace, the trace violates the related LTL formula. Automata created by the algorithm presented in [40] deal with infinite traces and cannot be used for execution of finite traces like DecSerFlow traces. Therefore, a variation of this algorithm that enables work with finite traces is used [41]. A more detailed introduction to automata theory and the creation of Büchi automata from LTL formulas is out of scope of this article and we refer the interested reader to [26, 40, 41, 48].

Figure 2.12 shows a graph representation of the automaton which is generated for the *response* constraint [40].[2] Automaton states are represented as nodes, and transitions as edges. An initial state is represented by an incoming edge with no source node. An accepting state is represented as a node with a double-lined border. The automaton in Fig. 2.12 has two states: *p1* and *p2*. State *p1* is both the initial and accepting state. Note that such automaton can also be generated for a DecSerFlow model with multiple constraints, i.e., for more than one LTL formula, by constructing one *big* LTL formula as a *conjunction* of each of the constraints.

$$response(curse, pray) = \Box(curse \Rightarrow \Diamond(pray))$$

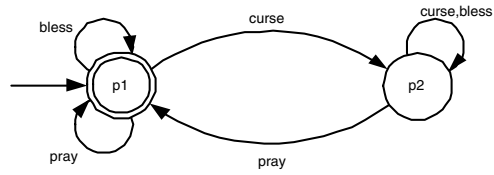


Fig. 2.12. Automaton for the formula response

[2] Note that the generated automaton is a non-deterministic automaton. For reasons of simplicity, we use a deterministic automaton with the same results.

Note that for illustration purposes, we only show a simplified automaton in Fig. 2.12. Any LTL expression is, actually, translated into a automaton, i.e, a *non-deterministic* automaton for *infinite words*. An automaton is deterministic if in each state there is exactly one transition for each possible input. In case of a deterministic automaton, we can simply change the state of the automata when executing an activity. To check the correctness of the execution, we check if the current state is an accepting one. In non-deterministic automata, there can be multiple transitions from a given state for a given possible input. In case of a DecSerFlow model, the fact that we are dealing with non-deterministic automata means that executing an activity might transfer an automaton to more that one next state—a *set* of possible states. To check the correctness of the execution, we need to check if the current *set* of possible states contains at least one accepting state. Another issue when executing automata for DecSerFlow models is the fact that we assume that every execution will be completed at some point of time, i.e., an execution of a DecSerFlow model is a *finite* one. The original algorithm for creating automata from LTL expressions generates automata for infinite words, i.e., for infinite executions [40]. That creates problems because the criteria for deciding which states are accepting are different for finite and infinite words. Therefore, we use a modified version of the original algorithm [41], which was constructed for verification of finite software traces. We use the Java PathExplorer (JPAX), a runtime verification tool, as a basis [41]. The algorithm in JPAX assumes that the system will start the execution, and does not consider empty traces. To allow an empty execution of a DecSerFlow model, we add an invisible activity *init* and a constraint *initiate* that specifies that activity *init* has to be executed as the first activity in the model. We automatically execute activity *init* at the beginning of the enactment of a DecSerFlow model. Another small complication is that in the JPAX implementation of [41], the $\bigcirc$ operator is slightly weaker (if there is no next step, $\bigcirc F$ evaluates to true by definition). This can be modified easily by redefining $\bigcirc F$ to $(\bigcirc F \wedge \Diamond F)$.

The mapping for LTL constraints onto automata allows for the guidance of people, e.g., it is possible to show whether a constraint is in an accepting state or not. Moreover, if the automaton of a constraint is not in an accepting state, indicate whether it is still possible to reach an accepting state. To do this, we can color the constraints *green* (in accepting state), *yellow* (accepting state can still be reached), or *red* (accepting state cannot be reached anymore). Using the automaton, some engine could even enforce a constraint, i.e., the automaton could be used to drive a classical workflow engine [7].

2.3.5 Using DecSerFlow to Relate Global and Local Models

In the first part of the chapter, we distinguished between *global* and *local* models. In the global model, interactions are described from the viewpoint of an external observer who oversees all interactions between all services. Local models are used to specify, implement, or configure particular services.

Clearly, both types of models can be represented using DecSerFlow. Moreover, as just shown, it is possible to construct an automaton to enact a DecSerFlow specification. This seems particularly relevant for local models. As we will see in the next section, both global and local models can be used for monitoring services. For example, given a DecSerFlow specification we can also check whether each party involved in a choreography actually sticks to the rules agreed upon. The ProM framework offers the so-called "LTL-checker" to support this (cf. Sect. 2.4.2). However, before focusing on the monitoring of service flows, we briefly discuss the relevance of DecSerFlow in relating global and local models.

Using DecSerFlow both global and local models can be mapped onto LTL expressions and automata. This allows for a wide range of model checking approaches. For example, it is possible to check if the constraints in the local model are satisfied by the global model and vice versa. Note that the set of activities in both models does not need to be the same. However, given the logical nature of DecSerFlow this is not a problem. Also, note that the different notions of inheritance of dynamic behavior can be used in this context [2] (e.g., map activities onto τ actions). The only constraints that seem problematic in this respect are chained relation formulas, i.e., *chain_response*, *chain_precedence*, and *chain_succession*. These use the "nexttime" ($\bigcirc F$) operator whose interpretation depends on the context, i.e., from a global perspective an activity in one service may be followed by an activity in another service thus violating some "nexttime" constraint. Nevertheless, it seems that the LTL foundation of DecSerFlow offers a solid basis for comparing global and local models and generating templates for local models from some partitioned global model.

2.4 Monitoring Service Flows

DecSerFlow can be used to create both local and global models. As shown in the previous section, these models can be used to specify a (part of some) service flow and to enact it. In this section, we show that DecSerFlow can also be used in the context of *monitoring service flows*.

In a service-oriented architecture, and also in classical enterprise systems, a variety of events (e.g., messages being sent and received) are being logged. This information can be used for *process mining* purposes, i.e., based on some event log some knowledge is extracted. In the context of service flows an obvious starting point is the interception of messages exchanged between the various services. For example, SOAP messages can be recorded using TCP Tunneling techniques [6] or, if middleware solutions such as IBM's Websphere are used, different events are logged in a structured manner [73]. Although possible, it is typically not easy to link events (e.g., SOAP messages) to process instances (cases) and activities. However, as pointed out by many researchers, the problem of correlating messages needs to be addressed anyway. Hence, in

the remainder, we assume that it is possible to obtain an event log where each event can be linked to some process instance and some activity identifier.

2.4.1 Classification of Process Mining

Assuming that we are able to monitor activities and/or messages being exchanged, a wide range of process mining techniques comes into reach. Before we focus on the relation between DecSerFlow and process mining, we provide a basic classification of process mining approaches. This classification is based on whether there is an a priori model (e.g., a DecSerFlow specification) and, if so, how it is used.

- *Discovery*: There is no a priori model, i.e., based on an event log some model is constructed. For example, using the α algorithm [15] a process model can be discovered based on low-level events. There exist many techniques to automatically construct process models (e.g., in terms of a Petri net) based on some event log [15, 17, 27, 28, 89]. Recently, process mining research also started to target the other perspectives (e.g., data, resources, time, etc.). For example, the technique described in [11] can be used to construct a social network.
- *Conformance*: There is an a priori model. This model is compared with the event log, and discrepancies between the log and the model are analyzed. For example, there may be a process model indicating that purchase orders of more than €1 million require two checks. Another example is the checking of the so-called "four-eyes" principle. Conformance checking may be used to detect deviations, to locate and explain these deviations, and to measure the severity of these deviations. An example is the conformance checker described in [79] which compares the event log with some a priori process model expressed in terms of a Petri net.
- *Extension*: There is an a priori model. This model is extended with a new aspect or perspective, i.e., the goal is not to check conformance but to enrich the model. An example is the extension of a process model with performance data, i.e., some a priori process model is used to project the bottlenecks on. Another example is the decision miner described in [80] which takes an a priori process model and analyzes every choice in the process model. For each choice the event log is consulted to see which information is typically available the moment the choice is made. Then classical data mining techniques are used to see which data elements influence the choice. As a result, a decision tree is generated for each choice in the process.

Figure 2.13 illustrates the classification just given in the context of DecSerFlow. The figure shows different web services together realizing a service flow. A DecSerFlow can be used to specify the whole service flow (global model) or individual services (local models). As shown in Fig. 2.13, we assume that we are able to record events which are stored on some event log. Given such

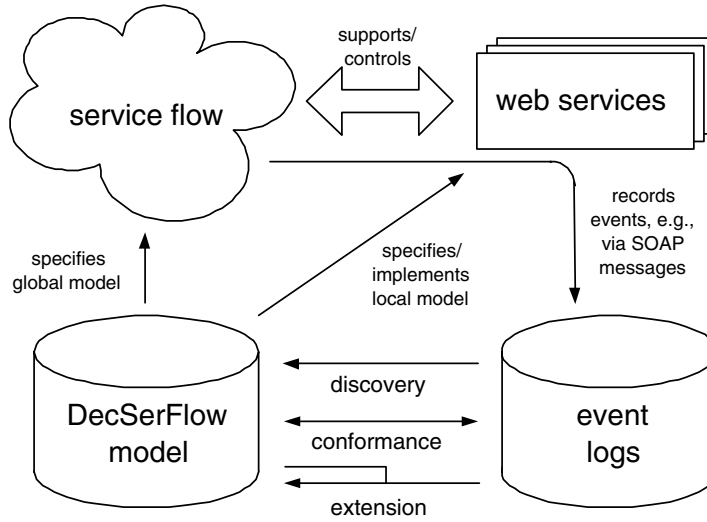


Fig. 2.13. Overview of the various process mining approaches related to DecSerFlow

an event log, the three types of process mining (discovery, conformance, and extension) become possible.

Discovery in the context of DecSerFlow would mean that, based on the event log, we discover a DecSerFlow model, i.e., by analyzing the log different constraints are discovered. For example, if an activity is always followed by another, this can be easily be deduced from the log. Currently, there exist many process discovery approaches [15, 17, 27, 28, 89]. Although none of them is tailored toward DecSerFlow, it is easy to modify these to yield a (partial) DecSerFlow model. Note that ordering relations discovered by the α algorithm [15] can easily be visualized in DecSerFlow.

Conformance checking requires an a priori DecSerFlow model, e.g., a global model showing the overall service flow. This model can easily be compared with the event logs, i.e., each constraint in the DecSerFlow specification is mapped onto an LTL expression and it is easy to check whether the LTL expression holds for a particular process instance. Hence it is possible to classify process instances into conforming or non-conforming for each constraint. This way it is possible to show where and how frequent deviations occur. Moreover, the (non-)conforming process instances can be investigated further using other process mining techniques, e.g., to discover the typical features of cases that deviate.

The third type of process mining also requires an a priori DecSerFlow model. However, now the model is extended with complementary information. For example, performance data are projected onto the DecSerFlow model or decision trees are generated for decision points in the process.

As suggested by Fig. 2.13, DecSerFlow can be used in combination with various process mining approaches. *It is important to note that the*

autonomous nature of services, the declarative style of modeling (avoiding any overspecification), and process mining fit well together. The autonomous nature of services allows services to operate relatively independently. In many cases it is not possible to enforce control. At best one can agree on a way of working (the global model) and hope that the other parties involved will operate as promised. However, since it is often not possible to control other services, one can only observe, detect deviations, and monitor performance.

In the remainder of this section, we discuss some of the features of ProM [29]: a process mining framework offering plug-ins for discovery, conformance, and extension.

2.4.2 Linking DecSerFlow to the ProM LTL Checker

The ProM framework [29] is an open-source infrastructure for process mining techniques. ProM is available as open source software (under the Common Public License, CPL) and can be downloaded from [75]. It has been applied to various real-life processes, ranging from administrative processes and healthcare processes to the logs of complex machines and service processes. ProM is plug-able, i.e., people can plug-in new pieces of functionality. Some of the plug-ins are related to model transformations and various forms of model analysis (e.g., verification of soundness, analysis of deadlocks, invariants, reductions, etc.). Most of the plug-ins, however, focus on a particular process mining technique. Currently, there are more than 100 plug-ins of which about half are mining and analysis plug-ins.

Starting point for ProM are event logs in MXML format. The MXML format is system independent and using ProMimport it is possible to extract logs from a wide variety of systems, i.e., systems based on products such as SAP, Peoplesoft, Staffware, FLOWer, WebSphere, YAWL, ADEPT, ARIS PPM, Caramba, InConcert, Oracle BPEL, Outlook, etc. and tailor-made systems. It is also possible to load and/or save a variety of models, e.g., EPCs (i.e., event-driven process chains in different formats, e.g., ARIS, ARIS PPM, EPML, and Visio), BPEL (e.g., Oracle BPEL, Websphere), YAWL, Petri nets (using different formats, e.g., PNML, TPN, etc.), CPNs (i.e., colored Petri nets as supported by CPN Tools), and Protos.

One of the more than 100 plug-ins offered by ProM is the so-called "LTL checker" [3]. The LTL checker offers an environment to provide parameters for predefined parameterized LTL expressions and check these expressions with respect to some event log in MXML format. For each process instance, it is determined whether the LTL expression holds or not, i.e., given an LTL expression all process instances are partitioned into two classes: conforming and non-conforming. We have predefined 60 typical properties one may want to verify using the LTL checker (e.g., the 4-eyes principle) [3]. These can be used without any knowledge of the LTL language. In addition the user can define new sets of properties. These properties may be application specific and may refer to data. Each property is specified in terms of an LTL expression. Formulas

may be parameterized, are reusable, and carry explanations in HTML format. This way both experts and novices may use the LTL checker.

Recall that each model element of the DecSerFlow is mapped onto an LTL expression. Therefore, it is possible to use the ProM LTL checker to assess the conformance of a DecSerFlow model in the context of a real log. All notations defined in Figs. 2.6, 2.7, and 2.8 map directly onto LTL expressions that can be stored and loaded into ProM. Currently, we do not yet provide a direct connection between the DecSerFlow editor and the ProM LTL checker. Hence, it is not yet possible to visualize violations on the DecSerFlow editor. However, it is clear that such integration is possible.

2.4.3 Other Process Mining Techniques in ProM

Clearly, the LTL checker is one of the most relevant plug-ins of ProM in the context of DecSerFlow. However, the LTL checker plug-in is only one of more than 100 plug-ins. In this subsection, we show some other plug-ins relevant to process mining of service flows. First, we show some plug-ins related to process discovery. Then, we show the ProM conformance checker that has been successfully used in the context of (BPEL) service flows.

The basic idea of process discovery is to derive a model from some event log. This model is typically a process model. However, there are also techniques to discover organization models, social networks, and more data-oriented models such as decision trees. To illustrate the idea of process mining consider the log shown in Table 2.5. Such a log could have been obtained by monitoring the SOAP messages the *shipper* service in Fig. 2.3 exchanges with it its environment. Note that we do not show the content of the message. Moreover, we do not show additional header information (e.g., information about sender and receiver).

Using process mining tools such as ProM, it is possible to discover a process model as shown in Fig. 2.14. The figure shows the result of three alternative process discovery algorithms: (1) the α miner shows the result in terms of a Petri net, (2) the multi-phase miner shows the result in terms of an EPC, and (3) the heuristics miner shows the result in terms of a heuristics net.[3] They are all able to discover the *shipper* service as specified in Fig. 2.3. Note that Fig. 2.14 shows the names of the messages rather than the activities because this is the information shown in Table 2.5. Note that the algorithms used in Fig. 2.14 can easily be modified to generate DecSerFlow models, i.e., constraints imposed by, e.g., a Petri net can be mapped onto DecSerFlow notations.

For process discovery, we do not assume that there is some a priori model, i.e., without any initial bias we try to find the actual process by analyzing some event log. However, in many applications there is some a priori model. For

[3] Note that ProM allows for the mapping from one format to the other if needed. Fig. 2.14 shows the native format of each of the three plug-ins.

Table 2.5. An event log

case identifier	activity identifier	time	data
order290166	s_request	2006-04-02T08:38:00	...
order090504	s_request	2006-04-03T12:33:00	...
order290166	s_confirm	2006-04-07T23:55:00	...
order261066	s_request	2006-04-15T06:43:00	...
order160598	s_request	2006-04-19T20:13:00	...
order290166	book_to_s	2006-05-10T07:31:00	...
order290166	book_to_c	2006-05-12T08:43:00	...
order160598	s_confirm	2006-05-20T07:01:00	...
order210201	s_request	2006-05-22T09:20:00	...
order261066	s_confirm	2006-06-08T10:29:00	...
order290166	notification	2006-06-13T14:44:00	...
order160598	book_to_s	2006-06-14T16:56:00	...
order261066	book_to_s	2006-07-08T18:01:00	...
order090504	s_decline	2006-07-12T09:00:00	...
order261066	book_to_c	2006-08-17T11:22:00	...
order210201	s_decline	2006-08-18T12:38:00	...
order160598	book_to_c	2006-08-25T20:42:00	...
order261066	notification	2006-09-27T09:51:00	...
order160598	notification	2006-09-30T10:09:00	...

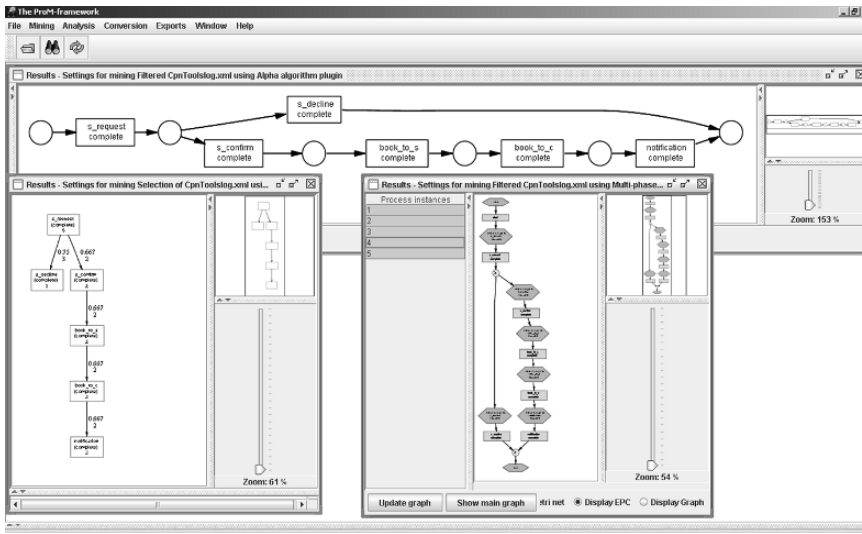


Fig. 2.14. The output of three process discovery algorithms supported by ProM when analyzing the event log shown in Table 2.5

example, we already showed that ProM's LTL checker can be used to check the conformance of a DecSerFlow model. However, ProM is not limited to DecSerFlow and can also be used to check the conformance of a specification in terms of abstract BPEL, EPC, or Petri nets. To illustrate this, assume that we add an additional process instance to Table 2.5 where the notification is sent before the book is shipped to the customer (i.e., in Fig. 2.3 activity *notify* takes place before activity *ship*).

If we assume there is some a priori model in terms of a Petri net, we can use the *conformance checker plug-in* of ProM. Figure 2.15 shows the result of this analysis (top-right corner). It shows that the *fitness* is 0.962 and also highlights the part of the model where the deviation occurs (the place connecting *ship*/*book_to_c* and *notify*/*notification*). An event log and Petri net "fit" if the Petri net can generate each trace in the log. In other words, the Petri net describing the choreography should be able to "parse" every event sequence observed by monitoring, e.g., SOAP messages. In [79] it is shown that it is possible to quantify fitness as a measure between 0 and 1. The intuitive meaning is that a fitness close to 1 means that all observed events can be explained by the model (in the example about 96 percent). However, the precise meaning is more involved since tokens can remain in the network and not all transactions in the model need to be logged [79].

Unfortunately, a good fitness does not only imply conformance, e.g., it is easy to construct Petri nets that are able to parse any event log (corresponding to a DecSerFlow model without any constraints, i.e., a model described by *true*). Although such Petri nets have a fitness of 1 they do

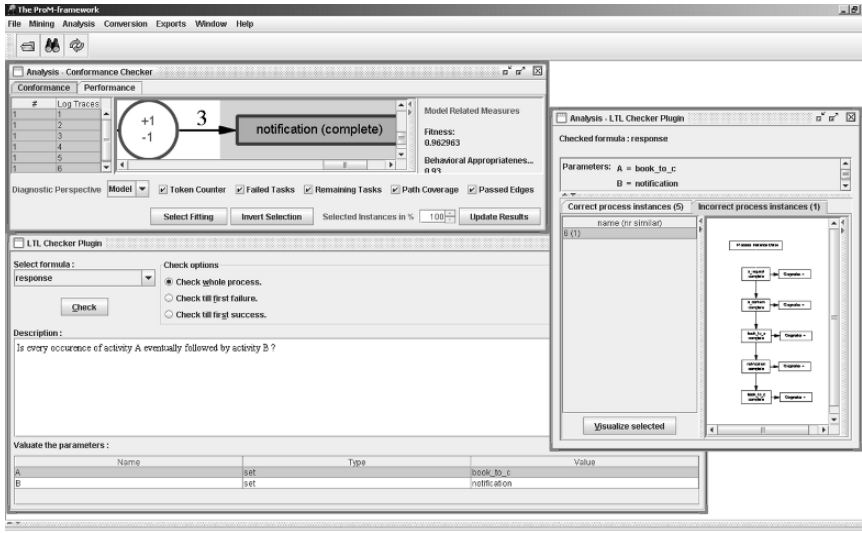


Fig. 2.15. Both the conformance checker plug-in and the LTL checker plug-in are able to detect the deviation

not provide meaningful information. Therefore, we use a second dimension: *appropriateness*. Appropriateness tries to capture the idea of *Occam's razor*, i.e., "one should not increase, beyond what is necessary, the number of entities required to explain anything." Appropriateness tries to answer the following question: "Does the model describe the observed process in a suitable way" and can it be evaluated from both a *structural* and a *behavioral* perspective? To explain the concept in more detail, it is important to note that there are two extreme models that have a fitness of 1. First of all, there is a model that starts with a choice and then has one path per process instance, i.e., the model simply enumerates all possibilities. This model is "overfitting" since it is simply another representation of the log, i.e., it does not allow for more sequences than those that were observed in the log. Therefore, it does not offer a better understanding than what can be obtained by just looking at the aggregated log. Secondly, there is the so-called "flower Petri net" [79] that can parse any log, i.e., there is one state in which all activities are enabled. This model is "underfitting" since it contains no information about the ordering of activities. In [79] it is shown that a "good" process model should somehow be minimal in structure to clearly reflect the described behavior, referred to as *structural appropriateness*, and minimal in behavior in order to represent as closely as possible what actually takes place, which will be called *behavioral appropriateness*. The ProM conformance checker supports both the notion of fitness and the various notions of appropriateness.

In [6] we have demonstrated that any (abstract) BPEL specification can automatically be mapped onto a Petri net that can be used for conformance checking using ProM's conformance checker.

Figure 2.15 also shows the LTL checker plug-in while checking the *response* property on *book_to_c* and *notification*. This check shows that indeed there is one process instance where activity *notify* takes place before activity *ship*. This example shows that it is possible to compare a DecSerFlow specification and an event log and to locate the deviations.

2.5 Related Work

Since the early 1990s, workflow technology has matured [39] and several textbooks have been published, e.g., [7, 30]. Most of the available systems use some proprietary process modeling language and, even if systems claim to support some "standard," there are often all kinds of system-specific extensions and limitations. Petri nets have been used not only for the modeling of workflows [7, 25, 30] but also for the orchestration of web services [65]. Like most proprietary languages and standards, Petri nets are highly procedural. This is the reason why we introduced the DecSerFlow language in this chapter.

Several attempts have been made to capture the behavior of BPEL [18] in some formal way. Some advocate the use of finite state machines [35], others

process algebras [34], and yet others abstract state machines [33] or Petri nets [71, 62, 83, 87]. (See [71] for a more detailed literature review.) For a detailed analysis of BPEL based on the workflow patterns [8], we refer to [90]. Few researchers have explored the other direction, e.g., translating (Colored) Petri nets into BPEL [9].

The work presented in this chapter is also related to the choreography language "Let's Dance" [94, 95]. Let's Dance is a language for modeling service interactions and their flow dependencies. The focus of Let's Dance is not so much on the process perspective (although a process modeling notation is added); instead, it focuses on interaction patterns and mechanisms. Similar to DecSerFlow it is positioned as an alternative to the Web Services Choreography Description Language (WS-CDL) [54].

Clearly, this chapter builds on earlier work on process discovery, i.e., the extraction of knowledge from event logs (e.g., process models [15, 17, 27, 37, 38, 47] or social networks [12]). For example, the well-known α algorithm [15] can derive a Petri net from an event log. In [6] we used the conformance checking techniques described in [79] and implemented in our ProM framework [29] and applied this approach to SOAP messages generated from Oracle BPEL. The notion of conformance has also been discussed in the context of security [10], business alignment [1], and genetic mining [66].

It is impossible to give a complete overview of process mining here. Therefore, we refer to a special issue of Computers in Industry on process mining [14] and a survey paper [13]. Process mining can be seen in the broader context of Business (Process) Intelligence (BPI) and Business Activity Monitoring (BAM). In [43, 44, 81] a BPI toolset on top of HP's Process Manager is described. The BPI toolset includes the so-called "BPI Process Mining Engine." In [69] Zur Muehlen describes the PISA tool which can be used to extract performance metrics from workflow logs. Similar diagnostics are provided by the ARIS Process Performance Manager (PPM) [53]. The latter tool is commercially available and a customized version of PPM is the Staffware Process Monitor (SPM) [85] which is tailored toward mining Staffware logs.

The need for monitoring web services has been raised by other researchers. For example, several research groups have been experimenting with adding monitor facilities via SOAP monitors in Axis [19]. Reference [56] introduces an assertion language for expressing business rules and a framework to plan and monitor the execution of these rules. Reference [21] uses a monitoring approach based on BPEL. Monitors are defined as additional services and linked to the original service composition. Another framework for monitoring the compliance of systems composed of web-services is proposed in [60]. This approach uses event calculus to specify requirements. Reference [59] is an approach based on WS-Agreement defining the Crona framework for the creation and monitoring of agreements. In [42, 31], Dustdar et al. discuss the concept of web services mining and envision various levels (web service operations, interactions, and workflows) and approaches. Our approach fits in their framework and shows that web-services mining is indeed possible. In [73] a

tool named the Web Service Navigator is presented to visualize the execution of web services based on SOAP messages. The authors use message sequence diagrams and graph-based representations of the system topology. Note that also in [5] we suggested to focus less on languages like BPEL and more on questions related to the monitoring of web services. In [6] we showed that it is possible to translate abstract BPEL into Petri nets and SOAP messages exchanged between services into event logs represented using the MXML format (i.e., the format used by our process mining tools). As a result, we could demonstrate that it is possible to compare the modeled behavior (in terms of a Petri net) and the observed behavior (in some event log). We used Oracle BPEL and demonstrated that it is possible to monitor SOAP messages using TCP Tunneling technique [6]. This comparison could be used for monitoring deviations and to analyze the most frequently used parts of the service/choreography.

This chapter discussed the idea of conformance checking by comparing the observed behavior recorded in logs with some predefined model. This could be termed "run-time conformance." However, it is also possible to address the issue of *design-time conformance*, i.e., comparing different process models before enactment. For example, one could compare a specification in abstract BPEL with an implementation using executable BPEL. Similarly, one could check at design-time the compatibility of different services. Here one can use the inheritance notions [2] explored in the context of workflow management and implemented in Woflan [88]. Axel Martens et al. [62, 63, 64, 82] have explored questions related to design-time conformance and compatibility using a Petri-net-based approach. For example, [63] focuses on the problem of consistency between executable and abstract processes and [64] presents an approach where for a given composite service the required other services are generated. Also related is [36] where Message Sequence Charts (MSCs) are compiled into the "Finite State Process" notation to describe and reason about web service compositions.

2.6 Conclusion

This chapter focused on *service flows* from the viewpoint of both specification/enactment and monitoring.

First, we discussed more traditional approaches based on Petri nets and BPEL. We showed that Petri nets provide a nice graphical representation and a wide variety of analysis techniques, and mentioned that BPEL has strong industry support making it a viable execution platform. We also showed that there are mappings from BPEL to Petri net for the purpose of analysis (cf. BPEL2PNML and WofBPEL [72]). Moreover, it is possible to translate graphical languages such a Petri nets to BPEL (cf. WorkflowNet2BPEL4WS [55]). Using such techniques, it is also possible to translate languages such as EPCs, BPMN, etc. to BPEL.

Although the first author has been involved in the development of these tools and these tools are mature enough to be applied in real-life applications, both Petri nets and BPEL are rather procedural and this does not fit well with the autonomous nature of services. Therefore, we proposed a new, more declarative language, *DecSerFlow*. Although DecSerFlow is graphical, it is grounded in temporal logic. It can be used for the *enactment* of processes, but it is particularly suited for the *specification* of a single service or a complete choreography. In the last part of this chapter, the focus shifted from languages to process mining. We showed that the combination of DecSerFlow and process mining (conformance checking in particular) is useful in the setting of web services. Moreover, we showed that DecSerFlow can be combined well with the conformance-checking techniques currently implemented in ProM (cf. the LTL checker plug-in).

DecSerFlow also seems to be an interesting proposal for linking global and local models. If both the global model (i.e., the view on the process as seen by some external observer) and one or more local models (i.e., the specification or implementation of a single service or service composition) are modeled in DecSerFlow, standard model checking techniques can be used to compare both.

To conclude, we would like to mention that all of the presented analysis and translation tools can be downloaded from various web sites: [75] (ProM), [20] (BPEL2PNML and WofBPEL), and [93] (WorkflowNet2BPEL4WS).

References

1. W.M.P. van der Aalst. Business Alignment: Using Process Mining as a Tool for Delta Analysis. In J. Grundspenkis and M. Kirikova, editors, *Proceedings of the 5th Workshop on Business Process Modeling, Development and Support (BPMDS'04)*, volume 2 of *Caise'04 Workshops*, pages 138–145. Riga Technical University, Latvia, 2004.
2. W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270 (1-2):125–203, 2002.
3. W.M.P. van der Aalst, H.T. de Beer, and B.F. van Dongen. Process Mining and Verification of Properties: An Approach based on Temporal Logic. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer-Verlag, Berlin, 2005.
4. W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Web Service Composition Languages: Old Wine in New Bottles? In G. Chroust and C. Hofer, editors, *Proceeding of the 29th EUROMICRO Conference: New Waves in System Architecture*, pages 298–305. IEEE Computer Society, Los Alamitos, CA, 2003.
5. W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, N. Russell, H.M.W. Verbeek, and P. Wohed. Life After BPEL? In M. Bravetti, L. Kloul, and

G. Zavattaro, editors, *WS-FM 2005*, volume 3670 of *Lecture Notes in Computer Science*, pages 35–50. Springer-Verlag, Berlin, 2005.
6. W.M.P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and H.M.W. Verbeek. Choreography Conformance Checking: An Approach based on BPEL and Petri Nets (extended version). BPM Center Report BPM-05-25, BPMcenter.org, 2005.
7. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
8. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
9. W.M.P. van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System Paper. In R. Meersman and Z. Tari et al., editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005*, volume 3760 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, 2005.
10. W.M.P. van der Aalst and A.K.A. de Medeiros. Process Mining and Security: Detecting Anomalous Process Executions and Checking Process Conformance. In N. Busi, R. Gorrieri, and F. Martinelli, editors, *Second International Workshop on Security Issues with Petri Nets and other Computational Models (WISP 2004)*, pages 69–84. STAR, Servizio Tipografico Area della Ricerca, CNR Pisa, Italy, 2004.
11. W.M.P. van der Aalst, H.A. Reijers, and M. Song. Discovering Social Networks from Event Logs. *Computer Supported Cooperative work*, 14(6):549–593, 2005.
12. W.M.P. van der Aalst and M. Song. Mining Social Networks: Uncovering Interaction Patterns in Business Processes. In J. Desel, B. Pernici, and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, Berlin, 2004.
13. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
14. W.M.P. van der Aalst and A.J.M.M. Weijters, editors. *Process Mining*, Special Issue of Computers in Industry, Volume 53, Number 3. Elsevier Science Publishers, Amsterdam, 2004.
15. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
16. W.M.P. van der Aalst and M. Weske. The P2P approach to Interorganizational Workflows. In K.R. Dittrich, A. Geppert, and M.C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 140–156. Springer-Verlag, Berlin, 2001.
17. R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*, pages 469–483, 1998.
18. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business

Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003.
19. Apache Axis, http://ws.apache.org/axis/.
20. BABEL, Expressiveness Comparison and Interchange Facilitation Between Business Process Execution Languages, http://www.bpm.fit.qut.edu.au/projects/babel/tools/.
21. L. Baresi, C. Ghezzi, and S. Guinea. Smart Monitors for Composed Services. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 193–202, New York, NY, USA, 2004. ACM Press.
22. T. Belwood and et al. UDDI Version 3.0. http://uddi.org/pubs/uddi_v3.htm, 2000.
23. D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/soap, 2000.
24. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.
25. P. Chrzastowski-Wachtel. A Top-down Petri Net Based Approach for Dynamic Workflow Modeling. In W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske, editors, *International Conference on Business Process Management (BPM 2003)*, volume 2678 of *Lecture Notes in Computer Science*, pages 336–353. Springer-Verlag, Berlin, 2003.
26. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.
27. J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.
28. A. Datta. Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. *Information Systems Research*, 9(3): 275–301, 1998.
29. B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. In G. Ciardo and P. Darondeau, editors, *Application and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag, Berlin, 2005.
30. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.
31. S. Dustdar, R. Gombotz, and K. Baina. Web Services Interaction Mining. Technical Report TUV-1841-2004-16, Information Systems Institute, Vienna University of Technology, Wien, Austria, 2004.
32. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
33. D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative control flow. In D. Beauquier and E. Börger and A. Slissenko, editor, *Proc. 12th International Workshop on Abstract State Machines*, pages 131–151, Paris, France, March 2005.

34. A. Ferrara. Web services: A process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.
35. J.A. Fisteus, L.S. Fernández, and C.D. Kloos. Formal verification of BPEL4WS business collaborations. In K. Bauknecht, M. Bichler, and B. Proll, editors, *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web '04)*, volume 3182 of *Lecture Notes in Computer Science*, pages 79–94, Zaragoza, Spain, August 2004. Springer-Verlag, Berlin.
36. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Composition. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 152–161, Montreal, Canada, October 2003.
37. W. Gaaloul, S. Bhiri, and C. Godart. Discovering Workflow Transactional Behavior from Event-Based Log. In R. Meersman, Z. Tari, W.M.P. van der Aalst, C. Bussler, and A. Gal et al., editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004*, volume 3290 of *Lecture Notes in Computer Science*, pages 3–18, 2004.
38. W. Gaaloul and C. Godart. Mining Workflow Recovery from Event Based Logs. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management (BPM 2005)*, volume 3649, pages 169–185. Springer-Verlag, Berlin, 2005.
39. D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.
40. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.
41. D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 412, Washington, DC, USA, 2001. IEEE Computer Society.
42. R. Gombotz and S. Dustdar. On Web Services Mining. In M. Castellanos and T. Weijters, editors, *First International Workshop on Business Process Intelligence (BPI'05)*, pages 58–70, Nancy, France, September 2005.
43. D. Grigori, F. Casati, M. Castellanos, U. Dayal, M. Sayal, and M.C. Shan. Business Process Intelligence. *Computers in Industry*, 53(3):321–343, 2004.
44. D. Grigori, F. Casati, U. Dayal, and M.C. Shan. Improving Business Process Quality through Exception Understanding, Prediction, and Prevention. In P. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. Snodgrass, editors, *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 159–168. Morgan Kaufmann, 2001.
45. K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. IEEE Computer Society Press, Providence, 2001.
46. K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the*

Construction and Analysis of Systems (TACAS 2002), volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, Berlin, 2002.
47. J. Herbst. A Machine Learning Approach to Workflow Management. In *Proceedings 11th European Conference on Machine Learning*, volume 1810 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag, Berlin, 2000.
48. G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5): 279–295, 1997.
49. G.J. Holzmann. An Analysis of Bitstate Hashing. *Form. Methods Syst. Des.*, 13(3):289–307, 1998.
50. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, USA, 2003.
51. G.J. Holzmann and D. Peled. An Improvement in Formal Verification. In *FORTE 1994 Conference*, Bern, Switzerland, 1994.
52. G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *The Spin Verification System, Proceedings of the 2nd Spin Workshop.)*, pages 23–32. American Mathematical Society, 1996.
53. IDS Scheer. ARIS Process Performance Manager (ARIS PPM): Measure, Analyze and Optimize Your Business Process Performance (whitepaper). IDS Scheer, Saarbruecken, Gemany, `http://www.ids-scheer.com`, 2002.
54. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web Services Choreography Description Language, Version 1.0. W3C Working Draft 17-12-04, 2004.
55. K.B. Lassen and W.M.P. van der Aalst. WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. BETA Working Paper Series, WP 167, Eindhoven University of Technology, Eindhoven, 2006.
56. A. Lazovik, M. Aiello, and M. Papazoglou. Associating Assertions with Business Processes and Monitoring their Execution. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 94–104, New York, NY, USA, 2004. ACM Press.
57. F. Leymann. Web Services Flow Language, Version 1.0, 2001.
58. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques.* Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
59. H. Ludwig, A. Dan, and R. Kearney. Crona: An Architecture and Library for Creation and Monitoring of WS-agreements. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 65–74, New York, NY, USA, 2004. ACM Press.
60. K. Mahbub and G. Spanoudakis. A Framework for Requirents Monitoring of Service Based Systems. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 84–93, New York, NY, USA, 2004. ACM Press.
61. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.
62. A. Martens. Analyzing Web Service Based Business Processes. In M. Cerioli, editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer-Verlag, Berlin, 2005.
63. A. Martens. Consistency between executable and abstract processes. In *Proceedings of International IEEE Conference on e-Technology, e-Commerce, and e-Services (EEE'05)*, pages 60–67. IEEE Computer Society Press, 2005.

64. P. Massuthe, W. Reisig, and K. Schmidt. An Operating Guideline Approach to the SOA. In *Proceedings of the 2nd South-East European Workshop on Formal Methods 2005 (SEEFM05)*, Ohrid, Republic of Macedonia, 2005.
65. M. Mecella, F. Parisi-Presicce, and B. Pernici. Modeling E-service Orchestration through Petri Nets. In *Proceedings of the Third International Workshop on Technologies for E-Services*, volume 2644 of *Lecture Notes in Computer Science*, pages 38–47. Springer-Verlag, Berlin, 2002.
66. A.K.A. de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Using Genetic Algorithms to Mine Process Models: Representation, Operators and Results. BETA Working Paper Series, WP 124, Eindhoven University of Technology, Eindhoven, 2004.
67. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.
68. M. zur Muehlen. *Workflow-based Process Controlling: Foundation, Design and Application of workflow-driven Process Information Systems*. Logos, Berlin, 2004.
69. M. zur Mühlen and M. Rosemann. Workflow-based Process Monitoring and Controlling - Technical and Organizational Issues. In R. Sprague, editor, *Proceedings of the 33rd Hawaii International Conference on System Science (HICSS-33)*, pages 1–10. IEEE Computer Society Press, Los Alamitos, California, 2000.
70. OASIS Web Services Business Process Execution Language (WSBPEL) TC, `http://www.oasis-open.org/committees/tc\_home.php?wg\_abbrev=wsbpel`.
71. C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal Semantics and Analysis of Control Flow in WS-BPEL. BPM Center Report BPM-05-15, BPMcenter.org, 2005.
72. C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, and A.H.M. ter Hofstede. WofBPEL: A Tool for Automated Analysis of BPEL Processes. In B. Benatallah, F. Casati, and P. Traverso, editors, *Proceedings of Service-Oriented Computing (ICSOC 2005)*, volume 3826 of *Lecture Notes in Computer Science*, pages 484–489. Springer-Verlag, Berlin, 2005.
73. W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J.F. Morar. Web Services Navigator: Visualizing the Execution of Web Services. *IBM Systems Journal*, 44(4):821–845, 2005.
74. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, Providence, 1977.
75. Process Mining Home Page, `http://www.processmining.org`.
76. Process Modelling Group, `http://process-modelling-group.org`.
77. A. Puri and G.J. Holzmann. A Minimized automaton representation of reachable states. In *Software Tools for Technology Transfer*, volume 3. Springer-Verlag, Berlin, 1993.
78. 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.
79. A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In C. Bussler et al., editor, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*,

volume 3812 of *Lecture Notes in Computer Science*, pages 163–176. Springer-Verlag, Berlin, 2006.
80. A. Rozinat and W.M.P. van der Aalst. Decision Mining in ProM. In S. Dustdar, J.L. Faideiro, and A. Sheth, editors, *International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 420–425. Springer-Verlag, Berlin, 2006.
81. M. Sayal, F. Casati, U. Dayal, and M.C. Shan. Business Process Cockpit. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 880–883. Morgan Kaufmann, 2002.
82. B.H. Schlingloff, A. Martens, and K. Schmidt. Modeling and model checking web services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems*, 126:3–26, mar 2005.
83. C. Stahl. Transformation von BPEL4WS in Petrinetze (In German). Master's thesis, Humboldt University, Berlin, Germany, 2004.
84. S. Thatte. XLANG Web Services for Business Process Design, 2001.
85. TIBCO. TIBCO Staffware Process Monitor (SPM). `http://www.tibco.com`, 2005.
86. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *In Proceedings of the 1st Symposium on Logic in Computer Science*, pages 322–331, Cambridge, Massachusetts, USA, 1986.
87. H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL Processes using Petri Nets. In D. Marinescu, editor, *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78. Florida International University, Miami, Florida, USA, 2005.
88. 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, 2001.
89. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.
90. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In I.Y. Song, S.W. Liddle, T.W. Ling, and P. Scheuermann, editors, *22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer-Verlag, Berlin, 2003.
91. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. 5th Int. Conference on Computer Aided Verification*, pages 59–70, 1993.
92. P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about Infinite Computation Paths. In *Proceedings of the 24th IEEE symposium on foundation of cumputer science*, pages 185–194, Tucson, Arizona, November 1983.
93. WorkflowNet2BPEL4WS, `http://www.daimi.au.dk/~krell/WorkflowNet2BPEL4WS/`.
94. J.M. Zaha, A. Barros, M. Dumas, and A.H.M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. QUT ePrints 4468, Faculty of Information Technology, Queensland University of Technology, 2006.
95. J.M. Zaha, M. Dumas, A.H.M. ter Hofstede, A. Barros, and G. Dekker. Service Interaction Modeling: Bridging Global and Local Views. QUT ePrints 4032, Faculty of Information Technology, Queensland University of Technology, 2006.