

Comparing and Evaluating Petri Net Semantics for BPEL

Niels Lohmann*

Universität Rostock, Institut für Informatik 18051 Rostock, Germany

E-mail: niels.lohmann@uni-rostock.de

*Corresponding author

Eric Verbeek

Technische Universiteit Eindhoven

Department of Mathematics and Computer Science

P.O. Box 513, 5600 MB Eindhoven, The Netherlands

E-mail: H.M.W.Verbeek@tue.nl

Chun Ouyang

Queensland University of Technology

Faculty of Information Technology

GPO Box 2434, Brisbane QLD 4001, Australia

E-mail: c.ouyang@qut.edu.au

Christian Stahl

Humboldt-Universität zu Berlin, Institut für Informatik

Unter den Linden 6, 10099 Berlin, Germany

Technische Universiteit Eindhoven

Department of Mathematics and Computer Science

E-mail: stahl@informatik.hu-berlin.de

Abstract: We compare two Petri net semantics for the Web Services Business Process Execution Language (BPEL). The comparison reveals different modelling decisions. These decisions together with their consequences are discussed. We also give an overview of the different properties that can be verified on the resulting models. A case study helps to evaluate the corresponding compilers which transform a BPEL process into a Petri net model.

Keywords: BPEL, Petri net semantics, formal methods, verification

Reference to this paper should be made as follows: Lohmann, N., Verbeek, H.M.W., Ouyang, C., Stahl, C. (2008) ‘Comparing and Evaluating Petri Net Semantics for BPEL’, Int. J. Business Process Integration and Management, Vol. ?, Nos. 1/2/3, pp.??-??.

Biographical notes: Niels Lohmann works since 2007 as research assistant in the group of Karsten Wolf at University of Rostock. Before, he worked in the group of Wolfgang Reisig at Humboldt-Universität zu Berlin where he received his Masters in 2005. His research interests include diagnosis and correction of service and choreography models and formal semantics of service languages.

Eric Verbeek is a scientific engineer at the section of Information Systems of the department of Technology Management at the Technische Universiteit Eindhoven, where he also received his Ph.D. His research interests include workflow management, business process management, and process verification.

Dr. Chun Ouyang received her PhD in Computer Systems Engineering from the University of South Australia in 2004. She currently works as a post-doctoral research fellow at the Faculty of Information Technology in Queensland University of Technology, Australia. Her research interests are in the areas of workflow management, process modelling and analysis, Web services, Petri nets, and formal specification and verification.

Christian Stahl studied Computer Science at Humboldt-Universität zu Berlin, Germany. He received a Masters in 2004. Since then, he is working as a Research Assistant in the group of Wolfgang Reisig in Berlin, and since 2006, also in the group of Kees van Hee and Wil van der

1 Introduction

The Web Services Business Process Execution Language (BPEL) (Alves et al., 2007) is emerging as the de facto standard for implementing business processes on top of the Web service technology. However, BPEL lacks formal semantics as it has been defined informally. Especially its predecessor specification, BPEL 1.1 (Andrews et al., 2003), contained many ambiguities. Fortunately, the current BPEL 2.0 specification (Alves et al., 2007)¹ is more precise (many of the BPEL 1.1 ambiguities have been resolved in BPEL 2.0) although it is still informally defined. Over the years, several attempts have been made to formalise BPEL, using among others Petri nets, process algebra, abstract state machines, or state machines. For example, Breugel and Koshkina (2006) survey more than 100 papers. However, there are only three *feature-complete* semantics; that is, semantics that cover both the standard and the exceptional behaviour of BPEL. These semantics are two Petri net semantics (Lohmann, 2007; Ouyang et al., 2007) and a semantics based on abstract state machines (Fahland, 2005).

The language constructs found in BPEL, especially those related to control flow, are close to those found in workflow definition languages (Aalst and Hee, 2002). In the area of workflows, it has been shown that Petri nets (Reisig, 1985) are appropriate both for modelling and analysis. More specifically, with Petri nets several elegant technologies such as the theory of workflow nets (Aalst, 1998), a theory of controllability (Martens, 2005; Schmidt, 2005; Wolf, 2008), a long list of verification techniques, and tools (see (Girault and Valk, 2002) for an overview) become directly applicable.

In this paper, we restrict ourselves to compare the two feature-complete Petri net semantics for BPEL, because they provide—in contrast to the abstract state machine semantics—broad tool support. One semantics is developed by the Theory of Programming Group at the Humboldt-Universität zu Berlin (HUB) (Hinz et al., 2005) and later refined by the Theory of Programming Languages and Programming Group at University of Rostock (Lohmann, 2008), the other mainly by the BPM Group at the Queensland University of Technology (QUT) (Ouyang et al., 2007), which we hereafter simply refer to as HUB and QUT semantics. Each of the groups has developed tools to support the translation and the analysis of BPEL processes.

The goal of this paper is twofold. On the one hand, we compare both semantics. We reveal their differences and discuss their consequences with respect to the model and the verification. On the other hand, we also evaluate the compilers which translate a BPEL process into a Petri net model according to the respective semantics. By help of a case study, we compare the sizes of the nets and their state

spaces. This helps to identify the impact on the complexity of the analysis resulting from the different modelling approaches and decisions. In addition, we also give an overview what properties can be analysed on the models. The result can be used to guide future tool integration for better performance.

The remainder of the paper is organised as follows. Section 2 gives a brief overview of BPEL. Next, Sect. 3 discusses the basics of both QUT and HUB semantics. Then, a number of differences between the BPEL semantics of the two approaches are highlighted in Sect. 4. Section 5 discusses to what extent both approaches allow for verification of the original BPEL processes. In Sect. 6, we list validation results and present several case studies. Finally, Sect. 7 concludes and outlines future work.

2 Overview of BPEL

BPEL (Alves et al., 2007) is a language for describing the behaviour of business processes based on web services. Such a business process can be described either as an *executable business process* or as an *abstract business protocol*. In this paper, we focus on the executable business processes, which model the behaviour and the interface of some *partner* (a participant, a web service) in some business interaction. In contrast, an abstract business protocol only models the interface and the message exchange of this partner, while the actual behaviour remains hidden. Throughout this paper we use the term *BPEL process* instead of “executable business process specified in BPEL”. To execute a BPEL process means to create a *running instance* of this process.

2.1 Activities

For the specification of the internal behaviour of a business process, BPEL provides two kinds of *activities*: *basic* activities and *structured* activities. A basic activity models an elementary action in the process, whereas a structured activity defines some causal order between other activities. Note that structured activities can be nested, and that the activity structure in the process resembles a tree, where the process itself is its root and the basic activities are its leaves. As such, any structured activity contains a number of *child* activities.

Examples of elementary actions are: Sending a message to another partner (<invoke> or <reply>); Receiving a message from another partner (<receive>); Manipulating data (<assign>); Validating data (<validate>); Waiting for some time (<wait>); Doing nothing (<empty>); Throwing a fault (<throw> or <rethrow>); Ending the running instance (<exit>). Examples of structured activities are: The child activities have to be performed in the given order (<sequence>); The child activities can be performed in parallel (<flow>); Depending on some condition, one of the child activities has to be performed while the others have to be skipped (<if>); Depending on some message of

¹We use the term “BPEL” if we do not need to distinguish between the 1.1 and the 2.0 specification.

the environment or the lack thereof (time-out), one of the child activities has to be performed while the others have to be skipped (`<pick>`); As long as some condition holds the (only) child activity has to be executed (`<while>`); Until some condition holds the (only) child activity has to be executed (`<repeatUntil>`); For every element of some set, the (only) child activity has to be executed (`<forEach>`).

2.2 Scope activity

The `<scope>` activity is BPEL's most important structured activity. It provides fault, event, and compensation handling. Scopes are hierarchically ordered: every scope has a *parent scope* and an arbitrary number of *child scopes*. The `<process>` activity is the root scope. The scope hierarchy plays an important role in fault propagation and compensation.

Fault handling allows for reaction on faults that may occur during the execution of a BPEL process. There are many sources of runtime 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 (i.e. the execution of the scope that encloses the faulty activity) has to stop. 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 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 entire 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.

2.3 Links and Dead-Path-elimination

Another important concept in BPEL are *links*. A link can be used to define an additional causal order between two concurrent activities embedded in some `<flow>`. A link has a *source* activity and a *target* activity. The source may specify a Boolean expression, which evaluates to the *status* of that link. The target may also specify a Boolean expression (its join condition) which evaluates the status of all incoming links. The target activity will only be performed if its join condition evaluates to true, otherwise it will get skipped, in which case the status of all outgoing links will be set to false.

BPEL provides dead-path-elimination (DPE) (Leymann and Roller, 2000) to skip those parts of the process that cannot be performed any more. If some activity cannot be performed any more, then either its join condition evaluates false, or it is skipped as a result of some `<if>` or `<pick>`. In case of the former, the activity has become dead as a result of some link path, whereas in case of the

latter, it has become dead as a result of some structure path. DPE transfers from structure paths to link paths: If some path was not chosen in a `<pick>`, then all outgoing links for that path will be set to false, which starts the DPE for these skip paths. However, DPE does not transfer from skip paths to structure paths: If in a `<sequence>` the join condition of some child activity evaluates to false, then this activity will get skipped, but the next activity might still be performed. As such, there is a difference between a `<sequence>` of two activities and a `<flow>` with the same two activities with a link from the first to the second. In the remainder of this paper, to distinguish between these two modes of (structure paths vs. link paths) DPE, we use the terms dead-structure-path-elimination (DSPE) and dead-link-path-elimination (DLPE).

3 Semantics Basics

In both HUB and QUT groups, the main motivation to develop a formal semantics for BPEL is to detect inconsistencies in the BPEL specification on the one hand, and to formally analyse BPEL processes using techniques of computer-aided verification on the other hand. Furthermore, both groups translate real-life BPEL processes into Petri nets to validate their (Petri net-based) verification techniques for analysing service behaviour (see Sect. 6 for details). Besides BPEL-specific analysis goals, the QUT semantics is built for analysing soundness (Aalst, 1998), an established correctness property of workflows. In contrast, the HUB semantics is built for analysing the communication behaviour of BPEL processes (Lohmann et al., 2006).

Both semantics follow an hierarchical approach. The translation is guided by the tree structure of the BPEL process. Each activity in the tree is translated separately into a Petri net. Such a net forms a *pattern* of the respective BPEL activity. Each pattern has an *interface* for connecting it with other patterns as is done with BPEL activities. Also, patterns capturing BPEL's structured activities may carry any number of inner patterns as its equivalent in BPEL can do. Overall, the collection of patterns forms the *Petri net semantics* for BPEL. It is worth mentioning that both HUB and QUT semantics are *feature complete*, covering the standard and exceptional behaviour of BPEL, and both semantics are *formal*, meaning that they are suitable for computer-aided verification.

The HUB group developed a Petri net semantics for BPEL 1.1 (Hinz et al., 2005). This semantics was later enhanced (Lohmann, 2008, 2007) to also cover BPEL 2.0. The QUT group developed a semantics which is restricted to modelling the activities and constructs of BPEL 1.1 (Ouyang et al., 2007) based on the semantical description given by the (more precise) BPEL 2.0 specification.

Despite the above commonalities, one major difference between the two semantics is that they are specified at different levels of abstraction. The QUT semantics is re-

stricted to the control flow of a BPEL process; that is, it abstracts from data (i. e. no BPEL variables are modelled) and message exchange (i. e. the sending and receiving of messages is not explicitly modelled). The resulting semantics is defined as a low-level Petri net, where the control flow is modelled by indistinguishable tokens and a data dependent branch is modelled by a nondeterministic choice. Consequently, the QUT semantics leads to a direct implementation based on low-level Petri nets. In contrast, the HUB semantics is defined using a high-level Petri net. It allows on the one hand to model not only control flow but also data information (such as data values, message content, and Boolean expressions of the link semantics), while on the other hand, it is possible to apply data abstraction techniques during the translation process (Lohmann et al., 2008b). This way, in the HUB’s implementation, a data abstraction is performed, and as a result, it yields a low-level Petri net as in the QUT’s implementation. We focus on the compilers BPEL2PNML (QUT semantics) and BPEL2oWFN (HUB semantics) in Sect. 5.1.

In contrast to the QUT semantics, the HUB semantics models message exchange explicitly via a message interface. This results in the next difference of the two approaches. In the QUT semantics, a model of a BPEL process is a *workflow net* (WFN) (Aalst, 1998). In contrast, due to the definition of a message interface, the resulting model of a BPEL process in the HUB semantics has a structure of an *open workflow net* (oWFN) (Massuthe et al., 2005). oWFNs are a generalised form of WFNs. As a substantial difference, in an oWFN the interface of a service is explicitly represented as sets of input and output places. As it can be seen later on, the different models allow for the analysis of different properties. The main motivation in the QUT group to have a WFN is to apply existing verification techniques for WFNs, such as the analysis of soundness which is implemented in the tool Woflan (Verbeek et al., 2001). In the HUB group, the explicit representation of the interface is essential for analysing the communication behaviour of BPEL processes using the tool Fiona (Lohmann et al., 2006).

4 Different Modelling Concepts

In this section, we compare the modelling of three important BPEL constructs in the QUT semantics and in the HUB semantics:

1. basic activities,
2. links and DPE, and
3. scopes and fault handling.

For each construct, we first discuss its modelling details in each semantics, and then review the different modelling decisions made in the two semantics and if applicable, their impacts to the resulting behaviour of the construct.

4.1 Basic activities

4.1.1 QUT

Each basic activity is considered as an atomic action, for example, a `<receive>` activity performs a receive action, and an `<invoke>` performs a send action. Due to the abstraction from data and message exchanges, all basic activities share the general pattern shown in Fig. 1. For normal processing of an activity X , place r_x models an initial state when it is ready to start X , and place f_x indicates a final state when X has finished its execution. The transition labelled X models the action to be performed. It has an input place s_x capturing the state when X has started, and an output place c_x for the state when X is completed. Two unlabelled transitions (drawn as solid bars) model internal actions for checking pre-conditions or evaluating post-conditions for activity X . The skip path of X corresponds to the DSPE of X .

The rest of the pattern in Fig. 1 is modelled to capture the fact that activity X can only be executed if none of its enclosing scopes (Q_1 to Q_n) has faulted. The pattern of each scope has two places `to_continue` and `to_stop`. Place `to_continue` is marked as long as the scope is not faulted, otherwise place `to_stop` will be marked. As a result, the transition labelled X has read arcs (depicted as arcs with two arrowheads) to the `to_continue` place of every enclosing scope. Assume that one of these enclosing scopes Q_i gets faulted and has to stop before activity X can occur. The stop signal will be propagated through the hierarchy of the enclosed scopes of Q_i in such a way that, for each active scope, its `to_stop` place is marked. As a result, once the `to_stop` place of Q_1 (the immediately enclosing scope of X) is marked, the `bypass` transition will fire, indicating that activity X gets bypassed.

4.1.2 HUB

Each pattern of an activity has five interface places: initial, final, stop, stopped, and fault. The places initial and final correspond to the places r_x and f_x of the QUT semantics, respectively. Marking the initial place starts an activity.

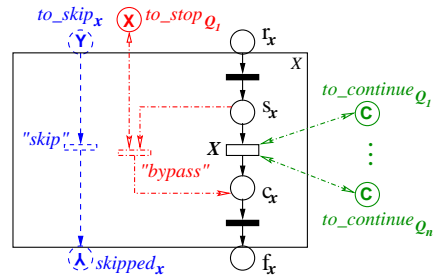


Figure 1: Pattern of a basic activity X . The structure of the pattern can be divided into three parts: one (drawn in solid lines) models the normal processing of X , one (drawn in dashed lines) models the DSPE of X , and one (drawn in dot-dashed lines) models whether to execute or to bypass X based on the status of all the enclosing scopes (Q_1 to Q_n) of X .

Upon faultless completion of the activity, the final place is marked. The places `stop` and `stopped` model the termination of activities. Faults thrown by the activity are signalled by marking the `fault` place.

In contrast to the QUT semantics, each BPEL activity is specified by a different high-level Petri net pattern, modelling both control-flow and data aspects of that activity, including data values, correlation sets, and messages. In addition, the HUB group implements several low-level versions of this high-level pattern which are used for computer-aided verification. The low-level patterns differ in their level of detail, and are chosen by the compiler BPEL2oWFN according to the verification goal. Two patterns for a `<receive>` activity with different degree of abstraction are depicted in Fig. 2.

Furthermore, the HUB semantics does not explicitly model the DSPE of the activity, which will be explained later on.

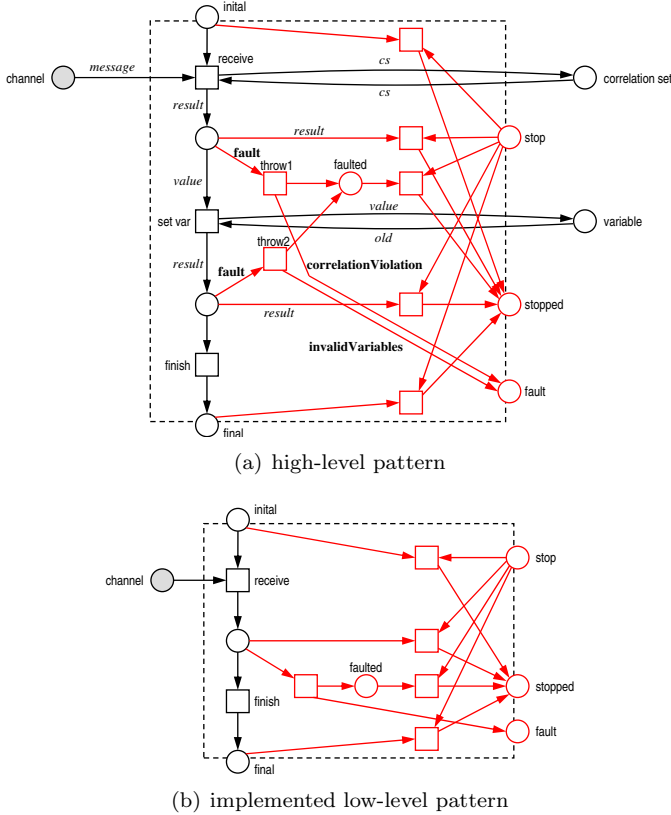


Figure 2: Patterns for a `<receive>` activity. The detailed behaviour is modelled with a high-level pattern (a). The message `channel` is explicitly modelled. When a `message` is received, its `correlation set` is read. Subsequently, the message is stored in a `variable`. During the execution, two faults can occur: the received message may not match the correlation set or the received message type may not match the variable type (transition `throw1`, `throw2`). In these cases, the respective BPEL standard fault is thrown on place `fault`. A simplified low-level pattern (b) abstracts from data and does not model variables or correlation sets.

4.1.3 Conclusion

The QUT semantics abstracts from data and actual communication, whereas the HUB semantics explicitly specifies them in the model. As a result, in the QUT approach all basic activities share the same general pattern, whereas the HUB semantics has different patterns for different activities.

4.2 Links and Dead-Path-Elimination

4.2.1 QUT

Each link is modelled by two places named `lst` and `lsf` capturing respectively the *true* and *false* status of the link. Figure 3 shows the pattern of a basic activity with links. Places named `tc` are used to hold the transition condition for each of the outgoing links. Since transition conditions are not explicitly specified in the pattern, their Boolean evaluation is modelled nondeterministically upon firing either of the two transitions that follow a `tc` place.

DLPE Places `jct` and `jcf` capture the *true* and *false* result of the join condition evaluation, respectively.

If the join condition evaluates to true, activity X starts. Otherwise, transition `sjf` can fire. Then, activity X is not executed and will end in the finished state (f_x) as if it was completed. This ensures that the processing of any following activity can still continue, thus capturing the semantics of dead-path-elimination.

As explained before, DLPE does not transfer to DSPE. Therefore, after having completed the DLPE, activity X

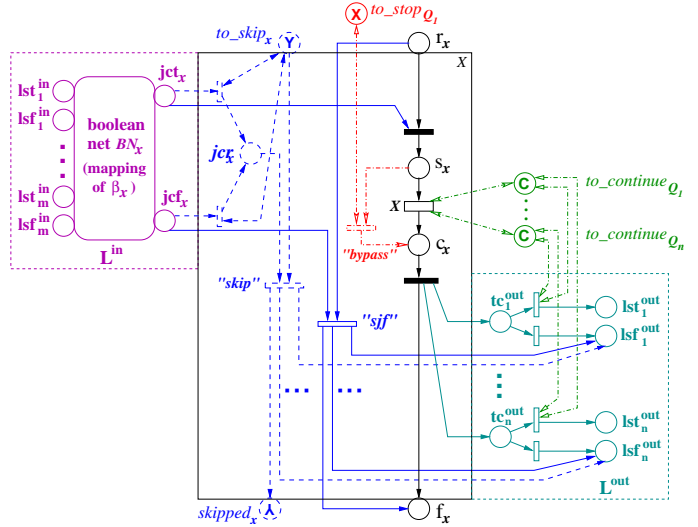


Figure 3: Pattern of a basic activity X with incoming links $L^{\text{in}} = \{l_1^{\text{in}}, \dots, l_m^{\text{in}}\}$ and outgoing links $L^{\text{out}} = \{l_1^{\text{out}}, \dots, l_n^{\text{out}}\}$. The subnet enclosed in the box labelled L^{in} specifies the mapping of incoming links and the join condition evaluation at X , and the one labelled L^{out} specifies the mapping of outgoing links and their transition condition evaluation. A join condition expression (e.g. $\beta_x(l_1^{\text{in}}, \dots, l_m^{\text{in}})$) is mapped onto a Boolean net (BN_x), which takes the status of all incoming links and produces an evaluation result.

is signalled to have completed as usual.

DSPE If activity X has to be skipped, the join condition is evaluated. The evaluation result will not affect the skipping behaviour: the `skip` transition—like transition `sjf`—will set all outgoing links to false. As a result, when skipping an activity, the outgoing links will be set to false (DSPE does transfer to DLPE), but only after all incoming links have been resolved.

4.2.2 HUB

In the HUB semantics, each link is modelled by a coloured place with Boolean type.² The status of the link can then be modelled by a token *true* or *false*. If the place is not marked, the link status is unknown. Similar to the QUT semantics, we model the setting and evaluation of links with patterns that wrap the corresponding source or target activity. An example of an activity that is both source and target of links is depicted in Fig. 4. As a graphical convention, dashed places indicate a place with the same label existing in the same pattern that has to be merged with the dashed place (e. g. place `stop`).

DLPE When implemented, the join condition is, just like in the QUT semantics, “unfolded” and the transition condition is replaced by nondeterminism. However, instead of recursively skipping activities and setting their outgoing links to false as it is required by the BPEL specification (Alves et al., 2007), we deviate from the specified

²In the implementation, this place is unfolded to two places, `link.true` and `link.false`.

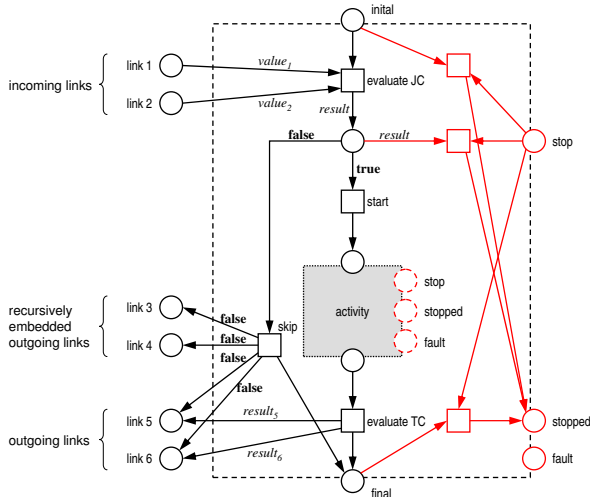


Figure 4: Wrapper pattern of an activity that is source and target of links. Firstly, transition `evaluate_JC` evaluates the join condition from the incoming links yielding a Boolean token with the *result*. If the result is **true**, the embedded activity is started. Upon completion of the inner activity, transition `evaluate_TC` evaluates the transition condition and sets the outgoing links accordingly. If the join condition evaluates to **false**, transition `skip` sets all outgoing links to false.

behaviour and set the status of all links recursively embedded to the skipped activity to false *at once*. This way, the modelling of dead-path-elimination reduces (concurrent) link status propagation, and as a result, the state space may be reduced.

DSPE In a similar way, if some activity was not chosen in a `<pick>` or `<if>`, then all outgoing links of this activity will be set to false by the `<pick>` or `<if>`. As a result, there is no need to explicitly capture the elimination of dead structure paths, as was needed by the QUT semantics.

4.2.3 A DPE Example

We use the example shown in Fig. 5 to demonstrate the different modellings of DPE in the two semantics. In this example, two scenarios are possible, depending on the condition of the `<if>` activity: In case that the condition evaluates to true, we have the execution order shown in Fig. 6(a). Firstly, activity **A** is executed and sets link **AtoB** to true, then **B** is executed and sets link **BtoC** to true, and finally, **C** and **D** are executed sequentially.

In case the condition evaluates to false, **E** is executed and, due to the DSPE, activity **B** will get skipped; that is, **B** gets skipped as soon as **A** has set its link **AtoB**. Then, DSPE sets **B**’s outgoing link, **BtoC**, to false and **C** is skipped due to DLPE. Finally, **D** is executed, as DLPE does not transfer to DSPE. This yields the execution order of Fig. 6(b). Due to a transitive control dependency, **D** is

```

<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>

```

Figure 5: An example for links and dead-path-elimination where `<activity>` is a placeholder for any BPEL activity.

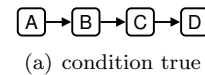


Figure 6: Possible execution orders of the activities of the example in Fig. 5. Skipped activities are depicted with dashed and shaded boxes.

executed *after* A. This execution is exactly modelled by the QUT semantics.

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 HUB semantics differs from the described behaviour of (Alves et al., 2007): an *overapproximation* of the process's exact behaviour is modelled. In the example, activity B is not skipped explicitly, but its outgoing link, BtoC, is set to false when E is selected. This yields the execution order of Fig. 6(c). Compared to the QUT semantics, no control dependency between A and D exists and two *additional* runs are modelled by the HUB semantics, namely A and D being executed concurrently, and D being executed before A. Due to the overapproximation, it is possible that the HUB model contains errors that are not present in the BPEL process. However, static analysis of the BPEL process can help to identify these pseudo-errors. We will come back to this in Sect. 5 where the tools are discussed.

4.2.4 Conclusion

When skipping an entire path (DSPE), or when a join condition evaluates to false (DLPE), the QUT semantics sets every outgoing link to false *after* all causal related incoming links have been resolved, whereas the HUB semantics sets all outgoing links to false immediately, i.e. without waiting for the incoming links to be resolved. While the QUT approach conforms the BPEL specification, the HUB semantics uses an overapproximation which still allows to find any design flaw of a BPEL process.

4.3 Scopes and Fault Handling

4.3.1 QUT

Figure 7 depicts the pattern of a scope Q that has a fault handler. It can be seen that a scope is a special structured activity being attached with four flags. Assume that no exception occurs. Scope Q remains in the status of `to_continue` during its normal performance (i.e. the execution of Q 's main activity X). Upon the completion of activity X , a `snapshot` is preserved for scope Q .

Consider the case when a fault occurs during the normal process of scope Q . The subnet enclosed in the dashed box labelled FH specifies the mapping of a fault handler. Transition e_{fault} models a fault event that may occur when scope Q is active. This fault event may signal any runtime fault or even a fault re-thrown from a child scope of Q . Upon the firing of e_{fault} , the status of Q changes from `to_continue` to `to_stop`. As a result, all active activities in Q need to be stopped, and the occurrence of any other fault is disabled. This ensures that no more than one fault handler can be invoked in the same scope.

A fault handler, once invoked, cannot start its main activity HF (i.e., handling fault) until the main activity of

the scope has been terminated. This results in an intermediate state, as captured by place $invoked_{FH}$, after the occurrence of event e_{fault} but before the execution of activity HF . All the active activities in the scope will be bypassed and finally end up in the finished state (f_x). Hence, the arc from place f_x to the input transition of place r_{HF} ensures that activity HF can be started only if the normal process of scope Q has been terminated.

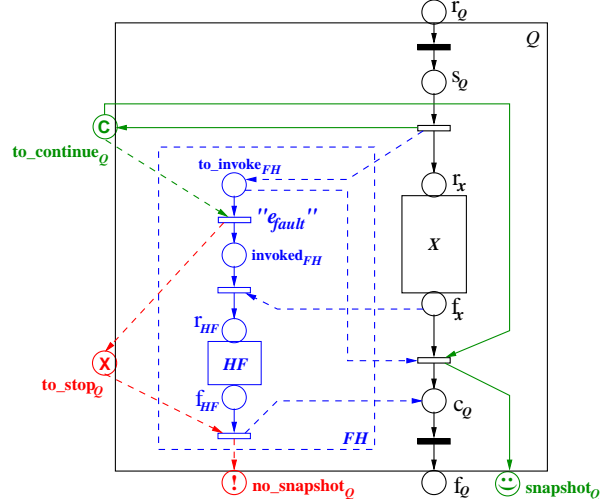


Figure 7: Pattern of a `<scope>` activity with a fault handler. Four flags are defined for a `<scope>`: `to_continue`, indicating that the scope is active and no fault has occurred; `to_stop`, signalling a fault has occurred and all active inner activities within the scope have to stop; `snapshot`, capturing the preserved state of a successfully completed uncompensated scope; and `no_snapshot`, indicating the absence of a scope snapshot.

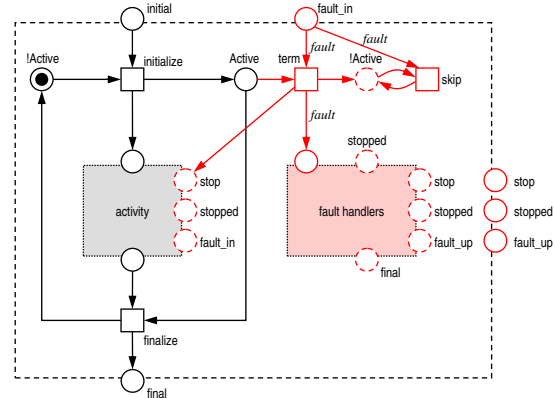


Figure 8: Pattern of a `<scope>` activity without event, termination, or compensation handlers. The `fault` place of the inner activity is merged with place `fault_in` of the scope. When a fault occurs, transition `term` stops the embedded activity and the scope enters state `!Active`. When the embedded activity has stopped, the fault handlers start. If the fault is handled successfully, the scope finishes and `final` is marked. Otherwise, the fault is propagated: `fault_up` is merged with place `fault_in` of the parent scope.

4.3.2 HUB

The pattern of a `<scope>` activity of the HUB semantics is depicted in Fig. 8. Firstly, the scope is initialized and the enclosed activity is started. The scope is then in state `Active`. When the activity completes faultlessly, the scope is completed (transition `finalize`) and is in state `!Active` again. Furthermore, the scope is marked to be ready for compensation, similar to the `snapshot` place of the QUT semantics. This place and the compensation handler is — to increase the legibility of the pattern — not depicted.

However, the embedded activity can throw faults to its embedding scope. The fault places of all enclosed activities are merged with the `fault_in` place of the scope. If an activity throws a fault, a token with the fault name is produced on this place. Transition `term` (forced termination) eventually propagates this token to the fault handlers and also stops the embedded activity. All further faults thrown to `fault_in` are ignored.

Every running activity can be stopped by marking place `stop`, which eventually results in a token on place `stopped`. Given that a BPEL process is a distributed system and the concurrency of multiple instances and scopes is underlined in the BPEL specification, the HUB semantics decides not to *instantly* stop activities, but to model forced termination concurrently. The motivation is to allow for maximal concurrency in the modelling of distributed systems.

After the inner activity has stopped (i. e. place `stopped` is marked), the fault handlers are started. If the fault can be caught by an enclosing `<catch>` or `<catchAll>` branch, the respective activity is executed. The scope is completed when the fault handler is completed faultlessly. However, if the fault cannot be caught or if an activity embedded in the fault handler throws a fault, this fault is propagated to the parent scope by marking place `fault_up`.

While the QUT semantics uses global state places `to_continue` and `to_stop` which have to be read by all embedded activities, the HUB stopping approach is *asynchronous* and is executed in three steps: Firstly, a fault is thrown by an activity. All concurrent activities can proceed as normal and even throw faults themselves. Secondly, the scope's inner activity is terminated; that is, its `stop` place is marked. The inner activity and the termination mechanism are concurrent: the moment the control flow stops is chosen nondeterministically (cf. Fig. 2). Finally, after the inner activity has stopped, the fault handlers start.

4.3.3 Conclusion

In the QUT semantics, a global stopping mechanism is applied to the mapping of termination. Once a fault is thrown during the execution of an activity within its directly enclosing scope Q , all remaining activities in scope Q stop immediately. For any activity that cannot execute any further, a “bypassing” mechanism is adopted to ensure the control flow reaches the end of scope Q . Then, the fault handling of scope Q starts. Since all remaining activities are bypassed, no further faults can be thrown.

In contrast, the HUB semantics models forced termination asynchronously; that is, upon the termination of a scope, all active activities within the scope do not have to stop immediately. However, the fault handlers are only executed if the inner activity has stopped.

5 Verification

5.1 Comparing the Compilers

Both semantics consist of a collection of patterns that are used to translate each BPEL activity into a Petri net pattern. Obviously, this translation is very laborious and takes too much time to be done manually. Therefore, for each semantics exists a compiler that automatically translates a BPEL process into a Petri net model. The HUB compiler supports both BPEL 1.1 and BPEL 2.0, whereas the QUT compiler is restricted to BPEL 1.1.

Currently, both implemented semantics abstract from data since variables in BPEL have an infinite data domain in general. If we would translate the process into a high-level Petri net, it may have an infinite state space which makes subsequent analysis impossible. To this account, both compilers abstract from time and instances due to the limitations of the semantics and they also abstract from data. As a result, models generated by the compilers are low-level Petri nets. Messages and the content of variables are modelled by indistinguishable black tokens. Data dependent branches (e. g. BPEL's `<if>`) are modelled by nondeterministic choices. In terms of Petri nets such an abstract net is a *skeleton net* (Vautherin, 1987).

5.1.1 QUT

The QUT semantics is implemented straightforwardly in the BPEL2PNML tool³. BPEL2PNML directly translates a BPEL process into a low-level Petri net in PNML format. Every activity is replaced by its predefined pattern, and nodes (places and transitions, respectively) with identical names are merged to one node.

As it can be seen from the QUT semantics in Sect. 4, the predefined pattern of an activity must suit every possible environment of that activity. Even if the actual activity cannot be skipped, the pattern will include a skip path, as in some other environment the activity can be skipped. Likewise, the pattern always includes support for links and termination, and the scope pattern always includes support for fault handlers, event handlers, and compensation handlers. As a result, the tool BPEL2PNML does not need to check the environment of an activity (and to possibly tune the pattern towards the activity) on the one hand; while on the other hand, the predefined pattern becomes very complicated, which in turn may lead to a very complicated Petri net in the end. Also, it is worth mentioning that a reduction of the unnecessary nodes, such as

³BPEL2PNML is available at www.bpm.fit.qut.edu.au/projects/babel/tools.

the skip path of the top level BPEL activity which cannot get skipped by definition, will be performed within the verification tool described in Sect. 5.2.

5.1.2 HUB

The HUB semantics is implemented in the tool BPEL2oWFN⁴, a compiler translating one or more⁵ BPEL processes into a Petri net model. This model can be exported as oWFN or in other file formats (e.g., PNML, PEP, LoLA, INA, SPIN) and thus supports a variety of analysis tools. It does not follow a brute-force mapping approach. Instead, BPEL2oWFN employs *flexible model generation* (Lohmann et al., 2008b), an approach that minimises the model both *during* and *after* the translation process, allowing to generate a compact model tailored to the analysis goal.

To this end, BPEL2oWFN builds a *control flow graph* (CFG) of the process and applies standard static analysis techniques (see Nielson et al. (2005) for an overview) to gain information about the activities, variables, scope hierarchy, and other aspects of the process. This gained information 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 modelling of the negative control flow or the occurrence of standard faults, whereas another goal does not. In the translation process, for each activity, a Petri net pattern is selected from this pattern repository according to the gained information.

After the translation process, structural reduction rules are applied to further scale down the size of the generated model. For each user-defined analysis goal (e.g., reachability), adequate reduction rules (adapted from Murata (1989)) 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 Petri net nodes that are not affecting the analysis.

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. Case studies show 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 (Lohmann et al., 2008b).

5.1.3 Conclusion

While the QUT compiler BPEL2PNML merges the patterns used by the input process, the HUB compiler BPEL2oWFN implements several sophisticated reduction techniques to reduce the model during its generation with respect to the verification goal. Furthermore, BPEL2oWFN applies standard Petri net reduction rules that may further reduce the model size. It is worthwhile mentioning that both compilers are integrated into the ProM framework (Dongen et al., 2005).

The ProM framework is a pluggable framework that is specifically tailored towards process mining; that is, the area that concerns deriving a process model from an event log (Aalst et al., 2003). At the moment of writing, the latest version of ProM is 4.2. This version contains several plug-ins that can mine some process model from an event log. Process models that can be mined using these plug-ins include Petri nets, EPCs (Keller et al., 1992), and Heuristic nets (Medeiros, 2006; Weijters et al., 2006). To be able to switch between different (process) models, ProM also contains a number of conversion plug-ins. For example, a Petri net can be converted into an EPC, a YAWL (Aalst and Hofstede, 2005) model, or a BPEL process. Each compiler has been implemented as a conversion plug-in and enables the users of ProM to convert a BPEL process into a Petri net (either a WF-net or an oWFN).

5.2 Possible Verification

5.2.1 QUT

The resulting Petri nets generated by BPEL2PNML are—as mentioned earlier—quite big and usually contain a lot of dead nodes, for instance unnecessary skip parts. For this reason, we “clean” the resulting Petri net from these dead skip paths. In addition, the structure of the net is used applying standard reduction rules (Murata, 1989). In contrast to the HUB approach, the model is only minimised after the translation.

The QUT semantics is designed to verify the soundness property (Aalst, 1998). The soundness property demands that the process can reach its final state from every reachable state. Thus, a sound process is deadlock-free. Furthermore, a sound net completes properly, i.e. without any tokens except on the final place. Soundness is a well-known sanity check for Petri nets that reflect workflows. To verify soundness, we use the tool Woflan (Verbeek et al., 2001).

Soundness is verified by generating the reachability graph for the given Petri net. If this is due to state explosion problems not possible, soundness cannot be checked. In such a situation, we might still be able to check relaxed soundness (Dehnert and Aalst, 2004) instead. The relaxed

⁴BPEL2oWFN is available at <http://service-technology.org/bpel2owfn>.

⁵If more than one process is given, the resulting models are composed to a single Petri net.

soundness property verifies that every transition is covered by *some* execution path that ends in the final state. If a transition is not on such a path, it cannot help in reaching the final state. Relaxed soundness can be approximated using techniques proposed in (Verbeek et al., 2007) and implemented in WofBPEL (Ouyang et al., 2005).

To verify the correctness of a BPEL process, we can use the proposed tools to translate the BPEL process into a Petri net. On the resulting net, we check soundness and relaxed soundness. If the net is not (relaxed) sound, a counterexample might help the BPEL designer to detect design flaws in the original BPEL process. The generated Petri nets can also be used as input by any Petri net-based model checkers to verify any temporal logic property.

Furthermore, if the reachability graph can be generated, WofBPEL also provides the analysis of BPEL-specific properties such as the existence of conflicting message-receiving activities (a situation which should throw a runtime fault at the BPEL level). In addition, messages that cannot be received anymore in future states can be determined. This information is annotated to the original BPEL model. These annotations can then be used by a BPEL engine to safely remove incoming messages in the message queue, thus optimising the resource management.

5.2.2 HUB

In addition to the translation, BPEL2oWFN also uses the CFG to check 54 of the 94 static analysis requirements⁶ proposed by the BPEL specification. They enable BPEL2oWFN to statically detect cyclic control links, read access to uninitialised variables, conflicting message-receiving activities, or other faulty constellations.

Due to the explicit modelling of the interface of a BPEL process, we are able to analyse its communication behaviour. Like the soundness property for WFNs, the existence of a partner process (formalised by the notion of *controllability* (Schmidt, 2005; Wolf, 2008)) is a minimal requirement for the correctness of an oWFN. An algorithm to decide controllability of an oWFN is implemented in the tool Fiona (Lohmann et al., 2006). This algorithm is constructive; that is, if the oWFN is controllable, a partner is generated. This partner can be translated back to BPEL using existing approaches, for example (Lassen and Aalst, 2006; Lohmann and Kleine, 2008).

Furthermore, to characterise all partners of an oWFN, the notion of an *operating guideline* (Lohmann et al., 2007) can be used. The operating guideline is a data structure that enables us to solve other problems concerning communicating services such as *consistency* (a composition of services can always reach a final state), *compatibility* (two services have equivalent external behaviour), or *exchangeability* of services (a service can be replaced by another service without losing a partner service).

⁶Most of the static analysis requirements that are not checked by BPEL2oWFN check aspects of XPath and are out of scope of the analysis goals presented in this paper.

Finally, BPEL2oWFN allows to create a closed system of the BPEL process; that is, a Petri net model without interface. This model is similar to a workflow net and can be used to check temporal logic properties using existing model checking tools (see Hinz et al. (2005) for an example).

5.2.3 Conclusion

While the QUT semantics is tailored to verify (relaxed) soundness, the HUB semantics using oWFNs allows for the analysis of the communication behaviour of a BPEL process. Furthermore, BPEL2oWFN can detect many design flaws directly on the structure of the BPEL process. Finally, the resulting nets of both semantics can be analysed by common model checkers.

6 Validation and Case Studies

Since BPEL is only informally specified, it is impossible to *prove* the correctness of the presented semantics. Instead, we validated the semantics by translating BPEL processes into Petri net models and comparing the behaviour of generated models with their respective expected behaviours.

In particular, Hinz et al. (2005); Lohmann et al. (2006, 2008b,a) present detailed case studies using the HUB semantics and their compiler BPEL2oWFN. The QUT semantics and the compiler have been used to check behavioural properties such as non-reachable activities, conflicting message-receiving activities, and resource management optimisation in Ouyang et al. (2005, 2007).

To underline the impact of the modelling decisions described earlier, we translated several BPEL processes⁷ and calculated their state spaces. The processes are toy examples taken from the BPEL specification as well as real-life processes used in other papers.

Table 1 summarises the results. For each process, we present the sizes of the resulting nets (in terms of places and transitions) after structural reduction. Furthermore, we used the model checker LoLA (Schmidt, 2000) to calculate the complete state space as well as the reduced state space using partial order reduction. To receive an impression of the size of the processes, the number of activities and in particular the number of scopes are shown.

In general, the nets of the HUB semantics have more nodes. This is due to the more detailed patterns and the stopping concept which introduces many transitions (cf. Fig. 2). For most of the processes with only one scope, the size of the state spaces of the QUT and the HUB nets are very similar. If, however, a lot of scopes and links are involved, the flexible model generation and the different modelling of DPE implemented in BPEL2oWFN permits very compact models. For instance, process “Identity” (a real-life process modelling the application of an identity card) consists of 8 scopes and 17 links: the resulting state

⁷To use both compilers BPEL2PNML and BPEL2oWFN, the processes only use features of BPEL 1.1.

Table 1: Comparison of net sizes (places and transitions) and state spaces (full state space and reduced state space using partial order reduction).

Process	Activities	Scopes	QUT				HUB			
			Petri net		state space		Petri net		state space	
			P	T	full	reduced	P	T	full	reduced
Booking	14	1	27	32	33	33	38	56	43	43
Loan 1	7	1	48	47	282	159	64	88	524	177
Loan 2	7	1	49	48	283	167	70	98	292	97
Identity	81	8	110	120	39,053	26,137	168	265	300	285
Purchase	35	4	53	50	190	86	48	80	42	42
Salesforce	22	2	38	43	100	95	68	117	71	71
Travel	8	1	34	32	73	41	37	52	30	28
Vacation	35	3	58	64	967	369	109	174	111	111
Shipping	16	1	21	21	73	43	27	53	23	23
Phone	43	1	74	79	1261	465	116	222	136	136
Auction	17	1	27	31	45	44	59	92	232	128

space of the HUB net (300 states) is just a fraction of the state space of the QUT net (39,053 states).

the activities may stop at any time after the fault occurrence, but before the `<faultHandlers>` start.

7 Conclusion

Many researchers have spent efforts in formalising and analysing of BPEL processes. In this paper, we compared two feature-complete Petri net semantics for BPEL, their compilers, and tools to support the analysis of certain properties of BPEL processes. The comparison has shown that due to some modelling decisions the two semantics differ in the following issues:

- The HUB semantics is a high-level Petri net which adequately models data. The current translation abstracts from data; thus, the resulting model is a low-level Petri net. In contrast, the QUT semantics is defined as a low-level Petri net from the start.
- The HUB semantics explicitly represents the interface whereas the QUT semantics does not. As a result, the QUT and the HUB semantics result in different models (workflow nets and open workflow nets, respectively) that allow for analysing different properties.
- In case of dead-path-elimination, the moment when an activity can set the status of its outgoing links to false is different in the two semantics. While the QUT implementation conforms to the BPEL specification, the HUB approach uses an overapproximation resulting in smaller models.
- When a scope has to be terminated due to a fault, the moment when all the activities within the scope stop is different in the two semantics. In the QUT semantics, the activities stop immediately as soon as a fault has occurred to the scope, because a global termination concept is used. In contrast in the HUB semantics, an asynchronous termination concept is used; thus,

Comparing the compilers, QUT's BPEL2PNML applies a brute-force translation and the resulting net is reduced after the translation. The HUB compiler BPEL2oWFN, in contrast, applies an approach of a flexible model generation that reduces the Petri net model during and after the translation with respect to the property to be analysed. For this purpose, for each BPEL activity, several Petri net patterns with different degree of abstraction are available in a pattern repository. Using static analysis on the BPEL code, we select the most abstract pattern applicable in a given context. As shown in the case study (see Sect. 6), flexible model generation combined with the more efficient implementation of the DPE usually results in very compact models for big BPEL processes. As another difference, BPEL2oWFN allows to check many properties statically on the BPEL code.

We further compared the different properties that can be analysed on the resulting Petri net model. All models can be analysed for temporal logic properties using existing model checkers. QUT models, in particular, can be checked for soundness. The workflow net structure of the QUT models allows to use efficient algorithms (i.e. performed on the structure of the net rather than on the reachability graph) for WFNs implemented in the tools Woflan and WofBPEL. An analysis based on the structure of the generated net is not possible in the HUB semantics, because its structure does not follow the requirements of workflow nets (e.g. oWFNs allow for arbitrary initial and final markings). However, similar properties such as liveness, boundedness or the absence of dead transitions could be checked using state space analysis. Furthermore, since the HUB models explicitly represent the interface of a BPEL process, they can be used to check controllability of the process and to calculate its operating guideline using the tool Fiona.

In ongoing research we work on some tool integration. For example, we think of implementing a global stop con-

cept in the HUB compiler BPEL2oWFN and study its impact combined with flexible model generation. We also want to spend additional efforts to detect more structural reduction rules that are specific for BPEL processes.

Acknowledgements

The authors wish to thank Christian Gierds and Jan Martijn van der Werf for their help with the case studies. Niels Lohmann is funded by the DFG project “Operating Guidelines for Services” (WO 1466/8-1). Chun Ouyang is funded by the Australia Research Council under Discovery Grant DP451092. Christian Stahl is funded by the DFG project “Substitutability of Services” (RE 834/16-1).

REFERENCES

- Aalst, W. M. P. v. d. (1998). The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66.
- Aalst, W. M. P. v. d., Dongen, B. F. v., Herbst, J., Maruster, L., Schimm, G., and Weijters, A. J. M. M. (2003). Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267.
- Aalst, W. M. P. v. d. and Hee, K. M. v. (2002). *Workflow Management: Models, Methods, and Systems*. MIT press, Cambridge, Massachusetts.
- Aalst, W. M. P. v. d. and Hofstede, A. H. t. (2005). YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275.
- Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Golland, Y., Guizar, A., Kartha, N., Liu, C. K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., Rijn, D. v. d., Yendluri, P., and Yiu, A. (2007). Web Services Business Process Execution Language Version 2.0. OASIS Standard, 11 April 2007, OASIS.
- Andrews, T., Curbera, F., Dholakia, H., Golland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM, Microsoft.
- Breugel, F. v. and Koshkina, M. (2006). Models and Verification of BPEL. <http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf>.
- Dehnert, J. and Aalst, W. M. P. v. d. (2004). Bridging the Gap Between Business Models and Workflow Specifications. *International Journal of Cooperative Information Systems*, 13(3):289–332.
- Dongen, B. F. v., Medeiros, A. K. A. d., Verbeek, H. M. W., Weijters, A. J. M. M., and Aalst, W. M. P. v. d. (2005). The ProM Framework: A New Era in Process Mining Tool Support. In Ciardo, G. and Darondeau, P., editors, *Applications and Theory of Petri Nets 2005, 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005, Proceedings*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer-Verlag.
- Fahland, D. (2005). Complete abstract operational semantics for the Web Service Business Process Execution Language. *Informatik-Berichte 190*, Humboldt-Universität zu Berlin.
- Girault, C. and Valk, R., editors (2002). *Petri Nets for System Engineering – A Guide to Modeling Verification and Applications*. Springer-Verlag Berlin Heidelberg New York.
- Hinz, S., Schmidt, K., and Stahl, C. (2005). Transforming BPEL to Petri Nets. In Aalst, W. M. P. v. d., Benattallah, B., Casati, F., and Curbera, F., editors, *Third International Conference on Business Process Management (BPM 2005), Nancy, France, 6–8 September 2005, Proceedings*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag.
- Keller, G., Nüttgens, M., and Scheer, A. W. (1992). Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, Germany.
- Lassen, K. B. and Aalst, W. M. P. v. d. (2006). WorkflowNet2BPEL4WS: A Tool for Translating Unstructured Workflow Processes to Readable BPEL. In Meersman, R. and Tari, Z., editors, *On the Move to Meaningful Internet Systems 2006, Montpellier, France, October 29 – November 3, 2006. Proceedings, Part I*, volume 4275 of *Lecture Notes in Computer Science*, pages 127–144. Springer-Verlag.
- Leymann, F. and Roller, D. (2000). *Production Workflow – Concepts and Techniques*. Prentice Hall.
- Lohmann, N. (2007). A Feature-Complete Petri Net Semantics for WS-BPEL 2.0 and its Compiler BPEL2oWFN. Techn. Report 212, Humboldt-Universität zu Berlin, Berlin, Germany.
- Lohmann, N. (2008). A feature-complete Petri net semantics for WS-BPEL 2.0. In Dumas, M. and Heckel, R., editors, *Web Services and Formal Methods, Forth International Workshop, WS-FM 2007 Brisbane, Australia, September 28-29, 2007, Proceedings*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag.

- Lohmann, N. and Kleine, J. (2008). Fully-automatic Translation of Open Workflow Net Models into Human-readable Abstract BPEL Processes. In Kühne, T., Reisig, W., and Steimann, F., editors, *Modellierung 2008, 12.-14. März 2008, Berlin, Proceedings*, volume P-127 of *Lecture Notes in Informatics (LNI)*, pages 57–72. GI.
- Lohmann, N., Kopp, O., Leymann, F., and Reisig, W. (2008a). Analyzing BPEL4Chor: Verification and participant synthesis. In Dumas, M. and Heckel, R., editors, *Web Services and Formal Methods, Forth International Workshop, WS-FM 2007 Brisbane, Australia, September 28-29, 2007, Proceedings*, volume 4937 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag.
- Lohmann, N., Massuthe, P., Stahl, C., and Weinberg, D. (2006). Analyzing Interacting BPEL Processes. In Dustdar, S., Fiadeiro, J. L., and Sheth, A., editors, *Fourth International Conference on Business Process Management (BPM 2006), Vienna, Austria, 5–7 September 2006, Proceedings*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag.
- Lohmann, N., Massuthe, P., Stahl, C., and Weinberg, D. (2008b). Analyzing interacting WS-BPEL processes using flexible model generation. *Data Knowl. Eng.*, 64(1):38–54.
- Lohmann, N., Massuthe, P., and Wolf, K. (2007). Operating guidelines for finite-state services. In Kleijn, J. and Yakovlev, A., editors, *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*, pages 321–341. Springer-Verlag.
- Martens, A. (2005). Analyzing Web Service based Business Processes. In Cerioli, M., editor, *Proceedings of Intl. Conference on Fundamental Approaches to Software Engineering (FASE'05), Part of the 2005 European Joint Conferences on Theory and Practice of Software (ETAPS'05)*, volume 3442 of *Lecture Notes in Computer Science*, Edinburgh, Scotland. Springer-Verlag.
- Massuthe, P., Reisig, W., and Schmidt, K. (2005). An Operating Guideline Approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics*, 1(3):35–43.
- Medeiros, A. K. A. d. (2006). *Genetic Process Mining*. PhD thesis, Eindhoven University of Technology, Eindhoven.
- Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580.
- Nielson, F., Nielson, H. R., and Hankin, C. (2005). *Principles of Program Analysis*. Springer-Verlag, 2nd edition.
- Ouyang, C., Verbeek, H. M. W., Aalst, W. M. P. v. d., Breutel, S., Dumas, M., and Hofstede, A. H. t. (2005). WofBPEL: A Tool for Automated Analysis of BPEL Processes. In Benatallah, B., Casati, F., and Traverso, P., editors, *Proceedings of the Third International Conference on Service Oriented Computing (ICSOC 2005)*, volume 3826 of *Lecture Notes in Computer Science*, pages 484–489, Amsterdam, The Netherlands. Springer-Verlag.
- Ouyang, C., Verbeek, H. M. W., Aalst, W. M. P. v. d., Breutel, S., Dumas, M., and Hofstede, A. H. t. (2007). Formal Semantics and Analysis of Control Flow in WS-BPEL. *Science of Computer Programming*, 67(1):162–198.
- Reisig, W. (1985). *Petri Nets*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, EATCS Monographs on Theoretical Computer Science edition.
- Schmidt, K. (2000). LoLA: A Low Level Analyser. In Nielsen, M. and Simpson, D., editors, *Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000)*, number 1825 in *Lecture Notes in Computer Science*, pages 465–474. Springer-Verlag.
- Schmidt, K. (2005). Controllability of open workflow nets. In Desel, J. and Frank, U., editors, *Enterprise Modelling and Information Systems Architectures*, number P-75 in *Lecture Notes in Informatics (LNI)*, pages 236–249. Entwicklungsmethoden für Informationssysteme und deren Anwendung (EMISA, RWTH Aachen), Bonner Köllen Verlag.
- Vautherin, J. (1987). Parallel systems specifications with coloured Petri nets and algebraic specifications. In Rozenberg, G., editor, *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*, pages 293–308. Springer-Verlag.
- Verbeek, H. M. W., Aalst, W. M. P. v. d., and Hofstede, A. H. t. (2007). Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. *The Computer Journal*, 50(3):294–314.
- Verbeek, H. M. W., Basten, T., and Aalst, W. M. P. v. d. (2001). Diagnosing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279.
- Weijters, A. J. M. M., Aalst, W. M. P. v. d., and Medeiros, A. K. A. d. (2006). Process mining with the heuristics miner-algorithm. BETA Working Paper Series, WP 166, Eindhoven University of Technology, Eindhoven.
- Wolf, K. (2008). Does my service have partners? *Transactions on Petri Nets and Other Models of Concurrency*. (Accepted for publication).