Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation*

Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
{nlohmann,massuthe,stahl,weinberg}@informatik.hu-berlin.de

Abstract. We address the problem of analyzing the interaction between WS-BPEL processes. We present a technology chain that starts out with a WS-BPEL process and translates it into a Petri net model. On the model we decide *controllability* of the process (the existence of a partner process, such that both can interact properly) and compute its *operating guideline* (a characterization of all properly interacting partner processes). To manage processes of realistic size, we present a concept of a *flexible model generation* which allows the generation of compact Petri net models. A case study demonstrates the value of this technology chain.

Key words: Business process modeling and analysis, Formal models in business process management, Process verification and validation, Petri nets, WS-BPEL

1 Introduction

The *Web Services Business Process Execution Language* (WS-BPEL) [1] is a language for describing the behavior of business processes based on web services. A *WS-BPEL process* or *process* for short can be seen as a workflow enhanced by an interface description specifying the interactional behavior of this process with other processes, its *partners*. The interaction between processes may be nontrivial. Thus, it is a challenging task to decide whether all processes *interact properly*; that is, the interaction is free of deadlocks and there are no messages being sent that cannot be received any more. There are two main reasons for improper interaction: (1) A process may have an erroneous design. For instance, the process may contain an internal choice relevant for the expected behavior of a partner, but the partner is not informed which decision is actually made. (2) The interactional behaviors of two processes exclude each other. For example, the processes run into a situation where one process waits for a message of the other one and vice versa. Therefore, tool support is needed to assist process designers during the modeling.

In [2], we presented a technology chain that starts out with a BPEL4WS [3] process[1], translates it into a Petri net model, and finally analyzes this model.

* Partially funded by the BMBF project "Tools4BPEL".

[1] BPEL4WS 1.1 is the predecessor of the current WS-BPEL 2.0 specification which provides a superset of the BPEL4WS 1.1 constructs.

We introduced two tools: BPEL2oWFN and Fiona. BPEL2oWFN translates a process into an *open workflow net* (oWFN), a special class of Petri nets modeling the interactional behavior of its corresponding BPEL process. Fiona then analyzes whether this oWFN is *controllable*. An oWFN is controllable [4, 5] if there exists a partner such that both can interact properly. Fiona can also calculate the *operating guideline* of a given oWFN. The operating guideline characterizes all properly interacting partners in a compact way [6]. In addition, BPEL2oWFN also supports the translation into Petri nets modeling the internal behavior only. These nets can be analyzed by common model checkers.

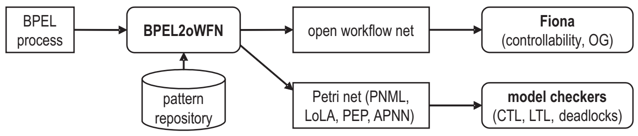

BPEL process
BPEL2oWFN
pattern repository
open workflow net
Petri net (PNML, LoLA, PEP, APNN)
Fiona (controllability, OG)
model checkers (CTL, LTL, deadlocks)

Fig. 1. Proposed tool chain to analyze WS-BPEL processes.

In this paper, we improve the proposed technology chain presented in Fig. 1. Firstly, BPEL2oWFN is no longer restricted to processes specified in BPEL4WS. It now *also* translates processes compliant to the current WS-BPEL specification. To the best of our knowledge, this is the only formal semantics for WS-BPEL 2.0. Compared to the BPEL4WS specification, WS-BPEL 2.0 contains many new activities and other constructs, and clarifies the semantics of several complex scenarios such as compensation. Secondly, we further improved the translation to generate compact Petri net models. We introduce our concept of a *flexible model generation* that was only sketched in [2]. The idea is to have, for each WS-BPEL construct, several patterns that are applicable in different contexts. Using static analysis on the WS-BPEL code, information is derived to select the most abstract pattern applicable in a given context. With the help of data abstraction, the complexity of WS-BPEL processes of realistic size is reduced in such a way that they can be efficiently analyzed.

It is worthwhile to mention that the concepts of this paper are not restricted to analyze WS-BPEL processes only. The concept of flexible model generation as implemented in BPEL2oWFN can be applied to any other process description language and likewise to any other target formalism. Our presented model, open workflow nets, is a general formalism that can be used to model various kinds of interacting processes. Since the algorithms of Fiona are based on oWFNs we are not restricted to a specific process description language like WS-BPEL.

The paper is organized as follows. In Sect. 2, we introduce the general concepts of WS-BPEL and our model, open workflow nets. We also explain controllability of oWFNs and the operating guideline of an oWFN. A WS-BPEL example process, an online shop, is presented in Sect. 3. Our main contribution is presented in Sect. 4, where we explain the concepts of our advanced translation and especially flexible model generation. It is exemplified by translating the online shop into an oWFN. The resulting oWFN is then analyzed in Sect. 5. We present a slightly modified version of that process in Sect. 6 and analyze it, too. In Sect. 7, we describe related work. Finally, we conclude with some directions for future research.

2 Background

2.1 WS-BPEL

The *Web Services Business Process Execution Language* (WS-BPEL) [1] is a language for describing the behavior of business processes based on web services. For the specification of a business process, WS-BPEL provides *activities* and distinguishes between basic and structured activities. A basic activity can communicate with the partners by messages (`invoke`[2], `receive`, `reply`), manipulate or check data (`assign`, `validate`), wait for some time (`wait`) or just do nothing (`empty`), signal faults (`throw`), or end the entire process instance (`exit`).

A structured activity defines a causal order on basic activities and can be nested in another structured activity itself. The structured activities are sequential execution (`sequence`), parallel execution (`flow`), data-dependent branching (`if`), timeout- or message-dependent branching (`pick`), and repeated execution (`while`, `repeatUntil`, `forEach`). The most important structured activity is a `scope`. It links an activity to a transaction management and provides fault, compensation, event, and termination handling. A `process` is the outmost scope of the described business process.

Activities embedded in a `flow` can be further ordered by *links*. A link connects a source activity with a target activity. The source may specify a Boolean expression, the status of the link. The target may also specify a Boolean expression (the join condition) which evaluates the status of all incoming links. The target activity is only executed when its join condition holds. WS-BPEL provides *dead-path elimination*; that is, the status of all outgoing links of a skipped source activity (e. g., due to an unchosen `pick` branch) are set to false.

A Fault handler is a component of a scope that provides methods to handle faults which may occur during the execution of its enclosing scope. Moreover, a compensation handler can be used to reverse some effects of successfully executed activities. Compensation of a process is started with the `compensate` activity which calls the compensation handlers of completed scopes in reverse execution order. With the help of event handlers, message events and specified timeouts can be handled. Finally, a termination handler controls the behavior during the termination of a scope.

2.2 Open Workflow Nets

Open workflow nets (oWFNs) [7] are a special class of Petri nets and can be seen as a generalized version of van der Aalst's workflow nets [8]. As a substantial

[2] We use a typewriter font for WS-BPEL activities.

difference, in an oWFN the interface of a process is explicitly represented as sets of input and output places. In our model, we assume asynchronous message passing. We concentrate on control flow aspects of processes and abstract from data like the content of messages, for instance. For data with finite domain, however, important message content can be represented by unfolding the data domain. Hence, oWFNs provide a simple but formal representation of processes, still preserving sufficient information to analyze proper interaction of such processes. We will describe our approach in a section later on.

We assume the usual definition of Petri nets. An *open workflow net* is a Petri net $N = (P, T, F)$, together with an *interface* $I = in \cup out$ such that $I \subseteq P$, $in \cap out = \emptyset$, and for all transitions $t \in T$ it holds: if $p \in in$ ($p \in out$), then $(t, p) \notin F$ ($(p, t) \notin F$); a distinguished marking m_0, called the *initial marking*; and a set Ω of distinguished markings, called the *final markings.* The places in *in* (*out*) are called *input* (*output*) places. The *inner* of an oWFN N can be obtained from N by removing all interface places, together with their adjacent arcs. As a convention, we label a transition t connected to an input (output) place x with $?x$ ($!x$). As an example, consider the oWFN N_1 depicted in Fig. 2.

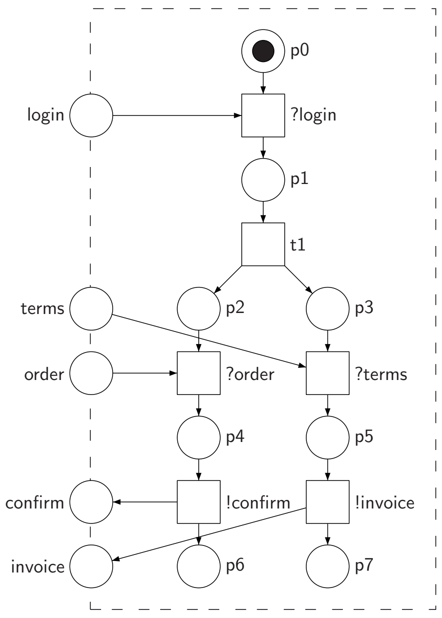


Fig. 2. An example oWFN N_1. The net has three input places, login, terms, and order and two output places, confirm and invoice. The initial marking m_0 is [p0] which denotes one token on place p0. N_1 has only one final marking, [p6,p7].
In m_0 the net waits for the login message from a partner. If the message arrives, transition ?login can fire and produces a token on place p1.
Then, firing transition t1 yields the marking [p2,p3]. This means that the net is ready to concurrently receive an order message (order) and a terms of payment message (terms). The order is confirmed (!confirm) and the terms of payment are followed by an invoice (!invoice). If both messages are received, the final marking [p6,p7] is reached.

The interplay of two oWFNs N and M is represented by their *composition*, denoted by $N \oplus M$. Thereby, we assume that the nets only share input and output places such that an input place of N is an output place of M and vice versa. Then, $N \oplus M$ can be constructed by merging joint places and merging the initial and final markings.

A marking (also called a *state*) m of N is called a *deadlock* if m enables no transition. An oWFN in which each deadlock is a final state is called *weakly terminating*. Obviously, the net N_1 in Fig. 2 itself is *not* weakly terminating. N_1 requires a partner who sends and receives messages. N_1 is not able to reach its final marking [p6,p7] on its own. Given an oWFN N, we call an oWFN S a *strategy* for N iff $N \oplus S$ is weakly terminating. A weakly terminating oWFN can still contain livelocks. In a livelock, the net changes its state without progressing and eventually terminating.

oWFNs can be canonically extended to high-level oWFNs to model data. However, during our analysis only low-level oWFNs are used. Throughout this paper we restrict ourselves to oWFNs N and S where $N \oplus S$ can reach only a finite number of states. Technically, we demand that the inner of N and the inner of S have finite state spaces and that there exists a number b, called *message bound*, such that any message exchanged between N and S is only sent at most b times.

2.3 Controllability of oWFNs

Intuitively, controllability of an oWFN N means that N can properly interact with some other net. Formally, N is *controllable* iff there exists a strategy for N. Like the soundness property for workflow nets (cf. [8]), controllability is a minimal requirement for the correctness of an oWFN.

In [5], we developed an algorithm to efficiently decide the controllability of an oWFN N. Intuitively, the algorithm tries to construct (synthesize) a strategy (i. e., an oWFN S), which imposes the weak termination property of $S \oplus N$. If the construction fails, N is not controllable. If it succeeds, N is controllable and we have constructed a strategy S. This construction is, in fact, a problem known in the literature as *controller synthesis* (see [9]). Technically, we do not construct a strategy S, but an *automaton* that reflects the interactional behavior of S instead. To avoid confusion, we call the constructed automaton *controller*, but denote it with S as well.

The idea of the construction algorithm is as follows. A node of the controller S represents the set of all states that N can reach by consuming (already present) messages or by producing messages itself. The actual state of N is hidden for S. S knows the history of sent and received messages only. From that information, in each node, S can deduce a *set* of states of N which *contains* the state that N is really in. Thus, a node of S represents a *hypothesis* of S with respect to the actual state of N. If a hypothesis contains a (nonfinal) deadlock, then S has to *resolve* that deadlock by receiving or sending messages.

As an example, a controller for the oWFN N_1 (see Fig. 2) is depicted in Fig. 3. It is easy to see that each deadlock in any node (except for the final marking in the last node) is resolved. Hence, a controller could be constructed and we conclude that N_1 is controllable. There are many other controllers for N_1 that could also be found.

As a last step, a controller could be transformed into an oWFN by using the theory of regions [10], for instance. This oWFN is then a strategy by construction.

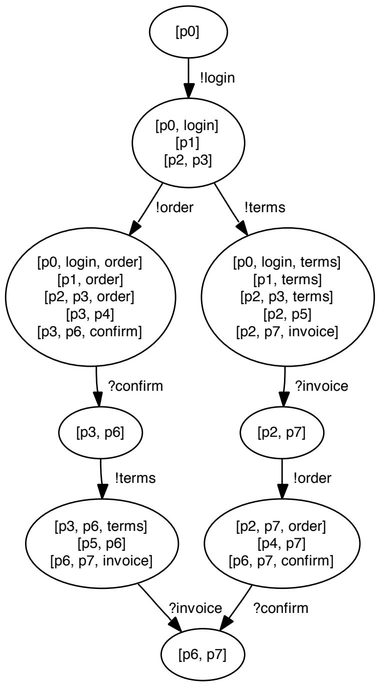


Fig. 3. A controller S for the net N_1 of Fig. 2. Its first node represents the hypothesis that S has about N_1 when neither messages have been sent nor received: the net must be in state [p0], which is a deadlock. Hence, the first node contains the state [p0] only.
However, sending a login message resolves the deadlock. So we add an edge labelled with the sending event !login and a new (yet empty) node to the controller. N_1 is now in state [p0,login] and may fire transition ?login reaching the state [p1]. After successively firing all enabled transitions, the next reachable deadlock is [p2,p3]. So the new node contains the states [p0,login], [p1], and [p2,p3].
Now one of two sending events is possible: !order or !terms. So we add two edges and two empty nodes, and so on.
The last node of the controller represents the states where N_1 can be in after all the messages are exchanged. There is only one deadlock, [p6,p7], in that node which is the final marking of N_1.

2.4 Operating Guidelines of oWFNs

While controllability aims at constructing *one* strategy for a given oWFN N, the *operating guideline* (OG) of N is used to characterize *all* strategies (i. e., all properly interacting partners) of N [6]. As we did in the section before, we do not directly represent the strategies as oWFNs, but represent their behaviors as automata. With the help of the OG of N, it can easily and efficiently be predicted whether or not an oWFN S is a strategy for N before actually composing S and N.

The construction idea is as follows. The basis of the OG is again a controller, denoted by $\mathcal{S}$. As the main difference to the controller which is constructed to decide controllability, $\mathcal{S}$ must be *most permissive.* That means, $\mathcal{S}$ must contain all possible controllers of N. Therefore, we first construct a full automaton $\mathcal{F}$ that performs *each* event at any node. This automaton is finite since N has only a finite number of states. This is true because, by assumption, the set of inner states of N is finite and the possible state of each interface place is bounded by b. $\mathcal{S}$ is now constructed from $\mathcal{F}$ by removing — in an iterative process — such nodes that contain (nonfinal) deadlocks which are not resolved. In [11], we have shown that the resulting $\mathcal{S}$ is indeed most permissive and well-suited as a basis for the OG. In a last step, we add *Boolean annotations* to the nodes of $\mathcal{S}$ that describe how we can obtain other controllers from $\mathcal{S}$ [11]. The most permissive controller together with the annotations is called the operating guideline of N.

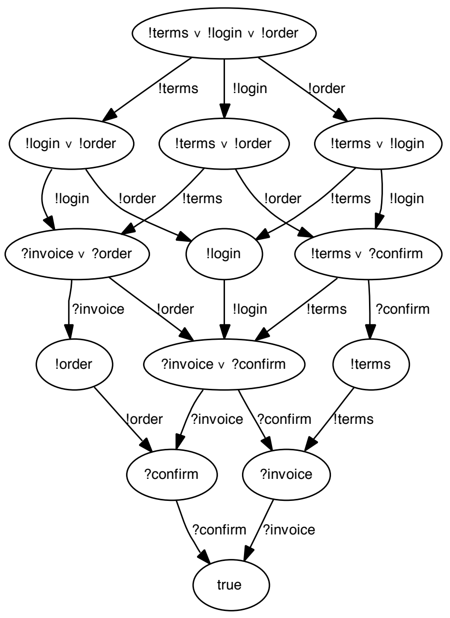


Fig. 4. The OG of the oWFN N_1 of Fig. 2. The annotation of the first node is the disjunction !terms ∨ !login ∨ !order. It means that every controller must, as its first event, send one of the three corresponding messages. The controller in Fig. 3, for instance, performs the event !login which is obviously correct according to the OG. The OG also allows a controller which first sends its order to N_1. This possibility results from the proposed asynchronous way of interaction. Even if the order was sent first, it would keep pending on the place order until N_1 has consumed the login message sent later.
The annotation true of the last node means that no event has to be performed any more.
In sum, the OG of N_1 characterizes 77 different controllers for N_1, including the controller in Fig. 3.

Figure 4 shows the operating guideline of the oWFN N_1 depicted in Fig. 2. In a node of the OG, the corresponding annotation is depicted. The reachable states of N_1 are hidden.

Given a controller representing an intended partner's behavior, we developed an algorithm to check whether it is characterized by a given OG or not [11].

3 Example Process: Online Shop

In this section, we present an online shop as our example process. It is a nontrivial extension of the process presented in [2]. The online shop's WS-BPEL process consists of 25 activities, a fault handler, an event handler, and a compensation handler. The shop is depicted in Fig. 5 in a common graphical notation.

When the online shop receives the login information from a customer, its business strategy distinguishes between premium and standard customers (upper `if` statement). The premium customer gets a special offer from the shop, for example a discount for the items to be purchased. The standard customer, however, receives a confirmation message (confirm) and has to accept the terms of payment first and does not get any kind of offer. Then, either customer can buy goods at the shop. This is modeled with a `repeatUntil` loop which initially demands the user to place an order (please_order). As the customer chooses an item to buy (order), all the necessary information is stored in a database. This procedure is not modeled in our WS-BPEL process. If the ordered item is available, the shop sends out a notice that the order has been accepted (order_accepted).

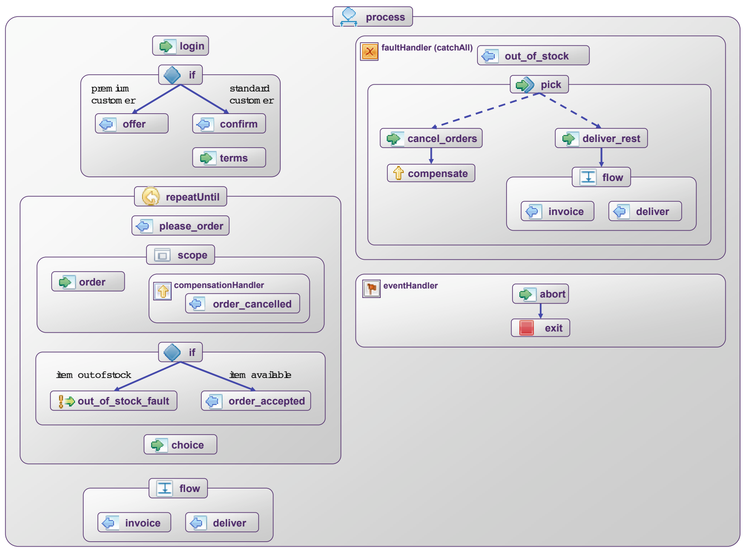


Fig. 5. The online shop process.

The customer may then choose another article to add to his cart or he may finish shopping (modeled by the Boolean choice message). In the end, the shop concurrently sends out the invoice and delivers the goods purchased (deliver).

In case an item is out of stock, the shop throws the out_of_stock_fault to its fault handler. Now the fault handler first informs the customer (out_of_stock) and gives him the possibility to cancel the whole order or to receive the articles chosen so far (out_of_stock). If the customer cancels all orders (cancel_orders), each order is compensated (compensate) by the shop, meaning that the respective database entries are deleted. To model this procedure, we scoped the order process and linked it to a compensation handler attached to it. That way, we make sure that each order will be compensated. In case the customer would like to get all chosen items (deliver_rest), an invoice is generated and the articles are sent out (deliver) concurrently.

The customer may send an abort message at any time. We modeled this as an event handler that receives the abort message and then terminates (exit) the whole process.

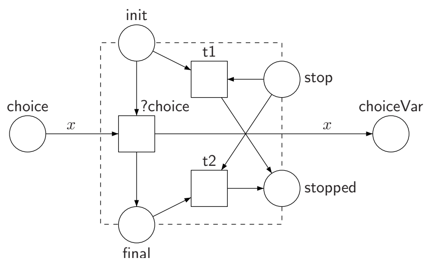
4 Translating WS-BPEL to Open Workflow Nets

4.1 Open Workflow Net Semantics for WS-BPEL

We aim at formally analyzing WS-BPEL processes. To achieve this aim, we translate a WS-BPEL process into an oWFN. The translation is guided by the syntax of WS-BPEL. In WS-BPEL, a process is built by plugging instances of WS-BPEL constructs together. Accordingly, we translate each construct of the language separately into an oWFN. Such a net forms a *pattern* of the respective WS-BPEL construct. Each pattern has an interface for joining it with other patterns as it is done with WS-BPEL constructs. The semantics aims at representing all properties of each WS-BPEL construct within its respective pattern. An example for an oWFN pattern is depicted in Fig. 6. As the semantics itself is not the focus of this paper, we only summarize the main ideas of it.

```
<variables>
  <variable name="choiceVar"
    type="xsd:boolean" />
</variables>
              ...
<receive
  partnerLink="customer"
  operation="choice"
  variable="choiceVar">
</receive>
```

(a) `receive` activity



(b) corresponding oWFN pattern

Fig. 6. The dotted box frames the pattern. The places on the frame (init, final, stop, and stopped) describe the interface of the pattern used to plug it with other patterns. The execution of the activity can be stopped at any time by marking place stop and firing either t1 or t2. The pattern models data: the input place choice and the variable place choiceVar are high-level places that are marked with tokens of type *xsd:boolean*.

The collection of patterns forms our oWFN semantics for WS-BPEL. The semantics is *complete*; that is, it covers all the standard and exceptional behavior of WS-BPEL such as fault, compensation, event, and termination handling, including arbitrarily nested scopes and repeatable constructs such as loops. In addition, all data aspects of WS-BPEL are modeled. Furthermore, the semantics is *formal*, meaning it is suitable for computer-aided verification. Currently, the semantics is subject to two restrictions. Firstly, only a single instance of a WS-BPEL process can be translated into an oWFN model. Thus, partner links cannot be dynamically changed. Secondly, time (e. g., for the `wait` activity) is not modeled.

The original semantics [12] was designed to formalize BPEL4WS rather than to create compact models that are necessary for computer-aided verification. Some patterns were easy to understand yet made use of quite "expensive" constructs such as reset arcs. We improved these patterns and replaced them by less intuitive patterns with simpler structure. In particular, the complex interplay of the fault, compensation, event, and termination handlers was condensed. As the main extension, the new semantics [13] covers all features of WS-BPEL 2.0, the successor of BPEL4WS 1.1.

4.2 Flexible Model Generation in BPEL2oWFN

The extended semantics is implemented in the tool BPEL2oWFN[3]. It is capable of generating oWFNs and other file formats (e. g., PNML, PEP, LoLA, INA, SPIN) and thus supports a variety of analysis tools. In contrast to its predecessor BPEL2PN [14], it does not follow a brute-force mapping approach which resulted in huge models for processes of realistic sizes. This enables a more efficient analysis.

The main reason for a vast model size is the translation of dead code; that is, parts of the WS-BPEL process that are unreachable. Furthermore, aspects that are not necessary for the analysis goal (e. g., controllability, deadlock-freedom) can be skipped. To scale down the model size, BPEL2oWFN employs *flexible model generation*, an approach to generate a compact model tailored to the analysis goal. The model is minimized both *during* and *after* the translation process. In contrast, similar tools (e. g., WofBPEL [15]) only apply structural reduction rules after performing a brute-force translation. Figure 7 presents an overview of our proposed framework.

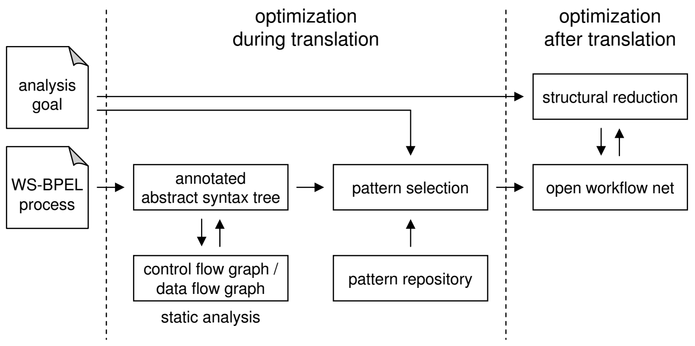

optimization during translation
optimization after translation
analysis goal
WS-BPEL process
annotated abstract syntax tree
pattern selection
structural reduction
open workflow net
control flow graph / data flow graph
static analysis
pattern repository

Fig. 7. Framework for flexible model generation implemented in BPEL2oWFN. For a WS-BPEL process the abstract syntax tree (AST) is built. For each WS-BPEL construct context information is gained from static analysis techniques and annotated to the AST. In the translation process, the annotated AST is used together with a user-defined analysis goal to select the most abstract patterns from a pattern repository. The resulting open workflow net is then structurally reduced.

Firstly, the input WS-BPEL process is parsed and an *abstract syntax tree* (AST) is built. It contains information about the syntax of the process such as structure, contained activities, and used attributes, for instance. All information gained from further analysis steps will be annotated to the AST. Later on the annotated AST is used to generate the oWFN model. The AST is also used to apply implicit transformation rules specified in the WS-BPEL specification to WS-BPEL constructs. Those transformation rules are a short-hand notation,

[3] available at `http://www.informatik.hu-berlin.de/top/tools4bpel`

that will be extended as the process is interpreted. For example, a `scope` can be specified without handlers. Then, an implicit transformation rule demands, that the standard handlers have to be added to this `scope`. Another example is that an `invoke` activity my have a nested fault handler. Here an additional `scope` has to be added according to a transformation rule.

However, the AST is not suitable for more sophisticated analysis steps as it only represents the syntactical structure of the input process. To this end, BPEL2oWFN builds a *control flow graph* (CFG) of the process. This graph represents all execution paths (i. e., all orders of activities) the process might traverse. Hence, CFG analysis results can be safely mapped back to the original process. All obtained results are annotated to the AST and affect the subsequent translation process.

On the CFG, standard static analysis techniques (see [16] for an overview) are applied. Among others, the scope hierarchy and the link dependencies can be derived from the CFG and used for further analysis. In addition to the control flow, we also add the data flow between activities and variables to the CFG. In our current work, we only model the data *flow*, and do not take data *values* into account. However, storing information about the data flow, for example from a `receive` to a variable and then to a `reply` allows to detect read access to uninitialized variables. The CFG with annotated data flow is the data structure for the following static analysis techniques.

On the CFG, we can estimate the control dependency relation which describes the partial execution order of the activities. This relation is used to calculate the compensation order of scopes. WS-BPEL provides the concept of dead-path elimination which is applied if an activity is skipped (e. g., due to an unchosen `pick` branch). In this case, all directly or indirectly embedded links have to be set to false. Instead of setting theses links to false recursively one by one, the control dependency relation allows to set these links to false *immediately* when the considered activity is skipped. Replacing the recursive setting of links to one step avoids many intermediate states in the resulting model.

It is also possible to detect dead code. The scope hierarchy together with the control flow allows to identify compensation handlers that will never be called since an according `compensate` activity is missing. Similarly, termination and fault handlers can be marked unreachable. Thus, they will be skipped during the subsequent translation process. Moreover, the CFG is also used to optimize the resulting model. For example, the flow of faults from their source (e. g., a `throw` activity) to the fitting fault handler can be explicitly modeled, avoiding unnecessary intermediate states or routing constructs.

So far we only gained context information and used this information to annotate the AST. In the next step, the AST 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 node of the AST, a Petri net pattern is selected from this pattern repository according to the annotations. The process is finally translated by composing all selected patterns to a single oWFN.

Currently, the implemented patterns abstract from data since variables in WS-BPEL have an infinite data domain in general. If we would translate the process to a high-level oWFN, it may have an infinite state space which makes subsequent analysis impossible. On this account, BPEL2oWFN abstracts from time and instances due to the limitations of the oWFN semantics and it also abstracts from data. As a result, models generated by BPEL2oWFN are low-level oWFNs. Messages and the content of variables are modeled by undistinguishable black tokens. Data dependent branches (e. g., WS-BPEL's `if`) are modeled by nondeterministic choices. In terms of Petri nets such an abstract net is a *skeleton net* [17]. Figure 8(c) illustrates this idea. Unfortunately, the skeleton net only weakly preserves controllability: if the skeleton net is controllable, so is the high-level oWFN and therefore the WS-BPEL process is controllable, too. The converse does not hold in general. The proof of weak preservation is subject of current research. In case a process has a finite data domain (e. g., data of type Boolean), the resulting high-level oWFN net can be unfolded into a low-level oWFN without loss of information. Figure 8(d) illustrates this.

After the translation process, structural reduction rules are applied to further scale down the size of the generated oWFN model. For each user-defined analysis goal (e. g., controllability or reachability), adequate reduction rules (adapted from [18]) preserving the considered property are selected. Combined with a removal of structurally dead nodes, the rules are applied iteratively. In addition, the information gathered by static analysis allows for further removal of oWFN nodes that are not affecting the analysis. For example, we remove places that model the negative state of a link which will never be set to true.[4]

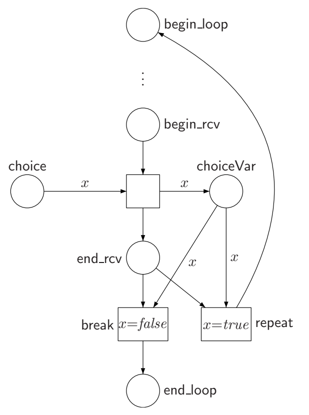
The annotation of the AST with the information gained by static analysis introduces domain knowledge into the translation process. In contrast, only domain independent — and thus usually weaker — reduction techniques such as structural reduction are applicable once the model is generated. We claim that 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. Furthermore, the concept of flexible model generation is independent from the output formalism, and can be adapted to other formalisms, for example process algebras or finite state machines. In addition, also the input language is not restricted to WS-BPEL.

[4] In some cases, it is more pragmatic to remove a node at this stage of the translation process, because gathering more information and not creating the node in the first place would require higher effort.

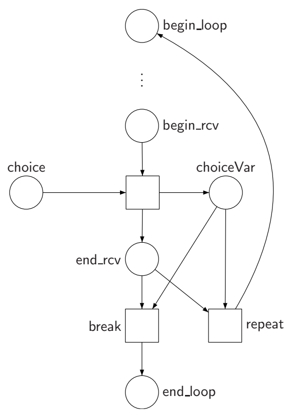
```
<variables>
  <variable name="choiceVar"
    type="xsd:boolean" />
</variables>
            ...

<repeatUntil>
            ...
  <receive
    partnerLink="customer"
    operation="choice"
    variable="choiceVar" />
  <condition>
    $choiceVar = false
  </condition>
</repeatUntil>
```

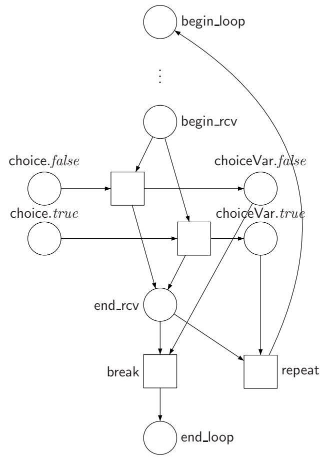
(a) WS-BPEL



(b) high-level



(c) skeleton



(d) unfolded

Fig. 8. A snippet of the `repeatUntil` activity taken from the online shop in Fig. 5 is presented in (a). The customer controls the loop by sending the message choice. The message content is stored in the variable choiceVar and is used to evaluate the loop condition. In case of a message with content *true* (*false*), the loop is repeated (exited). (b) shows the high-level model of the presented code. The loop condition is modeled by two transition guards depicted inside the transitions repeat and break. For purposes of simplification the stop places are not shown. The skeleton net is depicted in (c). Messages are undistinguishable tokens and the condition is evaluated nondeterministically. As the data domain is of type *xsd:boolean*, the high-level places of (b) can be unfolded into two places *true* and *false*, shown in (d).

The generated model can now be analyzed according to the analysis goal. In this paper, we focus on the analysis of the communicational behavior of WS-BPEL processes and introduce the tool Fiona in Sect. 5. However, detection of unreachable activities by using model checkers such as LoLA is possible, too.

In addition to the presented CFG-based algorithms, BPEL2oWFN also uses the CFG to check 44 of the 94 static analysis requirements[5] proposed by the WS-BPEL specification. They enable BPEL2oWFN to statically detect cyclic control links, read access to uninitialized variables, conflicting receive activities (two concurrent receive activities are waiting for the same input message), or other faulty constellations.

4.3 Translating the Online Shop

Using BPEL2oWFN, we now translate the online shop example process into an oWFN. Since we want to calculate the operating guideline of the process, we use patterns preserving all strategies. The translation generates a skeleton net except for the input message choice and its associated variable choiceVar (cf. Fig. 8(a)), which are unfolded as depicted in Fig. 8(d). Thus, the loop can be controlled by the customer by sending messages to the unfolded input places choice.*true* and choice.*false*, respectively.

The flexible model generation approach especially optimizes the translation of several handlers of the online shop. For example, the `scope` has a standard fault and termination handler that are added when building the AST. However, CFG analysis marks these handlers unreachable, as the `scope` can neither throw a fault nor can it be forced to abort due to a fault. Thus, a very compact pattern can be chosen from the pattern repository to translate this scope.

Table 1. Comparison of the different translation approaches: brute-force translation without optimization, optimization during translation and optimization during and after translation (flexible model checking).

translation approach	places	transitions	arcs
brute-force translation, no structural reduction	217	245	882
brute-force translation, structural reduction	155	181	646
flexible model generation, no structural reduction	140	176	558
flexible model generation, structural reduction	100	122	379

Table 1 compares the sizes of the resulting oWFNs of the different translation approaches. Flexible model generation with subsequent structural reduction rules yields the smallest oWFN. Also the state space of the inner of the resulting oWFN is affected: The number of reachable states is reduced from 6358 (brute force, no structural reduction) to 543 (flexible model generation, structural reduction). As the nodes of the IG consist of subsets of reachable states, this reduction rigorous for the analysis of WS-BPEL processes of realistic size.

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

5 Analyzing the Interaction of oWFNs

5.1 The Tool Fiona

Fiona[6] is a tool to automatically analyze the interactional behavior of a given oWFN N. The input format of Fiona is the oWFN output format of BPEL2oWFN. Hence, we can easily analyze WS-BPEL processes. Fiona provides two techniques: it checks for the controllability of N or it calculates the operating guideline of N.

The algorithm to construct the OG as described in Sect. 2.4 performs three steps: first, a full automaton $\mathcal{F}$ is constructed, then, nodes with unresolved deadlocks are iteratively removed from $\mathcal{F}$ to get the controller $\mathcal{S}$, and finally, the annotations are computed from the state information in the nodes of $\mathcal{S}$. Since $\mathcal{F}$ is a theoretical construct which is enormously big for practical processes, the implemented algorithm in Fiona is different:

1. The construction of $\mathcal{F}$ is based on a depth-first approach, with the three steps of the theoretical algorithm being interleaved. The data structure that we use for the controller is called interaction graph (IG) [5]. The graph in Fig. 3 is the IG of the oWFN N_1 as it is computed by Fiona.
2. Static analysis information of the process calculated by BPEL2oWFN can be used to suppress events at a node to substantially reduce the number of computed nodes of the IG.
3. In some cases we can conclude that a node has to be removed later on before knowing all of its successors. Then, the successors do not have to be computed since they would be removed as well.
4. To speed up the computation, the nodes that contain (nonfinal) deadlocks are not removed. They are stored and marked. That way, a repeated computation of that node does not mean a repeated computation of its successors. Obviously, this increments memory consumption but is reasonable.
5. To compute the reachable states of N inside a node of the IG, we use techniques and data structures that were implemented in the model checking tool LoLA [19]. For example, partial-order reduction techniques are applied.

If Fiona is used to decide controllability, it is sufficient to construct a much smaller controller. Thus, several further reductions can be applied while the IG is constructed [5]. For example, if it is possible to receive one message or to send another message at one particular node, it is sufficient to only consider the receiving event and to skip the sending event.

5.2 Analyzing the Online Shop Model

We now analyze our online shop example from Sect. 3. Firstly, we use Fiona to calculate the IG of the corresponding oWFN which we got from Sect. 4.3. The computed IG consists of 14 nodes and 14 edges (see Fig. 9). Since the controller is nonempty, the online shop is controllable. The IG reflects one particular controller that orders exactly one item.

[6] available at `http://www.informatik.hu-berlin.de/top/tools4bpel`

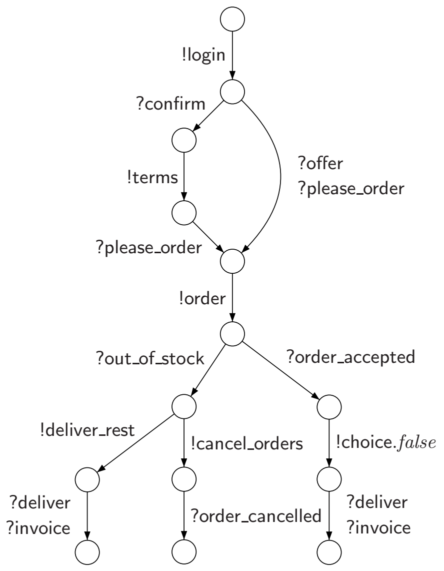


Fig. 9. A controller of the online shop. For better readability we do not show the set of states (the hypothesis) stored in each node. The labels of the edges represent the communication between the controller and the online shop (cf. Sect. 2.3). At some edges of the IG we depicted two receiving message. That means, the controller may receive those messages in arbitrary order. This is possible because we assume asynchronous message passing.

The controller of Fig. 9 reflects the intended behavior of a customer. First, he sends a login (!login). Afterwards, he must be able to either receive the confirmation (?confirm) in case he is a standard customer or to receive the shop's offer (?offer) in case he is a premium customer. If he has received the confirmation, he must send a terms of payment message (!terms). In either case the customer receives a message from the shop that he can start ordering (?please_order). After placing the order, the customer has to be able to receive an out of stock message (?out_out_stock) or to receive a message stating that the order is accepted (?order_accepted). After the order has been accepted, this customer finishes by sending an appropriate message (!choice.*false*). Then he receives the delivery notification (?deliver) and the invoice (?invoice). In case the item is out of stock (?out_of_stock), the customer can choose between receiving all ordered items (!deliver_rest) or he cancels the whole order (!cancel_orders). After canceling, he must be able to receive the cancel confirmation (?order_canceled). If he wants to receive the order, he gets the items and the invoice.

We let Fiona compute the OG of the shop, too. The OG consists of 67 nodes and 146 edges and includes customers that buy multiple items (bounded by the message bound b). The graph is too big to be depicted in this paper. Compared to the IG of Fig. 9, the OG also contains more interleavings of sending or receiving messages. For instance, a customer may reverse the order of sending the login and the order message.

6 The Online Shop Revised

Let us take a look at the online shop presented in Sect. 3 once again. The shop now slightly modifies its business strategy: every premium customer may choose a gift after login. The modified online shop is depicted in Fig. 10(a).

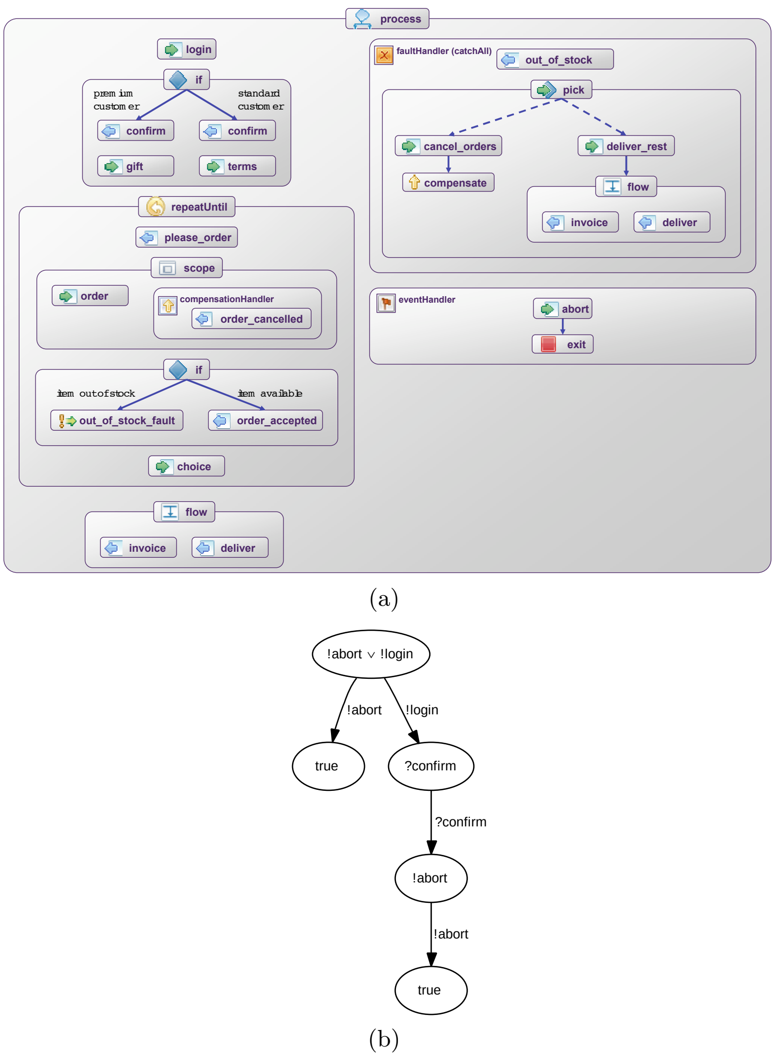


Fig. 10. The modified online shop (a) and its operating guideline (b).

The changes only affect the left branch of the first `if`-statement. The shop sends a login confirmation (confirm) first and then expects the customer to choose a gift. The rest of the process is as in Fig. 5.

The analysis with Fiona reflects that this simple change has a crucial effect on the behavior of the process. Our algorithm concludes that the process is controllable, too. The IG of the revised online shop has only 5 nodes and 4 edges, which is less than the controller for the original shop. However, the reflected strategy is not an expected one. The controller in the IG represents a customer who sends an abort message during the interaction.

The IG represents only one customer's behavior. For further information we need Fiona to calculate the OG. It is depicted in Fig. 10(b). Independent from the message bound it consists of the same 5 nodes and the same 4 edges as the IG. The OG reveals that actually *every* customer of the modified shop must eventually send an abort message. This surely means that the process is controllable. However, the way this is done is obviously not intended. There is no way that a customer can order an item and gets an item delivered.

Let us take a look at what went wrong when we modified our online shop from Sect. 3. We can see that the shop does not communicate its inner decision about which branch (premium customer, standard customer) is chosen. In the original online shop (Fig. 5) the controller received an offer from the shop or a confirmation message according to which branch the shop has chosen before. That way, the controller can conclude which branch the shop is actually in and hence knows how to continue. In contrast in the modified shop, the controller receives undistinguishable confirmation messages in either case. The modified shop expects a choice of a gift in case it decided for the premium customer branch. In the other case it expects the terms of payment. The controller, however, does not know about the decision of the shop. That means, it does not know which message to send. This is reflected by the OG of the new shop (see Fig. 10(b)): in the situation, where a partner receives the confirmation he does not know whether the shop decided for the left or the right branch. Hence, he could choose either to send a gift choice or the terms of payment. In either case it is not guaranteed that the message will always be consumed, and therefore it should not be sent in the first place. However, sending an abort is always correct.

This simple example shows that even a small modification of a process may result in an unintended interactional behavior. The effect on the interactional behavior of a process is not obvious. Since this is not obvious in our shop example, it is even more challenging for processes from industry. In general, processes may have a complex structure that it is not possible to detect such erroneous structures in the process manually. With the help of the operating guideline, we can see if there exists a controller that interacts with our process as we have expected it during the process design.

7 Related Work

Several groups have proposed formal semantics for BPEL4WS 1.1. Among them, there are semantics based on finite state machines [20, 21], the process algebra Lotos [22], abstract state machines [23, 24], and Petri nets [25, 12]. However, to the best of our knowledge, the Petri net semantics proposed in this paper is the only formal semantics for WS-BPEL 2.0.

The groups of van der Aalst and ter Hofstede also follow a Petri net-based approach [25]. Their semantics, however, is restricted to BPEL4WS and does not cover the communication of processes. Furthermore, they do not consider data and, as in our approach (see Sect. 4.1), time is not modeled and only a single process instance is translated. Thus, the resulting models are (low-level) workflow nets [8]. The translation is automated by the tool BPEL2PNML. In contrast to the concept of a flexible model generation implemented in BPEL2oWFN, BPEL2PNML provides only a single pattern for each BPEL4WS construct and the resulting net is only minimized after the translation by applying structural reduction rules. The semantics enables several analysis methods including the detection of unreachable activities and design flaws such as conflicting receive activities. Further, it is possible to perform a reachability analysis for the garbage collection of inconsumable messages later on. Those methods are implemented in the tool WofBPEL [15]. However, WofBPEL does neither analyze the interactional behavior of processes (due to the restrictions of the semantics) nor does it support static analysis techniques.

Martens et al. [26] translate BPEL4WS processes into an annotated subset of oWFNs, *BPEL annotated Petri nets* (BPNs). The translation is based on a modified version of our old semantics in [12] and does not include most of the fault and compensation handling. For BPNs, a technique for analyzing the controllability has already been introduced in [27] — the *communication graph*. It is similar to our proposed IG. As a main difference, this graph performs communication *steps*, where each step consists of a (possibly empty) sending phase followed by a (possibly empty) receiving phase. Therefore, the communication graph tends to be more complex than our IG (cf. [5]). Finally, Martens' analysis tool Wombat does not support the construction of operating guidelines, but it can visualize the analysis results both in the Petri net and in the BPEL4WS code.

8 Conclusion and Further Work

We presented a framework to formally analyze the interactional behavior of WS-BPEL processes. Both the translation from WS-BPEL into compact Petri net models as well as the further analysis of controllability and the computation of the operating guideline are implemented which allows for a fully-automatic analysis. The results show that we can detect nontrivial model flaws of interacting WS-BPEL processes that would have been hard or impossible to find manually.

In BPEL2oWFN, we have implemented the concept of a flexible model generation (see Sect. 4.2). This approach tries to reduce the Petri net model during and after the translation with respect to the property to be analyzed. For this purpose, for each WS-BPEL activity, several Petri net patterns with different degree of abstraction are available in a pattern repository. Using static analysis on the WS-BPEL code, we select the most abstract pattern applicable in a given context. Static analysis is further used to provide additional information for the analysis. Furthermore, to deal with the complexity of WS-BPEL's data, we unfold finite data domains (e. g., Boolean) and construct the skeleton net to abstract from data. Flexible model generation results in compact models preserving the user-defined analysis goal. That way, it is possible to analyze WS-BPEL processes of realistic size.

In ongoing research we are working on abstraction techniques that allow to map an infinite data domain to a finite domain while preserving more properties than the skeleton net. For that purpose, we adapt state-of-the-art data abstraction techniques such as predicate abstraction [28] and integrate them into our translation approach. Furthermore, we are also planning to enhance our Petri net semantics to cover multiple instances.

To analyze interactions consisting of more than two interacting processes, existing theoretical results have to be integrated into Fiona. We are further working on more efficient reduction techniques, both for IG and OG. We also aim at supporting the redesign of erroneous (e. g., non-controllable) processes. For this purpose, the analysis results (e. g., counter-examples) have to be translated back into WS-BPEL source code. This will be extremely helpful to assist process designers during the modeling.

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. Committee Draft, 25 January, 2007, Organization for the Advancement of Structured Information Standards (OASIS) (2007)
2. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting BPEL Processes. In Dustdar, S., Fiadeiro, J.L., Sheth, A., eds.: Forth International Conference on Business Process Management (BPM 2006), 5–7 September 2006 Vienna, Austria. Volume 4102 of Lecture Notes in Computer Science., Springer-Verlag (2006) 17–32
3. 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. Technical report, BEA Systems, International Business Machines Corporation, Microsoft Corporation (2003)
4. Schmidt, K.: Controllability of Open Workflow Nets. In Desel, J., Frank, U., eds.: Enterprise Modelling and Information Systems Architectures. Number P-75 in Lecture Notes in Informatics (LNI), Entwicklungsmethoden für Informationssysteme und deren Anwendung (EMISA, RWTH Aachen), Bonner Köllen Verlag (2005) 236–249

5. Weinberg, D.: Reduction Rules for Interaction Graphs. Techn. Report 198, Humboldt-Universität zu Berlin (2006)
6. Massuthe, P., Schmidt, K.: Operating Guidelines – An Automata-Theoretic Foundation for the Service-Oriented Architecture. In Cai, K.Y., Ohnishi, A., Lau, M., eds.: Proceedings of the Fifth International Conference on Quality Software (QSIC 2005), Melbourne, Australia, IEEE Computer Society (2005) 452–457
7. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
8. 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
9. Ramadge, P., Wonham, W.: Supervisory control of a class of discrete event processes. SIAM J. Control and Optimization **25**(1) (1987) 206–230
10. Badouel, E., Darondeau, P.: Theory of Regions. In: Lectures on Petri Nets I: Basic Models. Volume 1491 of Lecture Notes in Computer Science., Springer-Verlag (1998) 529–586
11. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. Techn. Report 210, Humboldt-Universität zu Berlin (2006)
12. Stahl, C.: A Petri Net Semantics for BPEL. Techn. Report 188, Humboldt-Universität zu Berlin (2005)
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 (2007) to appear.
14. 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.: Proceedings of the Third International Conference on Business Process Management (BPM 2005). Volume 3649 of Lecture Notes in Computer Science., Nancy, France, Springer-Verlag (2005) 220–235
15. Ouyang, C., Verbeek, E., Aalst, W.M.P.v.d., Breutel, S., Dumas, M., Hofstede, A.H.M.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., Amsterdam, The Netherlands, Springer-Verlag (2005) 484–489
16. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. 2nd edn. Springer-Verlag (2005)
17. Vautherin, J.: Parallel systems specitications with coloured Petri nets and algebraic specifications. In Rozenberg, G., ed.: Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets, Oxford, UK, June 1986. Volume 266 of Lecture Notes in Computer Science., Springer-Verlag (1987) 293–308
18. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE **77**(4) (1989) 541–580
19. Schmidt, K.: LoLA: A Low Level Analyser. In Nielsen, M., Simpson, D., eds.: Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000). Number 1825 in Lecture Notes in Computer Science, Springer-Verlag (2000) 465–474
20. Arias-Fisteus, J., Fernández, L.S., Kloos, C.D.: Formal Verification of BPEL4WS Business Collaborations. In Bauknecht, K., Bichler, M., Pröll, B., eds.: E-Commerce and Web Technologies, 5th International Conference, EC-Web 2004, Zaragoza, Spain, August 31-September 3, 2004, Proceedings. Volume 3182 of Lecture Notes in Computer Science., Springer-Verlag (2004) 76–85

21. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004, ACM Press (2004) 621–630
22. Ferrara, A.: Web services: a process algebra approach. In: Service-Oriented Computing - ICSOC 2004, Second International Conference, New York, NY, USA, November 15-19, 2004, Proceedings, New York, NY, USA, ACM Press (2004) 242–251
23. Fahland, D., Reisig, W.: ASM-based semantics for BPEL: The negative Control Flow. In Beauquier, D., Börger, E., Slissenko, A., eds.: Proceedings of the 12th International Workshop on Abstract State Machines (ASM'05), Paris XII (2005) 131–151
24. Farahbod, R., Glässer, U., Vajihollahi, M.: Specification and Validation of the Business Process Execution Language for Web Services. In: Abstract State Machines 2004. Advances in Theory and Practice, 11th International Workshop, ASM 2004, Lutherstadt Wittenberg, Germany, May 24-28, 2004. Proceedings. Volume 3052 of Lecture Notes in Computer Science., Springer-Verlag (2004) 78–94
25. Ouyang, C., Verbeek, E., Aalst, W.M.P.v.d., Breutel, S., Dumas, M., Hofstede, A.H.M.t.: Formal Semantics and Analysis of Control Flow in WS-BPEL. Technical report (revised version), Queensland University of Technology (2005)
26. Martens, A., Moser, S., Gerhardt, A., Funk, K.: Analyzing Compatibility of BPEL Processes – Towards a Business Process Analysis Framework in IBM's Business Integration Tools. In: International Conference on Internet and Web Applications and Services (ICIW'06), February 23–25, 2006, Guadeloupe, French Caribbean, IEEE Computer Society Press (2006)
27. Martens, A.: Analyzing Web Service based Business Processes. In Cerioli, M., ed.: Proceedings of Intl. Conference on Fundamental Approaches to Software Engineering (FASE'05), Part of the 2005 European Joint Conferences on Theory and Practice of Software (ETAPS'05). Volume 3442 of Lecture Notes in Computer Science., Edinburgh, Scotland, Springer-Verlag (2005) 19–33
28. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In Grumberg, O., ed.: Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22–25, 1997, Proceedings. Volume 1254 of Lecture Notes in Computer Science., Springer-Verlag (1997) 72–83