A Petri net semantic for BPEL4WS – validation and application

Karsten Schmidt and Christian Stahl

Humboldt–Universität zu Berlin
Institut für Informatik
D–10099 Berlin

Abstract. We translated a small business process into a recently defined Petri net semantic. Then we used the tool LoLA for validating the semantic as well as for proving relevant properties of the particular process.

1 Introduction

The *Business Process Execution Language for Web Services* (*BPEL* for short) [CGK$^+$03] is a language describing a stand alone business process as well as the composition of business processes. It makes a syntax available to model business processes based on Web services. This language is very expressive, but three lacks are known: First understanding BPEL is difficult, because "it offers (too) many overlapping constructs" [WADH02]. Second BPEL builds on IBM's WSFL [Ley01] and Microsoft's XLANG [Tha01] and combines the features of both languages. That is the reason why BPEL has some inconsistencies. Last a mathematically sound semantic for BPEL does not exist. Thus, formal analysis of in BPEL specified business processes is impossible.

BPEL is already used by business process developers. Accordingly it is necessary to find the language's inconsistencies and make tool support available for verifying such processes. The main problem is the lack of a formal BPEL semantic. Most approaches ignore complicated but important concepts of BPEL like compensation and fault handlers.

We have developed a pattern-based Petri net semantic for BPEL [Sta04]. Therefore we can translate every business process specified in BPEL into a Petri net. Such a Petri net can be analyzed by existing tools. In order to validate the semantic and to verify translated BPEL processes we use the model checker LoLA [Sch00a].

2 Petri net semantic for BPEL4WS

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

In the following subsections, we give a glimpse on our semantic, using a basic activity (invoke) and a structured activity (flow) as examples.

2.1 Example of a basic activity

Let us have a more detailed look at the general pattern's design. Figure 1 depicts the pattern for the BPEL's asynchronous invoke construct[1]. Invoke is responsible for sending a request to a partner. In the asynchronous version, it does not wait for a result.

In arc inscriptions of nets shown throughout this paper, a variable with small letter, e.g., `x` symbolizes a single variable and a variable with a capital letter, e.g., `X` symbolizes a tupel of variables.

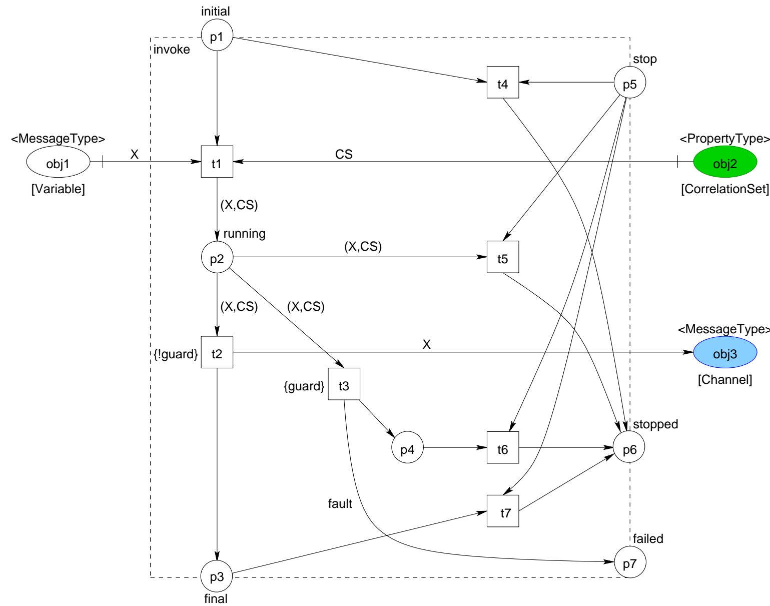

Fig. 1. Pattern for BPEL's asynchronous invoke. When the pattern is activated, it is executed in two steps. First the message is taken from the variable (`obj1`) and the correlation set (`obj2`) is read (`t1`). In the second step this information is analyzed. Either a fault occurs (`t2`) or the message is send (`t3`). In both cases, the pattern is finished.

In general, a pattern is framed by a dashed box. Inside the frame, the structure of the corresponding BPEL construct is modelled. The interface is established by the nodes depicted directly on the frame. Control flows from top to bottom while communication between processes flows horizontally. Outside the frame, there are external objects which relate to the scope, e.g., `obj1`. The label on the top of an object defines its sort whereas the role is defined at the bottom of the object. A sort is a set of tokens lying on and arriving at a place. The object's role is independent of its sort.

The pattern shown in Figure 1 reads a message saved in a variable and either invokes another BPEL process by sending this message or a fault is thrown because of a mangled message or some other error.

Looking at the figure the meaning of place `stop`, `stopped` and `failed` needs to be explained. In BPEL, a process is forced to stop its control flow, e.g., when a fault occurs or activity terminate is activated. However, the specification [CGK+03] does not tell how to do so. Thus we have to make some modelling decisions in our model:

[1] In BPEL, an invoke activity may be configured such that it turns an idle process active. This is not shown in our picture.

The pattern of BPEL's scope is extended by a stop pattern, which has no equivalent construct in BPEL. If a scope needs to be stopped, the stop pattern controls this procedure. Our idea is to remove all tokens from the patterns, embedded in the scope pattern; thus the patterns of BPEL's activities and compensation handler contain a subnet – a so called *stop component*. In the case of Figure 1 the stop component is established by transitions t4 – t7 using the interface stop and stopped.

In order to explain how a stop component works imagine a scope that contains just an asynchronous invoke and the latter throws a fault. This leads to place failed being marked – the token is an object that consists of the fault's name. This place is joined with a place in the stop pattern; thus this pattern gets the control of the scope. First it stops the inner activity of the scope and consequently a token is produced on the asynchronous invoke's stop place. Transition t6 fires and stopped is marked. This place is also joined with a place in the stop pattern. If control flows in a pattern and stop is marked, the transitions of the stop component are in conflict to them. This is in fact unavoidable, due to the involved concurrency. In [Sta04] we proved that using stop components every process can be stopped.

2.2 Example of a structured activity

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

The pattern in Figure 2 can carry a number of n inner activities which are executed concurrently. Such an embedded activity can be any BPEL construct; thus only the interface is visualized and all other information of the pattern are faded out. Therefore only the frame and places initial, final, stop, stopped and if needed

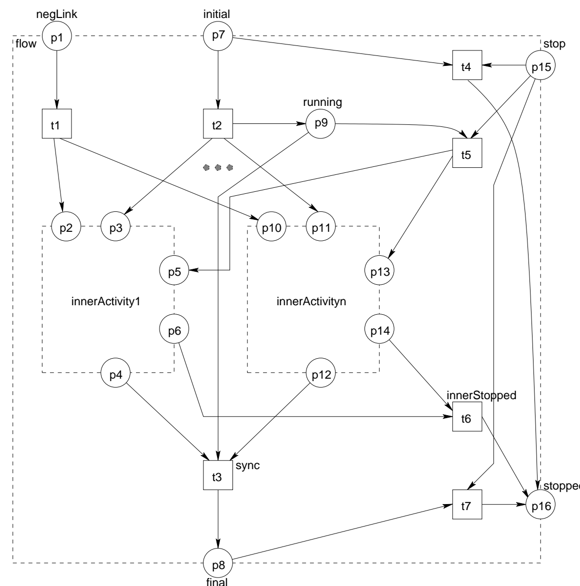

Fig. 2. Pattern for BPEL's flow embeds n inner activities. There are two possible scenarios: Either all inner activities are executed concurrently (t2) and afterwards they are synchronized (t3) or the status of all source links embedded in the flow is set to negative (t1).

negLink are visible (see innerActivity1 in Figure 2). The interface of each embedded pattern is joined with the surrounding flow pattern.

negLink is an abbreviation of negative link. It is an optional place that is only part of a link pattern's or of a structured activity pattern's interface when it embeds at least one activity that is source of a link. With the help of negLink the status of all source links of an inner activity that are not executed anymore is set to negative, e.g., imagine an activity within a branch that is not taken in a switch activity. In other words, negLink is a place for modelling *Dead-Path-Elimination* [LR99]. In Figure 2 we assume that innerActivity1 and innerActivityn contain at least one activity that is source of a link.

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

3 Validation

For validating the semantic, we generated a business process and verified several properties concerning the correct interplay between different patterns. We use the Petri net based model checker LoLA that features powerful reduction techniques like symmetries [Sch00b] partial order reduction using stubborn sets [Sch99] and the sweep-line method [Sch04].

In Figure 3 our example process is depicted – a modification of the purchase order process given in the specification [CGK$^+$03, pp. 14]. A box frames an activity. In the case of a scope (see the box around the synchronous invoke) or the process itself we use a bold frame. Sequential flow is depicted by dashed arcs, whereas concurrent activities are grouped in parallel. Arcs with solid lines symbolizes links.

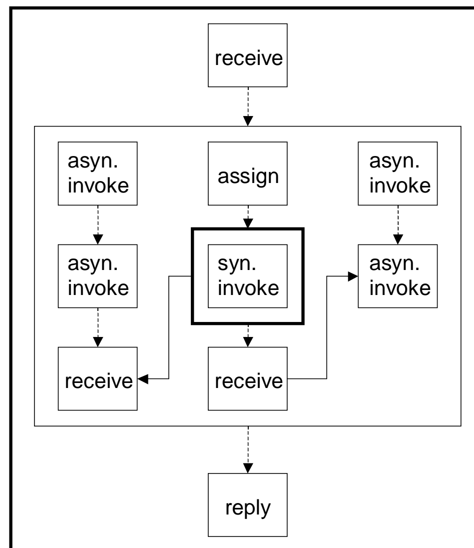

Fig. 3. When the purchase order process receives an order from a customer, it initiates three tasks concurrently: calculating the final price for the order (left sequence), selecting a shipper (middle sequence), and scheduling the production and shipment for the order (right sequence). There are two dependencies between the three tasks, realized by links. After the completion of the tasks the invoice is sent to the customer.

This is a small example, yet most of BPEL's activities including fault handler and links are used. This process is modified by enclosing the synchronous invoke, which can send a fault message within a scope. This helps us validating especially the concept of stop in nested scopes. Furthermore we decided to abstract from data, i.e. messages are modelled as black tokens, because we directed our attention to the control flow.

The Petri net of the example process consists of 158 places and 249 transitions. It was generated manually. The whole state space consists of 9991 states and is calculated in less than a second. LoLA's state space reduction techniques, in particular partial order reduction and the sweep-line method, work well on the net. Applied together, state space is reduced to 1286 states. The results presented that the Petri net of a transformed BPEL process has too many places and transitions to represent it graphically, and given further that the nets behave well w.r.t. reduction techniques, deriving conjectures about the behavior of the net and model checking them turns out to be a reasonable technique for validating the semantic.

We first checked for dead transitions. LoLA found 101 such transitions. Comparing these results to our patterns, we verified that exactly the right transitions are dead: Every pattern covers the whole behavior including all special cases of the respective BPEL construct. Consequently, in the case of such a small and simple process like in Figure 3 its patterns has some overhead not needed for the respective process. For a further use of the semantic, it would therefore be beneficial to reduce the stuff not necessary in a particular context away before doing verification.

Likewise, LoLA found 15 dead places which was expected as well. Next, we computed the 196 terminal states of the system and found them to be expected end states of the system.

We finally verified several temporal properties such as "stop leads to stopped", "the target activity of a link occurs always after the respective source activity" and so on. With the exception of some errors that were due to the manual generation of the net, LoLA confirmed all results. For approving the source/target property of links, LoLA needed between 386 and 4614 states.

4 Verification of a BPEL process

The objective in the previous section was to confirm that the semantic itself is plausible. The results, however, encouraged us to go one step further and show that computer aided verification of a BPEL process, using its Petri net semantic is indeed possible. We were able to verify relevant properties like termination, "the customer will always get an answer", and "he will always get the correct result unless an error occurs". The latter properties are complex temporal logic properties. To date, LoLA has only little support for such properties, so the state space amounted to between 8728 and 8840 states. For termination, 1286 states were computed.

5 Conclusion

This paper summarizes our work of verifying a business process specified in BPEL. We introduced in our pattern-based Petri net semantic of BPEL.

An approach to validate this semantic was presented next. For an example process transformed into a Petri net, we verified several conjectures about the behavior of the patterns. Our example is also used to prove properties of the process, like termination. Furthermore it is shown that such Petri nets have relatively small state space; and state space reduction techniques behave well. Thus use of a model checking tool like LoLA is feasible.

Further work of our research group is an ongoing validation of the semantic by transforming further BPEL processes. In order to handle larger processes, we plan to build a parser.

It should be mentioned that the definition of a Petri net semantic, in connection with a concurrent approach based on ASM [Fah04], enabled us to discover a number of inconsistencies and ambiguities in the informal specification. The results have been incorporated into recent working drafts of the document.

References

[CGK+03] Curbera, Goland, Klein, Leymann, Roller, Thatte, and Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Technical report, BEA Systems, Interantional Business Machines Corporation, Microsoft Corporation, May 2003.

[Fah04] Dirk Fahland. Ein Ansatz zur formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines. Studienarbeit, Humboldt-Universität zu Berlin, July 2004.

[Ley01] Frank Leymann. *WSFL – Web Services Flow Language.* IBM Software Group, Whitepaper, May 2001. `http://ibm.com/webservices/pdf/WSFL.pdf`.

[LR99] F. Leymann and D. Roller. *Production Workflow – Concepts and Techniques.* Prentice Hall, 1999.

[Rei85] W. Reisig. *Petri Nets.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, EATCS Monographs on Theoretical Computer Science Edition, 1985.

[Sch99] Karsten Schmidt. Stubborn set for standard properties. *Proc. 20th Int. Conf. Application and Theory of Petri nets, LNCS 1639*, pages 46–65, 1999.

[Sch00a] Karsten Schmidt. Lola – a low level analyser. In Nielsen, M. and Simpson, D., editors, *International Conference on Application and Theory of Petri Nets*, LNCS 1825, page 465 ff. Springer-Verlag, 2000.

[Sch00b] Karsten Schmidt. How to calculate symmetries of petri nets. *Acta Informatica 36,*, pages 545–590, 2000.

[Sch04] Karsten Schmidt. Automated generation of a progress measure for the sweep-line method. In *Proc. 10th Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2004*, volume 2988 of *LNCS*, pages 192–204. Springer-Verlag, 2004.

[Sta04] Christian Stahl. Transformation von BPEL4WS in Petrinetze. Diplomarbeit, Humboldt-Universität zu Berlin, April 2004.

[Tha01] Satish Thatte. *XLANG – Web Services for Business Process Design.* Microsoft Corporation, Initial Public Draft, May 2001. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c`.

[WADH02] Petia Wohed, Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede. Pattern based Analysis of BPEL4WS, 2002.