Transforming BPEL to Petri Nets

Sebastian Hinz, Karsten Schmidt, and Christian Stahl

Humboldt–Universität zu Berlin,
Institut für Informatik, D–10099 Berlin
{hinz, kschmidt, stahl}@informatik.hu-berlin.de

Abstract. We present a Petri net semantics for the Business Process Execution Language for Web Services (BPEL). Our semantics covers the standard behaviour of BPEL as well as the exceptional behaviour (e.g. faults, events, compensation). The semantics is implemented as a parser that translates BPEL specifications into the input language of the Petri net model checking tool LoLA. We demonstrate that the semantics is well suited for computer aided verification purposes.

keywords: Business process modeling and analysis, Formal models in business process management, Process verification and validation, BPEL, Petri nets.

1   Introduction

The *Business Process Execution Language for Web Services* (BPEL) is part of ongoing activities to standardize a family of technologies for web services. A textual specification [1] appeared in 2003 and is subject to further revisions. The language contains features from previous languages, for instance IBM's WSFL [2] and Microsoft's XLANG [3]. The textual specification is, of course, not suitable for formal methods such as computer aided verification. With computer aided verification, in particular model checking, it would be possible to decide crucial properties such as composability of processes, soundness, and controllability (the possibility to communicate with the process such that the process terminates in a desired end state). For a formal treatment, it is necessary to resolve the ambiguities and inconsistencies of the language which occurred particularly due to the unification of rather different concepts in WSFL and XLANG.

Several groups have proposed formal semantics for BPEL. Among the existing attempts, there are some based on finite state machines [4,5], process algebras [6], and abstract state machines [7,8]. Though all of them are successful in unravelling weaknesses in the informal specification, they are of different significance for formal verification. The semantics based on abstract state machines are feature-complete. However, Petri nets provide a much broader basis for computer aided verification than abstract state machines. Most of the other approaches typically do not support some of BPEL's most interesting features such as fault, compensation, and event handling.

In this paper, we consider a *Petri net semantics* for BPEL. The semantics is *complete* (i.e., covers all the standard and exceptional behaviour of BPEL),

and *formal* (i.e., feasible for model checking). With Petri nets, several elegant technologies such as the theory of workflow nets [9], a theory of controllability [10,11], a long list of verification techniques [12] and tools [13,14,12] become directly applicable. The Petri net semantics provides patterns for each BPEL activity. Compound activities contain slots for the patterns of their subactivities. This way, it is possible to translate BPEL processes automatically into Petri nets. Using high-level Petri nets, data aspects can be fully incorporated while these aspects can as well be ignored by switching to low-level Petri nets.

We first explain the general concepts of BPEL. Afterwards we introduce the principles of our Petri net semantics and explain the Petri net patterns for a few typical BPEL activities. Then we report first experiences with an automated translation of BPEL into Petri nets, and subsequent model checking. Finally, we discuss some ideas for an extension of our technology that aims at models which are better suitable for model checking.

2   Introduction to BPEL

BPEL is a language for describing the behaviour of business processes based on web services. Such a business process can be described in two different ways: either as *executable business process* or as *business protocol*. An executable business process which is the focus of this paper models the behaviour and the interface of a *partner* (a participant), in a business interaction. A business protocol, in contrast, only models the interface and the message exchange of a partner. The rest of its internal behaviour is hidden. Throughout this paper, we will use the term *BPEL process* instead of "executable business process specified in BPEL". Executing a BPEL process means to create an *instance* of this process which is executed.

For the specification of the internal behaviour of a business process, BPEL provides two kinds of *activities*. An activity is either an *elementary activity* or a *structured activity*. The set of elementary activities includes: `empty` [1] (do nothing), `wait` (wait for some time), `assign` (copy a value from one place to another), `receive` (wait for a message from a partner), `invoke` (invoke a partner), `reply` (reply a message to a partner), `throw` (signal a fault) and `terminate` (terminate the entire process instance).

A structured activity defines a causal order on the elementary activities. It can be nested with other structured activities. The set of structured activities includes: `sequence` (nested activities are ordered sequentially), `flow` (nested activities occur concurrently to each other), `while` (while loop), `switch` (selects one control path depending on data) and `pick` (selects one control path depending either on timeouts or external messages). The most important structured activity is a `scope`. It links an activity to a transaction management. It provides a `fault handler`, a `compensation handler`, an `event handler`, `correlation sets` and data `variables`. A `process` is a special `scope`. More precisely, it is the outmost `scope` of the business process.

---

[1] We use this type-writer font for BPEL constructs.

A `fault handler` is a component that provides methods to handle faults which may occur during the execution of its enclosing `scope`. In contrast, a `compensation handler` is used to reverse some effects which happened during the execution of activities. With the help of an `event handler`, external message events and specified timeouts can be handled. A `correlation set` is used for identifying the instance of a BPEL `process` only by the content of a message. Thus, a `correlation set` is an identifier – more precisely, it is a collection of properties – and all messages of an instance must contain it. It is either initialized by the first incoming or outgoing message.

Another important concept in BPEL are `links`. A `link` can be used to define an order between two concurrent activities in a `flow`. It has a *source* activity and 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 it evaluates its join condition to true. BPEL provides *dead-path-elimination* [15], i.e. the status of all outgoing `links` of a source activity that is not executed anymore is set to negative. Consider, for instance, an activity within a branch that is not taken in a `switch` activity.

3   Petri Net Semantics for BPEL

Our goal is to translate every BPEL process into a Petri net. The translation is guided by the syntax of BPEL. In BPEL, a process is built by plugging instances of language constructs together. Accordingly, we translate each construct of the language separately into a Petri net. Such a net forms a *pattern* of the respective BPEL construct. Each pattern has an *interface* for joining it with other patterns as is done with BPEL constructs. Some of the patterns are used with a parameter, e.g. there are some constructs that have inner constructs. The respective pattern must be able to carry any number of inner constructs as its equivalent in BPEL can do. We aim at keeping all properties of the constructs in the patterns. The collection of patterns forms our *Petri net semantics* for BPEL.

In the following subsections, we give a glimpse on our semantics, using a basic activity (receive), a structured activity (flow) and the stop pattern as examples. The complete version of the Petri net semantics is reported in [16,17].

3.1   Example of a Basic Activity

Let us have a more detailed look at the general design of a pattern. Figure 1 depicts the pattern for the BPEL's `receive` activity. `receive` is responsible for receiving a partner's request. To identify whether the request is sent to this receive pattern and not to another instance of the process, BPEL's `receive` specifies at least one `correlation set`. The pattern in Fig. 1 presents a `receive` with one `correlation set` which is already initialized[2].

---

[2] The pattern of BPEL's `receive` where a `correlation set` is initialized by the incoming message is very similar to Fig. 1 and can be found in [17].
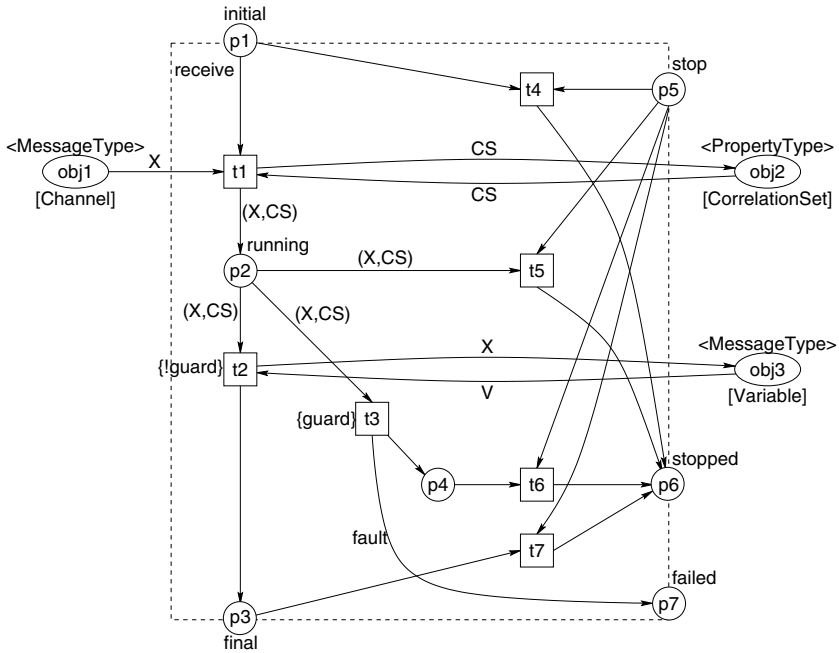
Fig. 1. Pattern for BPEL's `receive`. When the pattern is activated, it is executed in two steps. First, the message is taken from the channel (`obj1`) and the `correlation set` (`obj2`) is read (`t1`). Both values are saved in variables X and CS, respectively. In the second step, this information is analyzed. Either the message is saved in the `variable` (`t2`) or a fault occurs (`t3`). With it variable V holds the old value of `obj3` and `fault` holds the fault information. In both cases, the pattern is finished.

Before we discuss details of the receive pattern, we give some general comments on the notion of patterns. Firstly, we use the common graphical notations for Petri nets. Places and transitions are labelled with an identifier, e.g. `p1` [3] or `t1` which are depicted (contrary to common notation) inside the respective Petri net node. In addition, some nodes have a second label depicted outside the node, e.g. initial. This label is used to show the purpose of the node in the net. Secondly, a variable with small letter in arc inscriptions, e.g. fault, symbolizes a single variable and a variable with a capital letter, e.g. X, symbolizes a tupel of variables. Thirdly, there are transitions, e.g. `t2` which have a transition guard. Such a transition can only fire when its guard, a boolean expression, is evaluated to true. A guard is depicted (in braces) next to the transition it belongs to, e.g. {!guard}.

In general, a pattern is framed by a dashed box. Inside the frame, the structure of the corresponding BPEL construct is modelled. The interface is established by the nodes depicted directly on the frame. Positive control flows from top

---

[3] We use this serif-free font for labels in a Figure.

to bottom while communication between processes flows horizontally. In Fig. 1 the positive control flow starts with a token on initial and it ends either with a token on finish or failed. Outside the frame, there are external objects, e.g. obj1. An object is either a place of a scope pattern (`variable`, `correlation set`) or of the process pattern (channel). An activity's pattern as the receive pattern in Fig. 1 relates to those places. The label on the top of an object defines its sort whereas the role is defined at the bottom of the object. A sort is the domain of the tokens lying on and arriving at this place. The object's role is independent of its sort.

The pattern shown in Fig. 1 takes a message from the channel, reads the `correlation set` and either updates its `variable` by saving this message or a fault is thrown because of a mismatch between the values of the `receive`'s `correlation set` and the `correlation set` in the message or some other error.

The meaning of place stop, stopped and failed in Fig. 1 needs to be explained. In BPEL, a process is forced to stop its positive control flow, e.g., when a fault occurs or activity `terminate` is activated. However, the BPEL specification [1] tells only informally the requirements how to stop a scope. For instance, activity `receive` "is interrupted and terminated prematurely" [1, p. 79]. The specification does not describe how to realize those requirements. Thus, we had to make some modelling decisions in our model: The pattern of BPEL's `scope` is extended by a stop pattern (see Sect. 3.3 for more details), which has no equivalent construct in BPEL. If a `scope` needs to be stopped, the stop pattern controls this procedure. Our idea is to remove all tokens from the patterns, embedded in the scope pattern; thus the patterns of BPEL's activities and `event handler` contain a subnet – a so called *stop component*. In contrast, the patterns of BPEL's `compensation handler` and `fault handler` do not contain a stop component, because they both need not to be stopped. In [16] we proved that every process can be stopped using stop components. In the case of Fig. 1, the stop component is established by transitions t4 – t7 using the interface stop and stopped. Throughout this paper, we will call this the *negative control flow* of an activity.

In order to explain how a stop component works, consider a `scope` that contains just a `receive` and the latter throws a fault. This leads to place failed being marked – the token is an object that consists of the fault's name. This place is joined with a place in the stop pattern; thus this pattern gets the control of the `scope`. First of all it stops the inner activity of the `scope` and consequently a token is produced on the `receive`'s stop place. Transition t6 fires and stopped is marked. This place is also joined with a place in the stop pattern. In contrast, transitions t4, t5, t7 consume the token on stop by stopping the receive pattern wherever the control flow is in this pattern. As a result, a token is produced on stopped, too. One might assume that t4 obtains priority before t1 and t5 before t2. Indeed, this would destroy the model's asynchronous behaviour without changing the possible set of runs. We use this asynchronous behaviour in our patterns to model the aspect that sending the stop signal needs time, too. Consider, for instance, two receive patterns executed sequentially. It is possible that the first `receive` is finished (and so the second `receive` is activated) exactly in the moment signal stop is sent. In our patterns, however, this possibility is taken into

account. Alternatively, a different modelling approach is possible: A transition of the receive pattern's positive control flow is only enabled when no fault has been occurred in the surrounding scope pattern. This fact could be modelled by a place marked when no fault has been occurred. But this, of course, would destroy the asynchronous character of any BPEL process.

3.2 Example of a Structured Activity

Next we show the general pattern of BPEL's `flow`. `flow` is used to execute subtasks concurrently. The subtasks can be further synchronized by so-called `links`.

The pattern in Fig. 2 can carry n inner activities which are executed concurrently. An embedded activity can be any BPEL construct; thus only the interface is visualized and all other information of the pattern is hidden. Therefore only the frame and places initial, final, stop, stopped and if needed negLink are visible (see, for instance innerActivity1 in Fig. 2). The interface of each embedded pattern is joined with the surrounding flow pattern.

negLink is an abbreviation of negative link. It is an optional place that is only part of a pattern's interface when it embeds at least one activity that is source of a `link`. With the help of negLink the status of all outgoing `links` of
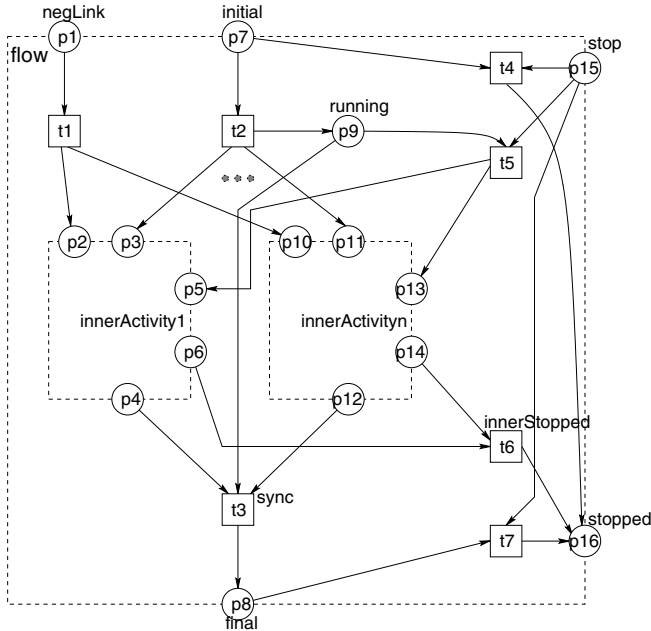

Fig. 2. Pattern for BPEL's `flow` embeds n inner activities. There are two possible scenarios: Either all inner activities are executed concurrently (t2) and afterwards they are synchronized (t3) or the status of all source `links` embedded in the `flow` is set to negative (t1)

an inner activity (i.e. all links for that the inner activity is source) that is not executed anymore are set to negative. Consider an activity within a branch that is not taken in a `switch` activity. In other words, `negLink` is a place for modelling dead-path-elimination. In Fig. 2 we assume that `innerActivity1` and `innerActivityn` contain at least one activity that is source of a `link`.

In our semantics, we model a `link` by a place of sort Boolean. If the link is set, the place is marked. The value of the token is the status of the `link` that depends on how the `transition condition` is evaluated. The `join condition` determines whether a target activity is executed or not. It is modelled by a transition guard. For modelling dead-path-elimination, we build a link pattern that embeds an activity.

If there is a token on `stop`, the `flow` and its embedded activities are stopped. After `t5` has fired, the token on `running` is consumed; thus `t3` cannot be activated. Furthermore the stop place of each inner activity is marked. So `innerActivity1`, . . . , `innerActivityn` can be stopped concurrently. Firing `t6` synchronizes them.

3.3   The Stop Pattern

After an activity has thrown a fault, the `fault handler` of the enclosing `scope` has firstly to finish the positive control flow inside the `scope` and secondly it has to handle the fault. We preserve this division and extend every `scope` by a so-called stop pattern which has no equivalent construct in BPEL. When the stop pattern receives the fault, it finishes its enclosing `scope` and afterwards it signals the fault to the `scope`'s `fault handler`. Furthermore the stop pattern is used to realize BPEL's `terminate` activity, i.e. to stop the entire process.

Figure 3 depicts the pattern of the stop pattern. It is quite complex, because the scope can be in different states when the fault signal occurs. For example, a fault can occur in the positive control flow or in a `fault handler`. For each scenario the stop pattern behaves differently. In order to explain how this pattern works it is useful to make the following commitment: The pattern we have a look at is embedded in a `scope` B. B itself embeds a `scope` C called the *child scope* of B. Furthermore B is child scope of A or in other words: A is the *parent scope* of B.

First of all we have a look at the interface of Fig. 3 which differs from the former patterns. On top there are four important places: `ft_in` (marked if A wants B to be stopped), `fault_in` (a fault is occurred in an enclosing activity of B, i.e. either a token on a failed place or C's `fault handler` rethrows a fault it cannot handle), `terminate_up` (a `terminate` activity embedded in A is activated) and `terminate` (a `terminate` activity either embedded in C or in B is activated). The place `fault_in` results from joining the failed places of all activities enclosed by B. All other interface places on top are state places of B. For the most part the state places take inspiration from the *business agreement protocol* (BAP) [18]. The BAP specifies a set of signals serving for communication between a scope and its parent scope. The places on the right are used to remove all tokens in B's `compensation handler` (`cleanCH`, `ch_cleaned`) and to stop the positive control flow of B (`stop`, `stopped`). On the bottom there are places to activate other
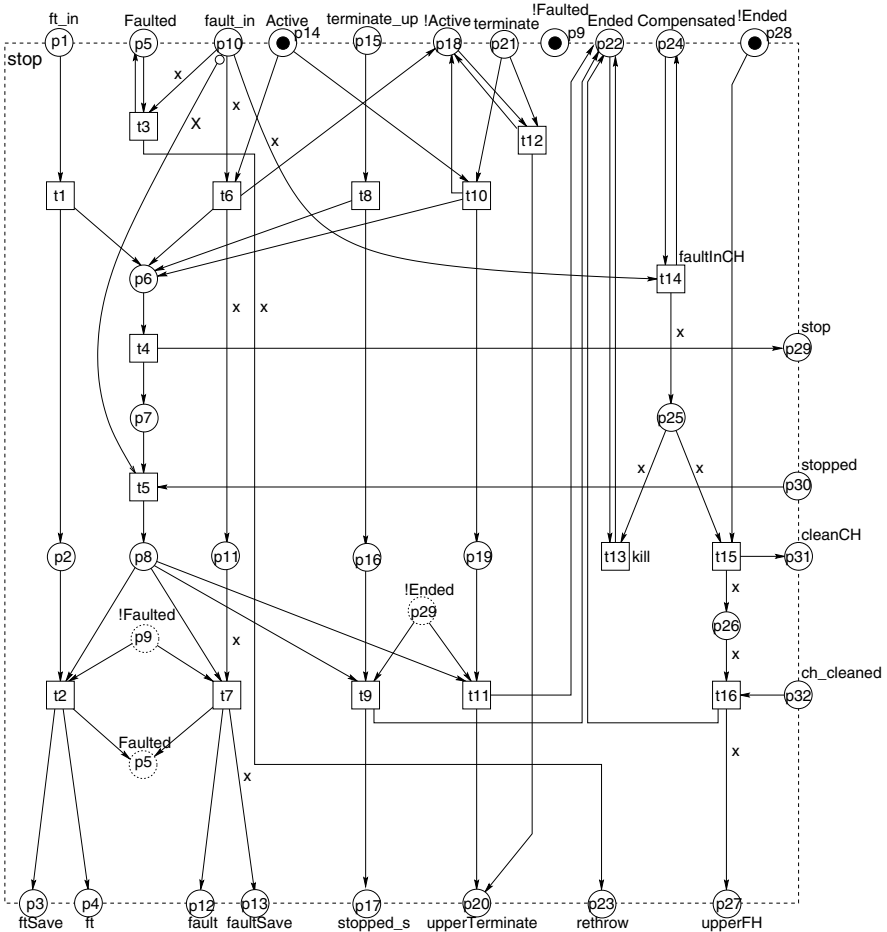
Fig. 3. Stop pattern embedded in a scope

patterns. ft and ft_fault (signalling that A wants to stop B), fault and faultSave (signalling the occurrence of a fault) and rethrow (signalling the occurrence of a fault during the execution of B's `fault handler`) activate the `fault handler` of B. In contrast, upperTerminate (signals `scope` A that it has to be terminated) and upperFH (rethrows a fault to A's `fault handler` that could not be handled by B's `fault handler`) activate the parent scope and the parent scope's `fault handler`, respectively. stopped_s is the stopped place of B.

The arc connecting p10 and t5 differs from the other arcs in its notation (a little circle at its source) and also in its semantics. It consumes all tokens of p10 making no difference if there are 0, 1 or more tokens on this place. In other words, p10 is emptied. This arc is a so-called *reset arc* [19].

Altogether 8 possible scenarios are modelled in this pattern: Either a fault is thrown, an activity `terminate` is activated or A wants to stop B. In the case of

an activated `terminate` activity we distinguish if this activity is embedded in an enclosing `scope` (here A) or not and if B's `fault handler` is activated or not. In the case of a thrown fault we distinguish a fault in the positive control flow, in the `compensation handler`, and in the `fault handler`. In this paper, we restrict ourselves to explain how a `scope` can be stopped if a fault in the positive control flow occurs. For details of the remaining scenarios, the interested reader is referred to [17].

Let us continue the scenario described in Sect. 3.1: Let B be the `scope` that encloses the `receive`. If the `receive` throws a fault, its failed place is marked – the token is an object that consists of the fault's name. As already mentioned, the failed place and the place fault_in in Fig. 3 are identical. It is the first fault occurred; thus B is in state Active, i.e. Active is marked. t6 can fire and variable x holds the fault information. Firing t4 produces a token on stop which leads to removing all tokens inside the receive pattern and to produce a token on place stopped. Place stopped in the receive pattern and stopped in Fig. 3 are identical, too. So t5 can fire and the positive control flow of B is finished. By firing t7 the stop pattern invokes the `fault handler` by signalling the fault information.

4   BPEL2PN

In [20], we translated a small BPEL process – it was a modification of the Purchase Order Process presented in the BPEL specification [1, pp. 14] – into a Petri net. This BPEL process consists of 17 activities. The resulting Petri net consists of 158 places and 249 transitions and it was generated manually. In fact this transformation was very laborious and took hours. Therefore tool support was necessary to transform a BPEL process automatically into a Petri net.

We built a parser, *BPEL2PN* [21], that can automatically transform a given BPEL process into a Petri net. The way BPEL2PN works is shown in Fig. 4: It takes a BPEL process process.bpel as an input. Then this process is transformed into a Petri net according to the Petri net semantics. In more detail, for each activity of process.bpel an instance of the corresponding pattern is generated and all these patterns are stuck together as done in the BPEL process. The resulting Petri net, process.lola, is the output of BPEL2PN where .lola is the data format of our model checker LoLA [12]. LoLA offers the user the opportunity to write out the net into the standard interchange format for Petri nets, the *Petri Net Markup Language* (PNML) [22].

As explained in Sect. 3.3, in the stop pattern a reset arc is used to remove all tokens from place fault_in. In the following we draft the idea how such an arc can be modelled as a high-level construct which can be, in turn, unfolded into a low-level construct: It is possible to safely over-approximate the maximal number k of tokens, i.e. the number of faults that can be produced on place fault_in. This is the number of activities of the enclosing scope that can throw a fault. Every scope encloses only a finite number of activities. Consequently k is bounded. So place fault_in is a high-level place that is *k-bounded*, i.e. the number of tokens on fault_in is never greater than k. Then, unfolding the reset arc means to replace
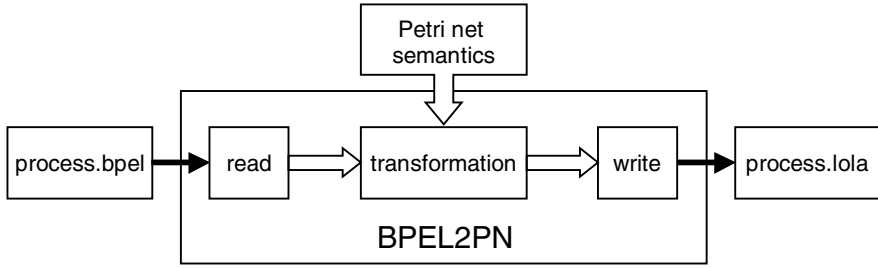
Fig. 4. Mode of operation of BPEL2PN

fault_in by $k+1$ places (0 tokens are possible, too). Furthermore every transition of the pre-set or post-set of fault_in has to be replaced by $k + 1$ transitions. It can be easily seen that a reset arc causes an increasing of the net size. The value of k can be narrowed, for instance, in the case of a sequence. Unaffected by the number of its inner activities only one fault can be thrown, because after this fault is thrown the control flow within the sequence is blocked. Calculating the best possible k of place fault_in is ongoing research. In order to avoid an increasing net size due to unfolding we could build an abstract stop pattern. In this pattern we could restrict the number of faults (and therefore k) to 1. Those ideas are explained in more detail in Sect. 6.

The current version of BPEL2PN has the following limitations: Firstly, as already mentioned in [20] we decided to *abstract from data*, i.e. messages and data are modelled as black tokens, because we directed our attention to the control flow. Consequently, all other high-level constructs like transition guards and variables were left out, too. So selecting one of two control pathes in the Petri net semantics, solved by the evaluation of data, is modelled by a nondeterministic choice, e.g. t2 or t3 in Fig. 1. Therefore the resulting Petri net is low-level[4]. Data aspects can be integrated later in our tool or analyzed by methods of static analysis. Secondly, every activity is limited to one `correlation set` (except the synchronous `invoke` that is limited to two `correlation sets`). And last, attribute *enableInstanceCompensation* is ignored. Therefore it is not possible to compensate a process instance, i.e. the entire BPEL process. This is, however, no real limitation: You only need to redefine the process as a `scope` and embed this `scope` in a `process`. Then, the old process can be compensated.

In fact, these are no serious limitations, because the control flow of the BPEL process is preserved. In the next section, we want to give the reader an impression what complex processes can be translated by BPEL2PN and analyzed by our model checker LoLA.

---

[4] Due to the high-level construct of the reset arc the net generated by BPEL2PN is high-level, but it is unfolded to a low-level Petri net by LoLA. Generating a low-level net by BPEL2PN would be possible, too. As a consequence, the complexity of the parser would be increased.

5 Case Study: Online Shop

In this section we present a case study. It shows how a given, realistic BPEL process can be analyzed by the use of our semantics. We generated a business process and verified several relevant properties of this process. We use the Petri net based model checker LoLA that features powerful state space reduction techniques like symmetries [23] partial order reduction using stubborn sets [24] and the sweep-line method [25].

In Fig. 5 our example process is depicted – a modification of the Online Shop Process presented in [10]. A box frames an activity. In the case of a `scope` or the `process` itself we use a bold frame. Sequential flow is depicted by dashed arcs, whereas concurrent activities are grouped in parallel. Arcs with solid lines symbolize `links`. The two nested `scopes` of the Online Shop Process are depicted in Figures 6(a) and 6(b).

This is a medium-sized example. It consists of 53 activities, yet most of BPEL's activities including `fault handler`, `event handler`, nested `scopes`, and `links` occur.

The Petri net of the example process consists of 410 places and 1069 transitions. It was generated by our tool BPEL2PN. LoLA takes this Petri net as an input and generates the state space, i.e. it calculates the *reachability graph* of the Petri net. The whole state space consists of 6,261,684 states and is calculated in ca. 96 minutes. By using LoLA's state space reduction techniques (partial order reduction and sweep line method in combination) a reduced state space consisting of 443,218 states could be generated in 50 minutes. More detailed, these reduction techniques do not work on the Petri net patterns, but on the reachability graph of the Petri net. We also generated a variant of the Online Shop Process where every place fault_in was 1-bounded, i.e. safe. That means, in every `scope` only one fault can occur. As a consequence, the net consists of only 382 places and 495 transitions. The state space reduced to 6,246,601 states (full state space) and 412,731 states (reduced state space), respectively.

If the state space can be fully explored by our tool, it is possible to analyze Petri net specific properties like dead places and dead transitions as well as any temporal property of the underlying process that can be expressed by a formula of the temporal logic CTL.

LoLA calculated dead places and dead transitions. These resulting places and transitions show which aspects of the patterns have been unused. Furthermore this result was used to prove whether there are activities inside the process that can never be activated. In fact, this is possible due to incorrect use of links. As an example consider the switch in Fig. 6(b). If the two assigns were ordered by a link, the target activity would never be activated: On the one hand the branch of the source activity is chosen and so the target activity is not executed. On the other hand the branch of the target activity is chosen, but due to dead-path-elimination the link is set to false. Thus, this activity is never activated, but the process will deadlock neither. In our example all activities can be activated.

We further verified relevant properties of the Online Shop Process like termination and "the customer will always get an answer". Of course, the formula
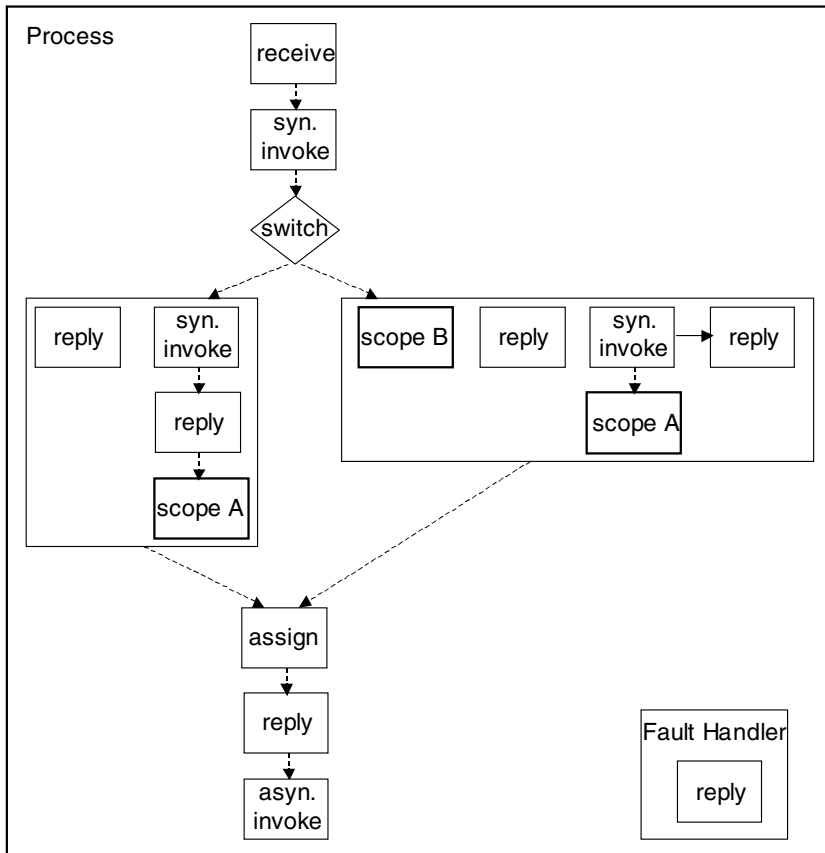
Fig. 5. When the Online Shop Process receives an order from a customer, it retrieves the customer's data. These data are analyzed, because the business strategy of the shop distinguishes new and already known customers. If it is a known customer (left switch branch) the shop initiates two tasks concurrently: The marketing department sends a special offer (on the left) and the customer department (right sequence) firstly takes the order and secondly send its discount level. Afterwards the shop invites offers from the suppliers (scope A). In the case of a new customer (right switch branch), the shop initiates four tasks concurrently: It collects the customer's bank data (scope B), the marketing department sends the customer a special offer (second task on the left). Furthermore the shop takes the order and then it invites offers from the suppliers (scope A). In addition, the terms of trade are sent to the customer (right task). After the completion of the flow the tasks of both, new and known customer are joined. The price information are saved and then the shop sends the supply information to the customer. The process finishes after the shop has invoked the shipper. There is a dependency between two tasks in the case of a new customer, realized by a link: The terms of trade are only sent after the shop has received the customer's order.
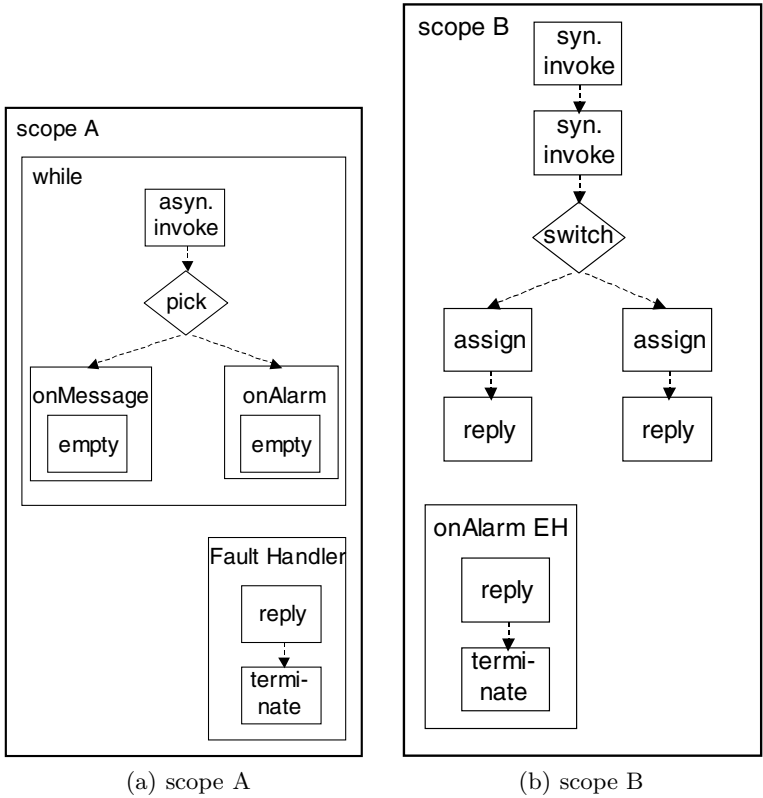
(a) scope A   (b) scope B

Fig. 6. (a) The shop invokes one supplier after the other to invite offers for the product. If the supplier does not answer in time, the next supplier is invoked. Additionally, if a fault occurs during the execution of the process, the customer is informed and the process instance is terminated. (b) The customer's bank data are analyzed whether he is credit worthy. The result is sent to the customer. If he is credit worthy, the process goes on. Otherwise, the process stops. If the bank does not reply in time, the process is terminated after the customer gets a message. One further dependency is modelled by a link (not depicted in the figure): The shop starts invoking the suppliers (scope A) only when the customer is credit worthy.

of the respective temporal properties were generated manually by ourself. The Online Shop behaves as expected: it always comes to an end and the costumer will always get an answer. By abstracting from data aspects as we did, a single process must always terminate, because a deadlock is not possible. Termination plays a more important role if we compose several BPEL processes. Then, it is possible that the composed processes run into a deadlock.

We also tried to analyze a business process that consists of 132 activities. Due to the huge net size we were not able to calculate the full state space of this process. In order to check such an extremely huge process it is necessary to get a smaller model. The next section presents some ideas how to do so.

6   Advanced Translation

The models generated by the present version of our parser can be seen as brute force models. The generated models are significantly larger than typical manually generated models. This is due to the fact that the Petri net patterns are complete, i.e. applicable in every context. For a particular process, many of the modelled features are unused. For instance, if a basic activity cannot throw any error, many of the error handling mechanisms in the surrounding compound activity can be spared. Furthermore, to decide if a specific property holds, it is often sufficient to restrict the patterns to specific aspects. To prove the correct inter-operation of two BPEL processes, for instance, it is sufficient to restrict the attention to communication aspects of the patterns, while internal actions can be abstracted away.

In ongoing projects, we aim at an improved translation where several Petri net patterns with different degree of abstraction are available for each BPEL activity. Using static analysis on the BPEL code, we want to select the most abstract pattern applicable in a given context. We believe that model sizes can be drastically reduced this way thus alleviating the state explosion problem inherent to model checking.

Data flow equations, the basis of static analysis, are already available for many features of BPEL [26]. It is, however, still necessary to select suitable abstraction techniques in order to make static analysis run.

7   Conclusion

We presented first experimental results for generating Petri net models of BPEL processes. The translation of BPEL to Petri nets follows a feature-complete Petri net semantics of BPEL. The translation is implemented, and we were able to present first results. The results show that it is necessary to complement the technology with an improved model generation. We have proposed the use of static analysis as a tool for providing process-specific information that can be exploited in a flexible model generator.

Our goal is a technology chain that, starting at a BPEL process, performs static analysis. Based on the analyzed information, the translator selects the most abstract pattern for each activity that is feasible in the analyzed context and synthesizes a Petri net model. On the Petri net model, a model checker evaluates relevant properties. The analysis results (e.g., counter example paths) are translated back to the BPEL source code.

References

1. Curbera, Goland, Klein, Leymann, Roller, Thatte, Weerawarana: Business Process Execution Language for Web Services, Version 1.1. Technical report, BEA Systems, International Business Machines Corporation, Microsoft Corporation (2003)

2. Leymann, F.: WSFL – Web Services Flow Language. IBM Software Group, Whitepaper. (2001) `http://ibm.com/webservices/pdf/WSFL.pdf`.
3. Thatte, S.: XLANG – Web Services for Business Process Design. Microsoft Corporation, Initial Public Draft. (2001) `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c`.
4. Fisteus, J.A., Fernández, L.S., Kloos, C.D.: Formal Verification of BPEL4WS Business Collaborations. In: Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web '04). LNCS, Springer (2004)
5. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, ACM Press (2004) 621–630
6. Ferrara, A.: Web services: a process algebra approach. In: ICSOC, ACM (2004) 242–251
7. Fahland, D., Reisig, W.: ASM-based semantics for BPEL: The negative Control Flow. In D. Beauquier, E.B., Slissenko, A., eds.: Proc. 12th International Workshop on Abstract State Machines, Paris, March 2005. Lecture Notes in Computer Science, Springer-Verlag (to appear, 2005)
8. Farahbod, R., Glässer, U., Vajihollahi, M.: Specification and Validation of the Business Process Execution Language for Web Services. In: Abstract State Machines. Volume 3052 of Lecture Notes in Computer Science., Springer (2004) 78–94
9. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems and Computers **8** (1998) 21–66
10. Martens, A.: Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services. Dissertation, WiKu-Verlag Stuttgart (2004)
11. Schmidt, K.: Controlability of Business Processes. Technical Report 180, Humboldt-Universität zu Berlin (2004)
12. Schmidt, K.: LoLA – A Low Level Analyser. In Nielsen, M., Simpson, D., eds.: International Conference on Application and Theory of Petri Nets. LNCS 1825, Springer-Verlag (2000) 465 ff.
13. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In: Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003), Eindhoven, The Netherlands, June 23-27, 2003 — Volume 2679 of Lecture Notes in Computer Science / Wil M. P. van der Aalst and Eike Best (Eds.), Springer-Verlag (2003) 450–462
14. Starke, P.H., Roch, S.: Ina et al. In Mortensen, K.H., ed.: Tool Demonstrations 21st International Conference on Application and Theory of Petri Nets, Department of Computer Science, University of Aarhus (2000) 51–56
15. Leymann, F., Roller, D.: Production Workflow – Concepts and Techniques. Prentice Hall (1999)
16. Stahl, C.: Transformation von BPEL4WS in Petrinetze. Diplomarbeit, Humboldt-Universität zu Berlin (2004)
17. Stahl, C.: A Petri Net Semantics for BPEL. Technical report, Humboldt-Universität zu Berlin (to appear June, 2005)
18. Cabrera, Copeland, Cox, Freund, Klein, Storey, Thatte: Web Services Transaction. Vorschlag zur Standardisierung, Version 1.0. (2002) `http://ibm.com/developerworks/webservices/library/ws-transpec/`.

19. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset nets between decidability and undecidability. In Spies, K., Schätz, B., eds.: Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98), Aalborg, Denmark, July 1998. Lecture Notes in Computer Science 1443, Springer (1998) 103–115

20. Schmidt, K., Stahl, C.: A Petri net semantic for BPEL4WS - validation and application. In Kindler, E., ed.: Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04), Universität Paderborn (2004) 1–6

21. Hinz, S.: Implementation einer Petrinetz-Semantik für BPEL4WS. Diplomarbeit, Humboldt-Universität zu Berlin (2005)

22. Billington et al., J.: The Petri Net Markup Language: Concepts, Technology, and Tools (2003)

23. Schmidt, K.: How to calculate symmetries of petri nets. Acta Informatica (2000) 545–590

24. Schmidt, K.: Stubborn set for standard properties. In: Proc. 20th Int. Conf. Application and Theory of Petri nets. Volume 1639 of LNCS., Springer-Verlag (1999) 46–65

25. Schmidt, K.: Automated Generation of a Progress Measure for the Sweep-Line Method. In: Proc. 10th Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 2988 of LNCS., Springer-Verlag (2004) 192–204

26. Heidinger, T.: Statische Analyse von BPEL4WS-Prozessmodellen. Studienarbeit, Humboldt-Universität zu Berlin (2003)