Specifying, Discovering, 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. However, both are rather procedural and this does not fit well with the autonomous nature of services. Therefore, we propose *DecSerFlow* as a *Declarative Service Flow Language*. Moreover, we link this language to process mining where we focus on *process discovery* and *conformance testing*. This makes it possible to uncover processes emerging in a service-oriented architecture. Moreover, it can be used to expose services that do not follow the rules of the game.

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. 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, WSDL and UDDI [3, 18]. Once deployed, web services provided by various organizations can be inter-connected in order to implement business collaborations, leading to *composite web services*.

Today workflow management systems are readily available [6, 49, 59] 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) [48] building on FlowMark/MQSeries and Microsoft proposed XLANG (Web Services for Business Process Design) [71] building on Biztalk. BPEL [16] 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 [16]. 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 [22] will be realized using BPEL. Whilst being a powerful language, BPEL is difficult to use. Its XML representation is very verbose and only readable 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 [66] and Pi calculus [58] 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 (`process-modelling-group.org`) in June 2004. 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. 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. 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*. 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.

Moreover, there is a clear link between more declarative languages such as DecSerFlow and process mining. It is possible to discover (part of) a DecSerFlow model or to project e.g. performance data on such a model. Moreover, given a DecSerFlow model it is possible to measure conformance and to locate deviations.

The remainder of this chapter is organized as follows. Section 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 3 first discusses the need for a more declarative language and then introduces the DecSerFlow language. In Section 4 the focus shifts from languages to process mining, i.e., first process discovery (Section 4.1) and then conformance checking (Section 4.2) are discussed in the context of web services. Finally there is a section on related work (Section 5) and a conclusion (Section 6).

2 BPEL and Petri Nets

2.1 Petri Nets

Petri nets [66] were among the first formalisms to capture the notion of concurrency. They combine an intuitive graphical notation with formal semantics and an wide range of analysis techniques. In recent years they have been applied in the context of process-aware information systems [22], workflow management [6, 8], and web services [55].

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 [14]. Figure 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*.

The circles 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 [6]). 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 Figure 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

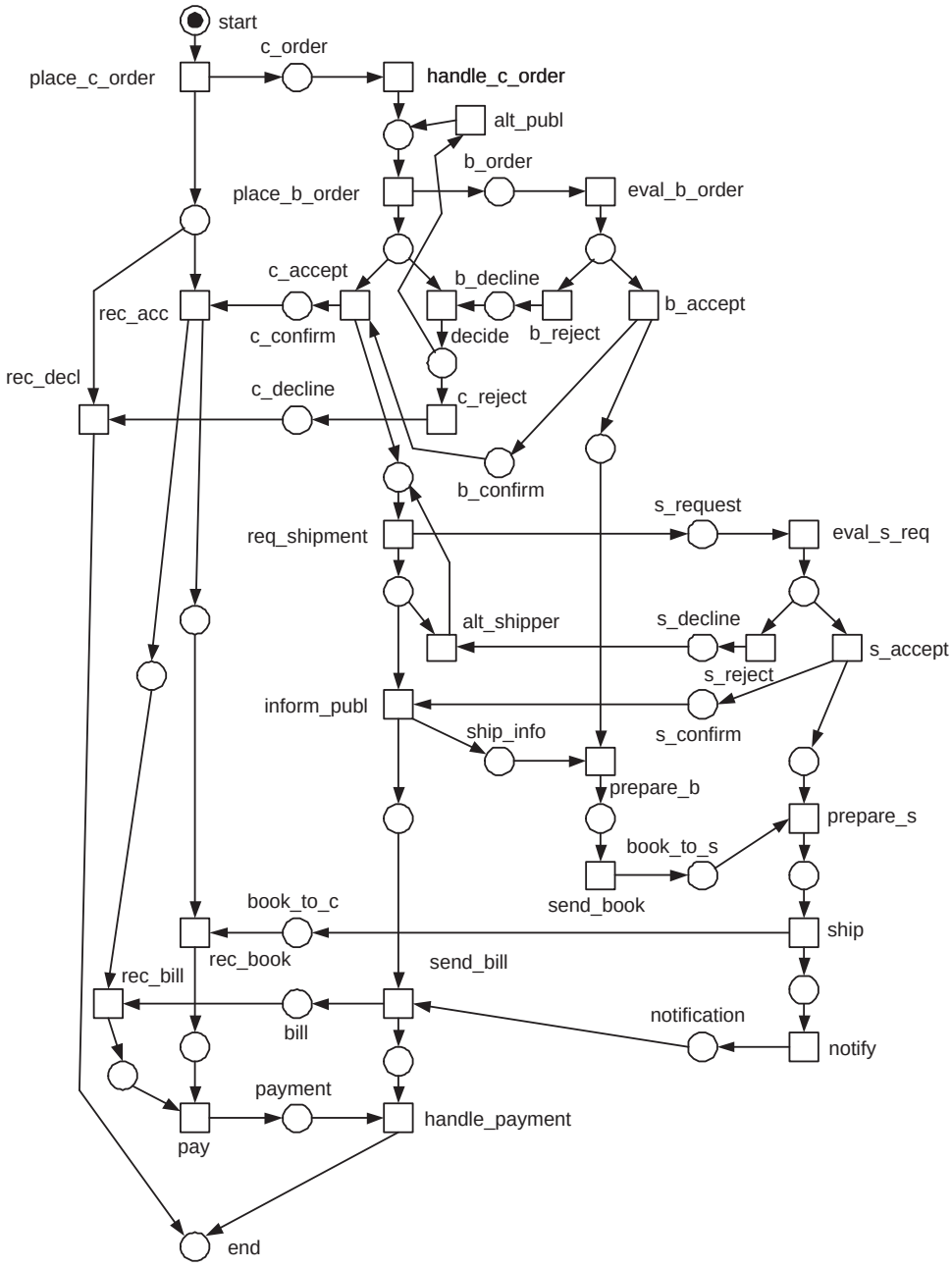


Fig. 1. A Petri net describing the process as agreed upon by all four parties.

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.

Figure 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 can not 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 Figure 1 is a 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 [75] or ProM [21] we can show that the process is *sound* [2, 6]. Figure 2 shows a screenshot of the Woflan plug-in of ProM. This 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.

One can think of the Petri net shown in Figure 1 as the contract between the customer, the bookstore, the publisher, and the shipper. Clearly there are many

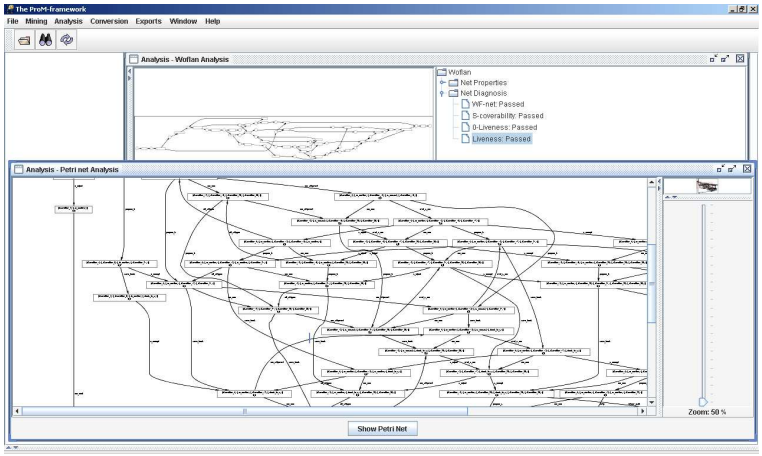


Fig. 2. Two analysis plug-in of ProM indicate that the Petri net shown in Figure 1 is indeed sound. The top window shows some diagnostics related to soundness. The bottom window shows part of the state space.

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 one shipper (if any) are involved. Note that Figure 1 abstracts from a lot of relevant things. However, given the purpose of this chapter we do not add more details.

Figure 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. We can think of the Petri-net fragments inside these rectangles as specifications of the corresponding services. We can think of the whole diagram as the *choreography* or *orchestration* of the four services.

2.2 BPEL

BPEL [16] supports the modeling of two types of processes: executable and abstract processes. An *abstract*, (not executable) *process* is a business protocol, specifying the message exchange behavior between different parties without revealing the internal behavior for 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 *mes-*

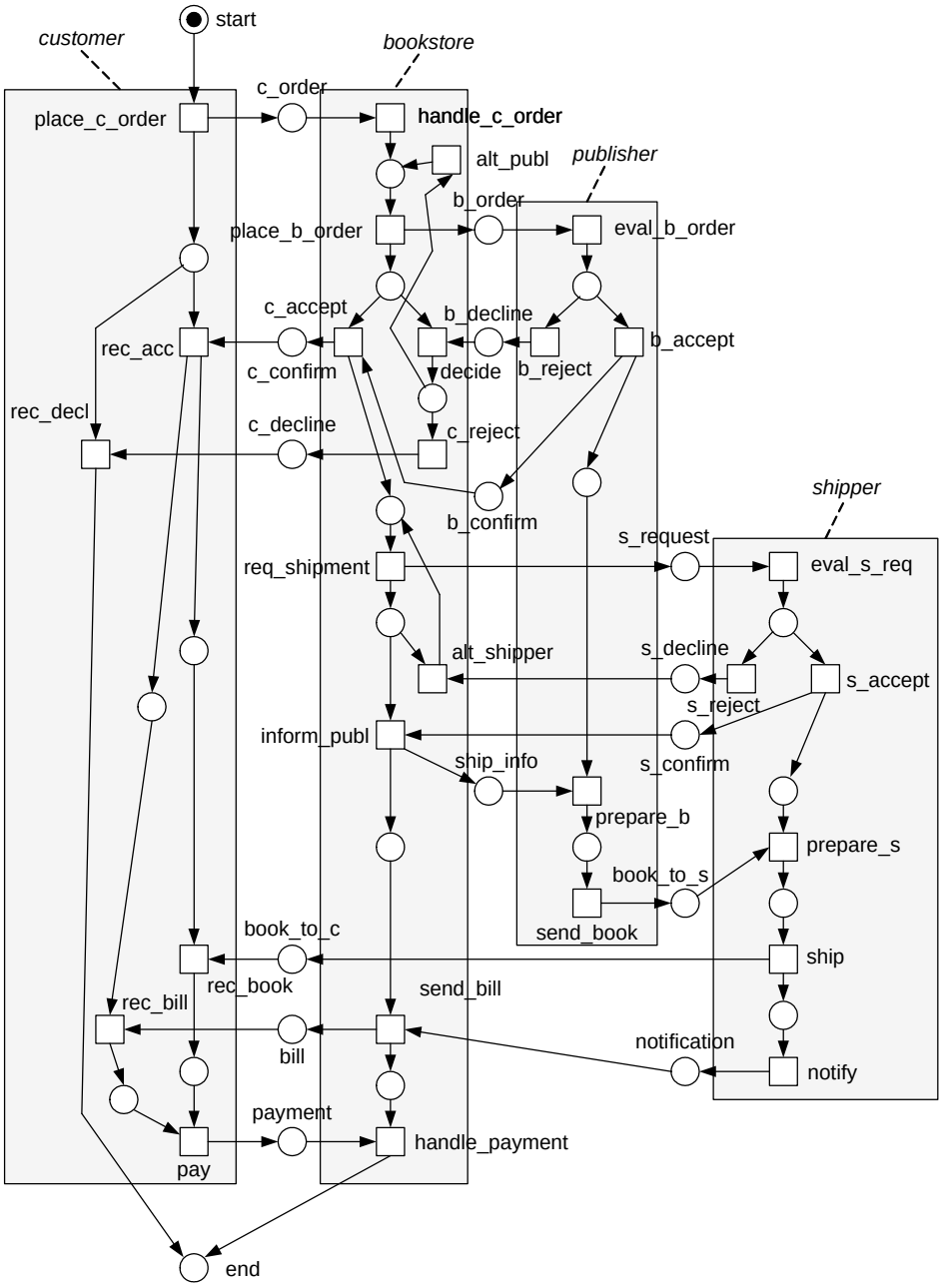


Fig. 3. The process as partitioned over (1) the *customer*, (2) the *bookstore*, (3) the *publisher*, and (4) the *shipper*.

sages exchanged between these partners, and the *fault* and *exception handling* required in cases of errors and exceptions.

In terms of Figure 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 the introduction, BPEL builds on IBM's WSFL (Web Services Flow Language) [48] and Microsoft's XLANG (Web Services for Business Process Design) [71] 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* [7] can only be modeled using the *pick* construct. For other constructs one is forced to use the 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, that is, 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, that is, 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 page 64 in [16]). 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 [16] and various web sites such as the web site of the OASIS technical committee on WS-BPEL: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

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. Figure 2). Therefore, it is interesting to look at the relation between 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 Section 5). As a example, we briefly describe the combination formed by BPEL2PNML and WofBPEL developed in a collaboration with QUT [62]. BPEL2PNML translates BPEL process definitions into Petri nets represented in the Petri Net Markup Language (PNML). WofBPEL, built using Woflan [75], applies static analysis and transformation techniques on the output produced by BPEL2PNML. WofBPEL can be used to (i) simplify the Petri net produced by BPEL2PNML by removing unnecessary silent transitions, and (ii) convert the Petri net into a 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 [5].

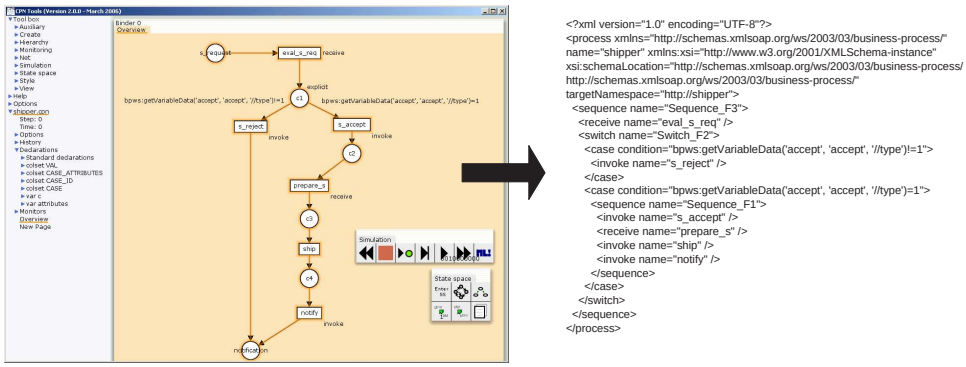


Fig. 4. The Petri net describing the service offered by the shipper is mapped onto BPEL code using WorkflowNet2BPEL4WS, a tool to automatically translate colored Petri nets into BPEL template code.

Few people have been working on the translation from Petri nets to BPEL. In fact, [8] is the only work we are aware of that tries to go from (colored) Petri nets to BPEL. Using our ProM tool [21] 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 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* [8]. To illustrate this, consider the shipper service shown in Figure 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 4 shows both the annotated WF-net in CPN Tools (left) and the automatically generated BPEL template code (right).

The presence of the tools and systems mentioned in this section make it possible to support *service flows*, i.e., the process-aspect of web services, at the design, analysis, and enactment level. 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 the introduction we would now like to move the focus to more *declarative languages* (Section 3) and *process mining* (Section 4).

3 Towards a Declarative Language for Specifying, Enacting, and Monitoring of Web Services

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.

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 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

[1] Note that both BPEL and Petri nets support the deferred choice pattern [7], i.e., it is possible to put the system in a state where several alternative activities are 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.

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 Figure 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 [46]. 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." [46]. 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 Figure 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 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 5 times while B is not executed at all. Second, the moment of choice is irrelevant for 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 over-specify things.

Therefore, we propose a more declarative approach based on *temporal logic* [52, 64] as described in the following subsection.

3.2 DecSerFlow: A Declarative Service Flow Language

Languages such as *Linear Temporal Logic* (LTL) [32, 36, 37] allow for the a more declarative style of modeling. These languages include temporal operators such as nexttime ($\bigcirc F$), eventually ($\Diamond F$), always ($\Box F$), and until ($F \sqcup G$). However, such languages are difficult to read. Therefore, we define a graphical syntax for some 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 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 specify sequential, parallel, alternative, and iterative routing. 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 for example 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 constraints 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 [24]. 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 form 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.

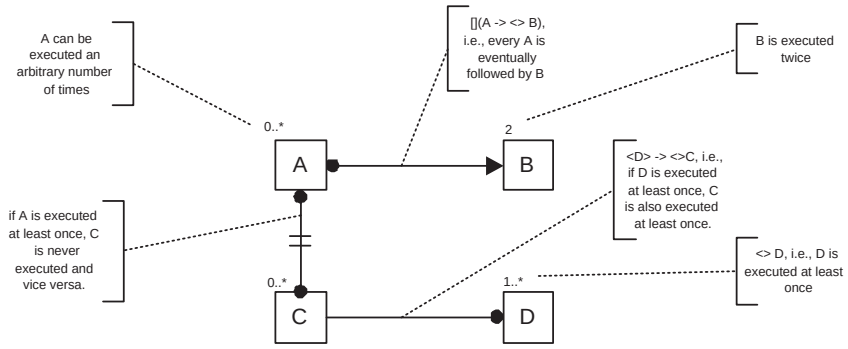


Fig. 5. A DecSerFlow model showing some example notations.

Before giving an overview of the initial set of formulas and their notation, we give a small example explaining the basic idea. Figure 5 shows a DecSerFlow model consisting of four activities: A, B, C, and D. Each activity is tagged with a constraint describing to the number of times the activity should 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 B and C denotes a "negation formula" (the LTL expression is not show here). Note that it is not easy to provide a classical procedural model (e.g., a Petri net) that allows for all behaviour modeled Figure 5.

Existence formulas. Figure 6 shows the so-called "existence formulas". These formulas define the cardinality of an activity. For example, the first formula

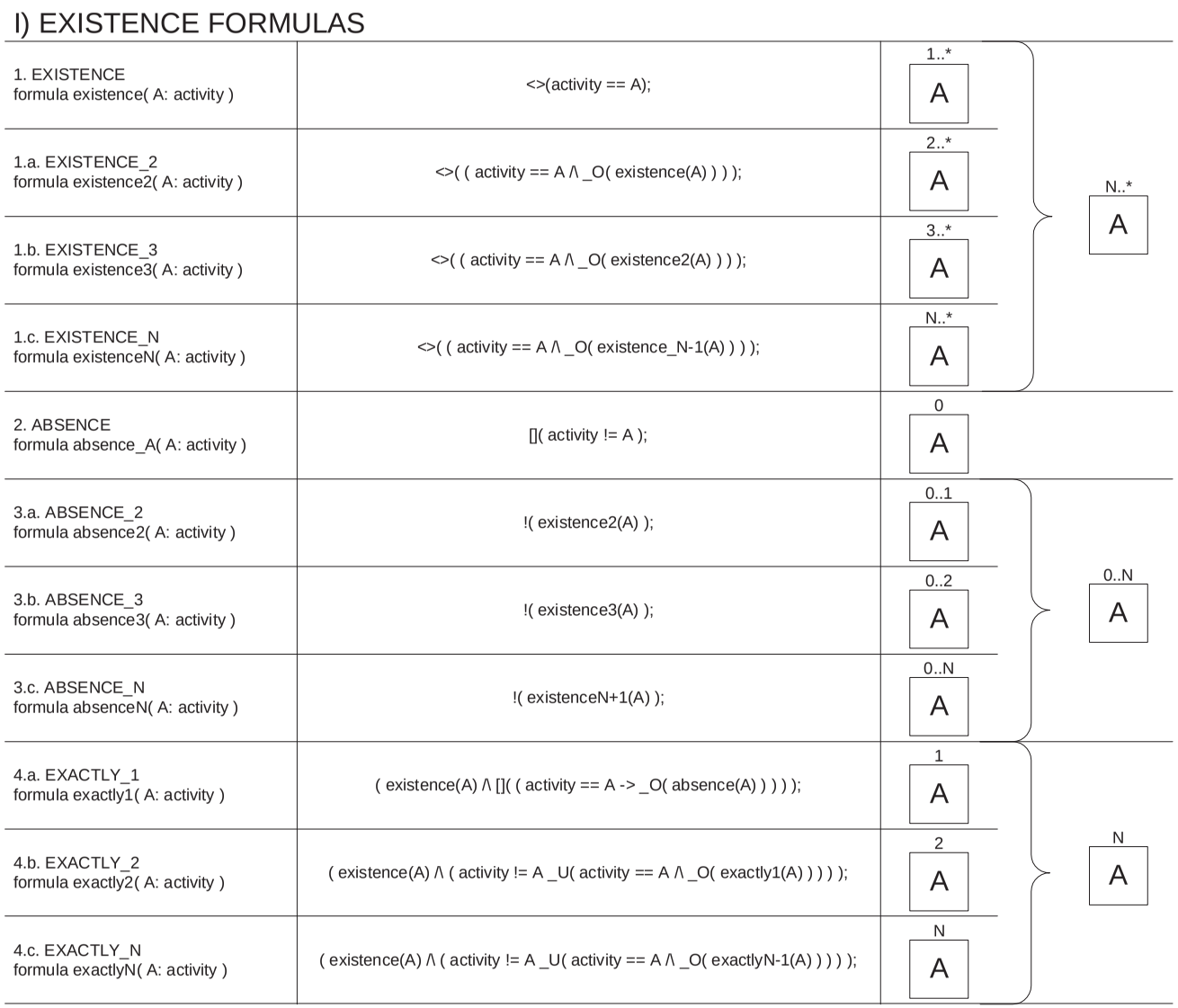


Fig. 6. Notations for the “existence formulas”.

is called *existence*. The name and the formula heading are shown in the first column. From this, we can see that it takes one parameter (A), which is the name of an activity. The body of the formula is written in LTL and can be seen in the second column. In this case the LTL expression $\Diamond(activity == A)$ ensures that the activity given as the parameter A will execute at least once. Note that we write $\Diamond(activity == A)$ rather than $\Diamond(A)$. The reason is that in a state we also want to access other properties, i.e., not just the activity name but also information on data, time, and resources. Therefore, we need to use a slightly more verbose notation ($activity == A$). The diagram in the third column is the graphical representation of the formula, which is assigned to the template. Parameter A is an activity and it is represented as a square with the name of the activity. The constraint is represented by a cardinality annotation above the square. In this case the cardinality is at least one, which is represented by *1..**. The first group of existence formulas are of the cardinality "N or more", denoted by *N..**. Next, the formula *absence* ensures that the activity should never execute in the service. The group of formulas with names *absenceN* uses negations of *existenceN* to specify that an activity can be executed at most *N-1* times. The last group of existence formulas defines an exact number of executions of an activity. For example, if a constraint is defined based on the formula *exactly2*, the referring activity has to be executed exactly two times in the service.

Relation formulas. Figure 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 two activities. All relation formulas have two activities as parameters and two activities in the graphical representation. The line between the two activities in the graphical representation should be unique for the formula, and reflect the semantics of the relation. The *responded existence* formula specifies that if activity A is executed, activity B also has to be executed either before or after the activity A. 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 the 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 ac-

II) RELATION BETWEEN EVENTS FORMULAS

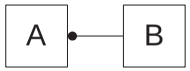
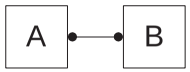
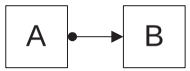
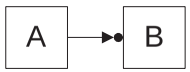
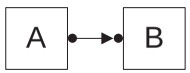
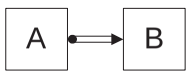
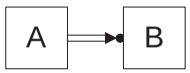
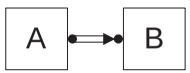
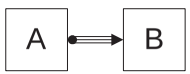
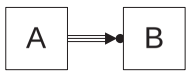
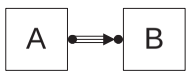
1. RESPONDED EXISTENCE formula existence_A_response_B(A: activity, B: activity)	(existence_A(A) -> existenceA(B));	
2. CO-EXISTENCE formula co_existence_A_and_B(A: activity, B: activity)	(existence(A) <-> existence(B));	
3. RESPONSE formula A_response_B(A: activity, B: activity)	[]((activity == A -> existence(B)));	
4. PRECEDENCE formula A_precedence_B(A: activity, B: activity)	(existence_A(B) -> (!(activity == B) _U activity == A));	
5. SUCCESSION formula A_succession_B(A: activity, B: activity)	(A_response_B(A,B) /\ A_precedence_B(A,B));	
6. ALTERNATE RESPONSE formula A_alternate_response_B(A: activity, B: activity)	(A_response_B(A,B) /\ B_always_between_A(A,B)*);	
7. ALTERNATE PRECEDENCE formula A_alternate_precedence_B(A: activity, B: activity)	(A_precedence_B(A,B) /\ B_always_between_A(B,A)*);	
8. ALTERNATE SUCCESSION formula A_alternate_succession_B(A: activity, B: activity)	(A_alternate_precedence_B(A,B) /\ A_alternate_response_B(A,B));	
9. CHAIN RESPONSE formula chain_A_response_B(A: activity, B: activity)	[]((activity == A -> _O(activity == B)));	
10. CHAIN PRECEDENCE formula chain_A_precedence_B(A: activity, B: activity)	(A_precedence_B(A,B) /\ []((_O(activity == B) -> activity == A)));	
11. CHAIN SUCCESSION formula chain_A_succession_B(A: activity, B: activity)	(chain_A_response_B(A,B) /\ chain_A_precedence_B(A,B));	
* subformula B_always_between_A(A: activity, B: activity)	[]((activity == A -> _O(A_precedence_B(B,A))));	

Fig. 7. Notations for the “relation formulas”.

tivity 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 the 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 can not 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 can not 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 formulas are *chain response*, *chain precedence* and *chain succession*, which require that the executions of the two activities (A and B) are next to each other. In the *chain response* the next activity after the 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 first preceding activity before 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.

Negation formulas. Figure 8 shows the "negation formulas", which are the negated versions of the "relation formulas". The first two formulas negate the *responded existence* and *co-existence* formulas. The *responded absence* formula specifies that if activity A is executed activity B must never be executed (not before nor after the activity A). The *not co-existence* formula applies *responded absence* from A to B and from B to A. However, if we look at the *responded absence* formula we can see that if existence of A implies the absence of B and we first execute activity B, it will not be possible to execute activity A anymore because the formula will become permanently incorrect. This means that the formula *responded absence* is symmetric with respect to the input, i.e., we can swap the roles of A and B without changing the outcome. Therefore formula *responded absence* will be skipped and we will use only the *not co-existence* formula. The graphical representation is a modified representation of the *co-existence* formula with the negation symbol in the middle of the line. An example of a correct execution sequence for the formula *not co-existence* is *[A,C,A,A]*, while the sequence *[A,C,A,A,B]* would not be correct.

The *negation response* formula specifies that after the execution of activity A, activity B can not be executed. According to the formula *negation precedence* activity B can not be preceded by activity A. These two formulas have the

III) NEGATION RELATION BETWEEN EVENTS FORMULAS

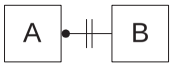
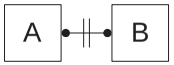
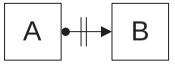
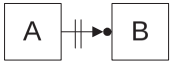
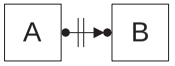
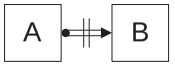
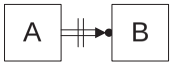
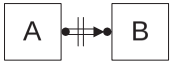
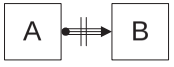
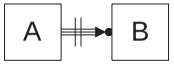
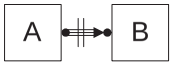
12.a. RESPONDED ABSENCE formula existence_A_response_notB(A: activity, B: activity)	(existence_A(A) -> absence(B));	
12.b. NOT CO_EXISTENCE formula existence_A_response_notB(A: activity, B: activity)	(existence_A_response_notB(A,B) /\ existence_A_response_notB(B,A));	
13.a. NEGATION RESPONSE formula A_response_notB(A: activity, B: activity)	[]((activity == A -> absence(B)));	
13.b. NEGATION PRECEDENCE formula notA_precedence_B(A: activity, B: activity)	[]((existence(B) -> activity != A));	
13.c. NEGATION SUCCESSION formula notA_succession_notB(A: activity, B: activity)	(A_response_notB(A,B) /\ notA_precedence_B(A,B));	
14. NEGATION ALTERNATE REPONSE formula A_not_alternate_response_B(A: activity, B: activity)	B_never_between_A(A,B)**;	
15. NEGATION ALTERNATE PRECEDENCE formula A_not_alternate_precedence_B(A: activity, B: activity)	B_never_between_A(B,A)**;	
16. NEGATION ALTERNATE SUCCESSION A_not_alternate_succession_B(A: activity, B: activity)	(A_not_alternate_precedence_B(A,B) /\ A_not_alternate_response_B(A,B));	
17.a.. NEGATION CHAIN RESPONSE formula chain_A_response_notB(A: activity, B: activity)	[]((activity == A -> _O(activity != B)));	
17.b. NEGATION CHAIN PRECEDENCE formula chain_notA_precedence_B(A: activity, B: activity)	[]((_O(activity == B) -> activity != A));	
17.c. NEGATION CHAIN SUCCESSION formula chain_A_notsuccession_B(A: activity, B: activity)	(chain_A_response_notB(A,B) /\ chain_notA_precedence_B(A,B));	
** subformula B_never_between_A(A: activity, B: activity)	[]((activity == A -> _O((<>(activity == A) -> (activity != B _U activity == A)))));	

Fig. 8. Notations for the “negations formulas”.

same effect because it is not possible to have activity B executed after activity A and it is not possible to have activity A executed before activity B. Since the formula *negation succession* combines these two formulas, it also has the same effect and we will use only the *negation succession* formula. The graphical representation of this formula is a modified representation of the *succession* formula with a negation symbol in the middle of the line. The execution sequence *[B,B,C,A,C,A,A]* is an example of a correct sequence, while *[A,C,B]* would be an incorrect execution.

Formulas *negation alternate response*, *negation alternate precedence* and *negation alternate succession* are easy to understand. The formula *negation alternate response* specifies that the activity B can not be executed between the two subsequent executions of the activity A. According to this formula the execution sequence *[B,A,C,A,B]* is correct. In the case of the *negation alternate precedence* activity A can not be executed between two subsequent executions of the activity B. The execution sequence *[A,B,C,B,A]* is correct for *negation alternate precedence*. The formula *negation alternate succession* requires both *negation alternate response* and *negation alternate precedence* to be satisfied. An example of a correct execution sequence for the *negation alternate succession* formula is *[B,C,B,A,C,A]*. Graphical representations of these three formulas are similar to the representations of *alternate response*, *alternate precedence* and *alternate succession* with the negation symbol in the middle of the line.

The last three formulas are negations of formulas *chain response*, *chain precedence* and *chain succession*. According to the formula *negation chain response*, activity B can not be executed directly after the activity A. Formula *negation chain precedence* specifies that activity B can never be directly preceded by activity A. These two formulas have the same effect because they forbid the activities A and B to be executed directly next to each other. Since the formula *negation chain succession* requires both *negation chain response* and *negation chain precedence* to be executed, these three formulas all have the same effect and we will use only *negation chain succession*. The graphical representation of this formula is a modified version of the representation of the *chain succession* formula with the negation symbol in the middle of the line. The execution sequence *[B,A,C,B,A]* is correct according to the *negation chain succession* formula, while the sequence *[B,A,B,A]* would not be correct.

Figures 7 and 8 show only binary relationships. However, these can easily be extended to deal with more activities. Consider for example the *response* relationship, i.e., $\Box(A \longrightarrow \Diamond B)$. We will allow multiple arcs to start from the same dot, e.g., an arc to B, C, and D. The meaning is $\Box(A \longrightarrow \Diamond(B \vee C \vee D))$, i.e., every occurrence of A is eventually followed by an occurrence of B, C, or D.

The amazon.com example in DecSerFlow. We use the amazon.com example to show how DecSerFlow language can be used to model services. For this purpose, we will model the customer service using existence, relation and negation formulas. In this way, we will use the defined templates for formulas, apply them to activities form our example and thus create real constraints in our DecSer-

Flow 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 modelling 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 9 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.

Constrains formulated as *relation* and *negation* formulas are added to describe dependencies between activities. There is a branched *response* from the activity *place_c_order*. It has two branches: one to the activity *rec_acc* and the other to the activity *rec_decl*. Figure 7 only defines a binary *response* relationship. However, as indicated these binary relationships can be extended in a straightforward manner. In this case every occurrence of *place_c_order* is eventually followed by at least one occurrence of *rec_acc* or *rec_decl*. However, it is possible that both activities are executed, and to prevent this we add the *not co-existence* constraint between activities *rec_acc* and *rec_decl*. So far, we have managed to make sure that after the 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 neither of the activities *rec_acc* and *rec_decl* can be executed before the 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 the activity *rec_acc* can be executed only after the activity *place_c_order* was executed, and (2) the one between activities *place_c_order* and *rec_decl* makes sure that the activity *rec_decl* can be executed only after the activity *place_c_order* was executed. 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 (the activity *rec_acc* is executed), the book does not arrive (the 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 the activity *rec_book* can be executed only after the activity *rec_acc* was 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 *responded existence* constraint between the *rec_book* activity and the *rec_bill* activity. This constraint forces that if the activity *rec_book* is executed, then the activity *rec_bill* must have been executed before or will be executed after the activity *rec_book*. Thus, if the execution of the activity *rec_book* exists, then also the execution of the activity *rec_bill* exists. The constraint *precedence* between the activities *rec_bill* and *pay* means that the customer will only pay after the bill was received. However, after the bill was 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 modelling of a single service, DesSerFlow langauge can as well be used to model the communication between services. In Figure 9 we can see how constraints specify the communication of the customer with the bookstore and the shipper. First, the *succession* constraint between the activity *place_c_order* and *handle_c_order* specifies that after the activity *place_c_order* the activity *handle_c_order* has to be executed, and that the activity *handle_c_order* can be executed only after the 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 was 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 was sent, but we can not 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 one is between the activity *ship* and *rec_book* to specify that the activity *rec_book* can be executed only after the activity *ship* was executed. The second one is between the activities *send_bill* and *rec_bill*, according to which the activity *rec_bill* can be executed only after the activity *send_bill* was executed.

Figure 9 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.

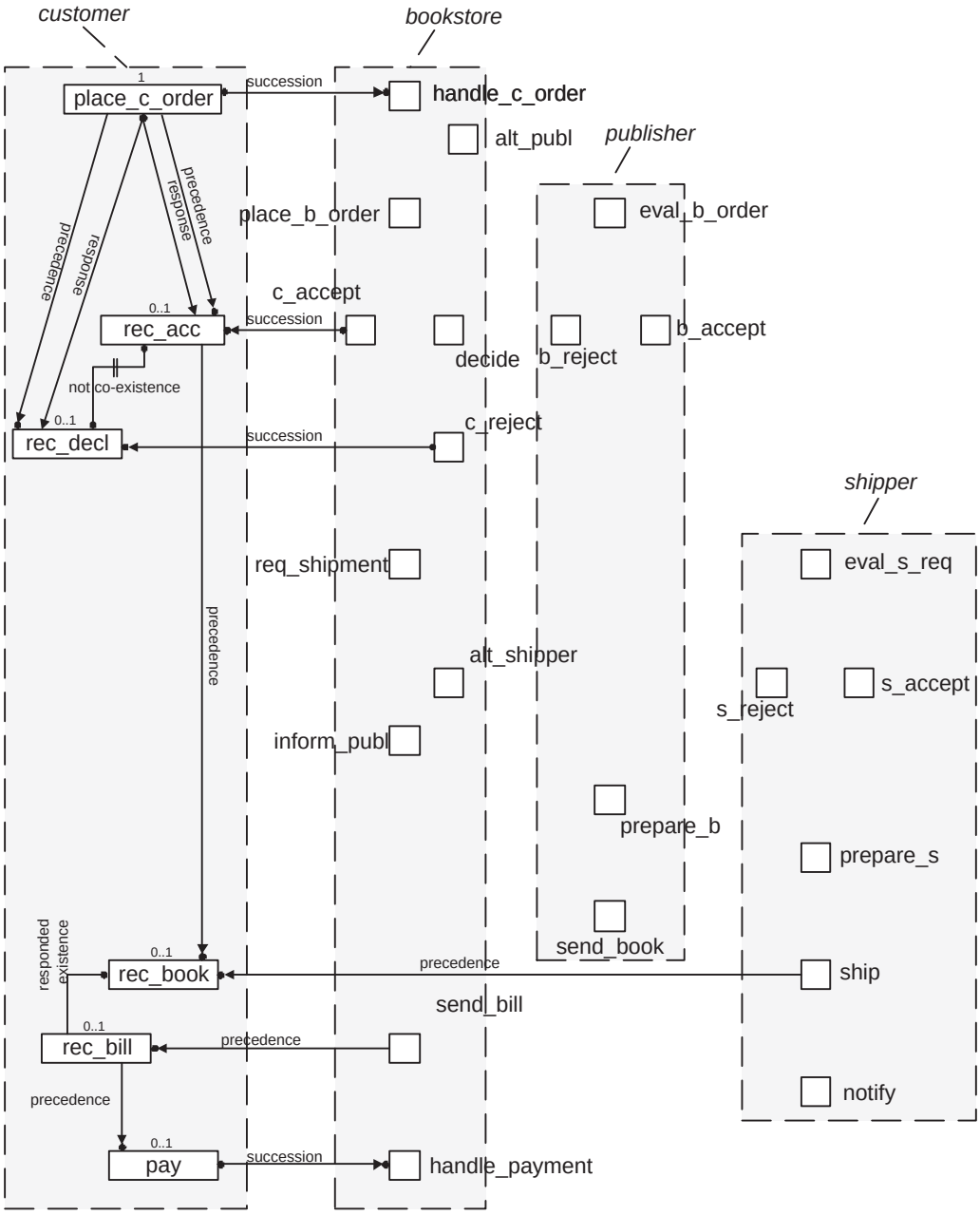


Fig. 9. DecSerFlow model.

3.3 Specifying, Enacting, and Monitoring of Service Flows

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 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 [45]. Algorithms for translating LTL expressions into automatons are given in [31, 78]. 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* [45]. 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 [41] 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 [73]. To use Spin, the system first has to be specified in the verification modelling language Promela (PROcess MEta LAnguage) [41]. Spin can verify the correctness of requirements, which are written as LTL formulas, in a Promela model using the algorithms presented in [31, 39, 40, 42, 43, 73, 65, 77]. When checking the correctness of an LTL formula, Spin first creates an automaton for the *negation* of the formula. If the intersection of the automaton and the system model 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.

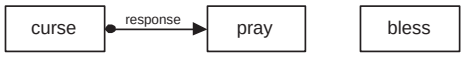


Fig. 10. A simple model in DecSerFlow.

Unlike Spin, which generates an automaton for the negation of the formula, we can execute a DecSerFlow model by constructing an automata for the *formula itself*. We will use a simple DecSerFlow model to show how processes can be executed by translating LTL formulas into automatons. Figure 10 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*. This constraint specifies that if a person curses, (s)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.

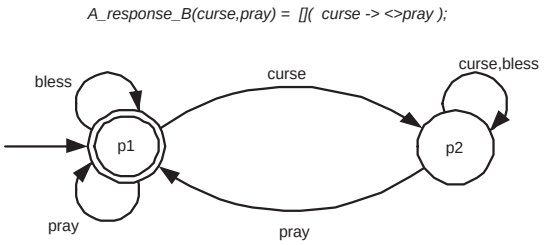


Fig. 11. Automaton for the formula response.

Using the example depicted in Figure 10, we briefly show the mapping of LTL formulas onto generalized Buchi automata [31]. A generalized Buchi automaton is a Buchi automaton with multiple accepting (final) states. Formally, a Buchi automaton is a five tuple $\langle \Sigma, Q, \Delta, Q^0, F \rangle$ such that [45]: (1) Σ is the infinite alphabet, (2) Q is the finite set of states, (3) $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, (4) $Q^0 \subseteq Q$ is the set of initial states, and (4) $F \subseteq Q$ is the set of accepting states. A more detailed explanation about the automata theory and the creation of the Buchi automatons from LTL formulas is out of scope of this article and we refer the interested readers to [31, 39, 45].

Figure 11 shows a graph representation of the automaton which is generated for the *response* constraint [31].[2]. The set of nodes of the graph corresponds to the set of states in the automaton. The set of edges in the graph corresponds to the transition relation. 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 Figure 11 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.

The mapping for LTL constraints onto Buchi 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, it is possible indicate whether it is still possible to reach an accepting

[2] Note that based on the LTL expression typically a non-deterministic Buchi automaton is generated. To simplify the presentation, we show a deterministic automaton. However, is is easy to develop a "workflow engine" able to deal with non-determinism by simply using a *set* of possible current states.

state. This way we can color the constraints *green* (in accepting state), *yellow* (accepting state can still be reached), or *red* (accepting state can not be reached anymore). Using the Buchi automaton some engine could even enforce a constraint.

As indicated in the introduction, we would like to link DecSerFlow to our work on process mining. Note that essentially a DecSerFlow specification is a set of rules, i.e., we could try to discover these rules by observing web services. Moreover, given a DecSerFlow specification we can also check whether each party involved in a choreography actually sticks to the rules agreed upon. Within the ProM framework there is a so-called LTL-checker. This LTL-checker can be used to check DecSerFlow specification relative to some execution log. In the next section, we will elaborate on these two aspects of process mining, i.e., process discovery and conformance checking.

4 Process Mining

In [5] we showed that it is possible to translate abstract BPEL into Petri nets and SOAP messages exchanged between services into event logs represented using our MXML format (i.e., the format used by our process mining tools). As a result we could compare the modeled behavior (in terms of a Petri net) and the observed behavior (in some event log). This comparison could be used for monitoring deviations and to analyze the most frequently used parts of the service/choregraphy. In this chapter, we would like to take a broader perspective without going into too much detail.

For process mining it is crucial that it is possible to obtain an event log giving insight into the *actual* processes as they are carried out. Based on our experiences reported in [5] this is possible, but not completely trivial. For example, when using Oracle BPEL it is possible to monitor SOAP messages using TCP Tunneling technique. In the context of WebSphere one can use IBM's Data Collector logging the content and context of SOAP messages and then load them into the Web Services Navigator [63]. 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.

Note that the collection of event logs can take place different levels. One can log the interactions of a *single service*. However, one can also log all interactions in a *choreography*. In the latter case it is essential have an infrastructure allowing for this (e.g., all interactions going through a central messaging service).

4.1 Process Discovery

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 1. Such a log could have been obtained by monitoring the SOAP messages the *shipper* service in Figure 3 exchanges with it its environment. Note that we do not show the content of the massage. Moreover, we do not show additional header information (e.g., information about sender and receiver).

Table 1. 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	...

Using process mining tools such as ProM it is possible to discover a process model as shown in Figure 12. The figure shows the result of three alternative process discovery algorithms: (1) the alpha 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 Figure 3. Note that Figure 12 shows the names of the messages rather than the activities because this is the information shown in Table 1.

4.2 Conformance Checking

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

[3] Note that ProM allows for the mapping from one format to the other if needed. Figure 12 shows the native format of each of the three plug-ins.

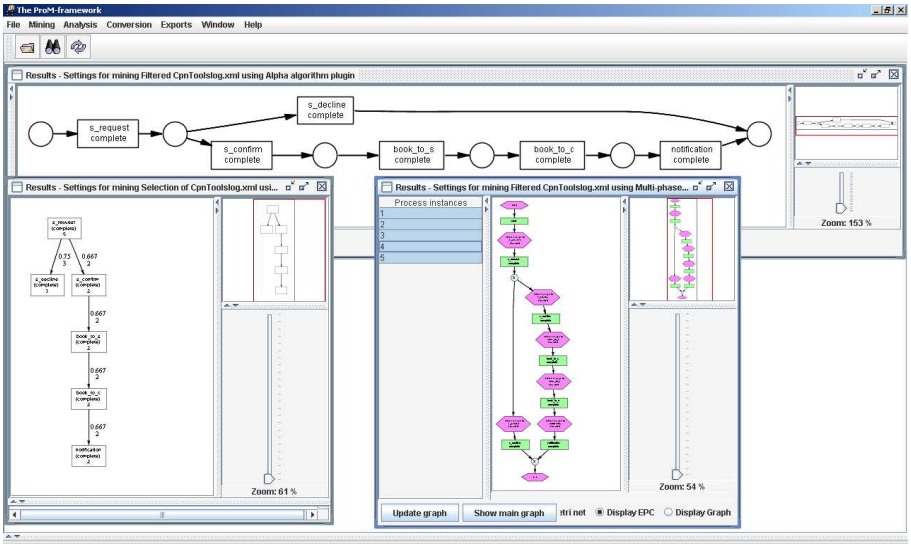


Fig. 12. The output of three process discovery algorithms supported by ProM when analyzing the event log shown in Table 1.

log. However, in many applications there is some a-priori model. For example, we can have some abstract BPEL process, a Petri net specification like in Figure 3, or some more declarative specification (e.g., a DecSerFlow model). If there is such an a-priori model, it is interesting the compare it with the event log. This is what we call *conformance checking*.

To illustrate this, assume that we add an additional process instance to Table 1 where the notification is sent before the book is shipped to the customer (i.e., in Figure 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 13 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 [67] it is shown that it is possible to quantify fitness as a measure between 0 and 1. Unfortunately, a good fitness only does not imply conformance, e.g., it is easy to construct Petri nets that are able to parse any event log. Although such Petri nets have a fitness of 1 they do 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". The ProM Conformance Checker supports both the notion of fitness and the notion of appropriateness.

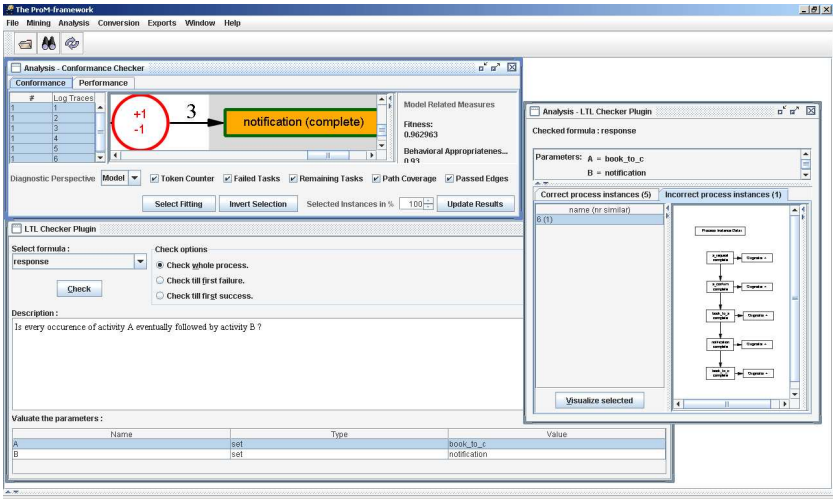


Fig. 13. Both the conformance checker plug-in and the LTL checker plug-in are able to detect the deviation.

The DecSerFlow language presented in this chapter shows that one can also think of processes in a more declarative manner, i.e., as something satisfying a set of LTL formulas. To illustrate this Figure 13 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, i.e., a declarative style of modeling fits well with conformance checking.

5 Related Work

Since the early nineties, workflow technology has matured [30] and several textbooks have been published, e.g., [6, 22]. 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 for the modeling of workflows [6, 19, 22] but also the orchestration of web services [56]. Like most proprietary languages and standards, Petri nets are highly procedural. This is the reason we introduced the DecSerFlow language in this paper.

Several attempts have been made to capture the behavior of BPEL [16] in some formal way. Some advocate the use of finite state machines [27], others process algebras [26], and yet others abstract state machines [25] or Petri nets [61, 53, 70, 74]. (See [61] for a more detailed literature review.) For a detailed analysis of BPEL based on the workflow patterns [7] we refer to [76]. Few approach go into the other direction, e.g., translating (Colored) Petri nets into BPEL [8].

Clearly, this chapter builds on earlier work on process discovery, i.e., the extraction of knowledge from event logs (e.g., process models [13, 15, 20, 28, 29, 38] or social networks [10]). For example, the well-known α algorithm [13] can derive a Petri net from an event log. In [5] we used the conformance checking techniques described in [67] and implemented in our ProM framework [21] and applied this approach to SOAP messages generated from Oracle BPEL. The notion of conformance has also been discussed in the context of security [9], business alignment [1], and genetic mining [57].

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 [12] and a survey paper [11]. Process mining can be seen in the broader context of Business (Process) Intelligence (BPI) and Business Activity Monitoring (BAM). In [34, 35, 68] a BPI toolset on top of HP's Process Manager is described. The BPI toolset includes a so-called "BPI Process Mining Engine". In [60] 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) [44]. The latter tool is commercially available and a customized version of PPM is the Staffware Process Monitor (SPM) [72] which is tailored towards 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 `http://ws.apache.org/axis/`. [47] introduces an assertion language for expressing business rules and a framework to plan and monitor the execution of these rules. [17] 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 [51]. This approach uses event calculus to specify requirements. [50] is an approach based on WS-Agreement defining the Crona framework for the creation and monitoring of agreements. In [33, 23], 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 [63] 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 [4] we suggested to focus less on languages like BPEL and more on questions related to the monitoring of web services.

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 [75]. Axel Martens et al. [53–55, 69] have explored questions related to design-time conformance and compatibility using a Petri-net-based approach. For example, [54] focuses on the problem of consistency between executable and abstract processes and [55] presents an approach where for a given composite service the required other services are generated.

6 Conclusion

This chapter presented *service flows* in both a more traditional and a more pioneering setting.

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). Moreover, it is possible to translate graphical languages such a Petri nets to BPEL (cf. WorkflowNet2BPEL4WS).

Although the first author has been involved in the development of these tools and that 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 both process discovery and conformance checking are useful in the setting of web services. Moreover, we showed that the declarative nature of DecSerFlow fits well with the conformance-checking techniques currently implemented in ProM (cf. the LTL checker plug-in).

To conclude we would like to mention that all of the presented analysis and translation tools can be downloaded from various websites: `www.processmining.org` (ProM), `www.bpm.fit.qut.edu.au/projects/babel/tools/` (BPEL2PNML and WofBPEL), and `www.daimi.au.dk/~krell/WorkflowNet2BPEL4WS/` (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, 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.
4. 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.
5. 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.
6. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, MA, 2002.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.

16. 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.
17. 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.
18. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001.
19. 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.
20. 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.
21. B. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. 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.
28. 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.

29. 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.
30. 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.
31. 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.
32. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. IEEE Computer Society Press, Providence, 2001.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.
38. 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.
39. G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
40. G.J. Holzmann. An Analysis of Bitstate Hashing. *Form. Methods Syst. Des.*, 13(3):289–307, 1998.
41. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, USA, 2003.
42. G.J. Holzmann and D. Peled. An Improvement in Formal Verification. In *FORTE 1994 Conference*, Bern, Switzerland, 1994.
43. 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.
44. 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.
45. E.M. Clarke Jr., O. Grumberg, and D.A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts and London, UK, 1999.

46. 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.
47. 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.
48. F. Leymann. Web Services Flow Language, Version 1.0, 2001.
49. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999.
50. 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.
51. 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.
52. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
53. 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.
54. 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.
55. 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.
56. 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.
57. 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.
58. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.
59. M. zur Muehlen. *Workflow-based Process Controlling: Foundation, Design and Application of workflow-driven Process Information Systems*. Logos, Berlin, 2004.
60. 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.
61. C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, , and H.M.W. Verbeek. Formal Semantics and Analysis of Control Flow in WS-BPEL. BPM Center Report BPM-05-15, BPMcenter.org, 2005.

62. 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.
63. 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.
64. 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.
65. A. Puri and G.J. Holzmann. A Minimized automaton representation of reachable states. In *Software Tools for Technology Transfer*, volume 3. Springer Verlag, 1993.
66. 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.
67. 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.
68. 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.
69. 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.
70. C. Stahl. Transformation von BPEL4WS in Petrinetze (In German). Master's thesis, Humboldt University, Berlin, Germany, 2004.
71. S. Thatte. XLANG Web Services for Business Process Design, 2001.
72. TIBCO. TIBCO Staffware Process Monitor (SPM). http://www.tibco.com, 2005.
73. 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.
74. 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.
75. 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.
76. 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.
77. P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. 5th Int. Conference on Computer Aided Verification*, pages 59–70, 1993.
78. 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.