Conformance Checking of Service Behavior

WIL M.P. VAN DER AALST
Eindhoven University of Technology and Queensland University of Technology
MARLON DUMAS and CHUN OUYANG
Queensland University of Technology
and
ANNE ROZINAT and ERIC VERBEEK
Eindhoven University of Technology

A service-oriented system is composed of independent software units, namely services, that interact with one another exclusively through message exchanges. The proper functioning of such system depends on whether or not each individual service behaves as the other services expect it to behave. Since services may be developed and operated independently, it is unrealistic to assume that this is always the case. This paper addresses the problem of checking and quantifying how much the actual behavior of a service, as recorded in message logs, conforms to the expected behavior as specified in a process model. We consider the case where the expected behavior is defined using the BPEL industry standard (Business Process Execution Language for Web Services). BPEL process definitions are translated into Petri nets and Petri net-based conformance checking techniques are applied to derive two complementary indicators of conformance: *fitness* and *appropriateness*. The approach has been implemented in a toolset for business process analysis and mining, namely ProM, and has been tested in an environment comprising multiple Oracle BPEL servers.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*; *Distributed Debugging*; *Monitors*; H.4.1 [**Information Systems Applications**]: Office Automation—*Workflow Management*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed Systems*; *Information Networks*

General Terms: Languages, Measurement, Theory, Verification

Additional Key Words and Phrases: Web services, Conformance, BPEL, Petri nets, ProM

1. INTRODUCTION

A service-oriented system is composed of software services that interact with one another for a given purpose. To ensure that this purpose is attained, designers

Corresponding author's address: W.M.P. van der Aalst, Department of Mathematics & Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
E-mail addresses of authors: {w.m.p.v.d.aalst,a.rozinat,h.m.w.verbeek}@tue.nl and {c.ouyang,m.dumas}@qut.edu.au.

The development of ProM is supported by EIT, NWO-EW, the Technology Foundation STW, and the IOP program of the Dutch Ministry of Economic Affairs. This work was also funded by an Australian Research Council (ARC) Discovery Grant (DP0451092).

specify these interactions and their dependencies in some form. In principle, the participating services are implemented, adapted, or configured to comply with this specification. However, services may be developed, operated, and evolved by independent teams or organizations. Thus, there is no guarantee that once a system is under operation, some services will not deviate from the specification. For example, after sending a request, a service may receive a reply of the wrong type, a service may reject a message sent by another service, messages may be received out of order, etc. Furthermore, each of these unexpected behaviors may cascade into other deviations. In general terms, service independence raises the question of *conformance*: "Do all services in a service-oriented system operate as expected?".

This paper addresses the following question: Given an expected service behavior captured as one or several *process models*, and an observed behavior as registered in a *message log*, does the observed behavior conform to the expected behavior?

We use the term *service choreography* to refer to a specification of the expected behavior of an individual service or collection of services. Choreographies may be captured in a number of languages. In this paper, we consider a standard language for service behavior specification, namely the Business Process Execution Language for Web Services (BPEL) [Jordan et al. 2006], but the results can well be applied to other languages. Also, the paper assumes that messages are represented according to the XML and SOAP standards [Box et al. 2000], but the proposed techniques could be applied to other message formats. Finally, the paper focuses on checking the fulfillment of control flow dependencies captured in the choreography. We do not consider the issue of checking that each individual message conforms to its expected message type as this is a well-understood problem.

When a choreography and a message log do not conform, two scenarios are possible. First of all, the model may be assumed to be correct because it represents the way partners should work, and the question is whether the events in the log are consistent with the process model. For example, the log may contain event sequences that are not possible according to the model. This may indicate violations of the choreography. Second, the event log may be assumed to be "correct" because it is what really happened. In the latter case the question is whether the choreography that has been agreed upon is no longer valid and should be modified. In this paper, we provide techniques for addressing both of the above scenarios.

The following examples illustrate the need for conformance checking of service behavior by comparing (service) choreographies with message logs.

—To manage its interactions with suppliers and carriers, a large retail company (LRC) has set up a number of logistics services. One of these services is responsible for tracking shipments. This service is activated when a supplier sends a "request for routing instructions" for a particular replenishment order. The service gathers details from a human operator, and responds to the supplier with the corresponding "routing instructions" indicating which carrier should be used and where should the products be shipped. Subsequently, the supplier interacts with the nominated carrier. Once the shipment has been picked up by the carrier from the supplier's premises, the supplier is expected to send an "Advanced Shipment Notification" (ASN) to LRC's shipment tracking service. Subsequently, the carrier may send a number of "shipment status notifications" to LRC. Normally, such status notifications would only be received by LRC after a corresponding ASN has been received. However, some suppliers may be late in sending their ASN or may fail to send it altogether, or messages may come out of order. By checking the conformance of the actual message logs with the "ideal choreography", LRC may detect and quantify such deviations.

- Consider a supplier's order management service and the corresponding procurement service on the customer's end. The customer starts an interaction by placing an order. The supplier acknowledges through an initial order response, possibly followed by one or several order updates. The customer can change or cancel the order under some circumstances. However, requests for changes or cancellations to an order cannot be accepted once the supplier has issued an "order confirmation". Also, the supplier expects that the customer will not send requests for changes or cancellations for orders which have not yet been acknowledged through an initial order response. By using conformance checking, the buyer and/or the supplier may find out if their services actually follow this agreed-upon choreography. Later in the paper, we will consider a variant of this example to illustrate our approach and its corresponding tools.

The above examples hint to a possible link between Service Level Agreements (SLAs) and Quality of Service (QoS) monitoring on the one hand, and conformance checking on the other. However, SLAs and QoS metrics typically focus on performance indicators such as time, failure rate or cost of activities and processes [Cardoso et al. 2004]. Such indicators assume that the process conforms to some predefined model. Thus, conformance checking of service behavior is complementary to SLA and QoS monitoring.

This paper addresses the problem of choreography conformance checking by building on earlier work on checking the conformance of a formal model (e.g., a Petri net) with respect to a set of traces. To link this work to the technologies currently used in service-oriented systems, we provide a mapping from BPEL to Petri nets and a mapping from SOAP messages to event logs.

Figure 1 illustrates the contributions and the structure of the paper. Section 2 gives an overview of the approach used in this paper. Section 3 elaborates on the notion of conformance and introduces the ProM Conformance Checker. Then, in Section 4, we discuss the mapping of BPEL onto WF-nets [Ouyang et al. 2005a; Ouyang et al. 2005b], a subclass of Petri nets. Section 5 discusses ways of extracting high-level event logs from SOAP message logs. Section 6 describes a case study demonstrating the feasibility of our approach and the tools we have developed. Related work is discussed in Section 7 and Section 8 concludes the paper.

2. OVERVIEW OF APPROACH

The goal of choreography conformance checking is to verify that all parties behave as expected, i.e., to answer the question "Does the service behavior match the service specification?". Both the service behavior and the service specification refer to occurrences of activities. Hence, choreography conformance checking suggests capabilities with respect to observing activities. Depending on the setting, the activities themselves may be recorded or it is just possible to see the messages being exchanged. Middleware products such as IBM's Websphere, Oracle

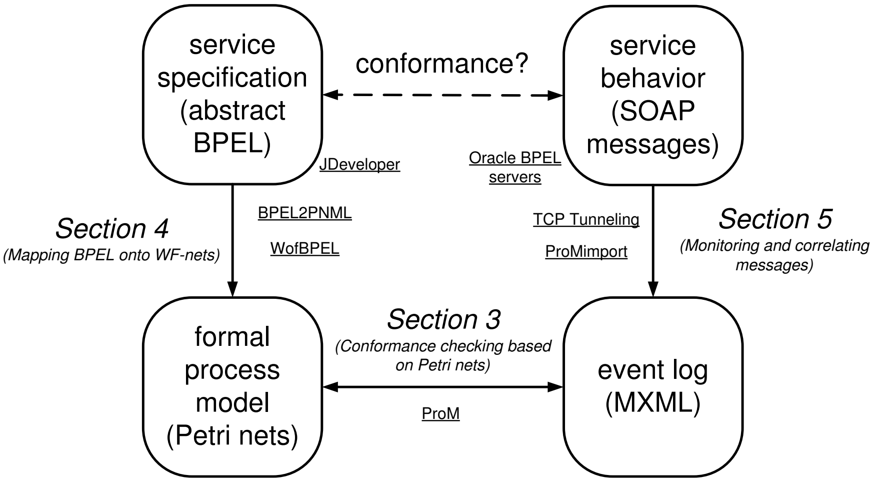

service specification (abstract BPEL)
conformance?
service behavior (SOAP messages)
JDeveloper
Oracle BPEL servers
BPEL2PNML
WofBPEL
Section 4
(Mapping BPEL onto WF-nets)
TCP Tunneling
ProMimport
Section 5
(Monitoring and correlating messages)
formal process model (Petri nets)
Section 3
(Conformance checking based on Petri nets)
ProM
event log (MXML)

Fig. 1. Outline of the paper showing the role of the core sections of the paper and the tools used.

BPEL, and Colombo, maintain detailed logs of activities. However, in many cases only SOAP messages can be observed. Fortunately, it is typically possible to link SOAP messages to activities and implicitly derive activity occurrences. Message exchanges and activity occurrences may be seen from the perspective of a global observer or from that of a local observer. The global observer sees activities and/or messages at the process level, i.e., involving multiple services. A local observer only sees the activities and/or messages related to a single service. Based on these two viewpoints (activities/messages and global/local) at least *four possible settings for choreography conformance checking* can be derived: (a) relevant messages exchanged between all services involved in a choreography are visible, (b) relevant activities executed inside all services involved in a choreography are visible, (c) relevant messages sent or received by a single service are visible, and (d) relevant activities executed within a single service are visible. In this paper, we will focus on the third setting (observing messages locally). Moreover, we assume that the service specification is expressed in terms of abstract BPEL [Jordan et al. 2006]. However, our ideas and techniques are applicable to all four settings and do not depend on the use of BPEL.

Figure 2 describes the approach proposed in this paper. Based on a process model described as an abstract BPEL process (i.e., the service specification), we generate a Petri net [Desel et al. 2004]. We use a translation described in [Ouyang et al. 2005a; Ouyang et al. 2005b] and implemented in a tool called BPEL2PNML[1]. We also propose an approach to monitor and to correlate SOAP messages in order to construct events logs. Conformance checking is performed by comparing the obtained event logs with the Petri net. The reason for using Petri nets as a intermediate step is that it is simpler to check the conformance of a simple formal model than checking the conformance of a complicated language like BPEL.

The paper considers two notions of conformance: *fitness* and *appropriateness.*

[1] Documentation and software available from www.bpm.fit.qut.edu.au/projects/babel/tools/.

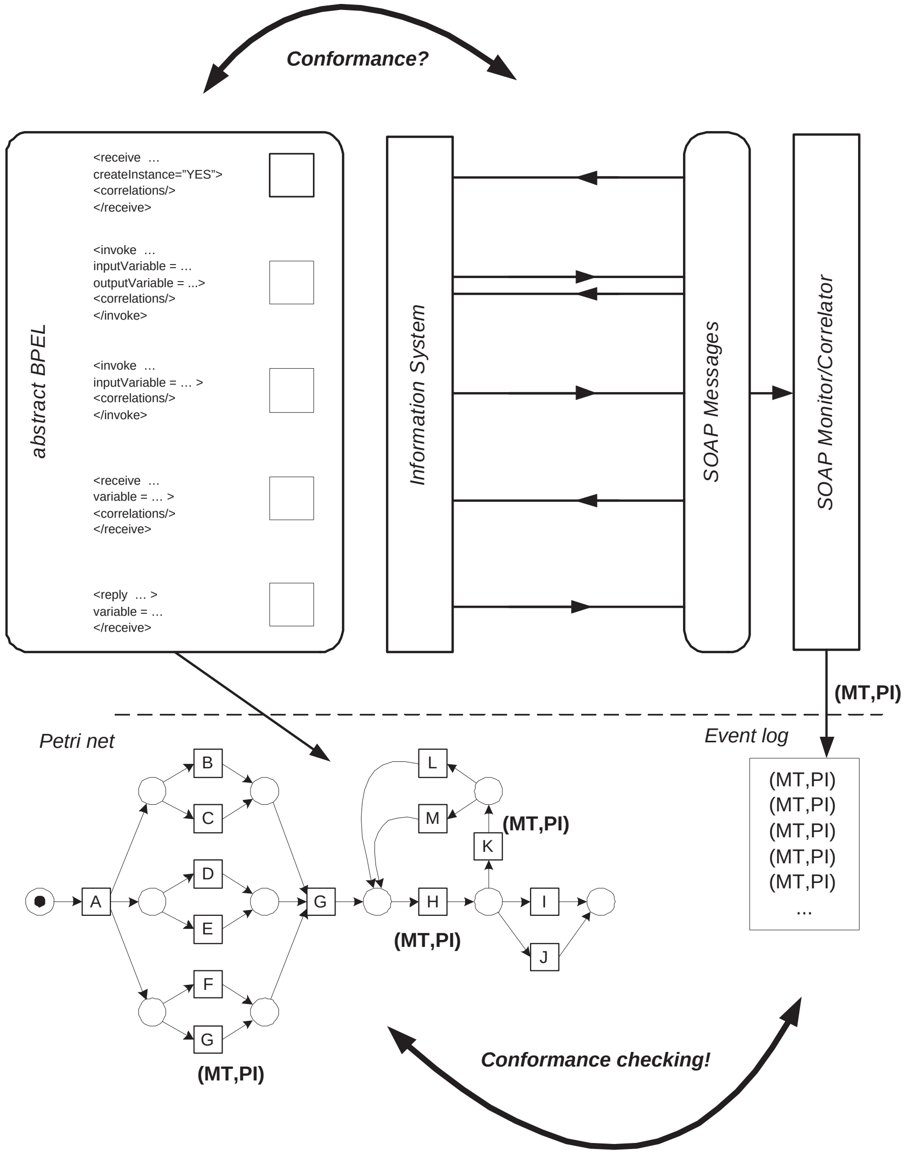


Fig. 2. Overview of the approach. The top level shows the process model in BPEL and the recorded behavior in the form of SOAP messages. The BPEL specification is mapped onto Petri nets and the SOAP messages are put in an event log. Finally, both are compared using conformance checking.

An event log and a 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 extracted from the message logs. In [Rozinat and Aalst 2006] it is shown that it is possible to quantify fitness, e.g., an event log and Petri net may have a fitness of 0.66 indicating that 66 percent of the events in the log are possible according to the model. Unfortunately, high fitness does not imply conformance: It is easy to construct Petri nets that are able to parse any event log. Although such Petri nets have a fitness of 1.0, they do not provide meaningful information about the service's behavior. This is why we consider a second dimension, namely appropriateness. Appropriateness captures the idea of *Occam's razor*: "one should not increase, beyond what is necessary, the number of entities required to explain anything". A model is appropriate if it is the "simplest" one explaining the observed behavior. Thus, overfitting and underfitting models are avoided.

The techniques proposed in this paper are implemented in a tool called *Conformance Checker*. This tool is integrated into the *ProM framework*[2]. Although ProM offers a wide range of tools related to process mining [Aalst et al. 2003] (e.g., LTL checking, process discovery, verification, etc.), in this paper we focus on ProM's Conformance Checker and its application to monitoring services.

For conformance checking, it is crucial that each event recorded in the log can be linked to (i) a *process instance* (also called a *case*) and (ii) *a process model element* (e.g., an activity in BPEL terms or a transition in Petri-net terms).[3] In Figure 2 this is indicated by the pairs *(MT,PI)*. *PI* refers to a specific process instance, i.e., a unique identifier of the case being processed. Examples of a *PI* are a customer id, customer order reference, a social security number, a patient id, etc. *MT* is the message type that can be linked to some activity *AT*. Examples of an *MT* are "request for information", "approval message", and "decline offer". Note that, depending on the setting, it may be possible to directly capture activity occurrences and have events of the form *(AT,PI)* where *AT* refers to an activity. When middleware products such as IBM's Websphere and Oracle BPEL are being used this is realistic. Otherwise, it is likely that only messages can be intercepted and events are of the form *(MT,PI)*. In this paper we focus on the latter situation.

It may seem trivial to capture events of the form *(MT,PI)* or *(AT,PI)*. In the presence of process-aware information systems such as workflow management systems (e.g., Staffware, Filenet, FLOWer, etc.) and dedicated middleware products (e.g., MQSeries and Oracle BPEL) it is indeed easy to extract the desired information. However, in many other situations this turns out to be much more complicated. Section 5 discusses different ways of extracting event logs from SOAP message logs.

In an ideal situation the abstract BPEL process and the observed messages conform (a precise definition will be given in the next section). However, there may be discrepancies between the actual service behavior recorded in the event log and the service specification represented in BPEL. There are two possible causes for non-conformance: (1) the service implements a process different from the specification given by the abstract BPEL process; and (2) the environment behaves different from what could be expected based on the specification given by the abstract BPEL process. In the remainder, we will show that it is indeed possible to measure conformance and track down discrepancies between the abstract BPEL process and the observed message exchanges. Although, we have implemented this in the context of BPEL, other languages for service interaction specification could be used. The only

[2] Documentation and software available from `www.processmining.org`.

[3] This requirement is also found in process mining techniques [Aalst et al. 2003] (e.g., the α algorithm [Aalst et al. 2004]).

requirement is that these languages should be suitable for describing all message exchanges between the services involved in the choreography, and there should be a mapping from the control-flow subset of that language to Petri nets.

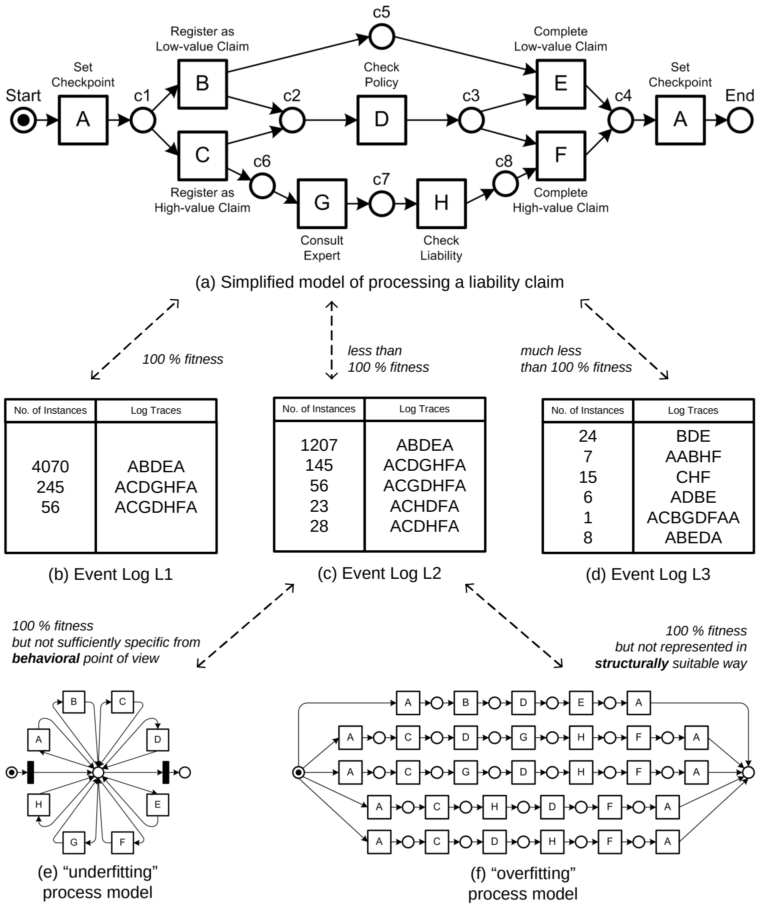
As shown earlier in Figure 1, the next three sections present how conformance checking can be applied to service behavior. First, we present an approach to do conformance checking given a Petri net and an event log (Section 3). Then we show a mapping from abstract BPEL to Petri nets (Section 4) and discuss the various ways in which service behavior (e.g., SOAP messages) can be captured and mapped onto a format suitable for conformance checking (Section 5).

3. CONFORMANCE CHECKING BASED ON PETRI NETS

The starting point for conformance checking is the presence of both an explicit process model, describing how some business process *should be* executed, and some kind of event log, giving insight into how it *was actually* carried out. Clearly, it is interesting to know whether they conform to each other. In [Rozinat and Aalst 2006] this question has been explored using Petri nets to represent process models [Desel et al. 2004], and assuming some abstract event log where log events are only expected to (i) refer to an activity from the business process, (ii) refer to a case (i.e., a process instance), and (iii) be totally ordered. As indicated before, we assume that messages can be associated to activities and cases. Moreover, there is no interaction among cases. Therefore, we can assume that an event log is simply a multiset of activity traces, i.e., each case refers to a sequence of activities and a log is simply a collection of such sequences. By mapping BPEL onto Petri nets and SOAP messages onto multisets of traces, we can focus on the core idea of conformance checking.

We have identified two dimensions of conformance, *fitness* and *appropriateness* [Rozinat and Aalst 2006]. Fitness relates to the question whether the observed process behavior complies with the control flow specified by the process model, while appropriateness can be used to evaluate whether the model describes the observed process in a suitable way (cf. Occam's razor as discussed in Section 1).

To illustrate both dimensions of conformance we use the example process shown in Figure 3(a). The process is represented as a Petri net [Desel et al. 2004]. The squares in Figure 3(a) are transitions and represent activities. The circles are places and represent pre- and post-conditions (i.e., partial states). In a Petri net, places may hold tokens. The marking of a Petri net is the distribution of tokens over places (i.e., the state). The network structure is static while the number of tokens and their location may change. A transition is enabled if there is a token on each of its input places. A transition may fire if it is enabled. Firing implies removing tokens from the input places and producing tokens for the output places. In Figure 3(a), transition A is enabled. Firing A implies moving a token from place *Start* to place *c1*, etc. Note that there are two transitions bearing the same label "Set Checkpoint". Each of these two transitions represents an activity that can be thought of as an automatic backup action within the context of a transactional system, i.e., activity A is carried out at the beginning to define a rollback point enabling atomicity of the whole process, and at the end to ensure durability of the results. Then, the actual business process is started with the distinction of low-value


c5
Register as Low-value Claim
Complete Low-value Claim
Set Checkpoint
Check Policy
Set Checkpoint
Start
A
c1
B
c2
D
c3
E
c4
A
End
C
F
c6
c8
c7
G
H
Register as High-value Claim
Complete High-value Claim
Consult Expert
Check Liability

(a) Simplified model of processing a liability claim

100 % fitness
less than 100 % fitness
much less than 100 % fitness

No. of Instances	Log Traces
4070	ABDEA
245	ACDGHFA
56	ACGDHFA

(b) Event Log L1

No. of Instances	Log Traces
1207	ABDEA
145	ACDGHFA
56	ACGDHFA
23	ACHDFA
28	ACDHFA

(c) Event Log L2

No. of Instances	Log Traces
24	BDE
7	AABHF
15	CHF
6	ADBE
1	ACBGDFAA
8	ABEDA

(d) Event Log L3

100 % fitness but not sufficiently specific from **behavioral** point of view
100 % fitness but not represented in **structurally** suitable way
B
C
A
D
H
E
G
F
A B D E A
A C D G H F A
A C G D H F A
A C H D F A
A C D H F A

(e) "underfitting" process model

(f) "overfitting" process model

Fig. 3. Two dimensions of conformance: fitness and appropriateness.

claims and high-value claims, which get registered differently (B or C). The policy of the client is always checked (D) but in the case of a high-value claim, additionally, the consultation of an expert takes place (G), and then the filed liability claim is being checked in more detail (H). Finally, the claim is completed according to the former choice between B and C (i.e., E or F).

Figures 3(b)-(d) show three example logs for the process described in Figure 3(a) at an aggregate level. This means that process instances exhibiting the same event sequence are combined as a logical log trace while recording the number of instances to weigh the importance of that trace. Note that each of the logs shown in Figures 3 represents a multiset of traces, e.g., log $L1$ contains 4070 occurrences of (A, B, D, E, A), 245 occurrences of (A, C, D, G, H, F, A), and 56 occurrences of (A, C, G, D, H, F, A). An event log can be viewed as a multiset of traces since only the control flow perspective is considered here. In a different setting like, e.g., mining social networks, the resources performing an activity would distinguish those instances from each other.

Event log *L1* completely *fits* the model in Figure 3(a) as every log trace can be associated with a valid path from *Start* to *End*. In contrast, event log *L2* does not match completely as the traces *ACHDFA* and *ACDHFA* lack the execution of activity *G*, while event log *L3* does not contain any trace corresponding to the specified behavior.

Now consider the two process models shown in Figure 3(e)-(f). Although event log *L2* fits both models quantitatively, i.e., the event streams of the log and the model can be matched perfectly, they do not seem to be *appropriate* in describing the observed behavior. The first one is much too generic ("underfitting") as it covers a lot of extra behavior, allowing for arbitrary sequences containing the activities *A*, *B*, *C*, *D*, *E*, *F*, *G*, or *H*, while the latter—although it does not allow for more sequences than those that were observed in the log—only lists the possible behavior instead of expressing it in a meaningful way ("overfitting"). Note that such underfitting and overfitting models could be constructed for any log, e.g., also *L1* and *L3* in Figure 3. Therefore, these extremes do not offer a better understanding than can be obtained by just looking at the aggregated log. So, there is also a qualitative dimension and we claim that a "good" process model should somehow be minimal in structure to clearly reflect the described behavior (i.e., *structural appropriateness*), and minimal in behavior to represent as closely as possible what actually takes place (i.e., *behavioral appropriateness*).

Conformance checking aims at both quantifying the respective dimension of conformance and locating the mismatch, if any. We have developed metrics for measuring the fitness and the behavioral and structural appropriateness of a given process model and event log [Rozinat and Aalst 2006]. But we also seek for suitable visualizations of the results as this is crucial to understand the sources of mismatches.

For example, we can quantify fitness by replaying the log in the model. For this, the replay of every log trace starts with marking the initial place in the model and then the transitions that belong to the logged events in the trace are fired one after another. While doing so, one counts the number of tokens that had to be created artificially (i.e., the transition belonging to the logged event was not enabled and therefore could not be *successfully executed*) and the number of tokens that were left in the model (they indicate that the process has not *properly completed*). Only if there were neither tokens left nor missing, the fitness measure evaluates to 1.0, which indicates 100% fitness.

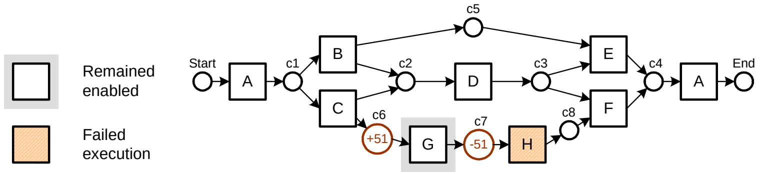


Fig. 4. Example process model after replay of event log *L2*.

Figure 4 shows that the places of missing and remaining tokens during log replay can also be used to provide insight into the location of error. Because of the remaining tokens (whose amount is indicated by a + sign) in place *c6* transition G has remained enabled, and as there were tokens missing (indicated by a − sign) in place *c7* transition H has failed seamless execution. This suggests that the expert consultation (activity G) did not take place for all the treated cases, and possible alignment actions would be to either enforce the specified process or to introduce the possibility to skip activity G in the model.

Both dimensions of conformance, i.e., fitness and appropriateness, have been implemented in the ProM Conformance Checker [Rozinat and Aalst 2006]. Note that the checker supports *duplicate activities*, e.g., in Figure 3(a) there are two activities with label A. This is important because multiple activities in a BPEL specification can exchange messages of a given type and are therefore indistinguishable. Similarly, it is important that the Conformance Checker supports *silent steps*, i.e., activities that are not logged. Note that the presence of silent activities makes it necessary to construct parts of the state space to find the most likely path.

4. MAPPING BPEL ONTO WF-NETS

To provide tool support for conformance checking of BPEL processes we rely on two tools developed by the authors of this paper: BPEL2PNML and WofBPEL. BPEL2PNML translates BPEL process definitions into Petri nets represented in the Petri Net Markup Language (PNML). WofBPEL, built using Woflan [Verbeek et al. 2001], applies static analysis and transformation techniques on the output produced by BPEL2PNML. For the purpose of conformance checking, WofBPEL is 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 and is the input format required by the ProM Conformance Checker.

Below, we discuss the mapping from BPEL to WF-nets and illustrate it using a BPEL process definition of a supplier service that we will use as a running example in the remainder of this paper.

4.1 The Supplier Service

Figure 5 provides an overview of a process definition capturing the behavior of a "Supplier Service". This figure uses a visual notation reflecting the syntax of BPEL. This service provides a purchase order and change order service for customers, where the purchase order that has been placed may be changed once.

The Supplier process is initiated upon receiving a purchase order that contains one or several line items. The supplier may accept or reject any ordered item, possibly suggesting alternative products, quantities or delivery dates in the latter case. The supplier replies to the purchase order either with a single response listing the outcome for all items, or with multiple responses corresponding to subsets of the items. The rationale for having multiple responses is that the supplier may be unable to determine outright if it can accept a line item. In this case, the supplier sends a first response listing the items of which the outcomes have been determined. Additional responses are then sent as information becomes available. After receiving an order response, the customer may request to change the previous purchase order because of some item(s) being rejected. A change order is an updated purchase order that overrides the previous one. Similarly to the processing

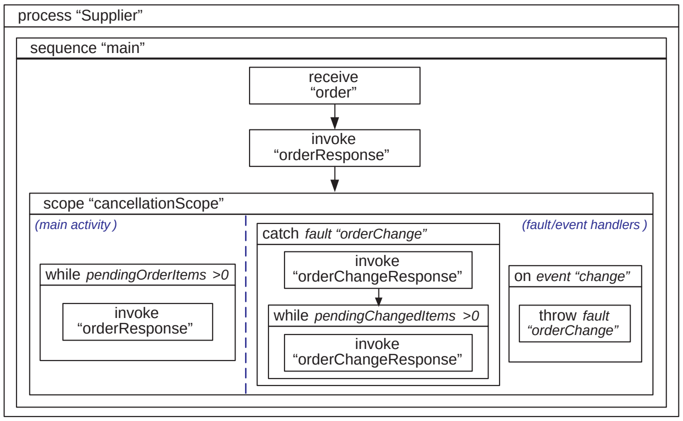

process “Supplier”
sequence “main”
receive “order”
invoke “orderResponse”
scope “cancellationScope”
(main activity)
(fault/event handlers)
while pendingOrderItems >0
invoke “orderResponse”
catch fault “orderChange”
invoke “orderChangeResponse”
while pendingChangedItems >0
invoke “orderChangeResponse”
on event “change”
throw fault “orderChange”

Fig. 5. An abstract view of the Supplier process.

of a purchase order, the supplier may reply with a single response or with multiple responses to a change order.

For the readers that want to see the actual definitions of the Supplier service both as an abstract and as an executable BPEL process, we refer to a technical report where we include the full BPEL specifications [Aalst et al. 2005].[4] An abstract process is defined at the level of abstraction required to capture public aspects of the service (i.e., message exchanges with the environment). In the working example, the abstract process specifies that the service receives orders and change orders and sends order responses and change order responses, and captures the control dependencies between these messages. Meanwhile, an executable process represents a possible implementation of the abstract process. However, services are not always coded as BPEL executable processes.

4.2 Mapping BPEL to Petri Nets

We first map BPEL processes to Petri nets, which can be then converted to WF-nets. When using Petri nets to capture the formal semantics of BPEL, we allow the usage of both labeled and unlabeled transitions. The labeled transitions model events and basic activities. The unlabeled transitions (τ-transitions, also known as *silent steps*) represent internal actions that cannot be observed by external users. This section presents only selected parts of the mapping. A complete version of the formal specification of the mapping can be found in [Ouyang et al. 2005b].

4.2.1 *Activities.* We start with the mapping of a basic activity (X) shown in Figure 6, which also illustrates our mapping approach for structured activities. The net is divided into two parts: one (drawn in solid lines) models the normal processing of X, the other (drawn using dashed lines) models the skipping of X.

In the normal processing part, the four places are used to capture four possible states for the execution of activity X: r_X for “ready” state, s_X for “started” state, c_X for “completed” state, and f_X for “finished” state. The transition labeled X models

[4] This report can be downloaded from `BPMcenter.org`.

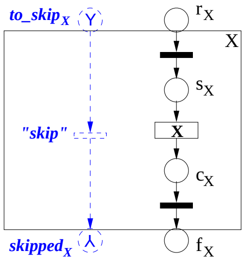


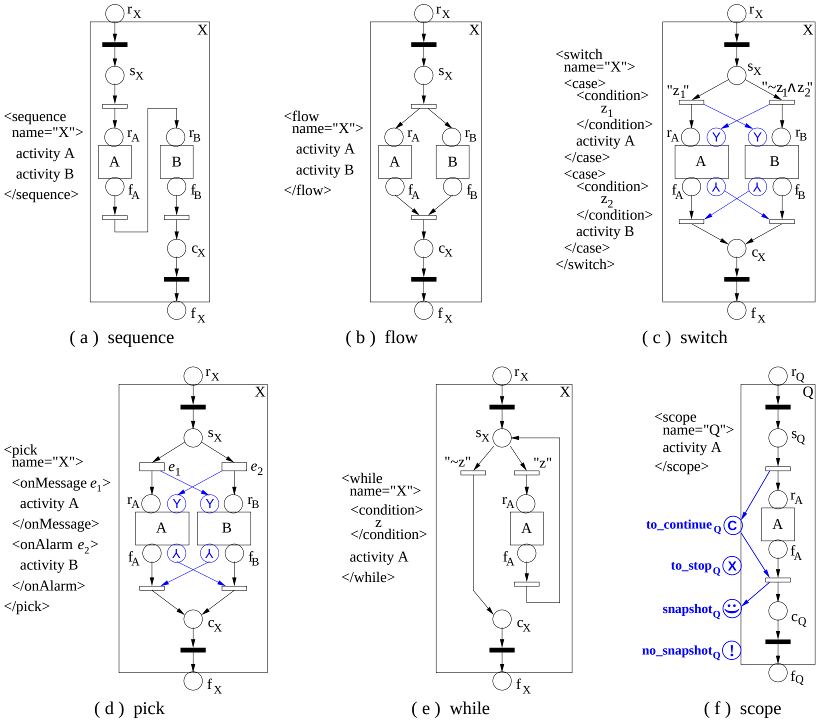
Fig. 6. A Basic activity.

the action to be performed. This is an abstract way of modeling basic activities, where the core of each activity is considered as an atomic action. Two τ-transitions (drawn as solid bars) model silent steps, i.e., internal actions for checking preconditions or evaluating post-conditions for activities. In the mapping of BPEL to Petri nets, we will introduce many silent steps to model the "logical wiring" among transitions representing the actual activities. The skip path in Figure 6 is mainly used to facilitate the mapping of control links. Note that the to_skip and skipped places are respectively decorated by two patterns (a letter Y and its upside-down image) so that they can be graphically identified. In Figure 6, hiding the subnet enclosed in the box labeled X yields an abstract graphic representation of the mapping for activities. This is used in the rest of the paper.

Figure 7 depicts the mapping of structured activities. Next to the mapping of each activity is a BPEL snippet of the activity. More τ-transitions (drawn as hollow bars) are introduced for the mapping of routing constructs. In Figure 7 and subsequent figures, the skip path of the mapping is not shown if it is not used. A detailed description of the mapping [Ouyang et al. 2005b] is outside the scope of this paper. However, to give some insight into the mapping, we describe the mappings of *while* and *scope* activities in some detail.

A *while* activity supports structured loops. In Figure 7(e), activity X has a sub-activity A that is performed multiple times as long as the while condition (z) holds and the loop construct ends if the condition does not hold anymore (∼z).

A *scope* provides event and fault handling. It has a main activity that defines its "normal" behavior. To map fault handling, we define four flags for a scope, each represented by a Petri net place, as shown in Figure 7(f). These flags are: to_continue, indicating the execution of the scope is in progress and no exception has occurred; to_stop, signaling an error has occurred and all active activities nested in the scope need to stop; snapshot, capturing the *scope snapshot* defined in [Jordan et al. 2006] which refers to the preserved state of a successfully completed uncompensated scope; and no_snapshot, indicating the absence of a scope snapshot. Specifically, if a fault occurs during the execution of the normal behavior associated to a scope, it will be caught by one of the fault handlers defined for the scope, and the scope switches from normal "processing" mode to "fault handling" mode. These two modes are represented by places to_continue and to_stop. A scope in which a fault has occurred is considered to have ended abnormally and thus cannot be compensated, even if the fault has been caught and handled successfully. This is represented by places snapshot and no_snapshot. For space reasons, we do not



(a) sequence (b) flow (c) switch

(d) pick (e) while (f) scope

Fig. 7. Mapping structured activities.

describe fault handlers and other advanced constructs. Full details, including a formal definition of the mapping, can be found in a separate technical report [Ouyang et al. 2005b].

4.2.2 *Event Handlers.* A scope can provide *event handlers* that are responsible for handling normal events (i.e., message or alarm events) that occur *concurrently* when the scope is running. Figure 8 depicts the mapping of a scope (Q) with an event handler (EH). The four flags associated with the scope are omitted. The subnet enclosed in the box labeled EH specifies the mapping of EH. As soon as scope Q starts, it is ready *to invoke* EH. Event $\mathsf{e}_{\mathsf{normal}}$ is *enabled* and may occur upon an environment or a system trigger. When $\mathsf{e}_{\mathsf{normal}}$ occurs, an instance of EH is created, in which activity HE ("handling event") is executed. EH remains active as long as Q is active. Finally, event $\mathsf{e}_{\mathsf{normal}}$ becomes disabled once the normal process (i.e., main activity A) of Q is finished. However, if a new instance of EH has already started before $\mathsf{e}_{\mathsf{normal}}$ is disabled, it is allowed to complete. The completion of the scope as a whole is delayed until all active instances of event handlers have completed.

4.2.3 *Example: Mapping of the Supplier Process.* Figure 9 depicts the mapping of the Supplier process shown in Figure 5. The complete mapping of the Supplier process, as obtained using BPEL2PNML, is summarized in Figure 9. This figure sketches the top-level structure including the top-level scope, the fault handler and

Example BPEL code 1:

```
<scope name="Q">
 <eventHandlers>
   <onMessage  e_normal >
     activity HE
   </onMessage>
 </eventHandlers>
    activity A
</scope>
```

Example BPEL code 2:

```
<scope name="Q">
 <eventHandlers>
   <onAlarm   e_normal>
     activity HE
   </onAlarm>
 </eventHandlers>
    activity A
</scope>
```

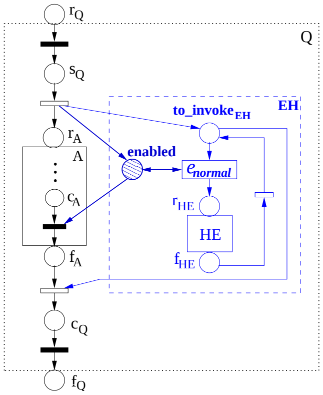


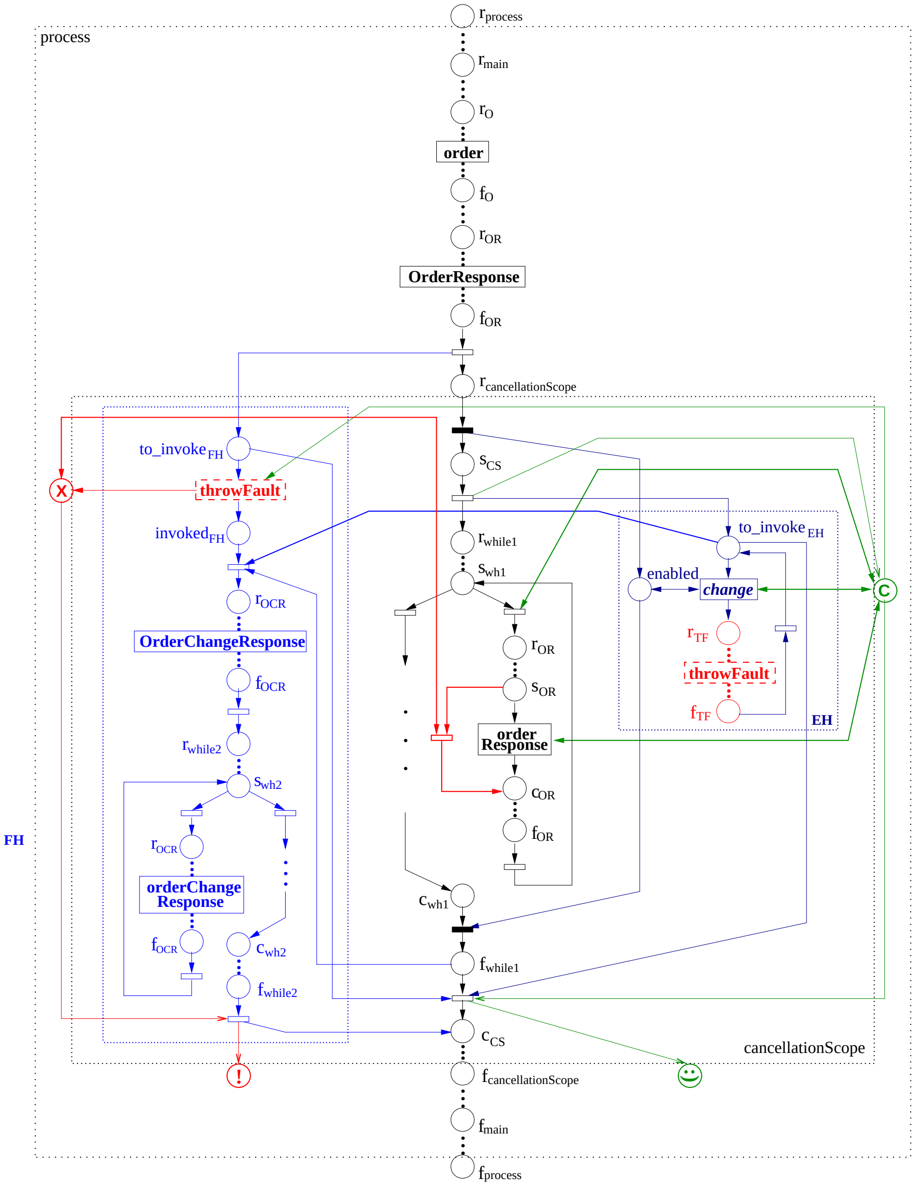
Fig. 8. Mapping event handlers.

the event handler in the process. For illustration purposes, some net details (e.g., those associated to the process scope and the skip paths) are omitted. Also, for the sake of readability the following conventions are used: place to_continue is labeled by a "C", to_stop by an "X", snapshot by a "smiley face" and no_snapshot by an exclamation mark. The reader does not need to understand this diagram in detail. What is important to retain is that any abstract BPEL specification can be mapped onto a Petri net, but in this mapping process a large number of silent steps (i.e., transitions with label τ) are introduced. Next we will show how to remove these and simplify the Petri net prior to performing conformance checking.

4.3 From Petri nets to WF-nets

The ProM Conformance Checker takes a WF-net [Aalst 1998] and an MXML log as input. A WF-net is a Petri net which models a workflow process definition. It has exactly one input place (called source place) and one output place (sink place). A token in the source place corresponds to a case (i.e., process instance) which needs to be handled, and a token in the sink place corresponds to a case which has been handled. Also, in a WF-net there are no dangling tasks and/or conditions. Tasks are modeled by transitions and conditions by places. Therefore, every transition/place should be located on a path from the source place to the sink place in a WF-net [Aalst 1998].

The Petri net obtained from the automated mapping to Petri nets is generally not a WF-net. For example, the Petri net sketched in Figure 9 contains four sink places: one at the bottom of the figure and four along the dotted line labeled "cancellationScope". Such additional source and sink places come from the mapping of constructs that can cause activities to be skipped, namely: control links and fault handlers attached to scopes. In order to facilitate the mapping of control links and fault handlers, and to be consistent in the way structured activities are mapped in BPEL, we have assumed in our mapping that any activity may be skipped. As a result, a skip path is generated for every activity in BPEL2PNML. However, not every activity can actually be skipped. A straightforward counter example is the root activity (i.e., the top-level process scope). By removing these idle skip



[Note]: The concrete action of ‘‘throwFault’’ is modelled by one transition, which is graphically represented by two transitions throwFault to avoid arc crossing.

Fig. 9. Mapping of the Supplier process shown in Figure 5.

fragments, the Petri net obtained from the initial phase of the mapping can be converted to a WF-net.

We use WofBPEL to convert the Petri nets returned by BPEL2PNML to WF-nets. WofBPEL has originally been built to perform analysis on the Petri nets produced as output from BPEL2PNML. Since it uses Woflan [Verbeek et al. 2001] and Woflan can only handle WF-nets, WofBPEL first needs to remove the idle skip fragments to obtain a WF-Net. In addition to this, WofBPEL also applies behavior preserving reduction rules based on the ones given by Murata [Murata 1989]. This way, the size of the net can be significantly reduced by removing unnecessary silent transitions and redundant places. Note that there is a difference between the rules given by Murata and the rules used in WofBPEL. The explanation for this difference is that in our case the non-silent transitions (represented by labeled transitions) should never be removed.

Figure 10 shows the reduction rules used in WofBPEL, where only silent transitions (τ-transitions) can be removed. The first rule shows that a (silent) transition connecting two places may be removed by merging the two places, provided that tokens in the first place can only move to the second place. The second rule shows that multiple alternative silent transitions can be reduced to a single one. Note that after applying the second rule one may be able to apply the first rule provided that the first place has only one remaining output arc (see Figure 10). The third rule shows that self-loops can be removed if the transition involved is silent. When applying the rules one should clearly differentiate between silent and non-silent transitions. For example, in the fourth and fifth rule at least one of the transitions should be silent, otherwise the rule should not be applied (as indicated). In the fourth rule the execution of y is inevitable once the silent transition has been executed. Therefore, it is only possible to postpone its occurrence. In the fifth rule the execution of x is always followed by the silent transition. Note that the silent transition cannot have any additional inputs. Therefore, it is only possible to postpone its occurrence. The two last rules do not remove any transitions but remove places. Transitions X and Y may be or may not be silent. The reduction rules shown in Figure 10 do not preserve the moment of choice and therefore assume trace semantics rather than branching or weak bisimulation semantics [Glabbeek and Weijland 1996].

Based on the above, Figure 11 depicts the WF-net automatically generated from the Supplier BPEL process of Figure 5 using first BPEL2PNML to obtain a Petri net and then WofBPEL to remove idle skip paths and reduce the size of the net. For reference, the Petri net generated by BPEL2PNML for the Supplier process contains 96 places and 84 transitions, while the reduced WF-net contains 27 places and 27 transitions.

5. MONITORING AND CORRELATING MESSAGES

In order to perform conformance checking, we assume that messages sent and received by a service are logged. The resulting logs should be ordered chronologically and should contain for each message, an indication of whether the message is inbound or outbound, as well as the message headers (e.g., HTTP and/or SOAP headers). The message payload is not relevant as we focus on behavioral rather

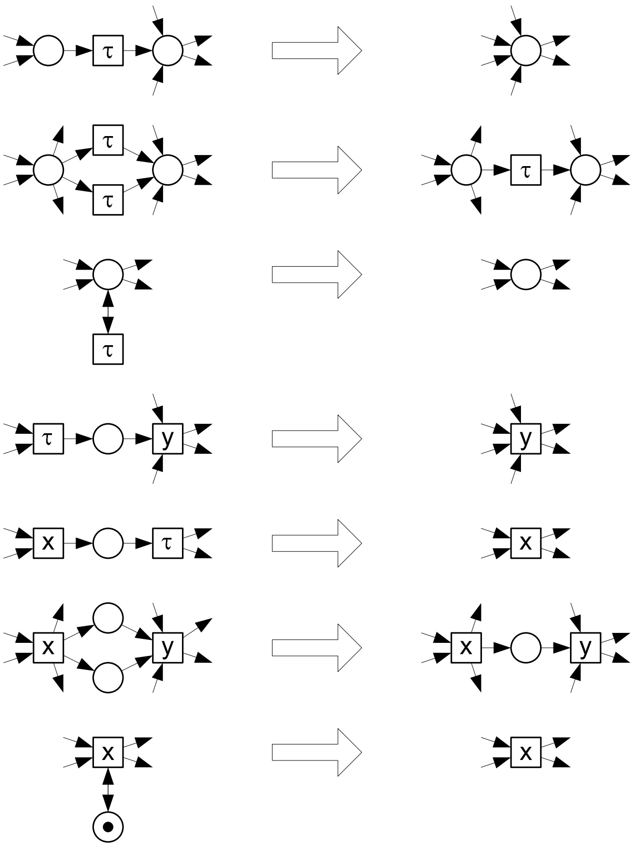


Fig. 10. Behavior preserving reduction rules used in WofBPEL.

than structural conformance.

Given such a message log and a BPEL abstract process definition that is presupposed to correspond to the message log, we need to extract log traces such as those depicted in Figures 3(b)-(d).[5] The labels in these log traces should correspond to labels in the Petri net obtained from the BPEL abstract process definition. These labels must allow one to determine the direction of messages and their message type. Thus, for each message we must determine:

—Its corresponding BPEL abstract process instance (herewith called its *service instance*). This is required because the event log needs to be structured as a set of log traces, each corresponding to one execution of the process capturing the expected behavior of the service.

—A label denoting the BPEL communication action in the abstract process definition to which the production or consumption of the message is attributed.

[5] Note that we will extract more information but this is the bare minimum for conformance checking. The MXML format also allows for the logging of timestamps, data, resources, and transactional aspects.

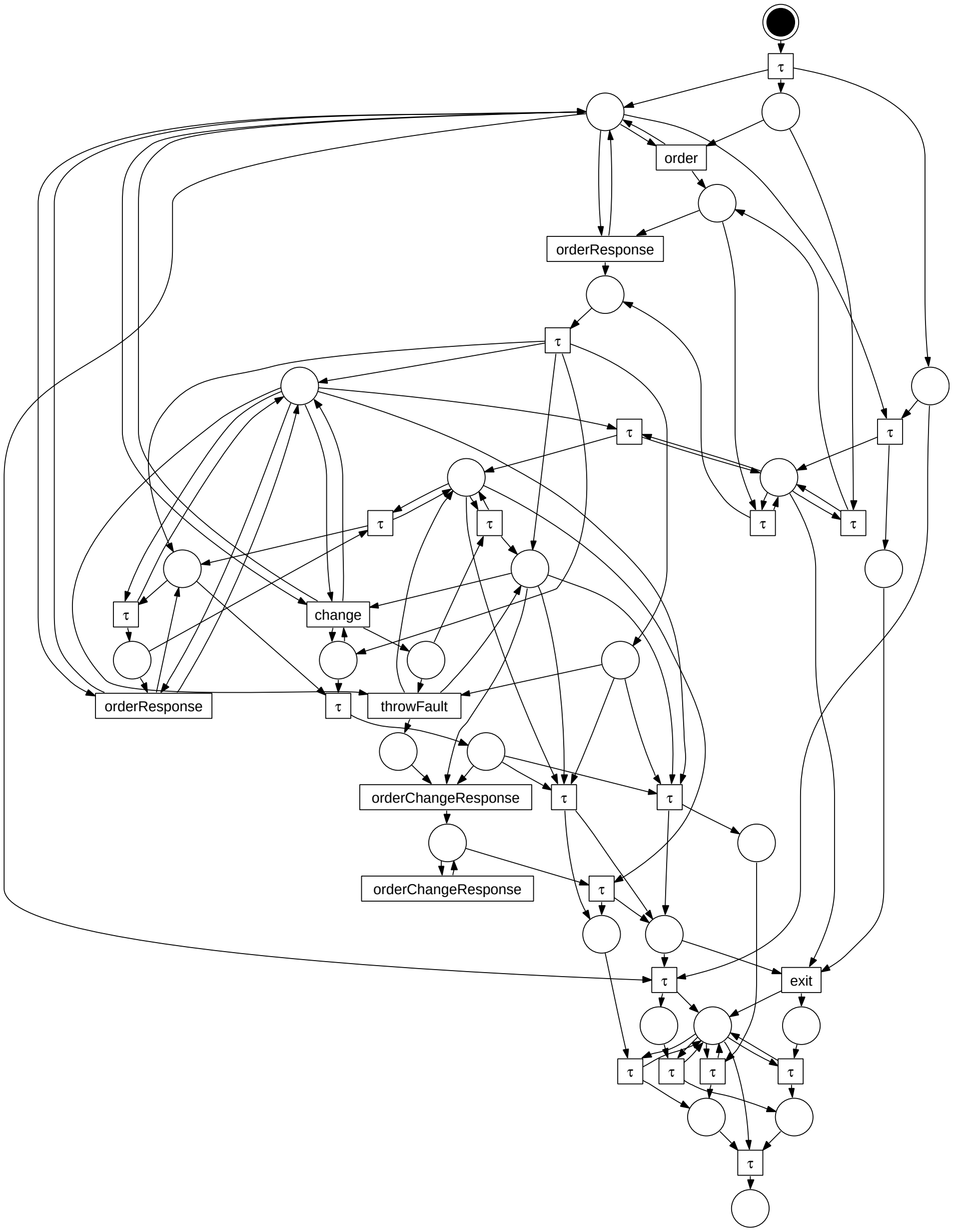


Fig. 11. The WF-net for the Supplier process.

In the remainder of this section we discuss both issues in detail.

5.1 Grouping messages into log traces

In order to apply the proposed conformance checking technique, messages need to be grouped into *log traces* each representing one execution of the service, i.e., each message needs to be associated to a process instance. If the service is implemented as an executable BPEL process, this grouping of messages is trivial. The process is executed by an engine that generates logs associating each communication action (and thus the message consumed or produced by that action) to a process instance. All messages consumed or produced by a process instance can then be grouped into a log trace.

If no executable BPEL process is available, we need to group messages into log traces just by looking at their contents. Current web service standards do not make a provision for messages to include a "service instance identifier", so assuming the existence of such identifier may be unrealistic in some situations. Other monitoring approaches in the field of web services have recognized this problem and have addressed it in different ways, but they usually end up relying on very specific and sometimes proprietary approaches. For example the Web Services Navigator [Pauw et al. 2005] uses IBM's Data Collector to log both the contents and context of SOAP messages. But to enable correlation, the Data Collector inserts a proprietary SOAP header element into messages.

In order to avoid relying on proprietary SOAP extensions, we use a generic grouping mechanism that we term *chained correlation*. The idea of chained correlation is that every message, except for the first message of a service instance, refers to at least one previous message belonging to the same service instance. In the context of contemporary web service standards and middleware this correlation information can be obtained in at least two ways:

—When using SOAP in conjunction with WS-Addressing, each message contains an identifier (*messageID* header) and may refer to a previous message through the *relatesTo* header. If we assume that these addressing headers are used to relate messages belonging to the same service instance in a chained manner, it becomes possible to group a raw service log containing all the messages sent or received by a service into log traces corresponding to service instances. This is the method used in our case study and more details will be given in Section 6. The method is applicable when using Oracle BPEL as well as various other web service middleware supporting the WS-Addressing standard. Note however web service middleware supporting WS-Addressing may use the *replyTo* header to correlate messages as opposed to the *relatesTo*. Specifically, the *replyTo* header of a given message (say M) may contain a URI uniquely identifying the message in question. Subsequently, when another message M' of the opposite directionality is observed that has the same URI in the *To* header, M and M' can be correlated.

—The second method is based on the identification of properties that a message has in common with another message belonging to the same service instance. In BPEL, properties shared by messages belonging to the same service instance are captured as *correlation sets*. A correlation set can be seen as a function that maps a message to a value of some type. Correlation sets are associated with communication actions. When a message is received which has the same value for a correlation set as the value of a message previously sent by a running service instance, the message in question is associated with this instance. This allows one to map messages to service instances, except for those messages that initialize a correlation set, that is, those messages that start a new instance. Assuming that in the BPEL abstract process of a service only the initial actions of the protocol initialize correlation sets, and all other actions refer to the same correlation sets as the initial action, each message produced or consumed by the service can be mapped to a service instance as follows: The full message log is scanned in chronological order. A message is either related to a new service instance if it corresponds to a communication action that initializes a correlation set, or related to a previously identified service instance if the values of its correlation set match those of a message sent by the previous service instance.

In some cases, neither of the techniques outlined above is applicable. In other words, there may be no way of defining a function that can determine whether or not a given message is related to a previously observed message. In this case, techniques from the area of Web session identification can be employed, but such techniques are not 100% reliable. This avenue is considered in [Gombotz and Dustdar 2005].

5.2 Abstracting messages as labels

Once the message log has been grouped into log traces corresponding to service instances, we associate each message in a log trace with a transition label used in the WF-net obtained from the BPEL abstract process definition. These transition labels represent communication actions seen at the level of abstraction used for conformance checking.

BPEL's communication action types are: *invoke*, *reply*, *receive*, and *onMessage* (or *onEvent* in BPEL 2.0). A receive or an onMessage action consumes one message, a reply produces one message, while an invoke can either produce a single message (*simple send*) or produce a message and consume another one in that order (*synchronous send-receive*). Without loss of generality, we assume that the BPEL abstract process given to the Conformance Checker does not contain any synchronous send-receive. For the purposes of conformance checking, a synchronous send-receive can be decomposed into a *sequence* activity containing a simple send followed by a receive. Also without loss of generality, we assimilate reply actions to send actions and onMessage handlers to receive actions, since these elements have the same effect in terms of message logs.

Thus, for conformance checking purposes, we view communication actions in a BPEL abstract process as being labeled by a pair $\langle$ D, MT $\rangle$ where D stands for the direction (inbound or outbound) and MT for message type. All non-communication actions are given τ-labels since their execution does not manifest itself as message log entries. Actions with τ-labels in the abstract process get translated to silent transitions.

Under this labeling scheme, it is possible that two actions in a BPEL process get the same label. Hence, the Petri net generated from a BPEL abstract process may have multiple (non-silent) transitions with the same label. Fortunately, this possibility is supported by the conformance checking technique, e.g., the example in Figure 3(a) contains two actions with label A.

Each communication action in a BPEL process definition is linked to a WSDL operation. A WSDL operation in turn is associated with *binding information* that determines how messages related to that operation are encoded and exchanged over a given communication protocol (e.g., SOAP over HTTP or XML over HTTP). The structure of an operation's binding information varies depending on the transport protocol, but in any case it normally provides a means to identify messages that pertain to that operation. In the case of SOAP over HTTP, the binding information for a WSDL operation maps this operation to a *SOAPAction* identifier. This makes it possible to reliably associate a SOAP message with a WSDL operation by inspecting the *SOAPAction* field in the HTTP header of the message. In the case of a communication protocol based on plain XML over HTTP, the binding information of a given WSDL operation may include a relative URL to be found in the HTTP headers of every message pertaining to that operation. Again, this makes it possible to associate a SOAP message to an operation by analyzing the "request URI" in the message's HTTP header.

In the general case, however, the SOAPAction header and the mapping between WSDL operations and SOAPAction identifiers are optional. In the absence of this information, associating SOAP messages to WSDL operations may require inspection of the message's body. Specifically, the top-level element in the SOAP message body needs to be compared with the message type associated to each operation supported by the service. This technique is only reliable if operations map to message types with different top-level elements. Otherwise, the user of the conformance checker would have to provide a function mapping each SOAP message in the log to a WSDL operation. This illustrates that the versatility of SOAP and WSDL make it difficult to achieve a general and reliable solution for the problem of mapping messages to operations. In some cases, tailor-made solutions are required.

Despite these potential obstacles, it is realistic to assume that every message produced or consumed by a service for which a BPEL abstract process is defined, can be mapped to a WSDL operation. With this information and the message direction, we can construct log traces such that each entry in the trace can be matched to a communication action label under the labeling scheme described above. Because we are able to map a BPEL specification onto a Petri net (cf. Section 4) and we can associate messages to both process instances and activities (cf. this section), we can now apply the conformance checking techniques described in Section 3.

6. EXPERIMENTAL APPLICATION OF THE APPROACH

This section discusses the applicability of the approach and tools described in previous sections using the example from Section 4.1. We focus specifically on the "local message observer" setting, that is, relevant messages exchanged between all services involved in the choreography (i.e. Supplier and Customer) are visible.

6.1 Obtaining event logs

To generate SOAP messages, we needed to implement services that would behave, at least presumably, according to the abstract BPEL processes describing the choreography. We could have used a conventional programming language to implement these services. However, we chose not to do so and implemented an executable process definition corresponding to the "supplier" role in the working example. Specifically, we took the abstract BPEL process definition of the supplier role as a starting point and we added into it manual tasks (e.g. for order entry and processing), data manipulation actions and other details. In this way, we obtained an executable BPEL process definition that we deployed into the Oracle BPEL Process Manager (version 10.1.2).[6].

SOAP messages are typically exchanged between two services, which can both run on the same server or on different servers. For this reason, we also had to implement a simple "Customer" executable BPEL process. The Customer process places an order, waits for an orderResponse, then places a changeOrder, waits for two orderChangeResponses, and then exits. We deployed the executable BPEL processes on two different Oracle BPEL servers.

Subsequently, we created instances of the executable BPEL processes and we executed these instances using the console and worklist handler provided by the Oracle BPEL platform. This allowed us to generate SOAP messages between two Oracle BPEL servers (one for the supplier and one for the customer), which we then altered to introduce different types of deviations.

Unfortunately, we were unable to obtain the SOAP messages directly from Oracle BPEL. No option existed to log all SOAP messages sent and/or received to a file, and they were also not stored in the database underlying the Oracle BPEL server. As a result, we had to use a TCP Tunneling technique to obtain the SOAP messages. With this technique, it is fairly easy to eavesdrop on a specific combination of host and port. Typically, incoming messages all go to the same combination of host and port, but outgoing messages can be directed to a multitude of combinations of hosts and ports. As a result, it is more convenient to eavesdrop on the incoming messages on each server. Examples of collected SOAP message logs from both servers are given in a technical report [Aalst et al. 2005]. From the SOAP message logs, it is straightforward to generate a log as shown in Figure 12. Since Oracle BPEL, by default, relies on WS-Addressing, the first message (the *order*) contains a unique message id (e.g. `bpel://localhost/default/Customer~1.1/301-BpInv0-BpSeq0.3-3`), and all other related messages refer to this message id.

Both the WF-net corresponding to the abstract Supplier process (cf. Figure 11) and the log from Figure 12 can be imported by the ProM framework to check their conformance.

6.2 Conformance checking

Having demonstrated that it is feasible to obtain an event log (such as in Figure 12) from service executions, we now use conformance checking techniques (see also Section 3) to validate the supplier service specification for a number of interaction scenarios. Table I shows five execution sequences which should be valid for the supplier service as specified in Section 4.1 and eight which should not.

Scenarios 1 – 5 reflect message sequences which should be compliant with the process specification (note that Scenario 5 corresponds to the example from Fig-

[6] See: `www.oracle.com/technology/products/ias/bpel/`.

```
<?xml version="1.0" encoding="UTF-8"?>
<WorkflowLog>
  <Source
    program="Oracle BPEL, using TCP Tunneling"
  />
  <Process
    id="http://services.qut.com/Supplier"
    description="Supplier 1.1, using Customer 1.1 as customer stub"
  >
    <ProcessInstance
        id="bpel://localhost/default/Customer~1.1/301-BpInv0-BpSeq0.3-3"
        description="Instance 301"
    >
      <AuditTrailEntry>
        <WorkflowModelElement>order</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2005-10-20T11:54:09-00:00</Timestamp>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>orderResponse</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2005-10-20T11:58:08-00:00</Timestamp>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>change</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2005-10-20T11:58:20-00:00</Timestamp>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>orderChangeResponse</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2005-10-20T11:58:35-00:00</Timestamp>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <WorkflowModelElement>orderChangeResponse</WorkflowModelElement>
        <EventType>complete</EventType>
        <Timestamp>2005-10-20T11:58:43-00:00</Timestamp>
      </AuditTrailEntry>
    </ProcessInstance>
  </Process>
</WorkflowLog>
```

Fig. 12. A small fragment of a SOAP-based log in MXML format.

ure 12). They all start with an initiating *order*, followed by one or more *orderResponses*, and potentially complete with a *change* request and one or more *orderChangeResponses*.

Scenarios 6 – 13 represent conceivable settings of misbehavior, whereas 6 – 9 correspond to possible violations by the supplier service and 10 – 13 contain violations by the client or environment of the service. Both Scenario 6 and 7 show situations

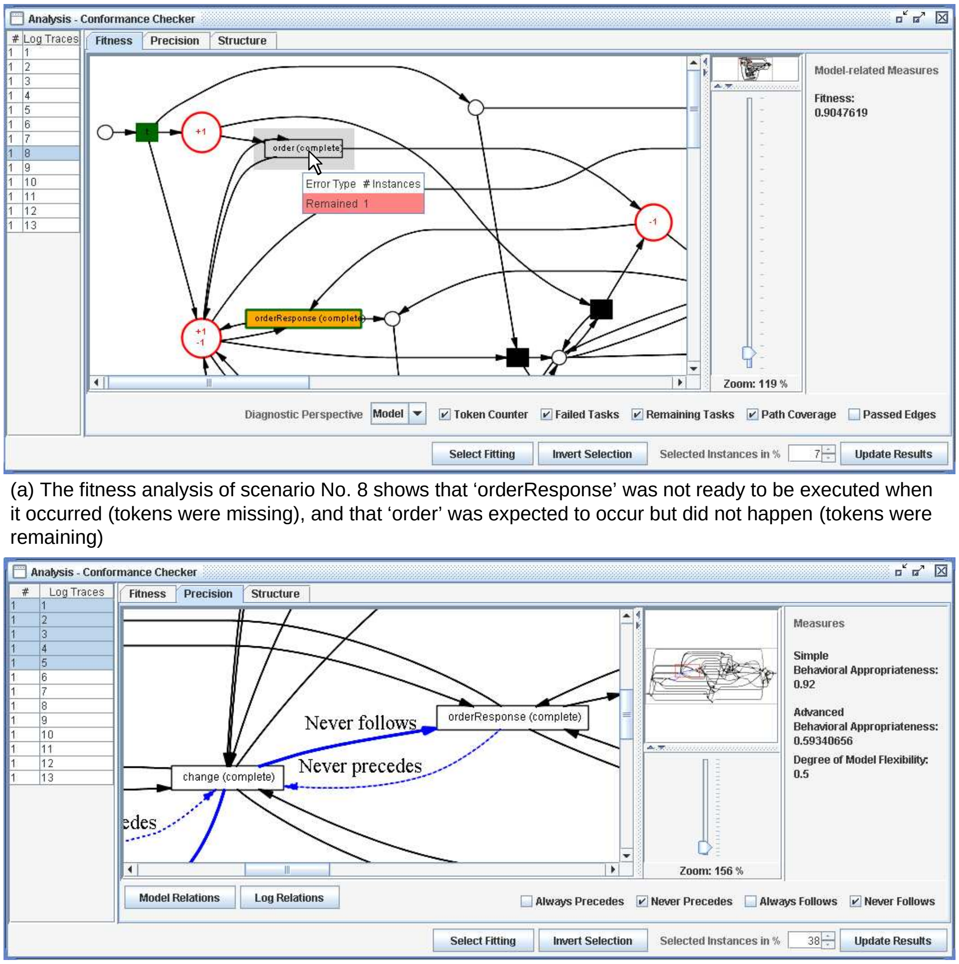
Table I. Desirable and undesirable scenarios for the supplier service execution.

	Scenario	Fitness	Log trace
desirable behavior	1	1.0	(order, orderResponse)
	2	1.0	(order, orderResponse, orderResponse, orderResponse)
	3	1.0	(order, orderResponse, change, orderChangeResponse)
	4	1.0	(order, orderResponse, orderResponse, change, orderChangeResponse)
	5	1.0	(order, orderResponse, change, orderChangeResponse, orderChangeResponse)
undesirable behavior	6	0.625	(order)
	7	0.749	(order, orderResponse, change)
	8	0.905	(orderResponse)
	9	1.0	(order, orderResponse, change, orderResponse, orderChangeResponse)
	10	0.759	(order, change, orderChangeResponse)
	11	0.0	(change)
	12	0.914	(order, orderResponse, change, orderChangeResponse, change)
	13	0.971	(order, orderResponse, change, change, orderChangeResponse)

where the conversation has not been completed properly as after having received the *order* request the service needs to send at least one *orderResponse* (missing in Scenario 6), and following a *change* request at least one *orderChangeResponse* must be sent (missing in Scenario 7). In Scenario 8 the supplier service sends an *orderResponse* which is not correlated with a previous *order*, and in Scenario 9 it still sends another *orderResponse* although a *change* request has been received already (and thus only *orderChangeResponses* should be sent). Scenario 10 shows the situation where the environment invokes a *change* request although the first *orderResponse* has not been sent by the service yet. In Scenario 11 a *change* request is invoked which is not even related to a previous *order*. Both Scenario 12 and 13 show a situation in which a second *change* is requested by the client, which is not allowed.

In order to verify the given scenarios with respect to the supplier service specification from Section 4.1 we use the reduced Petri net model generated from the abstract BPEL process, shown in Figure 11. Having imported it into the ProM framework, the Conformance Checker [Rozinat and Aalst 2006] is able to replay the log containing the scenarios in the model. Based on the number of missing and remaining tokens the fitness measurement is calculated indicating whether a scenario corresponds to a valid execution sequence for that process. If not, the depiction of missing and remaining tokens aids in locating the problem.

Consider for example Figure 13(a), in which the Conformance Checker shows a part of the model after the replay of Scenario 8. In this situation a single *orderResponse* was sent without having received any previous *order*, which is not allowed. The place in the upper left corner which has no incoming arcs represents the start place of the whole process (i.e., a token will be put there in order to start the replay of the scenario). Following the control flow of the model it can be observed that the *order* transition is supposed to fire first in order to produce a token in the enlarged place on the right, which can be consumed by the *orderResponse* transition afterwards. However, since the log replay is carried out from a log-based perspective the missing tokens (indicated by a − sign) are created artificially and the task belonging to the observed message in the model (i.e., the *orderResponse* transition) is executed immediately. The fact that it had been forced to do so is recorded and the task is marked as having failed successful execution (i.e., it was not enabled).



Fitness: 0.9047619

Diagnostic Perspective: Model; ☑ Token Counter ☑ Failed Tasks ☑ Remaining Tasks ☑ Path Coverage □ Passed Edges

Selected Instances in % 7

(a) The fitness analysis of scenario No. 8 shows that 'orderResponse' was not ready to be executed when it occurred (tokens were missing), and that 'order' was expected to occur but did not happen (tokens were remaining)

Simple Behavioral Appropriateness: 0.92

Advanced Behavioral Appropriateness: 0.59340656

Degree of Model Flexibility: 0.5

□ Always Precedes ☑ Never Precedes □ Always Follows ☑ Never Follows

Selected Instances in % 38

(b) The behavioral appropriateness analysis based on the desirable scenarios reveals that the model allows for more behavior than expected. Due to intermediate states it is possible to send an 'orderResponse' after a 'change' request has been received

Fig. 13. The Conformance Checker analyzing the scenarios from Table I.

Furthermore, there are tokens remaining in the enlarged places in the upper and the lower left corner (indicated by a + sign), which leads to the *order* transition remaining enabled after replay has finished. Remaining tasks are visualized with the help of a shaded rectangle in the background and they point to situations where a task was expected to be executed but did not occur.

Now reconsider Table I, where the Fitness column indicates for each scenario whether it corresponds to a valid execution sequence for our supplier service (i.e., during replay there were neither tokens missing nor remaining and therefore fitness = 1.0) or not (i.e., fitness < 1.0). As it shows 100% fitness for Scenario 1 – 5 the abstract BPEL process has been proven to be a valid specification with respect to the "well-behaving" conversation scenarios we thought of. However, it also allows for an execution sequence that we have classified as undesirable behavior, namely Scenario 9: Although another *orderResponse* is sent after a *change* request has been received already (and thus only *orderChangeResponses* should be sent) the scenario proved to comply with the given abstract BPEL process specification. This is an interesting result as it makes us aware of the fact that—due to a number of intermediate states—the chosen fault/event handler construct does not completely capture the intended constraint. The same conclusion can be drawn from the behavioral appropriateness analysis with the Conformance Checker based on the five desirable scenarios only, where a screenshot of the result is depicted in Figure 13(b). The displayed part visualizes that, although according to the model a *change* request could be followed by an *orderResponse*, this never happened in the log (as the analysis is based on scenarios 1 – 5).

The small case study presented in this section demonstrates that conformance checking not only helps to detect deviations in terms of violations of the specified control-flow, but can also point to undesirable behavior which is captured by the model.

7. RELATED WORK

Several attempts have been made to capture the semantics of BPEL by means of translations into formal languages. Some have defined translations from BPEL to finite state machines [Fisteus et al. 2004], others to process algebra [Ferrara 2004], abstract state machines [Fahland and Reisig 2005] or Petri nets [Ouyang et al. 2005a; Ouyang et al. 2005b; Hinz et al. 2005]. This paper uses the translation to Petri nets presented in [Ouyang et al. 2005b] which is very detailed in terms of its coverage of control-flow constructs.

This paper builds on earlier work on process mining, i.e., the extraction of knowledge from event logs (e.g., process models or social networks). For example, the α-algorithm [Aalst et al. 2004] can derive a Petri net from an event log. For an overview of process mining techniques, the reader is referred to [Aalst et al. 2003].

In this paper we use the conformance checking techniques described in preliminary form in [Rozinat and Aalst 2006] and implemented in the ProM framework [Dongen et al. 2005]. The notion of conformance has also been discussed in the context of security [Aalst and Medeiros 2004], business alignment [Aalst 2005], and genetic mining [Medeiros et al. 2006].

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 (`ws.apache.org/axis/`). [Lazovik et al. 2004] introduces an assertion language for expressing business rules and a framework to plan and monitor the execution of these rules. [Baresi et al. 2004] 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 [Mahbub and Spanoudakis 2004]. This approach uses event calculus to specify requirements. [Ludwig et al. 2004] is an approach based on WS-Agreement defining the Crona framework for the creation and monitoring of agreements. In [Gombotz and Dustdar 2005; Dustdar et al. 2004], 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 [Pauw et al. 2005] 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 Charts (MSCs) and graph-based representations of the system topology. Our work differs from these papers in two ways. First of all, we use a process model to check conformance rather than visualizing and analyzing frequent interaction patterns (i.e. scenarios). Typically, it is easier to specify a process rather than a complete set of scenarios, although scenarios can help in designing and analyzing a process. Moreover, a process specification enables a more intuitive visualization of the problem areas such as deviations from the intended behavior. Second, we consider the problem of correlation in more detail than these papers.

8. CONCLUSION

Service-oriented systems are composed of relatively autonomous entities (i.e., services). Unlike many classical systems there is not one entity controlling a monolithic system. Therefore, it is essential that each of the services involved in such a system behaves as the other services expect it to behave. In this paper we demonstrated the feasibility of conformance checking of service behavior, that is, comparing message logs with service behavior specifications to detect and to quantify deviations.

Although conformance checking of service behavior can be applied to a wide variety of settings, we focused on a particular usage scenario involving (1) abstract BPEL as the specification language of a single service and (2) SOAP messages exchanged between this service and other services. We demonstrated that specifications in terms of abstract BPEL can be mapped onto Petri nets and the SOAP messages exchanged between the various services can be mapped onto the MXML log format. Given a set of messages and an abstract BPEL specification we can then measure fitness and appropriateness. Moreover, if the observed behavior does not match the specified behavior, the deviations can be shown in both the log and the model. Using a case study utilizing Oracle BPEL as a process engine, we demonstrated that our approach is indeed applicable using current technology. We have implemented three tools to achieve this: (1) *BPEL2PNML* (for the mapping from BPEL to PNML), (2) *WofBPEL* (for process verification and cleaning up the automatically generated Petri net), and (3) the *ProM Conformance Checker*.

Although this paper focused on abstract BPEL, other languages could also be supported by replacing BPEL2PNML by a component providing the mapping onto Petri nets for the selected alternative language. In fact, we consider languages such as BPEL and WS-CDL not really suitable for the specification of services. They tend to describe things at a too low level, i.e., a level suitable for execution but less suitable for describing what the different services need to agree upon. Hence future research will aim at conformance checking in the context of more declarative languages such as *DecSerFlow* [Aalst and Pesic 2006] and *Let's Dance* [Zaha et al. 2006]. Moreover, we would like to apply our approach to more real-life case studies. One of the problems we are facing is that at this point in time only few organizations use BPEL and can provide us with SOAP logs. Clearly, conformance checking can be applied in many domains ranging from auditing (cf. the Sarbanes-

Oxley Act) to software testing. Therefore, we plan to consider a wide variety of applications and not limit ourselves to web services. Another topic for further research is the visualization of behavior conformance, for example, by combining the ideas presented in [Pauw et al. 2005] with our process-oriented approach.

REFERENCES

Aalst, van der W. 1998. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers 8,* 1, 21–66.

Aalst, van der W. 2005. Business Alignment: Using Process Mining as a Tool for Delta Analysis and Conformance Testing. *Requirements Engineering Journal 10,* 3, 198–211.

Aalst, van der W., Dumas, M., Ouyang, C., Rozinat, A., and Verbeek, H. 2005. Choreography Conformance Checking: An Approach based on BPEL and Petri Nets (extended version). BPM Center Report BPM-05-25, BPMcenter.org.

Aalst, van der W. and Medeiros, A. 2004. Process Mining and Security: Detecting Anomalous Process Executions and Checking Process Conformance. In *Second International Workshop on Security Issues with Petri Nets and other Computational Models (WISP 2004)*, N. Busi, R. Gorrieri, and F. Martinelli, Eds. STAR, Servizio Tipografico Area della Ricerca, CNR Pisa, Italy, 69–84.

Aalst, van der W. and Pesic, M. 2006. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *International Conference on Web Services and Formal Methods (WS-FM 2006)*, M. Bravetti, M. Nunez, and G. Zavattaro, Eds. Lecture Notes in Computer Science, vol. 4184. Springer-Verlag, Berlin, 1–23.

Aalst, van der W., Dongen, van B., Herbst, J., Maruster, L., Schimm, G., and Weijters, A. 2003. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering 47,* 2, 237–267.

Aalst, van der W., Weijters, A., and Maruster, L. 2004. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering 16,* 9, 1128–1142.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. 2003. Business Process Execution Language for Web Services, Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation.

Jordan, D., Evdemon, J., et al. 2006. Web Services Business Process Execution Language Version 2.0. Public Review Draft (August 2006), OASIS WS-BPEL Technical Committee.

Baresi, L., Ghezzi, C., and Guinea, S. 2004. Smart Monitors for Composed Services. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing.* ACM Press, New York, NY, USA, 193–202.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S., and Winer, D. 2000. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/soap.

Cardoso, J., Sheth, A., Miller, J., Arnold, J., and Kochut, K. 2004. Quality of service for workflows and web service processes. *Journal of Web Semantics 1,* 3, 281–308.

Desel, J., Reisig, W., and Rozenberg, G., Eds. 2004. *Lectures on Concurrency and Petri Nets.* Lecture Notes in Computer Science, vol. 3098. Springer-Verlag, Berlin.

Dongen, van B., Medeiros, A., Verbeek, H., Weijters, A., and Aalst, van der W. 2005. The ProM framework: A New Era in Process Mining Tool Support. In *Application and Theory of Petri Nets 2005*, G. Ciardo and P. Darondeau, Eds. Lecture Notes in Computer Science, vol. 3536. Springer-Verlag, Berlin, 444–454.

Dustdar, S., Gombotz, R., and Baina, K. 2004. Web Services Interaction Mining. Technical Report TUV-1841-2004-16, Information Systems Institute, Vienna University of Technology, Wien, Austria.

Fahland, D. and Reisig, W. 2005. ASM-based semantics for BPEL: The negative control flow. In *Proc. 12th International Workshop on Abstract State Machines*, D. Beauquier and E. Börger and A. Slissenko, Ed. Paris, France, 131–151.

Ferrara, A. 2004. Web services: A process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*. ACM Press, New York, NY, USA, 242–251.

Fisteus, J., Fernández, L., and Kloos, C. 2004. Formal verification of BPEL4WS business collaborations. In *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web '04)*, K. Bauknecht, M. Bichler, and B. Proll, Eds. Lecture Notes in Computer Science, vol. 3182. Springer-Verlag, Berlin, Zaragoza, Spain, 79–94.

Glabbeek, R. and Weijland, W. 1996. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM 43,* 3, 555–600.

Gombotz, R. and Dustdar, S. 2005. On Web Services Mining. In *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, C. Bussler et al., Ed. Lecture Notes in Computer Science, vol. 3812. Springer-Verlag, Berlin, 216–228.

Hinz, S., Schmidt, K., and Stahl, C. 2005. Transforming BPEL to Petri Nets. In *International Conference on Business Process Management (BPM 2005)*, W. van der Aalst, A. ter Hofstede, and M. Weske, Eds. Lecture Notes in Computer Science, vol. 2678. Springer-Verlag, Berlin, 220–235.

Lazovik, A., Aiello, M., and Papazoglou, M. 2004. Associating Assertions with Business Processes and Monitoring their Execution. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM Press, New York, NY, USA, 94–104.

Ludwig, H., Dan, A., and Kearney, R. 2004. 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*. ACM Press, New York, NY, USA, 65–74.

Mahbub, K. and Spanoudakis, G. 2004. A Framework for Requirents Monitoring of Service Based Systems. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM Press, New York, NY, USA, 84–93.

Medeiros, A., Weijters, A., and Aalst, van der W. 2006. Genetic Process Mining: A Basic Approach and its Challenges. In *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, C. Bussler et al., Ed. Lecture Notes in Computer Science, vol. 3812. Springer-Verlag, Berlin, 203–215.

Murata, T. 1989. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE 77,* 4 (April), 541–580.

Ouyang, C., Verbeek, H., Aalst, van der W., Breutel, S., Dumas, M., and Hofstede, ter A. 2005. WofBPEL: A Tool for Automated Analysis of BPEL Processes. In *Proceedings of Service-Oriented Computing (ICSOC 2005)*, B. Benatallah, F. Casati, and P. Traverso, Eds. Lecture Notes in Computer Science, vol. 3826. Springer-Verlag, Berlin, 484–489.

Ouyang, C., Aalst, van der W., Breutel, S., Dumas, M., Hofstede, ter A., and Verbeek, H. 2005. Formal Semantics and Analysis of Control Flow in WS-BPEL (Revised Version). BPM Center Report BPM-05-15, BPMcenter.org.

Pauw, W., Lei, M., Pring, E., Villard, L., Arnold, M., and Morar, J. 2005. Web Services Navigator: Visualizing the Execution of Web Services. *IBM Systems Journal 44,* 4, 821–845.

Rozinat, A. and Aalst, van der W. 2006. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, C. Bussler et al., Ed. Lecture Notes in Computer Science, vol. 3812. Springer-Verlag, Berlin, 163–176.

Verbeek, H., Basten, T., and Aalst, van der W. 2001. Diagnosing Workflow Processes using Woflan. *The Computer Journal 44,* 4, 246–279.

Zaha, J., Barros, A., Dumas, M., and Hofstede, ter A. 2006. Lets Dance: A Language for Service Behavior Modeling. In *OTM Proceedings, 14th International Conference on Cooperative Information Systems (CoopIS 2006)*, R. Meersman and Z. T. et al., Eds. Lecture Notes in Computer Science, vol. 4275. Springer-Verlag, Berlin, 145–162.

Received October 2005; August 2006; accepted January 2007