Comparing and evaluating Petri net semantics for BPEL

Document status and date:
Published: 01/01/2007

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

Comparing and Evaluating
Petri Net Semantics for BPEL

Niels Lohmann[1], Eric Verbeek[2], Chun Ouyang[3],
Christian Stahl[1], and Wil M. P. van der Aalst[2,3]

[1] Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
{nlohmann, stahl}@informatik.hu-berlin.de
[2] Technische Universiteit Eindhoven
Department of Mathematics and Computer Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{W.M.P.v.d.Aalst, H.M.W.Verbeek}@tue.nl
[3] Queensland University of Technology, Faculty of Information Technology
GPO Box 2434, Brisbane QLD 4001, Australia
c.ouyang@qut.edu.au

Abstract. We compare two Petri net semantics for the Web Services
Business Process Execution Language (BPEL). The comparison reveals
different modeling decisions. These decisions together with their conse-
quences are discussed. We also give an overview of the different properties
that can be verified on the resulting models. A case study helps to eval-
uate the corresponding compilers which transform a BPEL process into
a Petri net model.

Key words: BPEL, Petri net semantics, formal methods, verification

1 Introduction

The Web Services Business Process Execution Language (BPEL) [1] is emerging
as the de facto standard for implementing business processes on top of the Web
service technology. However, BPEL lacks of a formal semantics; that is, it has
been defined informally. Especially its predecessor specification, BPEL 1.1 [2],
contained many ambiguities. Over the years, several attempts have been made
to formalize BPEL, using among others Petri nets, process algebra, abstract
state machines, or state machines (see [3] for a survey). Fortunately, the current
BPEL 2.0 specification [1][4] is more precise although it is still informally defined.

The language constructs found in BPEL, especially those related to control
flow, are close to those found in workflow definition languages [4]. In the area of
workflows, it has been shown that Petri nets [5] are appropriate both for modeling
and analysis. More specifically, with Petri nets several elegant technologies such

---

[4] We use the term "BPEL" if we do not need to distinguish between the 1.1 and the
2.0 specification.

as the theory of workflow nets [6], a theory of controllability [7,8], a long list of verification techniques, and tools (see [9] for an overview) become directly applicable.

In this paper, we compare two Petri net semantics for BPEL. One is developed by the Theory of Programming Group at the Humboldt-Universität zu Berlin (HUB) [10], the other mainly by the BPM Group at the Queensland University of Technology (QUT) [11], which we hereafter simply refer to as HUB and QUT semantics, respectively. The BPEL semantics of both groups are *feature complete*; that is, covering the standard and the exceptional behavior of BPEL. Also, each of the groups has developed tools to support the translation and the analysis of BPEL processes.

The goal of this paper is twofold. On the one hand, we compare both semantics. We reveal their differences and discuss their consequences with respect to the model and the verification. On the other hand, we also evaluate the compilers which translate a BPEL process into a Petri net model according to the respective semantics. By help of a case study, we compare the sizes of the nets and their state spaces. This helps to identify the impact on the complexity of the analysis resulting from the different modeling approaches and decisions. In addition, we also give an overview what properties can be analyzed on the models. The result can be used to guide future tool integration for better performance.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of BPEL. Next, Sect. 3 discusses the basics of both QUT and HUB semantics. Then, a number of differences between the BPEL semantics of the two approaches are highlighted in Sect. 4. Section 5 discusses to what extent both approaches allow for verification of the original BPEL processes. In Sect. 6, we present the results of several case studies. Finally, Sect. 7 concludes and outlines future work.

2 Overview of BPEL

The *Web Services Business Process Execution Language* (BPEL) [1], is a language for describing the behavior of business processes based on Web services. For the specification of a business process, BPEL provides *activities* and distinguishes between basic and structured activities. A basic activity can communicate with the partners by message exchange (`<invoke>`, `<receive>`, `<reply>`), manipulate and validate data (`<assign>`, `<validate>`), wait for some time (`<wait>`) or just do nothing (`<empty>`), signal faults (`<throw>`, `<rethrow>`), or end the entire process instance (`<exit>`).

A structured activity defines a causal execution order on basic activities and can be nested in another structured activity itself. The structured activities include sequential execution (`<sequence>`), parallel execution (`<flow>`), data-dependent branching (`<if>`), timeout- or message-dependent branching (`<pick>`), and repeated execution (`<while>`, `<repeatUntil>`, `<forEach>`). Within activities executed in parallel, the execution order can further be controlled by the usage of a nonstructural construct called *control links* (`<link>`).

In addition, there is also a structured activity `<scope>`, which groups activities into a block, links this block to transaction management, and provides fault, compensation, termination, and event handling. The `<process>` is the outmost scope of the described business process. A `<faultHandler>` is a component of a scope that provides methods to handle faults which may occur during the execution of its enclosing scope. Moreover, a `<compensationHandler>` can be used to reverse some effects of successfully executed activities. The termination of scopes can be controlled with a `<terminationHandler>`. With the help of an `<eventHandler>`, external message events and specified timeouts can be handled.

3   Semantics Basics

In both HUB and QUT groups, the main motivation to develop a formal semantics for BPEL is to detect inconsistencies in the BPEL specification on the one hand, and to formally analyze BPEL processes using techniques of computer-aided verification on the other hand. Furthermore, both groups translate real-life BPEL processes into Petri nets to validate their (Petri net-based) verification techniques for analyzing service behavior. Besides BPEL-specific analysis goals, the QUT semantics is built for analyzing soundness [6], an established correctness property of workflows. In contrast, the HUB semantics is built for analyzing the communication behavior of BPEL processes [12].

Both semantics follow an hierarchical approach. The translation is guided by the syntax of BPEL. In BPEL, a process comprises a number of language constructs that are connected in certain execution orders. Each construct of the language is translated separately into a Petri net. Such a net forms a *pattern* of the respective BPEL construct. Each pattern has an *interface* for connecting it with other patterns as is done with BPEL constructs. Also, patterns capturing BPEL's structured activities may carry any number of inner patterns as its equivalent in BPEL can do. Overall, the collection of patterns forms the *Petri net semantics* for BPEL. It is worth mentioning that both HUB and QUT semantics are *feature complete*, covering the standard and exceptional behavior of BPEL, and both semantics are *formal*, meaning that they are suitable for computer-aided verification.

The HUB group developed a Petri net semantics for BPEL 1.1 [10]. This semantics was later enhanced [13] to cover BPEL 2.0. The QUT group developed a semantics for the activities and constructs of BPEL 1.1 [11] based on the description given by the (more precise) BPEL 2.0 specification.

Despite the above commonalities, one major difference between the two semantics is that they are specified at different levels of abstraction. The QUT semantics is restricted to the control flow of a BPEL process; that is, it abstracts from data (i. e., no BPEL variable is modeled) and message exchange (i. e., the sending and receiving of messages is not explicitly modeled). The resulting semantics is defined as a low-level Petri net, where the control flow is modeled by undistinguishable black tokens and a data dependent branch is modeled by

a nondeterministic choice. Consequently, the QUT semantics leads to a direct implementation based on low-level Petri nets. In contrast, the HUB semantics is defined using a high-level Petri net. It allows on the one hand to model not only control flow but also data information (such as data values, message content, and Boolean expressions of the link semantics), while on the other hand, it is possible to apply data abstraction techniques during the translation process [14]. This way, in the HUB's implementation, a data abstraction is performed, and as a result, it yields a low-level Petri net as in the QUT's implementation. We focus on the compilers BPEL2PNML (QUT semantics) and BPEL2oWFN (HUB semantics) in Sect. 5.1.

In contrast to the QUT semantics, the HUB semantics models message exchange explicitly via a message interface. This results in the next difference of the two approaches. In the QUT semantics, a model of a BPEL process is a *workflow net* (WFN) [6]. In contrast, due to the definition of a message interface, the resulting model of a BPEL process in the HUB semantics has a structure of an *open workflow net* (oWFN) [15]. oWFNs are a generalized form of WFNs. As a substantial difference, in an oWFN the interface of a service is explicitly represented as sets of input and output places. As it can be seen later on, the different models allow for the analysis of different properties. The main motivation in the QUT group to have a WFN is to apply existing verification techniques for WFNs, such as the analysis of soundness which is implemented in the tool Woflan [16]. In the HUB group, the explicit representation of the interface is essential for analyzing the communication behavior of BPEL processes using the tool Fiona [12].

4   Different Modeling Concepts

In this section, we compare the modeling of three important BPEL constructs in the QUT semantics and in the HUB semantics. For each construct, we first discuss its modeling details in each semantics, and then review the different modeling decisions made in the two semantics and if applicable, their impacts to the resulting behavior of the construct.

4.1   Patterns of Activities

We first discuss the patterns of basic activities modeled in the two semantics. To model BPEL's structured activities, the embedded activities are ordered and connected canonically. The patterns for the structured activities are very similar in both semantics and are thus not discussed.

QUT Each basic activity is considered as an atomic action, for example, a `<receive>` activity performs a receive action, and an `<invoke>` performs a send action. Due to the abstraction from data and message exchanges, all basic activities share the general pattern shown in Fig. 1. For normal processing of an activity X, place r_x models an initial state when it is <u>r</u>eady to start X, and

place f_x indicates a final state when X has finished its execution. The transition labeled X models the action to be performed. It has an input place s_x capturing the state when X has started, and an output place c_x for the state when X is completed. Two unlabeled transitions (drawn as solid bars) model internal actions for checking pre-conditions or evaluating post-conditions for activity X. The skip path of X is defined to facilitate the mapping of control links (see Sect. 4.2).

The rest of the pattern in Fig. 1 is modeled to capture the fact that activity X can only be executed if none of its enclosing scopes (Q_1 to Q_n) has faulted. The pattern of each scope has two places to_continue and to_stop. Place to_continue is marked as long as the scope is not faulted, otherwise place to_stop will be marked. As a result, the transition labeled X has read arcs (depicted as arcs with two arrowheads) to the to_continue place of every enclosing scope. Assume that one of these enclosing scopes Q_i gets faulted and has to stop before activity X can occur. The stop signal will get propagated through the hierarchy of the enclosed scopes of Q_i in such a way that, for each active scope, its to_stop place is marked. As a result, once the to_stop place of Q_1 (the immediately enclosing scope of X) is marked, the bypass transition will fire, indicating that activity X gets bypassed.
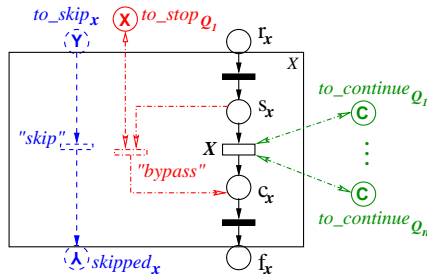
Fig. 1. Pattern of a basic activity X. The structure of the pattern can be divided into three parts: one (drawn in solid lines) models the normal processing of X, one (drawn in dashed lines) models the skipping of X, and one (drawn in dot-dashed lines) models whether to execute or to bypass X based on the status of all the enclosing scopes (Q_1 to Q_n) of X.

HUB Each pattern of an activity has five interface places: initial, final, stop, stopped, and fault. The places initial and final correspond to the places r_x and f_x of the QUT semantics, respectively. Marking the initial place starts an activity. Upon faultless completion of the activity, the final place is marked. The places stop and stopped model the termination of activities. Faults thrown by the activity are signaled by marking the fault place.

In contrast to the QUT semantics, each BPEL activity is specified by a different high-level Petri net pattern, modeling both control-flow and data aspects of that activity, including data values, correlation sets, and messages. In addition, the HUB group implements several low-level versions of this high-level pattern which are used for computer-aided verification. The low-level patterns differ in their level of detail, and are chosen by the compiler BPEL2oWFN according to

the verification goal. Two patterns for a `<receive>` activity with different degree of abstraction are depicted in Fig. 2.

(a) high-level pattern

(b) implemented low-level pattern
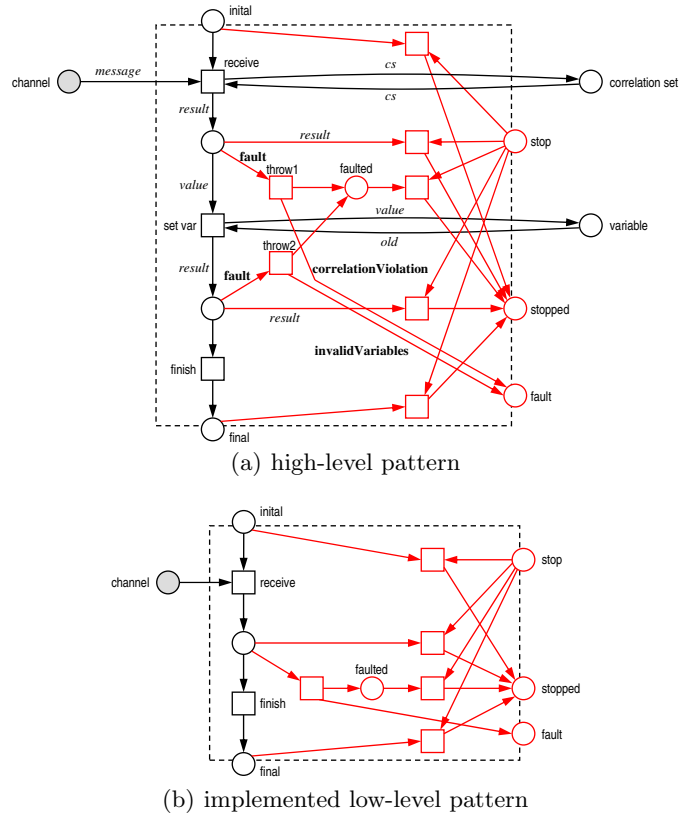
Fig. 2. Patterns for a `<receive>` activity. The detailed behavior is modeled with a high-level pattern (a). The message channel is explicitly modeled. When a *message* is received, its correlation set is read. Subsequently, the message is stored in a variable. During the execution, two faults can occur: the received message may not match the correlation set or the received message type may not match the variable type (transition throw1, throw2). In these cases, the respective BPEL standard fault is thrown on place fault. A simplified low-level pattern (b) abstracts from data and does not model variables or correlation sets.

Conclusion The QUT semantics abstracts from data and actual communication, whereas the HUB semantics explicitly specifies them in the model. As a result, in the QUT approach all basic activities share the same general pattern.

4.2 Links and Dead-Path-Elimination

Activities embedded within a `<flow>` activity are executed concurrently. However, it is possible to add control dependencies between these activities using *link*s. A link is a directed connection between a *source activity* and a *target activity*. After the source activity is executed, the link is set to true, allowing the target activity to start. As links express control dependencies, they may never form a cycle.

More precisely, when the source activity is executed faultlessly, the outgoing links are set according to their corresponding *transition conditions* which return a Boolean value for each outgoing link. After the status of all incoming links of a target activity is determined, a *join condition* — again a Boolean expression[5] — is evaluated. If this condition holds, the target activity is executed. Otherwise, if the condition is false, the activity is skipped. In this case, all outgoing links recursively embedded to the skipped activity are also set to false to avoid deadlocks. This concept is called *dead-path-elimination* (DPE) [17].

QUT Each control link is modeled by two places named lst and lsf capturing respectively the *true* and *false* status of the link. Figure 3 shows the pattern of a basic activity with links. Places named tc are used to hold the transition condition for each of the outgoing links. Since transition conditions are not explicitly specified in the pattern, their Boolean evaluation is modeled nondeterministically upon firing either of the two transitions that follow a tc place. Places jct and jcf capture the *true* and *false* result of the join condition evaluation, respectively.

If the join condition evaluates to true, activity X starts. Otherwise, transition sjf can fire. Then, activity X is not executed and will end up in the finished state (f_x) as if it was completed. This ensures that the processing of any following activity can still continue, thus capturing the semantics of dead-path-elimination.

If activity X has to be skipped (e. g., X is embedded in a non-chosen branch of an `<if>` activity), the join condition is evaluated. The evaluation result will not effect the skipping behavior: the skip transition — like transition sjf — will set all outgoing links to false. As a result, when skipping an activity, the outgoing links will be set to false, but only after all incoming links have been resolved.

HUB In the HUB semantics, each link is modeled by a colored place with Boolean type.[6] The status of the link can then be modeled by a token *true* or *false*. If the place is not marked, the link status is unknown. Similar to the QUT semantics, we model the setting and evaluation of links with patterns that wrap the corresponding source or target activity. An example of an activity that is both source and target of links is depicted in Fig. 4. As a graphical convention, dashed places indicate a place with the same label existing in the same pattern that has to be merged with the dashed place (e. g., place stop).

---

[5] While transition conditions are expressions over arbitrary variable values, join conditions only evaluate the status of the incoming links.

[6] In the implementation, this place is unfolded to two places, link.true and link.false.
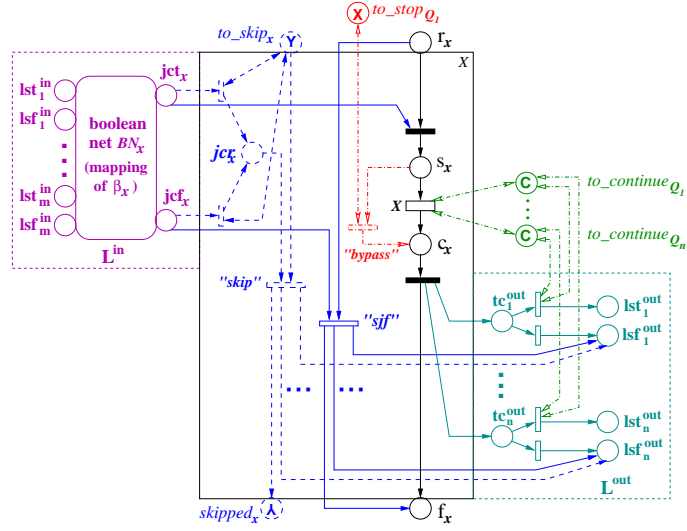
Fig. 3. Pattern of a basic activity X with incoming links $L^{\text{in}} = \{l_1^{\text{in}}, \ldots, l_m^{\text{in}}\}$ and outgoing links $L^{\text{out}} = \{l_1^{\text{out}}, \ldots, l_m^{\text{out}}\}$. The subnet enclosed in the box labeled L^{in} specifies the mapping of incoming links and the join condition evaluation at X, and the one labeled L^{out} specifies the mapping of outgoing links and their transition condition evaluation. A join condition expression (e. g., $\beta_x(ls_1^{\text{in}}, \ldots, ls_m^{\text{in}})$) is mapped onto a Boolean net (BN_x), which takes the status of all incoming links and produces an evaluation result.
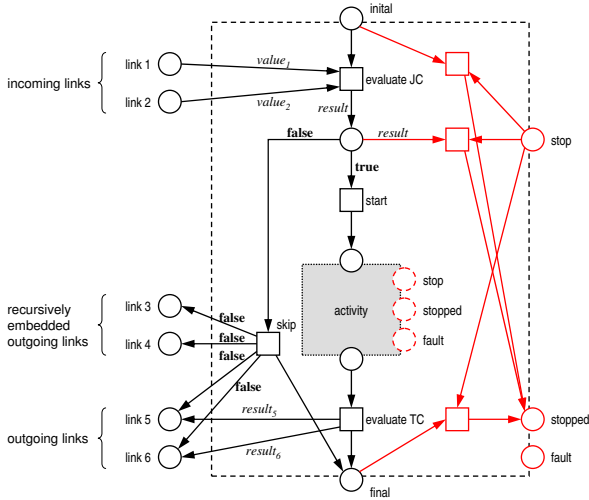
Fig. 4. Wrapper pattern of an activity that is source and target of links. Firstly, transition evaluate_JC evaluates the join condition from the incoming links yielding a Boolean token with the *result*. If the result is **true**, the embedded activity is started. Upon completion of the inner activity, transition evaluate_TC evaluates the transition condition and sets the outgoing links accordingly. If the join condition evaluates to **false**, transition skip sets all outgoing links to false.

When implemented, the join condition is, just like in the QUT semantics, "unfolded" and the transition condition is replaced by nondeterminism. However, instead of recursively skipping activities and setting their outgoing links to false as it is required by the BPEL specification [1], we deviate from the specified

behavior and set the status of all links recursively embedded to the skipped activity to false *at once*. This way, the modeling of dead-path-elimination reduces (concurrent) link status propagation, and as a result, the state space may be reduced.

```xml
<flow>
  <links> <link name="AtoB"/> <link name="BtoC"/> </links>
  <activity name="A">
    <sources> <source linkName="AtoB"/> </sources>
  </activity>
  <if>
    <condition>...</condition>
    <activity name="B">
      <targets> <target linkName="AtoB"/> </targets>
      <sources> <source linkName="BtoC"/> </sources>
    </activity>
    <else> <activity name="E"/> </else>
  </if>
  <sequence>
    <activity name="C">
      <targets> <target linkName="BtoC"/> </targets>
    </activity>
    <activity name="D"/>
  </sequence>
</flow>
```

Fig. 5. An example for links and dead-path-elimination.

A DPE Example We use the example shown in Fig. 5 to demonstrate the different modelings of DPE in the two semantics. In this example, two scenarios are possible, depending on the condition of the <if> activity: In case that the condition evaluates to true, we have the execution order shown in Fig. 6(a). Firstly, activity A is executed and sets link AtoB to true, then B is executed and sets link BtoC to true, and finally, C and D are executed sequentially.

(a) condition true   (b) condition false (QUT)   (c) condition false (HUB)
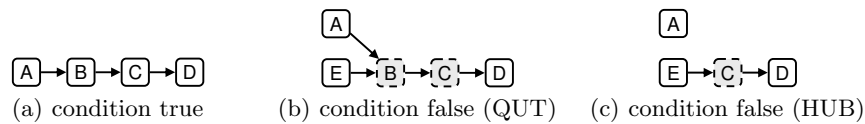
Fig. 6. Possbile execution orders of the activities of the example in Fig. 5. Skipped activities are depicted with dashed and shaded boxes.

In case the condition evaluates to false, E is executed and, due to the DPE, activity B is skipped; that is, B has to wait until A has set its link AtoB. Then, B's outgoing link, BtoC, is set to false and C is also skipped. Finally, D is executed. This yields the execution order of Fig. 6(b). Due to a transitive control dependency, D is executed *after* A. This execution is exactly modeled by the QUT semantics.

However, if the branches to be skipped are more complex, the skipping of activities yields a complex model due to the DPE. In particular, skipping of activities and execution of non-skipped activities is interleaved which might result in state explosion problems. To this end, the HUB semantics differs from the described behavior of [1]: an *overapproximation* of the process's exact behavior is modeled. In the example, activity B is not skipped explicitly, but its outgoing link, BtoC, is set to false when E is selected. This yields the execution order of Fig. 6(c). Compared to the QUT semantics, no control dependency between A and D exists and two *additional* runs are modeled by the HUB semantics, namely A and D being executed concurrently, and D being executed before A. Due to the overapproximation, it is possible that the HUB model contains errors that are not present in the BPEL process. However, static analysis of the BPEL process can help to identify these pseudo-errors. We will come back to this in Sect. 5 where the tools are discussed.

Conclusion When skipping an entire path (e. g., a path that was not chosen in a `<if>` activity), the QUT semantics sets all outgoing links to false until all causal related incoming links have been resolved, whereas the HUB semantics sets all outgoing links to false immediately; that is, without waiting for the incoming links to be resolved. While the QUT approach conforms the BPEL specification, the HUB semantics uses an overapproximation which still allows to find any design flaw of a BPEL process.

4.3 Scopes and Fault Handling

The `<scope>` activity is BPEL's most important structured activity. It provides fault, event, and compensation handling. Scopes are hierarchically ordered: every scope has a *parent scope* and an arbitrary number of *child scopes*. The `<process>` activity is the root scope. The scope hierarchy plays an important role in fault propagation and compensation.

Fault handling allows for reaction on faults that may occur during the execution of a BPEL process. There are many sources of runtime faults: wrongly typed data, WSDL faults, or explicitly thrown faults as a result of a `<throw>` activity, just to name a few. When a fault occurs, the positive control flow (i. e., the execution of the scope that encloses the faulty activity) has to stop. Subsequently, the fault handler of the enclosing scope may *catch* the fault and execute a user-defined activity to undo or "repair" the effects of the partial executed scope.

As the definition of fault handlers is optional, a fault might not be caught by the enclosing scope. Consequently, this fault is re-thrown to the parent scope until it is either handled by a fault handler or, if it can not be handled by the process's fault handler, yielding the termination of the entire BPEL process.

If more than one fault occurs in a scope, only the first is handled by the fault handlers; all subsequent faults are ignored. If a fault occurs inside a fault handler, it is re-thrown to the parent scope. We skip the explanation of how event and compensation handling is covered by the two semantics: the two approaches are very similar.

QUT Figure 7 depicts the pattern of a scope Q that has a fault handler. It can be seen that a scope is a special structured activity being attached with four flags. Assume that no exception occurs. Scope Q remains in the status of to_continue during its normal performance (i. e., the execution of Q's main activity X). Upon the completion of activity X, a snapshot is preserved for scope Q.

Consider the case when a fault occurs during the normal process of scope Q. The subnet enclosed in the dashed box labeled FH specifies the mapping of a fault handler. Transition e_{fault} models a fault event that may occur when scope Q is active. This fault event may signal any runtime fault or even a fault re-thrown from a child scope of Q. Upon the firing of e_{fault}, the status of Q changes from to_continue to to_stop. As a result, all active activities in Q need to be stopped, and the occurrence of any other fault is disabled. This ensures that no more than one fault handler can be invoked in the same scope.

A fault handler, once invoked, cannot start its main activity HF (i. e., handling fault) until the main activity of the scope has been terminated. This results in an intermediate state, as captured by place invoked$_{FH}$, after the occurrence of event e_{fault} but before the execution of activity HF. All the active activities in the scope will be bypassed and finally end up in the finished state (f_x). Hence, the arc from place f_x to the input transition of place r_{HF} ensures that activity HF can be started only if the normal process of scope Q has been terminated.
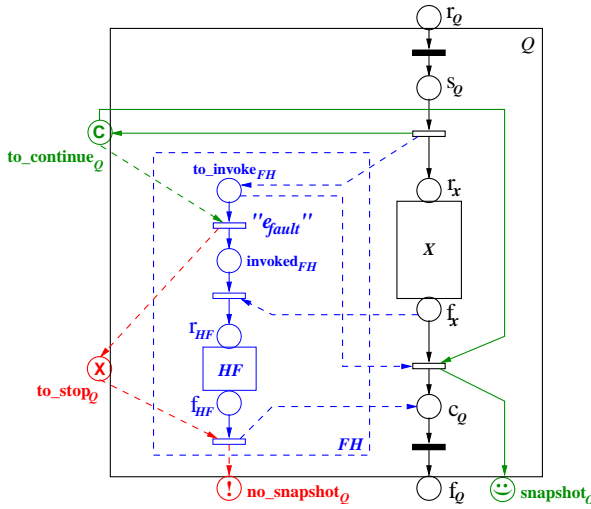
Fig. 7. Pattern of a `<scope>` activity with a fault handler. Four flags are defined for a `<scope>`: to_continue, indicating that the scope is active and no fault has occurred; to_stop, signaling a fault has occurred and all active inner activities within the scope have to stop; snapshot, capturing the preserved state of a successfully completed uncompensated scope; and no_snapshot, indicating the absence of a scope snapshot.

HUB The pattern of a `<scope>` activity of the HUB semantics is depicted in Fig. 8. Firstly, the scope is initialized and the enclosed activity is started. The scope is then in state Active. When the activity completes faultlessly, the scope is completed (transition finalize) and is in state !Active again. Furthermore, the scope is marked to be ready for compensation, similar to the snapshot place of

the QUT semantics. This place and the compensation handler is — to increase the legibility of the pattern — not depicted.
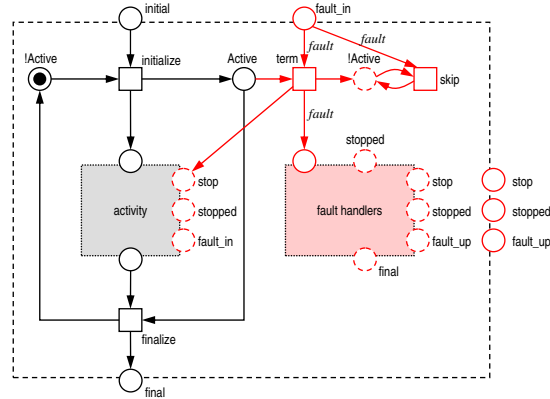
Fig. 8. Pattern of a `<scope>` activity without event, termination, or compensation handlers. The **fault** place of the inner activity is merged with place **fault_in** of the scope. When a fault occurs, transition **term** stops the embedded activity and the scope enters state !**Active**. When the embedded activity has stopped, the fault handlers start. If the fault is handled successfully, the scope finishes and **final** is marked. Otherwise, the fault is propagated: **fault_up** is merged with place **fault_in** of the parent scope.

However, the embedded activity can throw faults to its embedding scope. The **fault** places of all enclosed activities are merged with the **fault_in** of the scope. If an activity throws a fault, a token with the fault name is produced on this place. Transition **term** (forced termination) eventually propagates this token to the fault handlers and also stops the embedded activity. All further faults thrown to **fault_in** are ignored.

Every running activity can be stopped by marking place **stop**, which eventually results in a token on place **stopped**. Given that a BPEL process is a distributed system and the concurrency of multiple instances and scopes is underlined in the BPEL specification, the HUB semantics decides not to *instantly* stop activities, but to model forced termination concurrently. The motivation is to allow for maximal concurrency in the modeling of distributed systems.

After the inner activity has stopped (i.e., place **stopped** is marked), the fault handlers are started. If the fault can be caught by an enclosing `<catch>` or `<catchAll>` branch, the respective activity is executed. The scope is completed when the fault handler is completed faultlessly. However, if the fault cannot be caught or if an activity embedded in the fault handler throws a fault, this fault is propagated to the parent scope by marking place **fault_up**.

While the QUT semantics uses global state places **to_continue** and **to_stop** which have to be read by all embedded activities, the HUB stopping approach is *asynchronous* and is executed in three steps: Firstly, a fault is thrown by

an activity. All concurrent activities can proceed as normal and even throw faults themselves. Secondly, the scope's inner activity is terminated; that is, its stop place is marked. The inner activity and the termination mechanism are concurrent: the moment the control flow stops is chosen nondeterministically (cf. Fig. 2). Finally, after the inner activity has stopped, the fault handlers start.

Conclusion In the QUT semantics, a global stopping mechanism is applied to the mapping of termination. Once a fault is thrown during the execution of an activity within its directly enclosing scope Q, all remaining activities in scope Q stop immediately. For any activity that cannot execute any further, a "bypassing" mechanism is adopted to ensure the control flow reaches the end of scope Q. Then, the fault handling of scope Q starts. Since all remaining activities are bypassed, no further faults can be thrown. In contrast, the HUB semantics models forced termination asynchronously; that is, upon the termination of a scope, all active activities within the scope do not have to stop immediately. However, the fault handlers are only executed if the inner activity has stopped.

5 Verification

5.1 Comparing the Compilers

Both semantics consist of a collection of patterns that are used to translate each BPEL activity into a Petri net pattern. Obviously, this translation is very laborious and takes too much time to be done manually. Therefore, for each semantics exists a compiler that automatically translates a BPEL process into a Petri net model. The HUB compiler supports both BPEL 1.1 and BPEL 2.0, whereas the QUT compiler is restricted to BPEL 1.1.

Currently, both implemented semantics abstract from data since variables in BPEL have an infinite data domain in general. If we would translate the process into a high-level Petri net, it may have an infinite state space which makes subsequent analysis impossible. To this account, both compilers abstract from time and instances due to the limitations of the semantics and they also abstract from data. As a result, models generated by the compilers are low-level Petri nets. Messages and the content of variables are modeled by undistinguishable black tokens. Data dependent branches (e.g., BPEL's <if>) are modeled by nondeterministic choices. In terms of Petri nets such an abstract net is a *skeleton net* [18].

QUT The QUT semantics is implemented straightforwardly in the BPEL2PNML tool[7]. BPEL2PNML directly translates a BPEL process into low-level Petri net in PNML format. Every activity is replaced by its predefined pattern, and nodes (places and transitions, respectively) with identical names are merged to one node.

---

[7] BPEL2PNML is available at `www.bpm.fit.qut.edu.au/projects/babel/tools`.

As it can be seen from the QUT semantics in Sect. 4, the predefined pattern of an activity must suit every possible environment of that activity. Even if the actual activity cannot be skipped, the pattern will include a skip path, as in some other environment the activity can be skipped. Likewise, the pattern always includes support for links and termination, and the scope pattern always includes support for fault handlers, event handlers, and compensation handlers. As a result, the tool BPEL2PNML does not need to check the environment of an activity (and to possibly tune the pattern towards the activity) on the one hand; while on the other hand, the predefined pattern becomes very complicated, which in turn may lead to a very complicated Petri net in the end. Also, it is worth mentioning that a reduction of the unnecessary nodes, such as the skip path of the top level BPEL activity which cannot get skipped by definition, will be performed within the verification tool described in Sect. 5.2.

HUB The HUB semantics is implemented in the tool BPEL2oWFN[8], a compiler translating one or more[9] BPEL processes into a Petri net model. This model can be exported as oWFN or in other file formats (e. g., PNML, PEP, LoLA, INA, SPIN) and thus supports a variety of analysis tools. It does not follow a brute-force mapping approach. Instead, BPEL2oWFN employs *flexible model generation* [14], an approach that minimizes the model both *during* and *after* the translation process, allowing to generate a compact model tailored to the analysis goal.

To this end, BPEL2oWFN builds a *control flow graph* (CFG) of the process and applies standard static analysis techniques (see [19] for an overview) to gain information about the activities, variables, scope hierarchy, and other aspects of the process. This gained information together with the user-defined analysis goal is used to generate the most abstract model fitting to this analysis goal. For this purpose, we implemented a *pattern repository* which contains — in addition to a generic pattern — several patterns for each activity or handler, each designed for a certain context. As an example, consider the `<scope>` activity: For this activity, we provide a pattern with all handlers, a pattern without handlers, a pattern with an event handler only, etc. In general, specific patterns are much smaller than a generic pattern where the absent aspects are just removed. In addition, each pattern usually has diverse variants according to the user-defined analysis goal. For example, a certain analysis goal demands the modeling of the negative control flow or the occurrence of standard faults, whereas another goal does not. In the translation process, for each activity, a Petri net pattern is selected from this pattern repository according to the gained information.

After the translation process, structural reduction rules are applied to further scale down the size of the generated model. For each user-defined analysis goal (e. g., reachability), adequate reduction rules (adapted from [20]) preserving the considered property are selected. Combined with a removal of structurally dead

---

[8] BPEL2oWFN is available at `www.gnu.org/software/bpel2owfn`.

[9] If more than one process is given, the resulting models are composed to a single Petri net.

nodes, the rules are applied iteratively. In addition, the information gathered by static analysis allows for further removal of Petri net nodes that are not affecting the analysis.

The information gained by static analysis introduces domain knowledge into the translation process. In contrast, only domain independent — and thus usually weaker — reduction techniques such as structural reduction are applicable once the model is generated. Case studies show that flexible model generation combining domain-dependent and domain-independent reduction techniques is able to generate very compact models especially tailored to the considered analysis task [14].

Conclusion While the QUT compiler BPEL2PNML merges the patterns used by the input process, the HUB compiler BPEL2oWFN implements several sophisticated reduction techniques to reduce the model during its generation with respect to the verification goal. Furthermore, BPEL2oWFN applies standard Petri net reduction rules that may further reduces the model size.

5.2 Possible Verification

QUT The resulting Petri nets generated by BPEL2PNML are — as mentioned earlier — quite big and usually contain a lot of dead nodes, for instance unnecessary skip parts. For this reason, we "clean" the resulting Petri net from these dead skip paths. In addition, the structure of the net is used applying standard reduction rules [20]. In contrast to the HUB approach, the model is only minimized after the translation.

The QUT semantics is designed to verify the soundness property [6]. The soundness property demands that the process can reach its final state from every reachable state. Thus, a sound process is deadlock-free. Furthermore, a sound net completes properly; that is, without any tokens except on the final place. Soundness is a well-known sanity check for Petri nets that reflect workflows. To verify soundness, we use the tool Woflan [16].

Soundness is verified by generating the reachability graph for the given Petri net. If this is due to state explosion problems not possible, soundness cannot be checked. In such a situation, we might still be able to check relaxed soundness [21] instead. The relaxed soundness property verifies that every transition is covered by *some* execution path that ends in the final state. If a transition is not on such a path, it cannot help in reaching the final state. Relaxed soundness can be approximated using techniques proposed in [22] and implemented in WofBPEL [23].

To verify the correctness of a BPEL process, we can use the proposed tools to translate the BPEL process into a Petri net. On the resulting net, we check soundness and relaxed soundness. If the net is not (relaxed) sound, a counterexample might help the BPEL designer to detect design flaws in the original BPEL process. The generated Petri nets can also be used as input by any Petri net-based model checkers to verify any temporal logic property.

Furthermore, if the reachability graph can be generated, WofBPEL also provides the analysis of BPEL-specific properties such as the existence of conflicting message-receiving activities (a situation which should throw a runtime fault at the BPEL level). In addition, messages that cannot be received anymore in future states can be determined. This information is annotated to the original BPEL model. These annotations can then be used by a BPEL engine to safely remove incoming messages in the message queue, thus optimizing the resource management.

HUB In addition to the translation, BPEL2oWFN also uses the CFG to check 54 of the 94 static analysis requirements[10] proposed by the BPEL specification. They enable BPEL2oWFN to statically detect cyclic control links, read access to uninitialized variables, conflicting message-receiving activities, or other faulty constellations.

Due to the explicit modeling of the interface of a BPEL process, we are able to analyze its communicational behavior. Like the soundness property for WFNs, the existence of a partner process (formalized by the notion of *controllability* [8]) is a minimal requirement for the correctness of an oWFN. An algorithm to decide controllability of an oWFN is implemented in the tool Fiona [12]. This algorithm is constructive; that is, if the oWFN is controllable, a partner is generated. This partner can be translated back to BPEL using existing approaches, for example [24].

Furthermore, to characterize all partners of an oWFN, the notion of an *operating guideline* [25] can be used. The operating guideline is a data structure that enables us to solve other problems concerning communicating services such as *consistency* (a composition of services can always reach a final state), *compatibility* (two services have equivalent external behavior), or *exchangeability* of services (a service can be replaced by another service without loosing a partner service).

Finally, BPEL2oWFN allows to create a closed system of the BPEL process; that is, a Petri net model without interface. This model is similar to a workflow net and can be used to check temporal logic properties using existing model checking tools.

Conclusion While the QUT semantics is tailored to verify (relaxed) soundness, the HUB semantics using oWFNs allows for the analysis of the communication behavior of a BPEL process. Furthermore, BPEL2oWFN can detect many design flaws directly on the structured of the BPEL process. Finally, the resulting nets of both semantics can be analyzed by common model checkers.

---

[10] Most of the static analysis requirements that are not checked by BPEL2oWFN check aspects of XPath and are out of scope of the analysis goals presented in this paper.

6 Case Studies

To underline the impact of the modeling decisions described earlier, we translated several BPEL processes[11] and calculated their state spaces. The processes are toy examples taken from the BPEL specification as well as real-life processes used in other papers.

Table 1. Comparison of net sizes (places and transitions) and state spaces (full state space and reduced state space using partial order reduction).

| | | | QUT | | | | HUB | | | |
| | | | Petri net | | state space | | Petri net | | state space | |
Process	Activities	Scopes	P	T	full	reduced	P	T	full	reduced
Booking	14	1	27	32	33	33	38	56	43	43
Loan 1	7	1	48	47	282	159	64	88	524	177
Loan 2	7	1	49	48	283	167	70	98	292	97
Identity	81	8	110	120	39,053	26,137	168	265	300	285
Purchase	35	4	53	50	190	86	48	80	42	42
Salesforce	22	2	38	43	100	95	68	117	71	71
Travel	8	1	34	32	73	41	37	52	30	28
Vacation	35	3	58	64	967	369	109	174	111	111
Shipping	16	1	21	21	73	43	27	53	23	23
Phone	43	1	74	79	1261	465	116	222	136	136
Auction	17	1	27	31	45	44	59	92	232	128

Table 1 summarizes the results. For each process, we present the sizes of the resulting nets (in terms of places and transitions) after structural reduction. Furthermore, we used the model checker LoLA [26] to calculate the complete state space as well as the reduced state space using partial order reduction. To receive an impression of the size of the processes, the number of activities and in particular the number of scopes are shown.

In general, the nets of the HUB semantics have more nodes. This is due to the more detailed patterns and the stopping concept which introduces many transitions (cf. Fig. 2). For most of the processes with only one scope, the size of the state spaces of the QUT and the HUB nets are very similar. If, however, a lot of scopes and links are involved, the flexible model generation and the different modeling of DPE implemented in BPEL2oWFN permits very compact models. For instance, process "Identity" (a real-life process modeling the application of an identity card) consists of 8 scopes and 17 links: the resulting state space of the HUB net (300 states) is just a fraction of the state space of the QUT net (39,053 states).

---

[11] To use both compilers BPEL2PNML and BPEL2oWFN, the processes only use features of BPEL 1.1.

7 Conclusion

Many researchers have spent efforts in formalizing and analyzing of BPEL processes. In this paper, we compared two feature-complete Petri net semantics for BPEL, their compilers, and tools to support the analysis of certain properties of BPEL processes. The comparison has shown that due to some modeling decisions the two semantics differ in the following issues:

- The HUB semantics is a high-level Petri net which adequately models data. The current translation abstracts from data; thus, the resulting model is a low-level Petri net. In contrast, the QUT semantics is defined as a low-level Petri net from the start.
- The HUB semantics explicitly represents the interface whereas the QUT semantics does not. As a result, the QUT and the HUB semantics result in different models (workflow nets and open workflow nets, respectively) that allow for analyzing different properties.
- In case of dead-path-elimination, the moment when an activity can set the status of its outgoing links to false is different in the two semantics. While the QUT implementation conforms to the BPEL specification, the HUB approach uses an overapproximation resulting in smaller models.
- When a scope has to be terminated due to a fault, the moment when all the activities within the scope stop is different in the two semantics. In the QUT semantics, the activities stop immediately as soon as a fault has occurred to the scope, because a global termination concept is used. In contrast in the HUB semantics, an asynchronous termination concept is used; thus, the activities may stop at any time after the fault occurrence, but before the `<faultHandlers>` start.

Comparing the compilers, QUT's BPEL2PNML applies a brute-force translation and the resulting net is reduced after the translation. The HUB compiler BPEL2oWFN, in contrast, applies an approach of a flexible model generation that reduces the Petri net model during and after the translation with respect to the property to be analyzed. For this purpose, for each BPEL activity, several Petri net patterns with different degree of abstraction are available in a pattern repository. Using static analysis on the BPEL code, we select the most abstract pattern applicable in a given context. As shown in the case study (see Sect. 6), flexible model generation combined with the more efficient implementation of the DPE usually results in very compact models for big BPEL processes. As another difference, BPEL2oWFN allows to check many properties statically on the BPEL code.

We further compared the different properties that can be analyzed on the resulting Petri net model. All models can be analyzed for temporal logic properties using existing model checkers. QUT models, in particular, can be checked for soundness. The workflow structure of the QUT models allows to use efficient algorithms (i. e., performed on the structure of the net rather than on the reachability graph) for WFNs implemented in the tools Woflan and WofBPEL. Since

the HUB models explicitly represent the interface of a BPEL process, they can be used to check controllability of the and to calculate its operation guideline using the tool Fiona.

In ongoing research we work on some tool integration. For example, we think of implementing a global stop concept in the Berlin compiler BPEL2oWFN and study its impact combined with flexible model generation. We also want to spend additional efforts to detect more structural reduction rules that are specific for BPEL processes.

References

1. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., Rijn, D.v.d., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0. OASIS Standard, 11 April 2007, OASIS (April 2007)
2. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM, Microsoft (May 2003)
3. Breugel, F.v., Koshkina, M.: Models and Verification of BPEL. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf` (September 2006)
4. Aalst, W.M.P.v.d., Hee, K.M.v.: Workflow Management: Models, Methods, and Systems. MIT press, Cambridge, Massachusetts (2002)
5. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo (1985)
6. Aalst, W.M.P.v.d.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
7. Martens, A.: Analyzing Web Service based Business Processes. In Cerioli, M., ed.: Proceedings of Intl. Conference on Fundamental Approaches to Software Engineering (FASE'05). Volume 3442 of Lecture Notes in Computer Science., Edinburgh, Scotland, Springer-Verlag (April 2005)
8. Schmidt, K.: Controllability of Open Workflow Nets. In Desel, J., Frank, U., eds.: Enterprise Modelling and Information Systems Architectures (EMISA). Number P-75 in Lecture Notes in Informatics (LNI), Bonner Köllen Verlag (2005) 236–249
9. Girault, C., Valk, R., eds.: Petri Nets for System Engineering – A Guide to Modeling Verification and Applications. Springer-Verlag Berlin Heidelberg New York (January 2002)
10. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri Nets. In Aalst, W.M.P.v.d., Benatallah, B., Casati, F., Curbera, F., eds.: Third International Conference on Business Process Management (BPM 2005). Volume 3649 of Lecture Notes in Computer Science., Springer-Verlag (September 2005) 220–235

11. Ouyang, C., Verbeek, E., Aalst, W.M.P.v.d., Breutel, S., Dumas, M., Hofstede, A.H.t.: Formal Semantics and Analysis of Control Flow in WS-BPEL. BPM Center Report BPM-05-15, Queensland University of Technology, BPMcenter.org (October 2005) `http://www.BPMcenter.org/reports/2005/BPM-05-15.pdf`, accepted for publication by Science of Computer Programming.
12. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting BPEL Processes. In Dustdar, S., Fiadeiro, J.L., Sheth, A., eds.: Fourth International Conference on Business Process Management (BPM 2006). Volume 4102 of Lecture Notes in Computer Science., Springer-Verlag (August 2006) 17–32
13. Lohmann, N.: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0 and its Compiler BPEL2oWFN. Techn. Report 212, Humboldt-Universität zu Berlin (June 2007) to appear.
14. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation. Data Knowl. Eng. (2007) accepted for special issue of BPM 2006.
15. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
16. Verbeek, H.M.W.E., Basten, T., Aalst, W.M.P.v.d.: Diagnosing Workflow Processes using Woflan. Comput. J. **44**(4) (2001) 246–279
17. Leymann, F., Roller, D.: Prentice Hall. Production Workflow – Concepts and Techniques (2000)
18. Vautherin, J.: Parallel systems specifications with coloured Petri nets and algebraic specifications. In Rozenberg, G., ed.: Advances in Petri Nets 1987. Volume 266 of Lecture Notes in Computer Science., Springer-Verlag (1987) 293–308
19. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. 2nd edn. Springer-Verlag (2005)
20. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE **77**(4) (April 1989) 541–580
21. Dehnert, J., Aalst, W.M.P.v.d.: Bridging the Gap Between Business Models and Workflow Specifications. Int. J. Cooperative Inf. Syst. **13**(3) (2004) 289–332
22. Verbeek, H.M.W., Aalst, W.M.P.v.d., Hofstede, A.H.t.: Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. The Computer Journal **50**(3) (2007) 294–314
23. Ouyang, C., Verbeek, E., Aalst, W.M.P.v.d., Breutel, S., Dumas, M., Hofstede, A.H.t.: WofBPEL: A Tool for Automated Analysis of BPEL Processes. In Benatallah, B., Casati, F., Traverso, P., eds.: Proceedings of the Third International Conference on Service Oriented Computing (ICSOC 2005). Volume 3826 of Lecture Notes in Computer Science., Springer-Verlag (December 2005) 484–489
24. Lassen, K.B., Aalst, W.M.P.v.d.: WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. In Meersman, R., Tari, Z., eds.: On the Move to Meaningful Internet Systems 2006. Volume 4275 of Lecture Notes in Computer Science., Springer-Verlag (2006) 127–144
25. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. In: Proceedings of 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007), Siedlce, Poland, June 25-29, 2007. (2007) accepted.
26. Schmidt, K.: LoLA: A Low Level Analyser. In Nielsen, M., Simpson, D., eds.: Proceedings of 21st International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2000). Number 1825 in Lecture Notes in Computer Science, Springer-Verlag (June 2000) 465–474