A Feature-Complete Petri Net Semantics for WS-BPEL 2.0 and its Compiler BPEL2oWFN

★ Revision of of August 30, 2007 ★

Niels Lohmann

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

Abstract. We present an extension of a Petri net semantics for the Web Service Business Execution Language (WS-BPEL). This extension covers the novel activities and constructs introduced by the recent WS-BPEL 2.0 specification. Furthermore, we simplify several aspects of the Petri net semantics to allow for more compact models suited for computer-aided verification.
This technical report is the extended version of the papers [1, 2] and can be seen as the sequel of [3].

1 Introduction

Recently, the emerging standard to describe business processes on top of Web service technology, the Web Service Business Execution Language (WS-BPEL), has been officially specified [4]. This specification is much more detailed and more precise compared to the predecessor specification [5]. Still, WS-BPEL is specified informally using plain English. To formally analyze properties of WS-BPEL processes, however, a *formal* semantics is needed. Therefore, many work has been conducted to give a formal semantics for the behavior of WS-BPEL processes. The approaches cover many formalisms such as Petri nets, abstract state machines (ASMs), finite state machines, process algebras, etc. (see [6] for an overview). In addition to the possibility to analyze WS-BPEL processes, a formal semantics may also help to understand the original specification and to allow to find ambiguities.

The language constructs found in WS-BPEL, especially those related to control flow, are close to those found in workflow definition languages [7]. In the area of workflows, it has been shown that Petri nets [8] are appropriate both for modeling and analysis. More specifically, 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, and tools (see [12] for an overview) become directly applicable.

In this paper, we present an extension of the Petri net semantics of [3] (sometimes refered to as the "old semantics"). This extension is twofold: (1) we simplify

several patterns of the original semantics that resulted in huge nets, and (2) we introduce novel Petri net patterns for the constructs introduced by WS-BPEL 2.0 such as new activities or handlers.

The rest of this paper is organized as follows. In Sect. 2, we briefly introduce WS-BPEL, our formal model, and the basic concepts of the Petri net semantics and its graphical conventions. Then, in Sect. 3 and 4, we present the patterns for the basic and structured WS-BPEL activities, respectively. In Sect. 5, we present the new patterns for the most complex WS-BPEL constructs: the scope, the process, and handlers. Section 6 is devoted to the modeling of control links. The semantics was implemented in a compiler (BPEL2oWFN) which is presented in Sect. 7. The semantics and its compiler is compared with related work in Sect. 8. Finally, Sect. 9 concludes the paper and gives directions for future work.

2 Background

2.1 WS-BPEL

The *Web Services Business Process Execution Language* (WS-BPEL) [4], 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 activities* and *structured activities*. The basic activities are `<receive>` and `<reply>` to provide web service operations, `<invoke>` to invoke web service operations, `<assign>` to update partner links, `<throw>` to signal internal faults, `<exit>` to immediately end the process instance, `<wait>` to delay the execution, `<empty>` to do nothing, `<compensate>` and `<compensateScope>` to invoke a compensation handler, `<rethrow>` to propagate faults, `<validate>` to validate variables, and `<extensionActivity>` to add new activity types.

A structured activity defines a causal order on the basic activities and can be nested in another structured activity itself. The structured activities are `<sequence>` to process activities sequentially, `<if>` to process activities conditionally, `<while>` and `<repeatUntil>` to repetitively execute activities, `<forEach>` to (sequentially or in parallel) process multiple branches, `<pick>` to process events selectively, and `<flow>` to process activities in parallel. Activities embedded to a `<flow>` activity can further be ordered by the usage of *control links*.

Finally, the `<scope>` activity can add exception handling to an activity. For this purpose, there exist four kinds of handlers: a `<compensationHandler>` to compensate successfully executed scopes, `<faultHandlers>` to undo partial, unsuccessful executed scopes, a `<terminationHandler>` to control the forced termination of a scope, and `<eventHandlers>` to process message or timeout events. Though not listed as an activity, WS-BPEL's root element is the `<process>`, which is in fact a special `<scope>` activity.

2.2 Open Workflow Nets

Open workflow nets (oWFNs) are a special class of Petri nets. They generalize the classical workflow nets [9] by introducing an interface for asynchronous mes-

sage passing. oWFNs provide a simple but formal foundation to model services and their interaction. Open workflow nets — like common Petri nets — allow for diverse analysis methods of computer-aided verification. The explicit modeling of the interface further allows to analyze the communicational behavior of a service [13, 14].

To model data flow and data manipulation, Petri nets can be extended to algebraic high-level nets [15]. Similarly, open workflow nets can be canonically extended to high-level open workflow nets (HL-oWFNs).

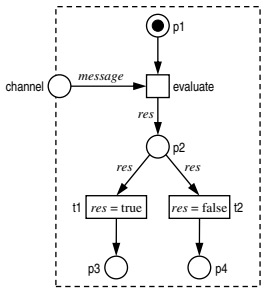


Fig. 1. A high-level oWFN.

An example for a high-level oWFN is depicted in Fig. 1. Transition evaluate receives a *message* (variable names are written in an *italic* font) from place channel and evaluates it. The evaluation process itself is not explicitly modeled. Still, the *res*ult of this evaluation (either the value 'true' or 'false') is produced on place p2. Then, depending on this value, either t1 (the guard "*res* = true", written inside the transition, holds) or t2 (the guard "*res* = false" holds) can fire. Throughout this paper, we refrain from depicting the concrete underlying Petri net schema. The domains of the places can be canonically derived from the patterns and the respective WS-BPEL activity.

2.3 A Petri Net Semantics for WS-BPEL

Both the semantics of [3] and the extension presented in this paper follow a hierarchical approach. The translation is guided by the syntax of WS-BPEL[1]. In WS-BPEL, a process is built by plugging instances of language constructs together. Accordingly, each construct of the language is translated separately into a Petri net. Such a net forms a *pattern* of the respective WS-BPEL construct. Each pattern has an *interface* for joining it with other patterns as is done with WS-BPEL constructs (cf. Fig. 2). Also, patterns capturing WS-BPEL's structured activities may carry any number of inner patterns as its equivalent in

[1] The semantics of [3] is only defined for BPEL4WS 1.1. As, however, the concept of the semantics is version-independent, we use "WS-BPEL" without version number unless we want to distinguish the two different versions.

WS-BPEL can do. The collection of patterns forms the *Petri net semantics* for WS-BPEL.

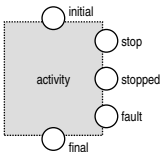


Fig. 2. The interface places of an activity: initial, final, stop, stopped, and fault. Marking the initial place starts an activity. Upon faultless completion of the activity, the final place is marked. The places stop and stopped model the termination of activities. Faults are signaled by marking the fault place.

Both semantics consist of high-level patterns which completely model WS-BPEL's control and data flow. As the data-domains of the variables can be infinite, abstract (low-level) patterns are implemented in the respective compilers BPEL2PN [16] and BPEL2oWFN. See Sect. 7.2 for detailed discussion.

2.4 Graphical Conventions

Experiences with the semantics from [3] show that patterns modeling the complete behavior of a WS-BPEL activity are — due to the large number of nodes and arcs — very hard to understand. One reason for illegible patterns can be too many crossings of arcs. We adapt the graphical notations of [3]: Dashed places indicate place duplications, and read arcs are used to avoid loop arcs, cf. Fig. 3.

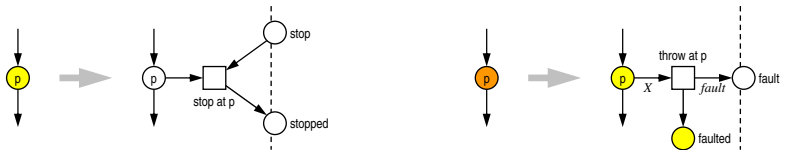
(a) place copies (b) read arcs

Fig. 3. Graphical conventions used to simplify patterns. (a) A dashed place is a copy of a place with the same label. (b) Read arcs are unfolded to loops.

Still, many structures or subnets of the patterns are very generic as they model a certain aspects of WS-BPEL that occur in many activities. To scale down the number of nodes of a pattern, and to focus on the original characteristic of each pattern, we introduce some additional graphical shortcuts for the patterns depicted in this paper.

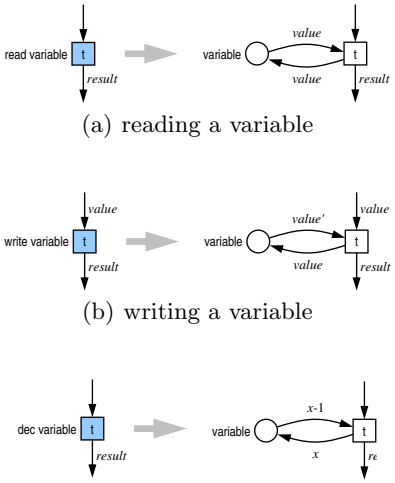
The mentioned subnets are usally connected with a single place or transition of the pattern and the interface places. To indicate the subnet, we color the respective place or transitions. For each color, we specify a transformation rule how

to insert the respective subnet. Figure 4 describes the implicit transformations for colored places and Fig. 5 and 6 for colored transitions.



(a) yellow places (b) orange places

Fig. 4. Graphical shortcuts: The control flow can be stopped at yellow places (a) by adding a stopping transition which is connected to the stop and stopped place of the respective activity. At orange places (b), a fault can be thrown on the activity's fault place (variable X and fault name *fault* depend on the context). Furthermore, the control flow can be stopped at the newly introduced yellow places.



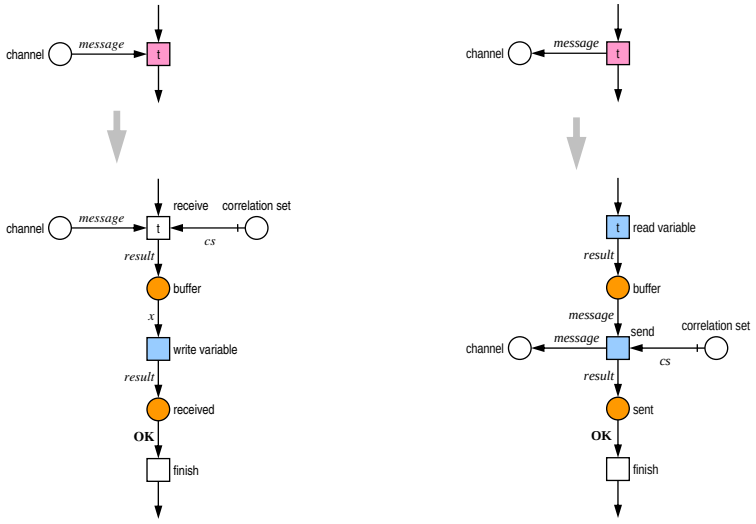
(a) reading a variable

(b) writing a variable

(c) decrementing a variable

Fig. 5. Graphical shortcuts: Blue transitions access variable places. The operation on the variable is indicated in the label of the transition, for example read (a), write (b), or decrement (c). The return variable *result* contains the current value of the connected variable place, or a fault name (e.g., `invalidExpressionValue`, `invalidVariables`, or `uninitializedVariable`) if the operation fails.

Finally, we simplify the representation of structured activities by "stacking" subnets. The patterns for structured activities usually consist of a repetition of identical subnets for each embedded activity. Instead of showing how each



(a) receiving a message (b) sending a message

Fig. 6. Graphical shortcuts: Pink transitions model communication; that is, receiving (a) and sending (b) messages. The variable and channel is provided by the respective activity. While faults thrown after reading/writing are of type `uninitializedVariable` or `invalidExpressionValue`, only `correlationViolation` faults can be thrown by the communicating transitions. The latter can occur if the message does not match the correlation set, which are not explicitly modeled.

of these subnets is connected to the rest of the pattern, we stack all activity's subnets and show how the topmost subnet is connected. This idea is presented in Fig. 7.

Fig. 7. A transition `t` connected to a stack place labeled `p` is "unfolded" to be connected with n instances of `p`. The actual value of n is determined by the context of the respective pattern, for instance the number of branches of a `<forEach>` activity.

3 Patterns for Basic Activities

3.1 Message Exchange: `<receive>`, `<reply>`, and `<invoke>`

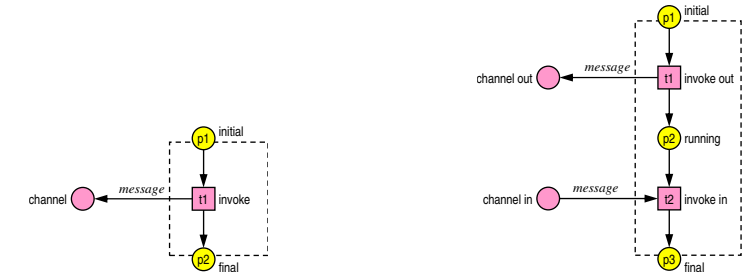
The `<receive>`, `<reply>`, and `<invoke>` activities are the basic activities to communicate with other web services by sending or receiving messages. While

the <receive> activity receives an incoming message from a calling service, the <reply> activity is used to "answer" this service calls.[2]

In contrast, the <invoke> activity can call other web services. WS-BPEL distinguishes two kinds of service invocation: synchronous and asynchronous invocations. Synchronous invocations are similar to handshakes: after sending the service call, the process blocks and waits for an answer of the called service. An asynchronous invocation does not block the calling service: it continues after sending the call. Possible results of the service call can be received later on using a <receive> activity.

(a) <receive> (b) <reply>

Fig. 8. Patterns for the <receive> and <reply> activity.



(a) asynchronous <invoke> (b) synchronous <invoke>

Fig. 9. Patterns for the asynchronous and synchronous <invoke> activity.

The corresponding patterns are depicted in Fig. 8 and 9. The input and output places are derived from the process description: for each occurring pair of `partnerLink` and `operation` attributes, a place partnerlink.operation is created and treated as input or output place according to the respective activities. In case of synchronous <invoke> activities, an input *and* output place has to be created.

Faults thrown when a conflicting receive or a dangling reply occur are not modeled in this semantics as they can be detected statically.

[2] Each <reply> activity has to have a matching <receive> activity.

3.2 Internal Activities: <empty> and <wait>

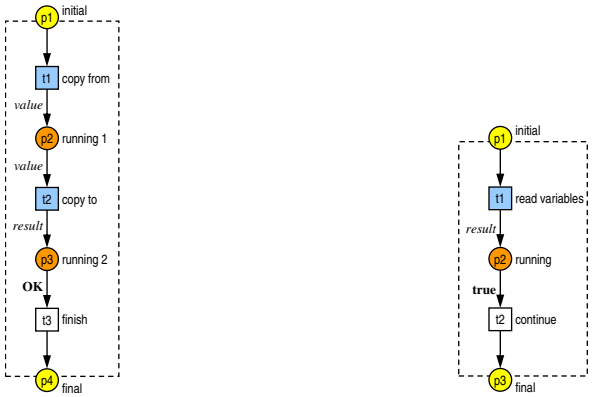
The <empty> can be seen as placeholder as it does nothing. The <wait> is used to delay the execution of the process and wait for a specified delay or until a deadline has passed. Figure 10 shows the patterns for these two activities. As we do not model time, the patterns coincide: Both consist of a single transition moving the control flow token from place initial (p1) to final (p2). Neither activity can throw a fault.

(a) <empty> (b) <wait>

Fig. 10. Patterns for the <empty> and <wait> activity.

3.3 Variable Handling: <assign> and <validate>

Variables values can be changed with the <assign> activity. Figure 11(a) shows the pattern using the graphical shortcuts (blue transitions) introduced in Fig. 5.



(a) <assign> (b) <validate>

Fig. 11. Patterns for the <assign> and <validate> activity.

The <validate> activity checks whether given variables matches their XML data definition. If the does not succeed, the WS-BPEL standard fault

`invalidVariables` is thrown. Figure 11(b) depicts the pattern for the `<validate>` activity. As the actual validation against the corresponding XML schema is out of the scope of this semantics, the decision whether or not to throw a fault is modeled by nondeterminism.

3.4 Signaling Faults: `<throw>` and `<rethrow>`

The `<throw>` activity explicitly signals an internal fault. The specification demands a fault name, and optionally, a fault variable to be specified. The activity can be modeled by a single place using the shortcut defined in Fig. 4(b) yielding the pattern depicted in Fig. 12(a). Figure 12(b) shows the explicit pattern for the `<throw>` activity without specified fault variable. Transition throw (t1) produces a token consisting of the fault name on place fault (p3). The arc inscription **faultName** is replaced by the specified `faultName` attribute.

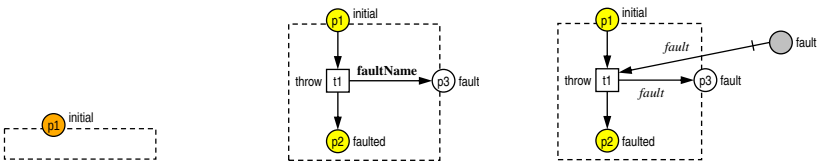


Fig. 12. Patterns for the `<throw>` and `<rethrow>` activity.

The `<rethrow>` activity may only be used inside a fault handler and is used to pass catched faults up to the parent scope. The corresponding pattern is depicted in Fig. 12(c). There only difference to the pattern of the `<throw>` activity is that transition rethrow (t1) produces a token with the fault name of the caught fault — modeled by the arc inscription *fault* — on place fault (p3). The name of the catched fault is read from place fault (depicted gray) of the fault handler (cf. Fig. 23).

In both patterns, the control flow can be stopped at places initial (p1) and faulted (p2). The pattern does not have a final place, because after signalling the fault, the enclosed scope will be stopped. Thus, no succeeding activities will be started.

3.5 Terminate Process Instance: `<exit>`

The `<exit>` activity demands to immediately end the process instance without termination handling. Figure 13 shows the corresponding pattern: The gray places are merged with the corresponding places of the `<process>` pattern, cf. Fig. 22.

We differentiate two scenarios:

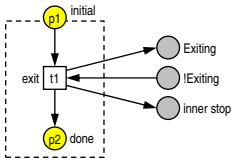


Fig. 13. Pattern for the `<exit>` activity.

- The process is running. In this case, transition exit (t1) changes the process's state to Exiting and marks place exit to invoke the termination of the process instance. The activity remains in state done (p2).
- The process is already exiting; that is, the process's !Exiting place is not marked. In this case, transition exit (t1) is not activated and the activity remains in state initial (p1).

The `<exit>` activity has no final place; that is, no subsequent activity will be started. If, for example, the `<exit>` activity is embedded to a `<flow>` activity, this flow will not be able to synchronize.

Instead of using a global place to organize the termination of the process or the stopping of activities in case of a fault, we follow an asynchronous, more distributed approach. Every running activity can be stopped by marking place stop, which *eventually* results in a token on place stopped. However, as a WS-BPEL process is a distributed system and the concurrency of multiple instances and scopes is underlined in the WS-BPEL specification, we decided not to *instantly* stop activities, but to model forced termination concurrently. This is a usual design decision for distributed systems and allows for maximal concurrency.

3.6 Invoke Compensation: `<compensateScope>` and `<compensate>`

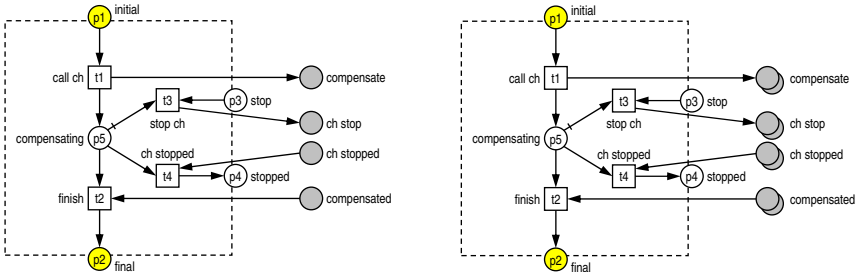
The `<compensateScope>` activity invokes the compensation handler of a child scope, specified using a `target` attribute. The activity may only be used inside a fault handler, a compensation handler, or a termination hander.

Figure 14(a) depicts the pattern for the `<compensateScope>` activity. The gray places compensate, ch stop, ch stopped and compensated are merged with the corresponding places of the scope pattern (cf. Fig. 21) addressed with the given `target` attribute. Firing call ch (t1) passes control to the respective compensation handler. If the target scope is successfully compensated transition finish (t2) completes the activity.

The `<compensateScope>` activity itself can not throw faults[3], but faults inside the called compensation handler are handled as if they occurred in the `<compensateScope>` activity.

When the `<compensateScope>` activity has to be stopped during compensation, that is, place compensating (p5) is marked, this request has to be passed

[3] In contrast to BPEL4WS 1.1, repeated compensation, or compensation of unsuccessful scopes are ignored instead of treated as faulty behavior.



(a) `<compensateScope>` (b) `<compensate>`

Fig. 14. Patterns for the `<compensateScope>` and the `<compensate>` activity.

to the active compensation handler. This is modeled by transition stop ch (t3) and ch stopped (t4), respectively.

The `<compensate>` activity invokes all child scopes' compensation handlers in default, that is, reverse order. As the `<compensateScope>` activity, it may only be used inside a fault handler, a compensation handler, or a termination hander.

Modeling the correct compensation order is a nontrivial task as it can usually not be derived from the WS-BPEL process description, that is, the XML document to be translated to a Petri net model. Therefore, instead of calling the compensation handlers in a specific order, we model *concurrent* calls. Thus, the resulting model has *more* behavior than the original process. However, this overapproximation is suitable for deadlock-detection and the analysis of other safety-properties.

The pattern for the `<compensate>` activity (cf. Fig. 14(b)) resembles the pattern of the `<compensateScope>` activity, but transitions call ch (t1), finish (t2), stop ch (t3), and ch stopped (t4) are now connected to the corresponding places of *all* child scopes.

Please note that — as all other activities — the `<compensate>` activity can be stopped at most once. Even if several of the called compensation handlers throw faults, only one of them is propagated to the scope that encloses the `<compensate>` activity. Thus, the stopping transition stop ch (t3) stops all child scopes' compensation handlers.

4 Patterns for Structured Activities

As described earlier, structured activities embed other activities. Thus, in patterns of this section, we depict the embedded activities by gray boxes and only show the interface places, that is, the places initial, final, and — if applicable or needed — the places stop, stopped and fault.

4.1 Sequential Execution: <sequence>

Activities embedded in a <sequence> activity are executed sequentially. Figure 15 shows the corresponding pattern. The final and initial places of consecutive embedded activities are merged. To stop the currently running activity of the sequence, the stop places (stopped places, respectively) are merged with those of the inner activities.

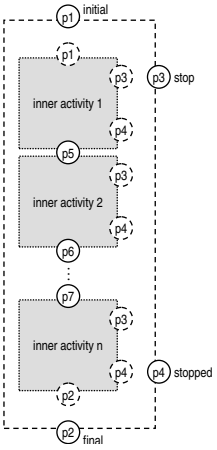


Fig. 15. Patterns for the <sequence> activity embedding n activities.

The pattern does not provide a transition to stop the control from places p1 (initial) or p5 (final): these places are merged with corresponding places of the embedded activities which provide those stop transitions.

4.2 Concurrent Execution: <flow>

The activities inside a <flow> activity are executed concurrently. However, it is possible to synchronize embedded activities with the help of control links. The links are not part of the <flow> activity itself and are subject of Sect. 6.

Fig. 16 shows the pattern of the <flow> activity. Transition split (t1) starts the embedded activities by producing a token on each activity's initial place. Transition t2 (join) synchronizes the finished activities. When the <flow> activity is stopped, transitions t3 (stop all) and t4 (all stopped) concurrently stop all embedded activities.

4.3 Conditionally Branching: <if>

The <if> activity consists of a list of conditions and list of activities. The conditions are checked sequentially. If a condition evaluates to true, the corresponding activity is executed and, after that activity finishes, completes the <if> activity.

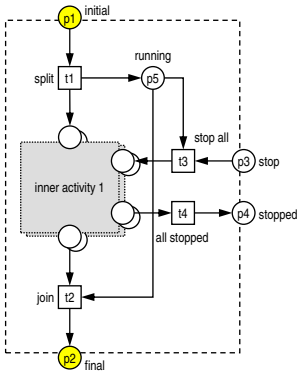


Fig. 16. Patterns for the `<flow>` activity embedding n activities.

More precisely, the `<if>` activity encloses at least one activity, an arbitrary number of `<elseif>` branches, and an optional `<else>` branch. The conditions of the mandatory activity and those of the `<elseif>` branches are checked sequentially. If no condition evaluates to true, the activity of the `<else>` branch — which has no condition attached — is executed. An unspecified `<else>` branch is treated as if it was specified with an `<empty>` activity. Thus, we can always assume a specified `<else>` branch when modeling the `<if>` activity.

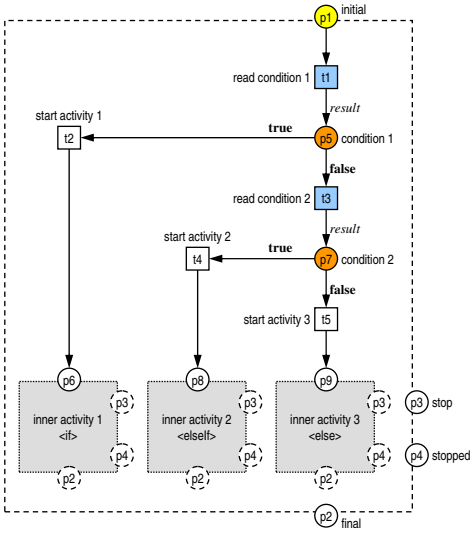


Fig. 17. Pattern for the `<if>` activity with three embedded activities.

Figure 17 shows the pattern of an `<if>` activity with three embedded activities (of which one is embedded to the `<else>` branch). Its behavior is an alteration between reading and evaluating expressions. If a condition evaluates to true, the corresponding activity is started. When the last condition evaluates to false, the activity embedded to the `<else>` branch is selected. The final places of the embedded activities are merged with the place final (p2).

Please note that the selection in the `<if>` activity is deterministic (i.e., the conditions are checked in the order they are specified in the XML document) and complete (i.e., exactly one activity will be selected). Completeness ensures deadlock-freedom at this point: even if all conditions evaluate to false, the `<if>` activity can complete.

We will discuss the handling of links inside the inner activities (i.e., dead-path elimination) in Sect. 6.

4.4 Selective Processing: `<pick>`

The `<pick>` activity waits for exactly one message or timeout event to occur. For each of the events an activity is associated which is executed if the corresponding event occurs. Similarly to the `<wait>` activity, we do not model time.

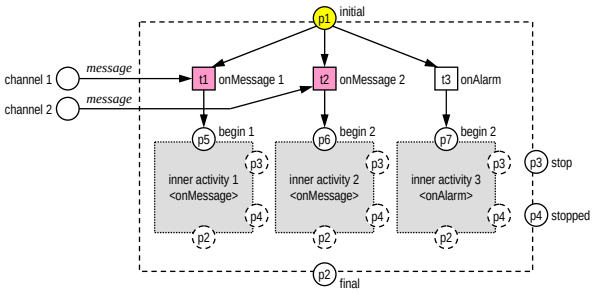
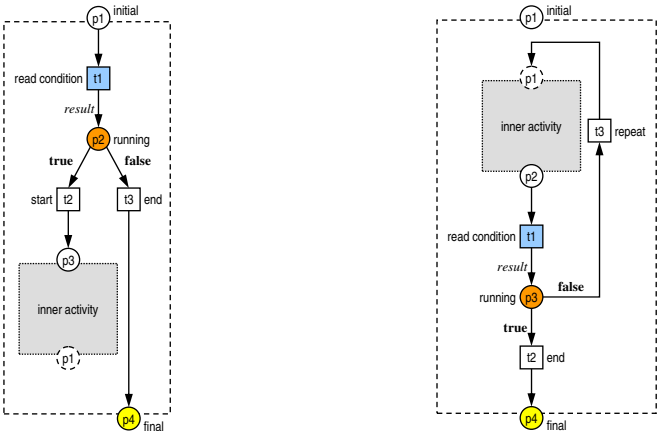


Fig. 18. Pattern for the `<pick>` activity with three embedded activities.

The pattern for a `<pick>` activity with two `<onMessage>` branches and one `<onAlarm>` branch is depicted in Fig. 18. As for the basic activities, we used a shorthand notation for the message-receiving transitions. Again, dead-path elimination is discussed in Sect. 6.

4.5 Repetitive Execution: `<while>` and `<repeatUntil>`

The `<while>` and the `<repeatUntil>` activity allow for repeated execution of an embedded activity. Whereas the embedded activity of the `<while>` activity is repeatedly executed *while* a given expression holds, the activity embedded in the `<repeatUntil>` activity is executed *until* an expression holds. Thus, the



(a) `<while>` (b) `<repeatUntil>`

Fig. 19. Patterns for the `<while>` and the `<repeatUnil>` activity.

`<while>`'s inner activity can be skipped (the condition initially evaluates to false) whereas the `<repeatUntil>`'s inner activity is executed at least once.

Figure 19 shows the patterns of both activities. In both patterns, transition t1 (read condition) reads the value of the variables occurring in the condition expression and produces the evaluation result — represented by the arc inscription *condition* — on place running. Dependent on the result, that is, *true* or *false*, either transition t2 or t3 can fire. This is again represented by the according arc inscription.

In addition to the Boolean values, the evaluation of the condition expression can fail. In this case, transition t1 (read condition) produces a token with value *failed*. This token is then consumed by a transition that produces a token *invalidExpressionValue* on the scope's fault in place. Again, this fault transition is not explicitly depicted as we use the short hand notations described in Fig. 4(b).

4.6 Processing Multiple Branches: `<forEach>`

The `<forEach>` activity allows to parallel or sequentially process several instances of an embedded `<scope>` activity. To this end, an integer counter is defined which is running from a specified start counter value to a specified final counter value. The enclosed `<scope>` activity is then executed according to the range of the counter. In addition, an optional *completion condition* specifies a number of successful executions of the `<scope>` activity after the `<forEach>` activity can be completed prematurely.

Sequential `<forEach>` The semantics of the sequential `<forEach>` activity can be simulated by a `<while>` or a `<repeatUntil>` activity which encloses a `<scope>` activity and an `<assign>` activity that organizes the counter. As

the resulting pattern is rather technical and straightforward, we refrain from a presentation.

Parallel `<forEach>` To model the parallel `<forEach>` activity, the number of instances of the embedded `<scope>` activity — that is, the range of the counter — has to be known in advance. It can be derived using static analysis, for instance. Due to the expressive power of XPath, static analysis of WS-BPEL processes with arbitrary XPath expressions is undecidable. Thus, if no upper loop bound can be derived, this bound has to be given explicitly. However, for the case where the loop bound is received as a message, existing work [17] can be adapted to create an exact model in this case.

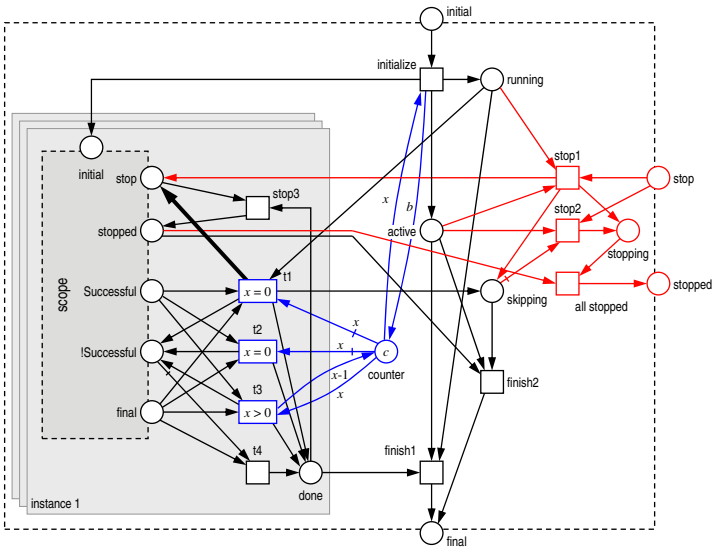


Fig. 20. The pattern for the parallel `<forEach>` activity. To ease the presentation, we use the colors to highlight certain aspects of the pattern. The counter, its adjacent arcs, and the transitions reading or increasing it are colored blue. All nodes organizing the stopping of the pattern are colored red.

Figure 20 depicts the generic pattern for an arbitrary but fixed number of scope instances. All nodes in the grey rectangle (the scope pattern as well as transitions t1–t4 and stop3) are present for each instance, whereas the other nodes of the pattern belong to the `<forEach>` activity itself and exist only once. To simplify the graphical representation, we merge arcs from or to instanced places. For example, the arc from transition initialize to the place initial of the scope pattern represents a single for each instance. Likewise, transition finish1 is connected to the done places of all instances. In addition, the bold depicted

arc connects each instance's t1 transition with *every* stop place of the instance's scope.

We now describe the possible scenarios of the parallel <forEach> activity and their respective firing sequences in the pattern of Fig. 20. Any scenario starts with the firing of transition initialize which initializes all embedded scope patterns and produces a token with the value b on place counter. This value describes the completion condition; that is, the number of scope instances that have to finish successfully to end the <forEach> activity prematurely. The <forEach> activity is now in state active and running.

- **Normal completion.** The instances are concurrently executing their embedded <scope> activities. When a scope completes, its final place is marked. In addition, either place Successful (the scope executed faultlessly) or place !Successful is marked (the scope's activity threw a fault that could be handled by the scope's fault handlers). In case of successful completion, transition t3 fires and resets the scope's state to !Successful and marks the instance's done place. Furthermore, the counter is decreased. If the scope was in state !Successful, transition t4 produces a token on the instance's done place without decreasing the counter. When all instances' scopes are completed, transition finish1 completes the <forEach> activity.
- **Premature completion.** When a sufficient number of scope instances have completed faultlessly, the <forEach> activity may complete prematurely; that is, it ends without the need to wait for the other still running scopes to complete. As mentioned before, the completion condition is modeled by the counter place. As this counter is decreased every time an instance's <scope> activity completed faultlessly, the counter value might reach 0. In this case, transition t3 is — due to its guard — disabled. Instead, transition t1 can fire which resets the scope as before and additionally sets the <forEach>'s state to skipping. Furthermore, it produces a token on the stop place of every instance's scope.[4] Thus, all running scopes are stopped. Eventually, the stopped place of all instances is marked — any tokens on the done places are also removed — and transition finish2 completes the <forEach> activity. Due to the asynchronous stopping mechanism, it is possible that other scopes complete while their stop place is marked. In this case, transition t2 behaves similarly to transition t1, but does not initiate the stopping sequence again.
- **Forced termination.** The <forEach> activity can — as all other activities — be stopped at any time by marking its stop place. Transitions stop1 and stop2 organize the stopping for the normal completion and the premature completion, respectively. The counter is not changed by the stopping mechanism, because its value is overwritten each time the <forEach> starts.

The <forEach> activity is mainly used to parallel or sequentially perform similar requests addressed to multiple partners and is thus an important construct to model service orchestrations or choreographies. To simplify the presen-

[4] This is depicted by the bold arc. Transition t1 also produces a token on the stop place of the scope that just finishes.

tation of the pattern, we do not depicted the subnet that organizes the compensation of the instance's scopes.

5 Process, Scope, and Handlers

5.1 Expressing Behavioral Contexts: `<scope>`

As the `<scope>` activity not only embeds an activity, but can also contain event, fault, compensation, and termination handlers, it is WS-BPEL's most complex activity. This complexity is reflected by the big `<scope>` pattern of the semantics of [3]. Though termination handlers were not introduced in BPEL4WS 1.1, this pattern still had to be distributed to several subpatterns, one for each handler. In addition, a *stop component* which has no equivalent in WS-BPEL was added to the `<scope>` pattern. This pattern by itself consists of 32 places, 16 transitions, and also uses a reset arc [18].[5] The main purpose of this component is to model the interactions of the several subpatterns in case of fault and compensation handling, or during the termination of the scope. In particular, the stop component uses several status places to "distribute" control and data tokens to the correct subpattern. Thus, it is possible to signal faults to a unique place of the scope. However, faults occurring in the embedded activity can be handled by the fault handler of the respective scope whereas faults of the compensation handler have to be handled by the parent scope's fault handler. This separation of positive control flow inside the activities' patterns and the negative control flow organized in the stop component allowed comprehensible patterns. Still, the stop pattern introduced several intermediate states. In addition to this possible state explosion, the scope pattern of [3] could not be nested inside repeatable constructs such as `<while>` activities or event handlers[6]. To this end, we decided not to extend the existing scope pattern, but to create a new pattern optimized for computer-aided verification while covering the semantics specified by WS-BPEL 2.0.

The main idea of the new pattern is to use as much information about the context of the activities as possible. For example, we refrain from a single place to signal faults to avoid a stop component to distribute incoming fault tokens. Instead, we use static analysis to derive information of the activities from the WS-BPEL process. If, for example, an activity is nested in a fault handler, faults should be signaled to the fault handler's parent scope *directly*. This way, we decentralize the aspects encapsulated in the stop component, resulting in patterns which are possibly less legible yet avoiding unnecessary intermediate states.

The new scope pattern is depicted in Fig. 21. It consists of four parts modeling the different aspects of the scope.

[5] This reset are can be unfolded as the connected place is bounded, resulting an even bigger subnet.

[6] The WS-BPEL 2.0 specification now actually demands activities in event handlers to be nested in a `<scope>` activity.

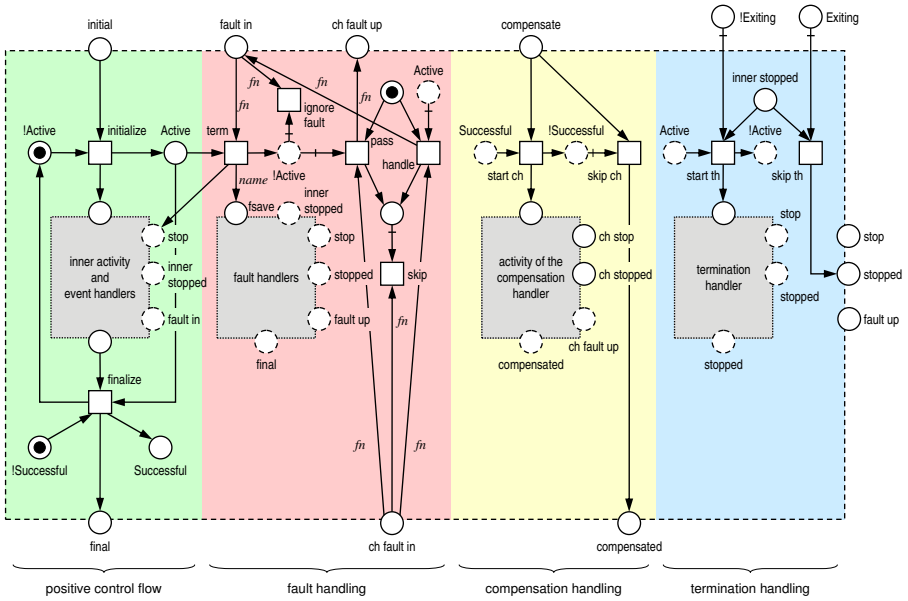


Fig. 21. The pattern for the <scope> activity. It consists of four parts modeling the different aspects of the scope: the positive control flow consisting of the embedded activity and the event handlers, the fault handlers, the compensation handler and the termination handler.

- The **positive control flow** consists of the inner activity of the scope and the optional event handlers. It is started by transition initialize which sets the scope's state to Active. The scope remains in this state while the inner activity and the event handlers are executed. Upon completion, transition finalize sets the scope's state to !Active (the positive control flow is not active) and Successful (the embedded activity ended faultlessly). The latter state is later used by the compensation handler.
- The **negative control flow** consists of the fault handlers and a small subnet organizing the stopping of the embedded activity. It can be seen as the remainder of the former stop component, yet it is integrated more closely to the rest of the pattern. When a fault occurs in the inner activity or the event handlers, a token consisting of the fault's name is produced on place fault in. As the positive control flow is active, place Active is marked. Thus, transition term is activated. Upon firing, the scope's state is set to !Active, and the stop place of the inner activity and the event handlers is marked. Furthermore, the fault's name *fn* is passed to the fault handlers (place fsave). When the positive control flow is stopped (place inner stopped is marked), the fault handlers are started. If they succeed, place final is marked and the scope has finished.[7] If, however, the fault could not be handled or the fault

[7] The scope is left in state !Sucessful to avoid future compensation.

handlers themselves signal a fault, place fault up is marked. This place is merged with the parent scope's or process's fault in place. Instead of using a reset arc to ignore any further faults occurring during the stopping of the embedded activity, transition ignore fault eventually removes all tokes from place fault in. See Sect. 5.3 for details about the pattern modeling the <faultHandlers>.

- The transitions pass, handle, and skip organize the fault propagation in case the compensation handler throws a fault. In this case, the fault is passed to the scope that called the faulted compensation handler. The **compensation handler** itself is not modeled by a special pattern, but its embedded activity is directly embedded to the scope. The compensation of the scope is triggered by a <compensate> or <compensateScope> activity that produces a token on place compensate. If the positive control flow of the scope completed faultlessly before (i. e., place Successful is marked), transition start ch starts the compensation handler's activity. If the scope did not complete faultlessly or the compensation handler was already called, transition skip ch skips the embedded activity. In any case, place compensated is marked. This place is again merged with the calling <compensate> or <compensateScope> activity.
- The **termination handler** is a new feature of WS-BPEL 2.0 and is discussed in the next section. The termination behavior of BPEL4WS 1.1 can, however, be simulated by embedding a <compensate> activity to the termination handler. See Sect. 5.4 for details about the pattern modeling the <terminationHandler>.

The new scope pattern is more compact as the pattern from the semantics of [3]. It correctly models the behavior of a <scope> activity for both BPEL4WS 1.1 and WS-BPEL 2.0 processes. Furthermore, it is easily possible to reset the status places which allows for scopes embedded in repeatable constructs (cf. the <forEach> pattern in Fig. 20). Finally, due to the absence of a stop component which is connected to all subpatterns, it is easy to derive parameterized patterns for any constellation of handlers, for example, a pattern for a scope without any handlers, a pattern for a scope with just an event handler, etc.

5.2 The Root Scope: <process>

The <process> is the root of the scope hierarchy. Thus, its pattern (cf. Fig. 22) is very similar to the original <scope> pattern (cf. Fig. 21).

However, there are three differences:

- The <process> construct does not embed a compensation handler. This is due to the fact that a compensation handler can only be invoked after successful completion of its embedding scope. However, the whole process instance will be terminated after the process completes successfully, so an embedded compensation handler would be unreachable.

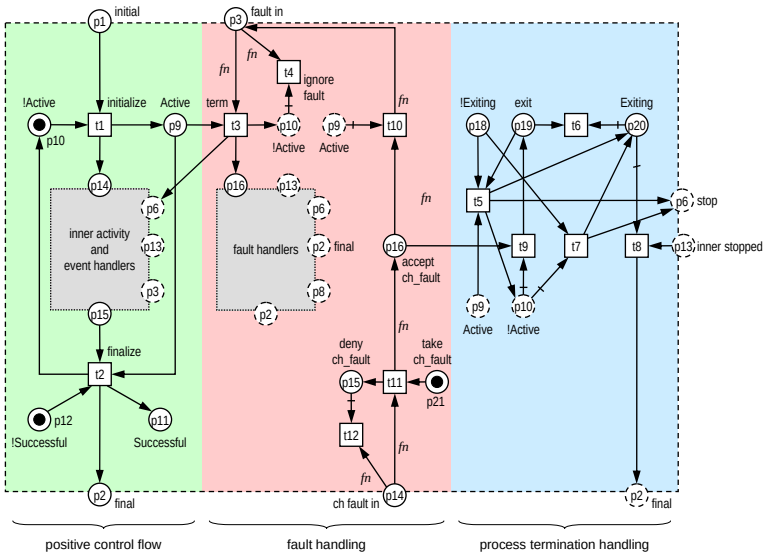


Fig. 22. The pattern for the `<process>` activity.

– Similarly, the `<process>` construct does not embed a termination handler, as there is no concurrently running scope that could force the process to terminate.
– Finally, the `<process>` construct embeds a subnet to organize the process termination. As the process is the root element of the scope hierarchy, it cannot rethrow faults that cannot be handled by the process's fault handlers. Instead, the whole process instance is terminated. The process termination is controlled by two status places, !Exiting (p18) and Exiting (p20). These places are also used by the `<exit>` pattern (cf. Fig.13).

5.3 Handle Runtime Faults: `<faultHandlers>`

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

As the definition of fault handlers is optional, a fault might not be caught by the enclosing scope. Consequently, this fault is re-thrown to the next parent scope until it is either handled by a fault handler or, if it can not be handled by the process's fault handler, yielding the termination of the whole WS-BPEL process. If more than one fault occurs in a scope, only the first is handled by the

fault handlers; all subsequent faults are ignored. If a fault occurs inside a fault handler, it is re-thrown to the parent scope.

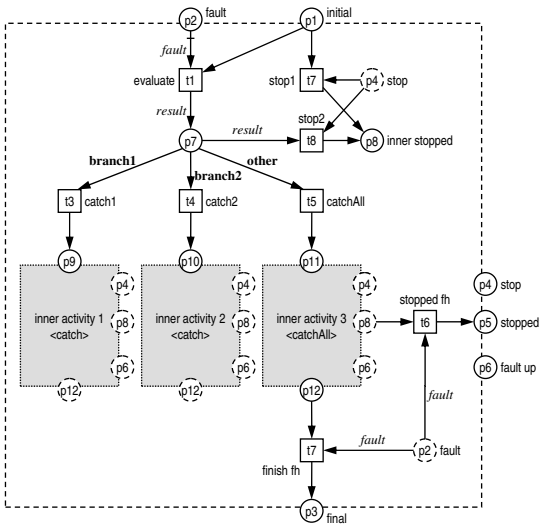


Fig. 23. The pattern for the fault handlers.

The pattern for the fault handlers is depicted in Fig. 23. When both place initial (p1) and fault (p2) are marked, transition evaluate (t1) evaluates the fault name and decides which branch the fault handlers should take to handle this fault. This decision is based on the name of the fault.[8] After firing transition evaluate, the fault name token stays on place fault (p2) until the fault handlers complete, because this fault has to be accessible to the `<rethrow>` activity (cf. Fig. 12(c)). When an activity enclosed to a `<catch>` or `<catchAll>` branch throws a fault, this fault is propagated to the parent scope by marking place fault up (p6).

5.4 Controlling Forced Termination: `<terminationHandler>`

By the help of a termination handler, the user can define how a scope behaves if it is forced to terminate by another scope. The termination handler is syntactically optional, but — if not specified — a standard termination handler consisting of a single `<compensate>` is deemed to be present.[9]

[8] Instead of a transition that choses the branch immediately, it would also be possible to model this decision with a cascade of transition comparable to the `<if>` pattern (cf. Fig. 17). This, however, would have complicated the pattern.

[9] This standard termination handler also models the behavior described in the BPEL4WS 1.1 specification.

The termination handler is only executed if (1) the scope's inner activity has stopped, (2) no fault occurred, and (3) no `<exit>` activity is active. In the scope pattern of Fig. 21, these prerequisites are fulfilled if the places inner stopped, Active, and !Exiting (a status place of the process that is marked unless an `<exit>` activity is active) are marked. Then, transition start th invokes the termination handler. In any other case, place stopped is marked. Unlike the compensation handler, the termination handler's activity cannot be embedded directly to the scope, but needs a wrapper pattern, depicted in Fig. 24.

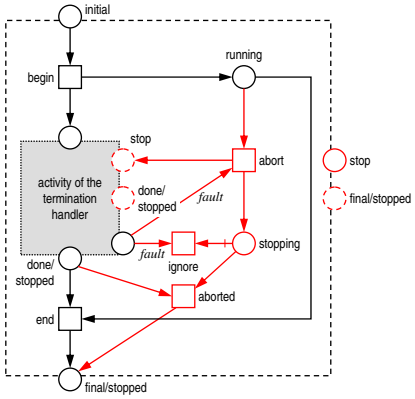


Fig. 24. The pattern for the termination handler. All nodes organizing the stopping of the pattern are colored red.

In the positive control flow, transitions begin and end start the embedded activity and end the termination handler, respectively. If the embedded activity throws a fault, it is not propagated to the scope's fault handler, because the scope is forced to terminate to handle a fault that occurred in a different scope. Thus, transition abort just stops the inner activity if a fault occurred, and transition ignore fault ignores further faults. When the inner activity is stopped, place done/stopped place is marked and transition aborted completes the termination handler similarly to transition end.

5.5 Processing Message and Timeout Events: `<eventHandlers>`

Event handlers allow for repeated reaction to inbound messages as well as timeout triggered behavior. They can be attached to scopes and are initialized together with its enclosing scope, and are active as long as their enclosed scope is executed. Event handlers are modeled similarly to a `<pick>` activity inside a loop. Multiple incoming messages or timeout events[10] can be handled concurrently. However, if two or more messages for a single message event handler

[10] Again, we do not explicitly model time.

arrive concurrently, they are handled sequentially. When the scope's embedded activity has completed faultlessly, the event handler is uninstalled: all active timeout or message event handlers can finish their execution; that is, they are not stopped.

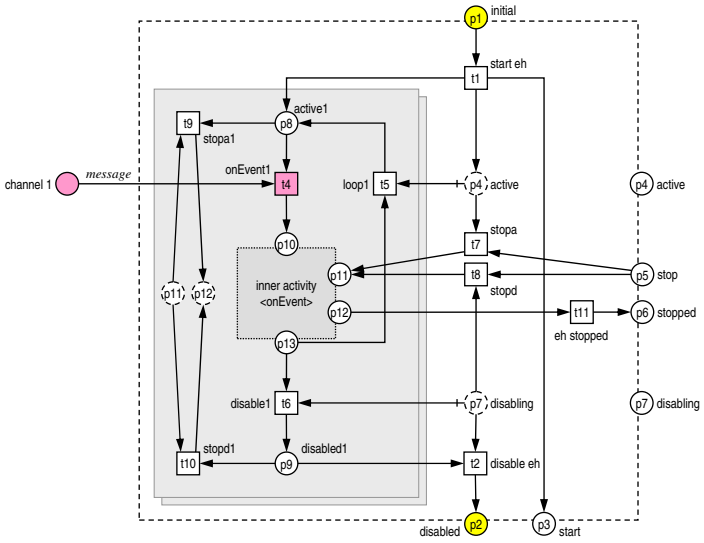


Fig. 25. The pattern for the event handlers.

The pattern for the `<eventHandlers>` is depicted in Fig. 25. In addition to the usual interface places, it has two additional places running (p2) and finishing (p3) to model the internal state of the `<eventHandlers>` and to make it accessible to the enclosing scope or process. Transition start eh (t1) initializes all embedded `<onEvent>` and `<onAlarm>` branches and changes the event handlers' state to running. In that state, the specified events (cf. transition event 1 (t2)) can trigger the respective embedded activity.

At any time, the enclosing scope or process can change the event handlers' state to finishing. This prevents future events to trigger activities. However, all currently running activities can finish. When all branches finished (i.e. place finished 1 (p10) is marked in all branches), transition finish eh (t7) finishes the event handlers. When the event handlers are stopped by marking p5 (stop), also the embedded activities are stopped.

Figure 26 shows the pattern for an timeout event handler. The pattern can be easily adjusted to model the semantics of the `repeatEvery` attribute that can be set to a timeout value.

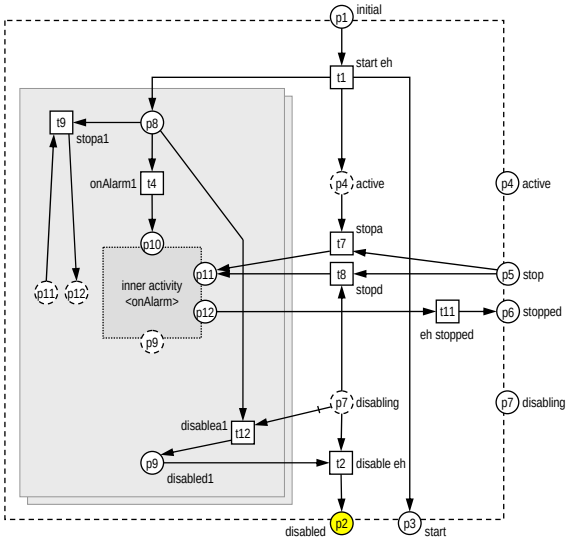


Fig. 26. The pattern for the event handlers.

6 Links and Dead-path-elimination

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

More precisely, when the source activity is executed faultlessly, the outgoing links are set according to their corresponding *transition conditions* which returns a Boolean value for each outgoing link. After the status of all incoming links of a target activity is determined, a *join condition* — again a Boolean expression[11] — is evaluated. If this condition holds, the target activity is executed. If, however, the condition is false, the activity is skipped. In this case, all outgoing links recursively embedded to the skipped activity are also set to false to avoid deadlocks. This concept is called *dead-path-elimination* (DPE) [19] and can be enabled for each target activity.

6.1 Dead-path Elimination

As an example, consider the `<flow>` of Fig. 27. Two scenarios are possible, depending on the condition of the `<if>` activity: If the condition evaluates to true, we have the execution order shown in Fig. 28(a). Firstly, A is executed and

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

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

Fig. 27. An example for links and dead-path-elimination. `<activity>` is a placeholder for any WS-BPEL activity.

sets link AtoB to true, then B is executed and sets link BtoC. Finally, C and D are executed sequentially.

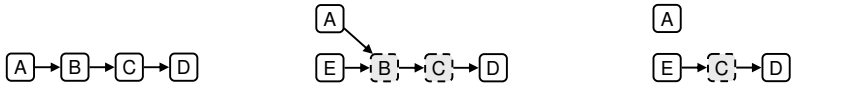


Fig. 28. Possible executions of the activities of the example in Fig. 27. Skipped activities are depicted with dashed lines. The executions (a) and (b) correctly model the specified behavior, whereas the execution (c) does neither skips nor executes activity B.

In case the condition evaluates to false, E is executed and, due to the DPE, activity B is skipped; that is, B has to wait until A has set its link AtoB. Then, B's outgoing link, BtoC, is set to false and C is also skipped. Finally, D is executed. This yields the execution order of Fig. 28(b). These two runs are correctly modeled by the semantics of [3] using a subnet in each pattern to bypass the execution of the activity and to set outgoing links to false.

However, if the branches to be skipped are more complex, the skipping of activities yields a complex model due to the DPE. In particular, skipping of activities and execution of non-skipped activities is interleaved which might result in state explosion problems. To this end, the new semantics differs from the described behavior of [4]: an *overapproximation* of the process's exact behavior is modeled. In the example, activity B is not skipped explicitly, but its outgoing link, BtoC, is set to false *directly* when E is selected. This yields the execution order of Fig. 28(c). Compared to the semantics of [3], two *additional*

runs are modeled by the new semantics, namely A and D being executed concurrently, and D being executed before A. Due to the overapproximation, it may be possible that the resulting model contains errors that are not present in the WS-BPEL process. For example, activity A and D could be `<receive>` activities that receive messages from the same channel. If they are active concurrently, a "conflicting receive" fault would be thrown. However, static analysis of the WS-BPEL process can help to identify these pseudo-errors (see [14] for details).

6.2 Link Wrapper Patterns

Figure 29–31 depict the wrapper patterns for activities with incoming or outgoing links. While there is a unique wrapper pattern for activities with only outgoing links (cf. Fig. 29), there exist two different patterns in case of incoming links, depending on the attribute `suppressJoinFailure` (cf. Fig. 30 and 31). The latter patterns are additional examples for the direct setting of recursively embedded links (transition skip). Again, transition evaluate_JC and evaluate_TC only implicitly model the evaluation of the join and transition condition, respectively. An explicit model of the evaluation would require to take XPath expressions, XML variables, etc. into account and is out of scope of this paper. However, it is possible to model some aspects of the link setting by low-level patterns. This is discussed later in Sect. 7.2.

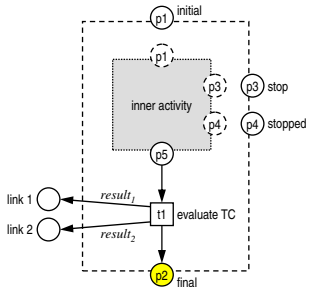
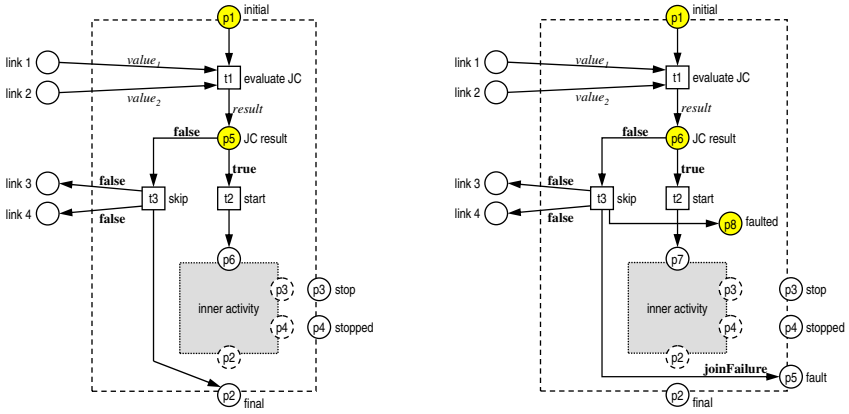


Fig. 29. Wrapper pattern for an activitiy with only outgoing links (link1 and link2).

7 The Compiler BPEL2oWFN

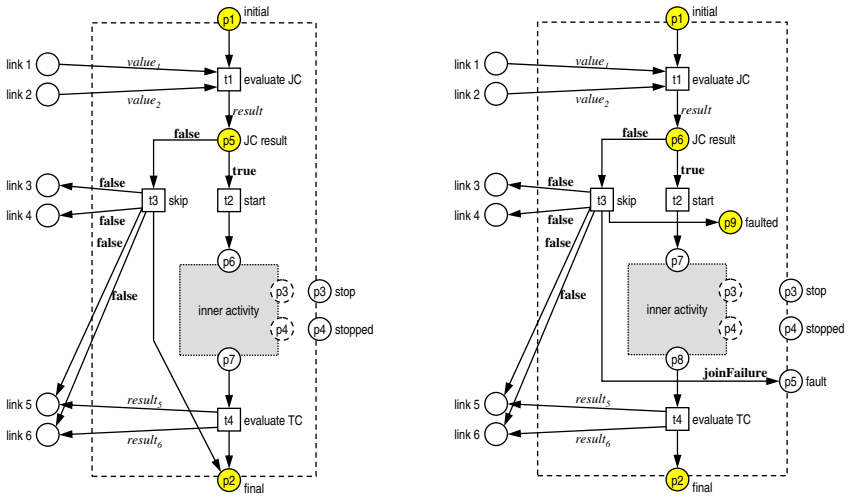
The presented semantics is implemented in the tool BPEL2oWFN[12]. It is capable of generating oWFNs and other file formats (e. g., PNML, PEP, LoLA, INA, SPIN) and thus supports a variety of analysis tools.

[12] BPEL2oWFN is available at `http://www.gnu.org/software/bpel2owfn`.



(a) `suppressJoinFailure="yes"` (b) `suppressJoinFailure="no"`

Fig. 30. Wrapper patterns for an activitiy with only incoming links (link1 and link2). The links link3 and link4 are recursively embedded to the inner activity and are set to false as soon as it is clear that the inner activity is skipped.



(a) `suppressJoinFailure="yes"` (b) `suppressJoinFailure="no"`

Fig. 31. Wrapper patterns for an activitiy with both incoming (link1 and link2) and outgoing links (link5 and link6). The links link3 and link4 are recursively embedded to the inner activity and are set to false as soon as it is clear that the inner activity is skipped.

7.1 Data Abstraction

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 decisions (e. g., in the `<if>` activity) are modeled by nondeterministic choices. In terms of Petri nets such an abstract net is a *skeleton net* [20].

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

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

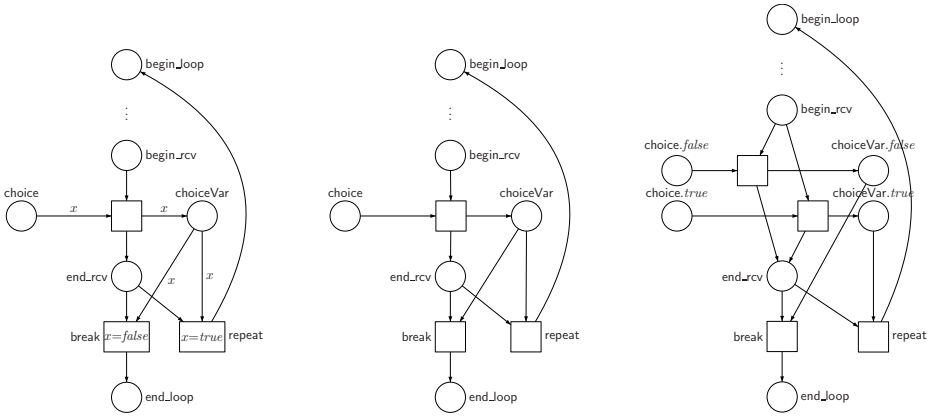
Fig. 32. An example snippet of a `<repeatUntil>` activity. The loop can be controlled by sending the message choice. The message content is stored in the variable choiceVar and is used to evaluate the loop condition. In case of a message with content *true* (*false*), the loop is repeated (exited).

Consider the WS-BPEL snippet of Fig. 32. The idea of the abstraction is illustrated in Fig 33. 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 (cf. Fig. 33(c)).

7.2 Low-Level Link Wrapper Patterns

While most data-dependend decisions have to be replaced by non-determinism, the join conditions and transition conditions can be treated differently. On the one hand, the state of a link has a finite domain as it is either *true* or *false*. On the other hand, join conditions and transition conditions have a special structure that usually allows for more precise low-level models.

As described earlier, variables with finite domain can be unfolded. Links can be treated as Boolean variables; that is, we can unfold them to two places, namely link.true and link.false for every high-level link place link. These unfolded link places are then connected to placeholder patterns for the join condition and the transition condition of the wrapper pattern. Figure 34 depicts the low-level pattern for an activity with incoming and outgoing links in case of



(a) high-level oWFN (b) skeleton oWFN (c) unfolded oWFN

Fig. 33. Several translations of the snipped of Fig. 32. (a) 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 (b). Messages are undistinguishable tokens and the condition is evaluated nondeterministically. As the data domain is of type *xsd:boolean*, the high-level places of (a) can be unfolded into two places *true* and *false*, shown in (c).

`suppressJoinFailure` set to `no`. It is the low-level version of the pattern depicted in Fig. 31(b) and can be easily changed to model the other four wrapper patterns.

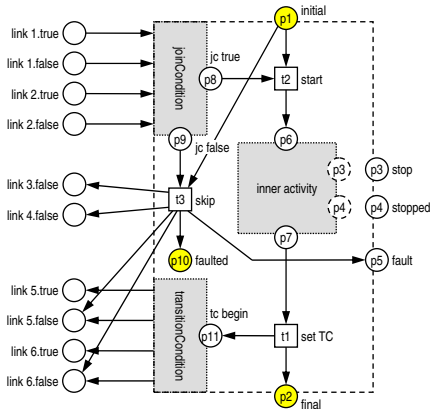


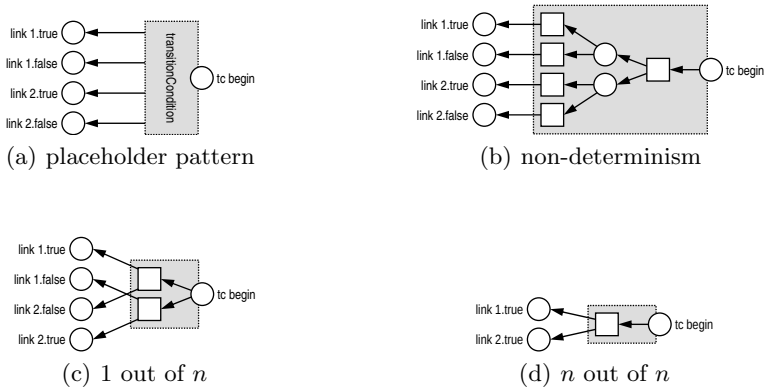
Fig. 34. Low-level link wrapper pattern for an activity with incoming and outgoing links and `suppressJoinFailure` set to `no`.

We already described in Sect. 6 that join conditions are not arbitrary XPath expressions, but Boolean expressions over the states of the incoming links. Thus, it is possible to exactly model a join condition by a low-level Petri net which then replaces the placeholder pattern of Fig. 34. Figure 35 gives an impression how this constructed low-level patterns look like. However, the size of such patterns can grow exponentially in the number of links mentioned by the join condition.

(a) placeholder pattern (b) conjunction: link1 ∧ link2

Fig. 35. Patterns modeling join conditions. The placeholder pattern (a) can be unfolded according to any join condition, for example a conjunction of the incoming links (b).

In contrast to join conditions, no constraints about the structure of transition conditions are specified. Therefore, transition conditions can be arbitrary complex XPath conditions which also involve variables other than control links. Nevertheless, some widespread design patterns can be modeled by more exact low-level patterns (cf. Fig. 36). For example, a transition condition setting exactly one link to true and all others to false can — though the decision which link is set to true is not known — be modeled more faithfully by the pattern depicted in Fig. 36(c).



(a) placeholder pattern (b) non-determinism

(c) 1 out of n (d) n out of n

Fig. 36. Patterns modeling transition conditions. As transition conditions can be arbitrary expressions, the placeholder pattern (a) is usually replaced by non-determinism (b). However, patterns for standard cases like "1 out of n" (c) or "n out of n" (d) can be used.

In the current implementation of BPEL2oWFN, such tailored low-level transition conditions have to be chosen manually. An automatic choice of the most suited low-level transition condition pattern could be implemented using static analysis algorithms, and is subject of future work.

7.3 Flexible Model Generation

In contrast to its predecessor BPEL2PN [16], 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 [21]) only apply structural reduction rules after performing a brute-force translation. Figure 37 presents an overview of our proposed framework.

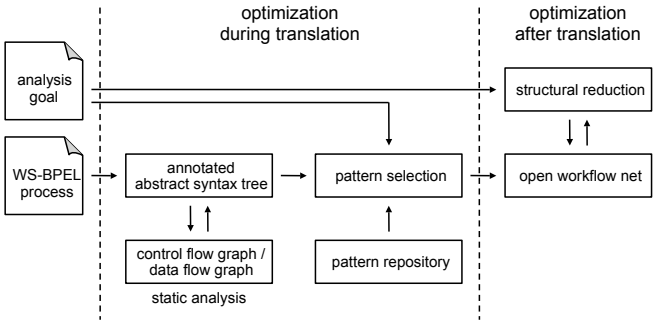


Fig. 37. Framework for flexible model generation implemented in BPEL2oWFN. For a WS-BPEL process the abstract syntax tree (AST) is built. For each WS-BPEL construct context information is gained from static analysis techniques and annotated to the AST. In the translation process, the annotated AST is 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, 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 [22] 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.

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 [23]) 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.[13]

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.

7.4 Further Features

The model generated by BPEL2oWNF can be analyzed according to the analysis goal. For example, the the communicational behavior (i. e., controllability [11] or operating guidelines [24]) of the WS-BPEL process can be analyzed using the tool Fiona[14]. For more information, see [13, 14].

Furthermore, detection of unreachable activities by using model checkers such as LoLA [25] is possible, too. In addition to the presented CFG-based algorithms, BPEL2oWFN also uses the CFG to check 56 of the 94 static analysis requirements[15] proposed by the WS-BPEL specification [4]. They enable BPEL2oWFN

[13] 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.

[14] Fiona is available at `http://www.informatik.hu-berlin.de/top/tools4bpel/fiona`.

[15] 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.

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.

Finally, BPEL2oWFN is capable of translating multiple WS-BPEL processes and compose the resulting service models according to a choreography description. In particular, BPEL4Chor choreographies [26] can be translated into open workflow nets or closed (i. e., classical) Petri nets. A detailed case study is reported in [27].

8 Related Work and Comparison

In this section, we sum up related work and compare the presented semantics with the old semantics of [3]. Then, we show the effect of using flexible model generation compared to a brute-force translation that is limited to structural reduction of the generated model. Finally, the compiler BPEL2oWFN is compared with other existing compilers.

8.1 Related Work

Though many formal semantics for WS-BPEL were proposed (see [6] for an overview), to the best of our knowledge, no formal semantics of the new constructs of WS-BPEL 2.0 was proposed yet.

Ouyang et al. present in [28, 21, 29] a pattern-based Petri net semantics. This semantics models the behavior of the activities and constructs of BPEL4WS 1.1 with the semantics described an early specification draft of the WS-BPEL 2.0. Thus, the semantics adequately models the behavior of BPEL4WS 1.1 processes and avoids the ambiguities of the earlier specification [5]. However, constructs such as the `<forEach>` activity or termination handlers are not covered by this semantics. See [30] for a detailed comparison.

8.2 New Semantics vs. Old Semantics

To compare the new patterns (especially the `<scope>` pattern the the different dead-path-elimination) with the old patterns, we investigated an example process described in [16]. This process models a small online shop consisting of 3 scopes, 2 links, and 46 activities. The authors of [16] translated it using the old Petri net semantics. We translated this process with BPEL2oWFN.

Table 1. Comparison of the old and the new semantics: the net size together with the size of the resulting state space (complete and reduced using partial order reduction).

patterns	places	transitions	states, complete	states, reduced
old semantics	410	1,069	6,261,648	443,218
new semantics	242	397	304,007	74,812

Table 1 compares the net sizes and the resulting states spaces. With the presented simplified patterns, we can verify processes of realistic size. Furthermore, structural reduction rules can be applied to further reduce the net size and — due to less intermediate states — also the state space.

8.3 Flexible Model Generation vs. Brute-Force Translation

To compare flexible model generation with a brute-force translation approach, we translated a WS-BPEL process with BPEL2oWFN with different parameters. The process consists of 25 activities, a fault handler, an event handler, and a compensation handler. The case study was taken from [14] where flexible model generation is discussed in detail.

Table 2. Comparison of the different translation options: 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 2 compares the sizes of the resulting oWFNs of the different translation options. 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 6,358 (brute force, no structural reduction) to 543 (flexible model generation, structural reduction).

8.4 BPEL2oWFN vs. Other Compilers

Besides BPEL2oWFN, there are two other compilers translating WS-BPEL processes into Petri net models: BPEL2PN and BPEL2PNML:

- BPEL2PN [31] is the predecessor of BPEL2oWFN. It is a brute-force implementation of the semantics of [3]. It was designed to validate the semantics. In [16], BPEL2PN was used to translate a large BPEL4WS 1.1 process into a Petri net without interface. This net was then analyzed by the model checker LoLA [25]. Deadlock freedom and several temporal logical formulae could be proven.
- BPEL2PNML [28, 29] implements the semantics of the Queensland Technical University (QUT). It is capable of translating a BPEL4WS 1.1 process into a workflow net. This workflow net can be analyzed for (relaxed) soundness. Furthermore, some WS-BPEL-related properties such as dead code or the

absence of conflicting message-receiving activities can be verified by the tool WofBPEL [21].

Table 3 provides an overview of the features of the three compilers. A more detailed overview of the similarities and differences between BPEL2oWFN (and the semantics presented in this paper) and BPEL2PNML/WofBPEL (and the respective semantics) can be found in [30].

Table 3. Comparison of the compilers: BPEL2oWFN, BPEL2PN, and BPEL2PNML.

category	feature	BPEL2 oWFN	BPEL2 PN	BPEL2 PNML
input	BPEL4WS 1.1	+	+	+
	WS-BPEL 2.0	+	–	–
	WS-BPEL Abstract Process	+	–	–
	WS-BPEL4Chor	+	–	–
output	open workflow net	+	+	–
	workflow net	–	–	+
	PNML file format	+	–	+
	LoLA file format	+	+	–
	other file formats (e. g. SPIN, PEP)	+	–	–
reduction	structural reduction	+	–	+[1)]
	flexible model generation	+	–	–
analysis	cyclic links	+[2)]	–	–
	unreachable activities	+[2)]	–	+[3)]
	conflicting receives	+[2)]	–	+[4)]
	WS-BPEL static analysis goals	+[2)]	–	–

[1)] postprocessing using WofBPEL; [2)] using static analysis; [3)] using transition invariants with WofBPEL; [4)] using state space analysis with WofBPEL

9 Conclusion

We presented a feature-complete Petri net semantics that models all data and control flow aspects of a WS-BPEL (version 1.1 or 2.0) process. The semantics is an extension of the semantics presented in [3]. To allow more compact model sizes, we simplified and reduced important aspects such as dead-path-elimination and the `<scope>` pattern. First experiments show that the resulting models are much more compact than the models presented in [16]. We further introduced patterns of the novel constructs such as the `<forEach>` activity and termination handlers. For computer-aided verification, we implemented a low-level version of the semantics in our compiler BPEL2oWFN which is used in several case studies [13, 14]. We only presented a few patterns of the semantics in this paper.

As WS-BPEL is only defined informally, the correctness of the presented patterns can not be proven. However, we validated the Petri net semantics in various case studies. We translated real-life WS-BPEL processes into Petri net models and analyzed the internal (cf. [16]) and interaction (cf. [13, 24, 14]) behavior as well as the interplay of several WS-BPEL processes in choreographies (cf. [27]).

9.1 Future Work

The presented semantics is feature-complete; that is, it models all data and control flow aspect of a WS-BPEL process.[16] However, the instantiation of process instances and message correlation is not covered by the semantics. In future work, we want to add a instantiation mechanism to the semantics, allowing to analyze the complete lifecycle of process instances.

As WS-BPEL is just a part of the web service protocol stack (cf. [32]), the underlying layers such as WSDL, WS-Policy, etc. may also influence the behavior of the WS-BPEL process under consideration. In ongoing research, we plan to incorporate the information derived from these layers (e. g., fault types and policy constraints) to our semantics to *refine* the resulting models and allow for more faithful analysis results.

Finally, static analysis techniques (see [22] for an overview) such as predicate abstraction might help to avoid the complete unfolding of a data domain. Instead, "relevant" intervals are extracted from the expressions used in the process. For instance, for a data-driven decision "$salary \geq 5,000$?", the exact value of the variable *salary* does not have to be modeled, but only whether it is greater or equal to $5,000$. Thus, the original integer domain can be replaced by a Boolean domain. Moser et al. report in [17] first results applying data abstraction techniques to WS-BPEL that could be incorporate to BPEL2oWFN.

9.2 Acknowledgements

The author wishes to thank Jeffrey Bedard, Sylvain Beucler, Maarten Boote, Antoon van Breda a.k.a. Jaded Hobo, Jan Bretschneider, Dirk Fahland, Carsten Frenkler, Christian Gierds, Thomas Heidinger, Mario Karrenbrock, Dieter König, Oliver Kopp, Axel Martens, Harald Meyer, Simon Moser, Chun Ouyang, Michael Piefel, Dennis Reinert, Arnab Roy, Christian Stahl, Eric Verbeek, Jan Martijn van der Werf, Ernesto Santana-Diaz, and Martin Znamirowski.

This work is funded by the German Federal Ministry of Education and Research (project Tools4BPEL, project number 01ISE08).

References

1. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In Hee, K.M.v., Reisig, W., Wolf, K., eds.: Proceedings of the Workshop on Formal Ap-

[16] We do not model aspects that are not part of the WS-BPEL language itself such as XPath or XSLT.

proaches to Business Processes and Web Services (FABPWS'07), University of Polandaise (2007) 21–35
2. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In Dumas, M., Heckel, R., eds.: Web Services and Formal Methods, Forth International Workshop, WS-FM 2007 Brisbane, Australia, September 28-29, 2007, Proceedings. Lecture Notes in Computer Science, Springer-Verlag (2007) accepted.
3. Stahl, C.: A Petri net semantics for BPEL. Techn. Report 188, Humboldt-Universität zu Berlin (2005)
4. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., Rijn, D.v.d., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0. OASIS standard, Organization for the Advancement of Structured Information Standards (OASIS) (2007)
5. 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)
6. Breugel, F.v., Koshkina, M.: Models and verification of BPEL. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf` (2006)
7. Aalst, W.M.P.v.d., Hee, K.M.v.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)
8. Reisig, W.: Petri Nets. Springer-Verlag (1985)
9. 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
10. Martens, A.: Analyzing Web service based business processes. In Cerioli, M., ed.: Proceedings of Intl. Conference on Fundamental Approaches to Software Engineering (FASE'05). Volume 3442 of Lecture Notes in Computer Science., Springer-Verlag (2005) 19–33
11. 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), EMISA, Bonner Köllen Verlag (2005) 236–249
12. Girault, C., Valk, R., eds.: Petri Nets for System Engineering – A Guide to Modeling Verification and Applications. Springer-Verlag (2002)
13. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In Dustdar, S., Fiadeiro, J.L., Sheth, A., eds.: Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5–7, 2006, Proceedings. Volume 4102 of Lecture Notes in Computer Science., Springer-Verlag (2006) 17–32
14. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data Knowl. Eng. (2007)
15. Reisig, W.: Petri nets and algebraic specifications. Theor. Comput. Sci. **80**(1) (1991) 1–34
16. 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., Springer-Verlag (2005) 220–235
17. Moser, S., Martens, A., Gorlach, K., Amme, W., Godlinski, A.: Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. In: 2007 IEEE International Conference on Services Computing (SCC 2007), 9-13 July 2007, Salt Lake City, Utah, USA, IEEE Computer Society (2007) 98–105

18. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset nets between decidability and undecidability. In Larsen, K.G., Skyum, S., Winskel, G., eds.: Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings. Volume 1443 of Lecture Notes in Computer Science., Springer-Verlag (1998) 103–115
19. Leymann, F., Roller, D., Schmidt, M.T.: Web services and business process management. IBM Systems Journal **41**(2) (2002)
20. Vautherin, J.: Parallel systems speciticications 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
21. 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., Springer-Verlag (2005) 484–489
22. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. 2nd edn. Springer-Verlag (2005)
23. Murata, T.: Petri nets: Properties, analysis and applications. Proc. of the IEEE **77**(4) (1989) 541–580
24. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In Kleijn, J., Yakovlev, A., eds.: 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25–29, 2007, Proceedings. Volume 4546 of Lecture Notes in Computer Science., Springer-Verlag (2007) 321–341
25. Schmidt, K.: LoLA: A low level analyser. In Nielsen, M., Simpson, D., eds.: Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000). Volume 1825 of Lecture Notes in Computer Science., Springer-Verlag (2000) 465–474
26. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for modeling choreographies. In: Proceedings of the IEEE 2007 International Conference on Web Services (ICWS 2007), Salt Lake City, Utah, USA, July 2007, IEEE Computer Society (2007) 1–8
27. Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and participant synthesis. In Dumas, M., Heckel, R., eds.: Web Services and Formal Methods, Forth International Workshop, WS-FM 2007 Brisbane, Australia, September 28-29, 2007, Proceedings. Lecture Notes in Computer Science, Springer-Verlag (2007)
28. Ouyang, C., Aalst, W.M.P.v.d., Breutel, S., Hofstede, A.H.M.t.: Formal semantics and analysis of control flow in WS-BPEL. BPM Center Report BPM-05-15, Business Process Management (BPM) Center (2005)
29. Ouyang, C., Aalst, W.M.P.v.d., Breutel, S., Hofstede, A.H.M.t.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. **67**(2-3) (2007) 162–198
30. Lohmann, N., Verbeek, E., Ouyang, C., Stahl, C., Aalst, W.M.P.v.d.: Comparing and evaluating Petri net semantics for BPEL. Computer science report, Technische Universiteit Eindhoven, The Netherlands (2007)
31. Hinz, S.: Implementierung einer Petrinetz-Semantik für BPEL. Diplomarbeit, Humboldt-Universität zu Berlin (2005) in German.

32. Wilkes, L.: The Web services protocol stack. Technical report, CBDI Web Services Roadmap (2005) http://roadmap.cbdiforum.com/reports/protocols.