A Feature-Complete Petri Net Semantics for WS-BPEL 2.0

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.

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 [1]. This specification is much more detailed and more precise compared to the predecessor specification [2]. 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 [3] 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 [4]. In the area of workflows, it has been shown that Petri nets [5] are appropriate both for modeling and analysis. More specifically, with Petri nets several elegant technologies such as the theory of workflow nets [6], a theory of controllability [7, 8], a long list of verification techniques, and tools (see [9] for an overview) become directly applicable.

In this paper, we present an extension of the Petri net semantics of [10] (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. Admittedly, we can only present a few aspects of this new semantics and refer to [11] where the complete semantics formalizing all activities of WS-BPEL.

Niels Lohmann. **A feature-complete Petri net semantics for WS-BPEL 2.0**. In Marlon Dumas and Reiko Heckel, editors, *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, September 2007. Note: accepted.

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 we extend in this paper. Then, in Sect. 3, we show how several aspects of the semantics can be simplified. Section 4 is devoted to the presentation of patterns for some novel activities and constructs of WS-BPEL 2.0. Finally, Sect. 5 concludes the paper, summarizes related work, and gives directions for future work.

2 Background

2.1 WS-BPEL

The *Web Services Business Process Execution Language* (WS-BPEL) [1], is a language for describing the behavior of business processes based on Web services. For the specification of a business process, WS-BPEL provides *activities* and distinguishes between *basic 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 [6] by introducing an interface for asynchronous message 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 [12, 13].

To model data flow and data manipulation, Petri nets can be extended to algebraic high-level nets [14]. Similarly, open workflow nets can be canonically extended to high-level open workflow nets (HL-oWFNs).

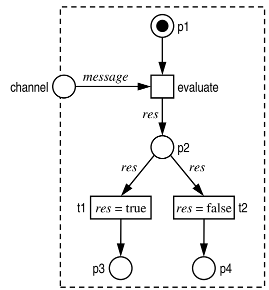


Fig. 1. A high-level oWFN.

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

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

[1] The semantics of [10] 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.

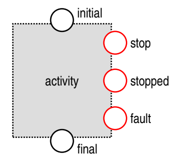


Fig. 2. The interface places of an activity: initial, final, stop, stopped, and fault. Marking the initial place starts an activity. Upon faultless completion of the activity, the final place is marked. The places stop and stopped model the termination of activities. Faults are signaled by marking the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

infinite, abstract (low-level) patterns are implemented in the respective compilers BPEL2PN [15] and BPEL2oWFN [11]. To simplify the presentation of the patterns, we use several graphical conventions, depicted in Fig. 3(a) and 3(b).

(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.

3 Simplifying Existing Patterns

The original semantics [10] was designed to formalize BPEL4WS 1.1 rather than to create compact models that are necessary for computer-aided verification. Some patterns were easy to understand yet made use of quite "expensive" constructs such as reset arcs [16]. We improved these patterns and replaced them by less intuitive patterns with simpler structure. In particular, the setting of control links and the complex interplay of the fault, compensation, event, and (the newly introduced) termination handlers was condensed.

3.1 Links and Dead-path-elimination

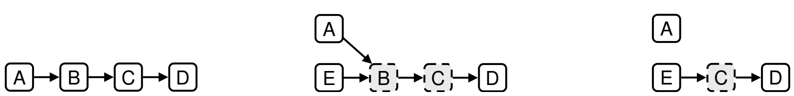
Activities embedded in a `<flow>` activity are executed concurrently. However, it is possible to add control dependencies by the help of *links*. A link is a directed connection between a *source activity* and a *target activity*. After the source activity is executed, the link is set to true, allowing 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[2] — 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) and can be enabled for each target activity.

```
<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. 4. An example for links and dead-path-elimination. `<activity>` is a placeholder for any WS-BPEL activity.

As an example, consider the `<flow>` of Fig. 4. 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. 5(a). Firstly, A is executed and sets link AtoB to true, then B is executed and sets link BtoC. Finally, C and D are executed sequentially.



(a) condition true (b) condition false ([10]) (c) condition false (new)

Fig. 5. Possible executions of the activities of the example in Fig. 4. 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

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

outgoing link, BtoC, is set to false and C is also skipped. Finally, D is executed. This yields the execution order of Fig. 5(b). These two runs are correctly modeled by the semantics of [10] 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 [1]: an *overapproximation* of the process's exact behavior is modeled. In the example, activity B is not skipped explicitly, but its outgoing link, BtoC, is set to false *directly* when E is selected. This yields the execution order of Fig. 5(c). Compared to the semantics of [10], 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 [13, 11] for details). Figure 6 depicts another example 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.

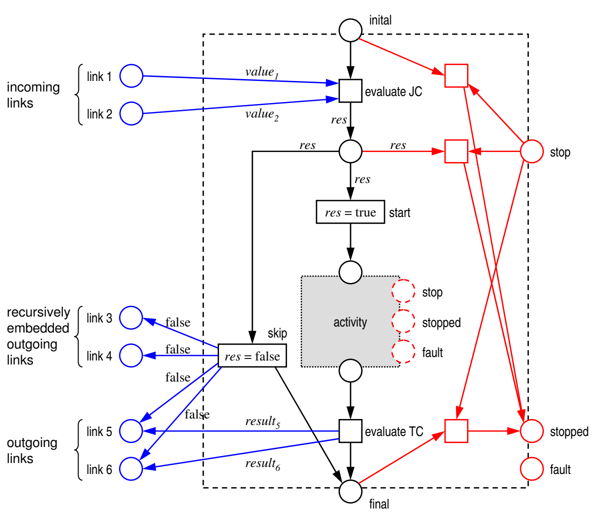


Fig. 6. Wrapper pattern of an activity that is source and target of links. Transition evaluate_JC evaluates the join condition. If the *res*ult is true, the embedded activity is started. Upon completion of the this activity, transition evaluate_TC evaluates the transition condition and sets the outgoing links accordingly. If, however, the join condition evaluates to false, transition skip does not only set all directly or recursively enclosed outgoing links to false.

3.2 Fault Handling and Termination of Scopes

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 [10]. 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 [16].[3] 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 [10] could not be nested inside repeatable constructs such as `<while>` activities or event handlers[4]. 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. 7. It consists of four parts modeling the different aspects of the scope.

– 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)

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

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

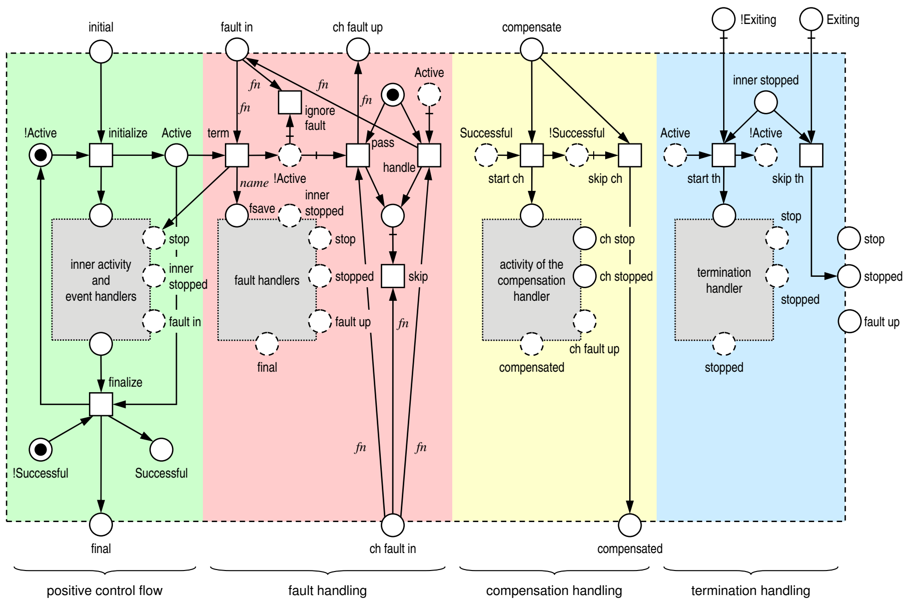


Fig. 7. The pattern for the `<scope>` activity. It consists of four parts modeling the different aspects of the scope: the positive control flow consisting of the embedded activity and the event handlers, the fault handlers, the compensation handler and the termination handler.

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.[5] If, however, the fault could not be handled or the fault 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.

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

– 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.

The new scope pattern is more compact as the pattern from the semantics of [10]. 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. 8). 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.

3.3 Comparison

To compare the new patterns for scopes and dead-path-elimination with the old patterns, we investigated an example process described in [15]. This process models a small online shop consisting of 3 scopes, 2 links, and 46 activities. The authors of [15] translated it using the old Petri net semantics and report a net size of 410 places and 1069 transitions, and a state space consisting of 6,261,648 states (443,218 states using partial order reduction). We translated this process with our compiler BPEL2oWFN[6] which implements the new semantics. Using the new patterns, the resulting net has 242 places and 397 transitions. The smaller net structure also results in a smaller state space consisting of 304,007 states (74,812 states using partial order reduction).

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.

[6] Available at `http://www.gnu.org/software/bpel2owfn`.

4 Modeling WS-BPEL's New Features

WS-BPEL 2.0 [1] clarified several scenarios and added or renamed a couple of activities. While most of the semantical details where already covered by the semantics of [10], the other changes are mainly of syntactic nature and can be modeled straightforwardly. For example, the new `<repeatUntil>` activity can be easily modeled by a `<while>` activity with adjusted loop condition. As such resulting patterns are not very surprising, we focus on those features that are entirely novel. In particular, the parallel `<forEach>` activity with its complex completion and cancelation behavior cannot be simulated with existing features. Furthermore, a termination handler now allows to execute an arbitrary activity when a scope is forced to terminate. In this section, we present patterns for the `<forEach>` activity and the termination handler and refer to [11] for the complete collection of patterns.

4.1 Modeling the `<forEach>` Activity

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.

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. Instead, we focus on the parallel `<forEach>` activity.

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.

Figure 8 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

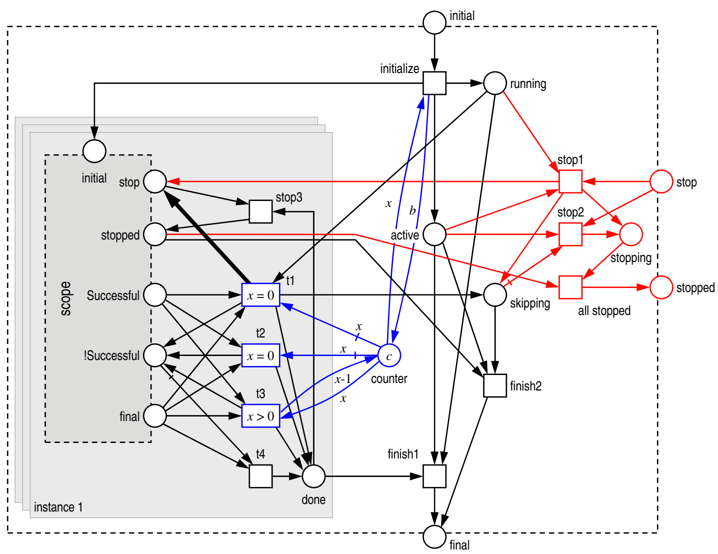


Fig. 8. The pattern for the parallel `<forEach>` activity.

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. 8. 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.[7] 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 presentation of the pattern, we do not depicted the subnet that organizes the compensation of the instance's scopes.

4.2 Modeling Termination Handlers

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.[8]

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. 7, 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. 9.

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

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

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

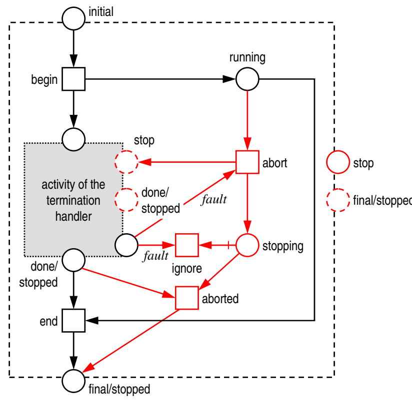


Fig. 9. The pattern for the termination handler.

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 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 [10]. 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 [15]. 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 [12, 13]. We only presented a few patterns of the semantics in this paper. The complete semantics is published in [11].

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. [15]) and interaction (cf. [12, 18, 13]) behavior as well as the interplay of several WS-BPEL processes in choreographies (cf. [19]).

5.1 Related Work

Though many formal semantics for WS-BPEL were proposed (see [3] 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 [20, 21] 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 [2]. However, constructs such as the `<forEach>` activity or termination handlers are not covered by this semantics.

5.2 Future Work

The presented semantics is feature-complete; that is, it models all data and control flow aspect of a WS-BPEL process.[9] 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. [22]), 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.

5.3 Acknowledgements

The author wishes to thank Christian Gierds, Eric Verbeek, Christian Stahl, and Simon Moser for valuable discussions and comments regarding the Petri net semantics. This work is funded by the German Federal Ministry of Education and Research (project Tools4BPEL, project number 01ISE08).

References

1. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS (2007)
2. Andrews, T., et al.: Business Process Execution Language for Web Services, Version 1.1. Technical report, BEA, IBM, Microsoft (2003)
3. Breugel, F., Koshkina, M.: Models and verification of BPEL. `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf` (2006)
4. Aalst, W.M.P.v.d., Hee, K.M.v.: Workflow Management: Models, Methods, and Systems. MIT press, Cambridge, Massachusetts (2002)
5. Reisig, W.: Petri Nets. Springer-Verlag (1985)
6. Aalst, W.M.P.v.d.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
7. Martens, A.: Analyzing Web service based business processes. In Cerioli, M., ed.: Proceedings of Intl. Conference on Fundamental Approaches to Software Engineering (FASE'05). Volume 3442 of Lecture Notes in Computer Science., Springer-Verlag (2005) 19–33

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

8. 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
9. Girault, C., Valk, R., eds.: Petri Nets for System Engineering – A Guide to Modeling Verification and Applications. Springer-Verlag (2002)
10. Stahl, C.: A Petri net semantics for BPEL. Techn. Report 188, Humboldt-Universität zu Berlin (2005)
11. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN. Techn. Report 212, Humboldt-Universität zu Berlin (2007)
12. 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
13. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data Knowl. Eng. (2007)
14. Reisig, W.: Petri nets and algebraic specifications. Theor. Comput. Sci. **80**(1) (1991) 1–34
15. 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
16. 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
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: IEEE International Conference on Services Computing (SCC 2007), IEEE Computer Society (2007) 98–105
18. 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
19. 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)
20. 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
21. 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
22. Wilkes, L.: The Web services protocol stack. Technical report, CBDI Web Services Roadmap (2005) `http://roadmap.cbdiforum.com/reports/protocols`.