2007 IEEE Asia-Pacific Services Computing Conference

Efficient Analysis of BPEL 2.0 Processes using π-Calculus

Matthias Weidlich, Gero Decker, Mathias Weske
Hasso-Plattner-Institute, University of Potsdam, Germany
{matthias.weidlich, gero.decker, mathias.weske}@hpi.uni-potsdam.de

Abstract

The Business Process Execution Language (BPEL) has become the de-facto standard for the description of Web Service compositions. A variety of formal approaches to decide compatibility and consistency for BPEL processes has been presented. Nevertheless, these approaches suffer from high complexity and state explosion. Therefore we present a lean formalization of BPEL 2.0 based on the π-calculus, that enables efficient reasoning. Due to our focus on behavioral compatibility and consistency checking (and not on comprehensive formalization), we are able to reduce effort needed for process verification. Besides the exemplary application of our approach, we also compare it to existing BPEL formalizations by means of complexity.

1 Introduction

The increasing influence of the service-oriented architecture (SOA) is at least partly driven by the growing demand for business-to-business process integration. Owing to the shift to distributed business processes, Web Services have been established as a standardized and widely accepted paradigm for implementing process collaborations. On a conceptual level, functionality is provided via Web Services, that can be invoked by high-level applications or by other Web Services. This paradigm is further underpinned through a stack of web standards and technologies.

Concerning the logical description of composite Web Services, the Web Services Business Process Execution Language (shortly BPEL) [15] has emerged as the leading standard. It defines a model and a grammar for defining the behavior of a business process based on interactions. Furthermore, the BPEL specification distinguishes between non-executable *abstract processes*, and fully-specified *executable processes*, which can be deployed and executed on BPEL engines.

In order to ensure successful interaction of certain services, their structural and behavioral *compatibility* has to be examined. In contrast to structural process verification BPEL is inappropriate as a basis for behavioral compatibility checking, due to its expressiveness and mergence of data and process flow. As a consequence, a demand for feasible abstractions of the process flow, restricting the state base for reasoning approaches, can be determined. To meet this demand, a wide variety of BPEL formalizations using different underlying concepts have been presented in recent years. However, these approaches focus on extensive formalizations that suffer from high complexity and state explosion resulting in inefficient reasoning. This paper argues, that any BPEL formalization should be driven by a certain aim instead of trying to capture all aspects. By focusing on behavioral compatibility checking, we present a lean and lightweight formalization of BPEL. In addition, we show how our approach can be used to decide *consistency* between abstract BPEL processes and a process implementation (e.g. as an executable BPEL process). Although our approach is based on the π-calculus, the goal driven restriction of formalization can also be applied to other formal foundations (e.g. Petri nets and finite state machines).

The next section discusses compatibility and consistency of BPEL processes by means of an example. Section 3 summarizes related work, while Section 4 introduces the π-calculus. Afterwards, our formalization of BPEL is defined. Section 6 elaborates on how our formalization can be used to decide compatibility and consistency. Addressing our aim for a lean approach, we compare different formalizations regarding the evolving state space. Finally, Section 8 concludes and discusses open issues.

2 Compatibility and Consistency

To introduce the topic of process verification we present an example from the financial domain, namely an investment spending scenario operated by a stockbroker. Figure 1 illustrates the example using the Business Process Modeling Notation (BPMN) [16]. At first the customer sends an investment request to a stockbroker. Depending on the investment request, the broker obtains further information about several stocks from rating agencies. Subsequently, a set of stocks is chosen and the broker retrieves the current

0-7695-3051-6/07 $25.00 © 2007 IEEE
DOI 10.1109/APSCC.2007.36

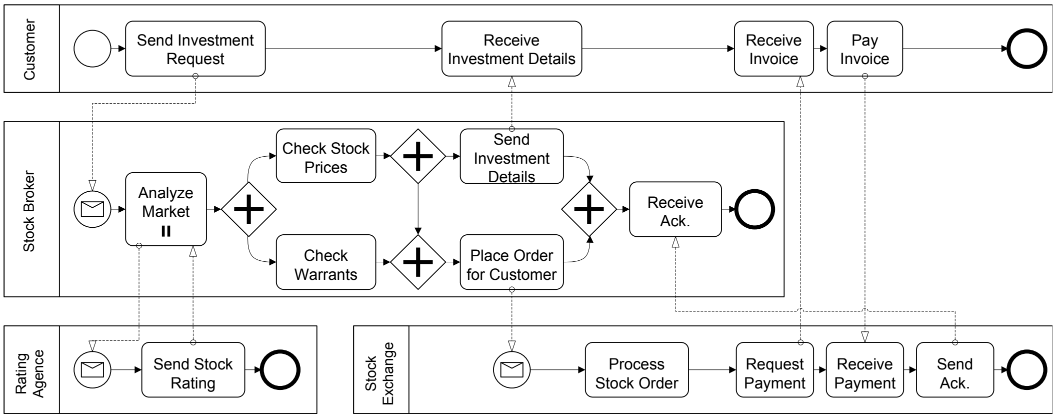

Figure 1. Stockbroker scenario

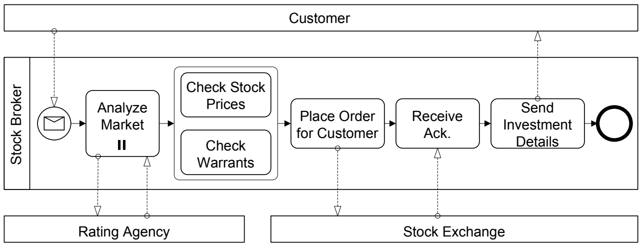

Figure 2. Incompatible stockbroker

stock prices (not necessarily from the stock exchange) and reviews his own stock warrants whether they can be used for the current investment. After the retrieval of stock prices, the broker sends the details about the investment to the customer. This activity is independent of the warrant analysis, as only the broker might benefit from the warrants. Later on, the broker compares his warrant prices with the current stock prices and places an order on behalf of the customer at the stock exchange. In the stock exchange process, the order is processed and an invoice is sent to the customer. After the customer met the account, the stock exchange sends an acknowledge message to the stockbroker indicating the successful proceeding of the order transaction.

Provided that the interconnected processes do not suffer from any structural incompatibility (e.g. different message formats), the introduced composition is compatible. Due to the absence of behavioral anomalies (e.g. deadlocks) all processes end properly. Nevertheless, we can imagine another process implementation for the stock broker as illustrated in Figure 2. In contrast to the first example, the stockbroker sends the investment details *after* he received the acknowledge message. As the customer requires these details before he is willing to pay for the stock order, which in turn must happen before the stock exchange sends the acknowledgment message, a deadlock occurs.

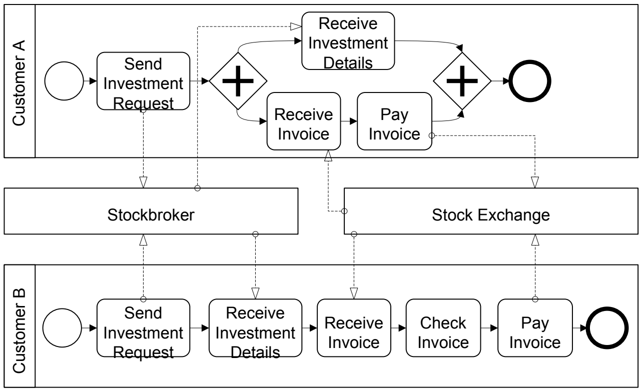

Figure 3. Different customer implementations

In other settings, abstract BPEL processes are given as behavioral specifications for a role. In such cases we need to check whether the fully executable BPEL is consistent with the abstract definition to ensure successful interaction. Regarding the introduced example, we can treat the introduced process for the customer as an abstract behavior description, while two specific process implementations are shown in Figure 3. In one process, the internal behavior has been changed, as the customer receives the investment details and handles the invoice concurrently (Customer A). In the other process, we added an internal activity, in which the customer inspects the invoice (Customer B). For each of these process implementations, consistency to the specification has to be decided.

Section 6 addresses the above mentioned issues. Hence, we show how our formalization is put into action to detect deadlocks and decide consistency for process implementations regarding an abstract behavioral specification.

3 Related Work

Related work comprises different formalizations of BPEL. A first group of approaches uses different kinds of automata as formal foundation. Fahland advocates the usage of abstract state machines and defined an extensive formalization in [7], while he discusses fault handlers and event handlers in detail in [8]. Moreover, Fisteus et al. use finite state machines for process verification in [1].

A second group of approaches prefers Petri nets as formal foundation for reasoning on BPEL processes. On the one hand, Aalst et al. define a mapping of BPEL 1.1 constructs to workflow nets in [3]. On the other hand, Stahl et al. introduced a Petri net based BPEL 1.1 formalization [21, 10]. The latter is considered in the discussion of formalization complexity in Section 7.

In the field of process algebras, Ferrara presents a two-way mapping from BPEL to LOTOS [9]. In contrast to most formalization approaches, the mapping includes both, the control flow and the data handling. Consequently, this approach is more complex than methods focusing only on the control flow and is therefore not considered in our discussion of mapping complexity. Focused on the control flow, Mazzara and Lucchi present a new language named $web\pi_\infty$ as an extension of the π-calculus with a transactional mechanism [13]. Based thereon, a mapping of BPEL constructs to this language is defined. Introducing $web\pi_\infty$ as a proposal for enhancements of BPEL, Mazzara and Lucchi restrict their mapping to a subset of BPEL activities and focus on fault, event and compensation handlers. Besides the extension of a known process algebra (and therefore missing tool support), the lack of support for control links has to be seen as a major drawback. Nevertheless, we discuss the complexity of this approach in Section 7. Fadlisyah provides a BPEL 1.1 formalization using the π-calculus [6]. Owing to the aim to provide a formalization, that supports error handling without an extension of the underlying process algebra, Fadlisyah's mapping leads to complex processes (please refer to Section 7) and does not include constructs with conditional and repetitive behavior or synchronization mechanisms.

4 Prerequisites: The π-Calculus

This section introduces the π-calculus, which is a process algebra developed to describe and analyze concurrent, interacting processes with dynamic or evolving structures in a formal way. It is based on names representing the communication channels as well as the messages sent over them. Hence, communication channels can be passed to other processes to support link passing mobility, the π-calculus is predestined to model dynamic binding in the SOA domain (an extended motivation can be found in [17]). Besides, simulation and bisimulation techniques for proving compatibility and consistency are available.

The grammar of the π-calculus is defined in the Backus normal form as follows:

$$\begin{array}{rcl} P & ::= & M \mid P|P' \mid \mathbf{v}zP \mid !P \mid K(y_1, \ldots y_n) \\ M & ::= & \mathbf{0} \mid \pi.P \mid M + M' \\ \pi & ::= & \overline{x}\langle\tilde{y}\rangle \mid x(\tilde{z}) \mid \tau \mid [x = y]\pi. \end{array}$$

The process semantic can be informally characterized as follows: The concurrent execution of two processes P and P' is denoted by $P|P'$. $C = \prod_{i=1}^{n} P_i$ is the short form for the definition of the composition C as the parallel execution of n processes P_i and $P \in C$ denotes that P is a concurrent subprocess of C. In the process $\mathbf{v}zP$, the operator $\mathbf{v}$ restricts the name z to P. The meaning of the process replication, given by $!P$, is that an infinite number of replicated process instances are acting in parallel. Recursion for P with the parameters $y_1, \ldots y_n$ is expressed via $P(y_1, \ldots y_n)$. The inaction $\mathbf{0}$ represents empty or inactive behavior. The summation operator specifies alternatives, as $M + M'$ evolves to M or M', while $\sum_{i=1}^{n}(M_i)$ evolves to M_j $(1 \leq j \leq n)$. All actions a process can do, are given by π. To model interactions, input and output prefixes are used. The output prefix $\overline{x}\langle\tilde{y}\rangle.P$ evolves to P after sending a sequence of names $\tilde{y}$ over the channel identified with x. The corresponding action is the input prefix $x(\tilde{z}).P$. This process receives a sequence of names and continues as $P\{^{\tilde{m}}/_{\tilde{z}}\}$ with all occurrences of $\tilde{z}$ replaced by the received names. Additionally, τ, the silent transition, models an internal action, which cannot be observed at all. The match prefix $[x = y]\pi.P$ is defined in the expected way: the process behaves as $\pi.P$, iff x and y are identical, and like $\mathbf{0}$ otherwise.

A complete formal definition of the π-calculus semantics based on a labeled transition system can be found in [20]. Advanced reasoning concepts, that are of high relevance for this paper include distinction of *free* and *bound* names, the definition of process *contexts* and the application of *weak open bisimulation*. Please refer to [20, 14] for details.

5 Formalization of BPEL

This section defines our π-calculus based formalization of BPEL processes. At first, the scope of our formalization is defined. Due to the extensive usage of contexts, we discuss this concept in detail afterwards. Subsequently, the BPEL standard elements, the basic activities and the structural activities are formalized.

Focusing on compatibility and consistency checking, our formalization captures the majority of BPEL control flow constructs, while abstracting from the data flow. Thus, we abstract from any data assignment in message activities and the formalization of the *assign* activity is restricted to the treatment of partner links.

In regard to control flow, we focus on the positive flow. Consequently, we do not formalize *compensation handlers*, *fault handlers* and *termination handlers*, as they require modeling of the internal state for every basic activity to allow arbitrary process termination. In the course of a lean formalization, we neglect these internal states. Hence, basic activities of the BPEL error-handling framework, namely *throw*, *rethrow*, *compensate* and *compensate scope* activities, are also not considered. For the same reason, the *exit* activity, that terminates the whole BPEL process instance immediately, is not formalized.

Minor restrictions of the presented formalization conclude the absence of *join conditions* and *transition conditions* in the formalization of *control links* and the disregard of message correlation. Further on, we have to restrict the definition of process instantiation to inbound message activities that are not embedded in conditional constructs.

5.1 The Context Concept

In general, a context C is a process containing a hole, denoted by $\langle . \rangle$, which can be replaced with an arbitrary process other than inaction ($C\langle P \rangle$ denotes the replacement with the process P). This replacement is literal, thus free names of P might be bound in $C\langle P \rangle$. Further on, name restrictions in the context process are not preserved and the logical ordering of the context parts before and after the hole cannot be determined unambiguously. Imagine a context $C = (\mathbf{v}b)a.\langle . \rangle.b$ and two processes $P_1 = (\mathbf{v}b)\overline{c}\langle b \rangle$ and $P_2 = c + \tau$. In the process $C\langle P_1 \rangle = (\mathbf{v}b)a.(\mathbf{v}b)\overline{c}\langle b \rangle.b$ the initial name restriction is abrogated, while in $C\langle P_2 \rangle = (\mathbf{v}b)a.c + \tau.b$ the action on b can occur *before* the action on a. To avoid these effects, we require two assumptions to hold. At first, any replacement of a process hole respects name restrictions via the operator $\mathbf{v}$. Indeed, restriction through name binding in input actions can be abrogated. Secondly, for any replacement of a process hole acting as an operation in a sequence, the sequential ordering has to be preserved. The latter constraint is of high relevance, as we allow contexts with more than one hole, for instance the context $C = \langle P_1 \rangle.\langle P_2 \rangle$. The aforementioned standard definition would allow any replacement, thus P_2 might be executed before P_1, while our second constraint ensures the initially defined ordering. Consequently, P_1 has to be executed before P_2. Based thereon, we use contexts respecting these constraints to enable a boundless nesting of structured activities through context replacement. In order to avoid confusion, a process P is denoted by $P^{\langle\rangle}$, if it can only be applied to replace a context hole.

5.2 Standard Elements

Each BPEL activity has optional containers *sources* and *targets* (referred to as standard elements in the BPEL specification [15]) to express synchronization relationships by means of *control links*. Both incoming (*target*) and outgoing (*source*) links can optionally be associated with Boolean formulas, that are not considered in our approach. Therefore our semantics for the *targets* container complies with the default interpretation, the disjunction of all *target* links. Further on, the status value of *source* links is assessed via non deterministic choices, if there exists a *transition condition*. In our formalization all incoming links of one activity are represented by a channel l on which a message is passed, if the status of the link is *true*. In addition, we use another channel h to indicate, that the status of the link has been determined. Consequently, communication *only* on h indicates a *false* status. We formalize *control links* as follows ($h, a, exec$ and $skip$ have to be restricted to the embracing process composition):

$$SRC^{\langle\rangle} = (\mathbf{v}s)sk.\overline{s} + ex.\overline{s} \mid s.\prod_{i=1}^{n}(\tau.\overline{l_i}.\overline{h_i}.\overline{go} + \tau.\overline{h_i}.\overline{go}) \mid go$$

$$TAR^{\langle\rangle} = \tau \mid skip.\overline{sk} + exec$$

$$TAR_c = \{h\}_m.\overline{a} \qquad\qquad TAR_d = l.(a.\overline{exec} \mid !l) + a.\overline{skip}$$

According to these processes, a *control link* l is established for $l_i = l$, which implies $h_i = h$. As every *incoming* link is represented by a channel l, the target process receives only on this channel name. In contrast the process $SRC^{\langle\rangle}$ models n outgoing links to n different activities. Thus, it sends messages on a set of link channels l_i. As $SRC^{\langle\rangle}$ should be used inside a certain context, we use the name go to synchronize the process. In the same matter, $TAR^{\langle\rangle}$ is used inside a context, thus the τ action is needed to connect the process part on the left side of the according context hole. Although there exists only one channel name for each incoming link, messages on it might be sent from several activities (all connected through one of their outgoing links). Moreover, receiving a message on the name sk will skip an activity to enable *dead path elimination*. Consequently, $SRC^{\langle\rangle}$ receives the message on sk, while the encapsulated process, actually realizing the BPEL activity, is not executed. In contrast, the activity is activated, if $TAR^{\langle\rangle}$ receives a message on the channel $exec$. TAR_c and TAR_d are agents (and therefore not used inside a context) needed to wait until the status of all links has been determined and to take a decision about the continuation.

The introduced processes $SRC^{\langle\rangle}$ and $TAR^{\langle\rangle}$ are applied, if an activity has outgoing *and* incoming links. Activities having either outgoing or incoming links need slightly adapted processes. An activity without incoming links cannot be skipped, therefore only $SRC^{\langle\rangle}$ is inserted in the corresponding context, defined as follows:

$$SRC^{\langle\rangle} = ex.\prod_{i=1}^{n}(\tau.\overline{l_i}.\overline{h_i}.\overline{go} + \tau.\overline{h_i}.\overline{go}) \mid go$$

In contrast we need to insert both processes $SRC^{\langle\rangle}$ and $TAR^{\langle\rangle}$ in the corresponding contexts, if the activity has only incoming links. Nonetheless, the process definition of $SRC^{\langle\rangle}$ can be reduced to $SRC^{\langle\rangle} = sk.\overline{go} + ex.\overline{go} \mid go.$

5.3 Basic Activities

For each basic activity two different formalizations are presented, an initial definition of the activity process and a second formalization taking standard elements into account (P_{SE} represents the second formalization for the process P). Therefore, the context holes $\langle SRC\rangle$ and $\langle TAR\rangle$ are appended to the initial definition.

A main question is, how information needed to identify a certain service is represented in our formalization. These information include the *partner link*, the *port type* and the name of the *operation*. This three tuple is encoded directly in a channel name $partnerLink\_portType\_operation$ (or shortly $pLpTo$), which is used to invoke services or receive service requests. Thus, a static *one-way interaction* is formalized as follows (please refer to the next part for the scope related issues of the invoke activity):

$$I^{\langle\rangle} = \overline{pLpTo} \qquad I_{SE}^{\langle\rangle} = (\mathbf{v}\ ex)\langle TAR\rangle.\overline{pLpTo}.\overline{ex} \mid \langle SRC\rangle$$
$$R^{\langle\rangle} = pLpTo \qquad R_{SE}^{\langle\rangle} = (\mathbf{v}\ ex)\langle TAR\rangle.pLpTo.\overline{ex} \mid \langle SRC\rangle$$

While the process I (or I_{SE}) is the *asynchronous* invoke activity, R (or R_{SE}) describes the receive activity. The introduced processes do not support dynamic binding of partner links via *assign* statements. Addressing this issue, we specify a memory cell generator $C_{pL} = !mem_{pl}(id).M_{pl}(id, \tilde{\bot})$ as a concurrent agent for every partner link name that is used in an *assign to partner link* statement. It creates a memory cell $M_{pL}(pid, \tilde{y}) = pid.(\overline{pL}\langle\tilde{y}\rangle.M(pid, \tilde{y}) + pL(\tilde{y}).M(pid, \tilde{y}))$ for every process instance. A process instance transmits its identifier pid (q.v. the definition of the process construct) once it requests a cell. The identifier unlocks the corresponding memory cell before any values can be written or read, respectively. As it would require extensive data flow analysis to determine the names, which actually need to be stored, we require all service information to be passed to the memory cell. Therefore the cell for a single partner link, contains all three tuples $pLpTo$ related to this partner link. Thus, the following processes define the one-way interaction respecting dynamic binding (the processes with standard elements are not shown, but they can easily be derived from the above mentioned processes):

$$I^{\langle\rangle} = \overline{pid}.pl(\tilde{y}).\overline{pLpTo}\langle\tilde{y}\rangle \quad R^{\langle\rangle} = pLpTo(\tilde{y}).\overline{pid}.\overline{pl}\langle\tilde{y}\rangle$$

The formalization $I^{\langle\rangle}$ of the invoke activity assumes, that the process contains an *assign to partner link* statement for the corresponding partner link and therefore uses a memory cell to ascertain the current binding of the partner link. The absence of such a statement, leads to an adapted formalization through $I^{\langle\rangle} = \overline{pLpTo}\langle\tilde{y}\rangle$. Again, R is the receive activity. Contrary to the mentioned processes, a formalization of a *request-response scenario* has to realize a *synchronous* invoke activity. As a consequence, the invoke activity transmits a name on which the synchronous invoke activity expects the response. In addition, the implicit correlation (potentially not stated in the BPEL code) between inbound (e.g. receive) and outbound (e.g. reply) message activities via *message exchanges* has to be considered. Due to potential concurrency of the outbound and inbound activity, a memory cell generator C_{me} creating cells $M_{me}(pid, \tilde{y})$ (defined as in the solution concerning dynamic binding) is introduced as a concurrent agent for each message exchange correlation. Evidently, the introduced mechanism to handle dynamic binding, is also applicable to the request-response scenario. Thus, a synchronous interaction with dynamic binding is formalized as follows (please note that the reply activity can also pass a partner link):

$$I^{\langle\rangle} = (\mathbf{v}\ res)pl(\tilde{y}).\overline{pLpTo}\langle res, \tilde{y}\rangle.res(\tilde{z}).\overline{pid}.\overline{pl}\langle\tilde{z}\rangle$$
$$R^{\langle\rangle} = pLpTo(res, \tilde{y}).\overline{pid_1}.\overline{pl}\langle\tilde{y}\rangle.\overline{pid_2}.\overline{me}\langle res\rangle$$
$$RP^{\langle\rangle} = \overline{pid_1}.pl(\tilde{z}).\overline{pid_2}.me(y).\overline{y}\langle\tilde{z}\rangle$$

$I^{\langle\rangle}$ is the synchronous invoke activity, while $R^{\langle\rangle}$ and $RP^{\langle\rangle}$ represent the related receive and reply activities.

It was already mentioned, that we restrict our formalization of the *assign* activity to the treatment of partner links. The approach presented above, however, leads to false formal representations in certain cases (e.g. a process receives a partner link, but invokes another service on this partner link, before binding the partner link to the received one). Consequently, the assumption, that any binding of a received partner link (through an assign activity) is applied before the next message activity uses this partner link, must hold. In addition, passing of partner links through input and output variables along several processes cannot be represented in our approach. Therefore, we require all sent partner link references to be bound to a partner link in the receiving process, if the reference is propagated further on.

An important side effect of inbound message activities (e.g. receive, pick, on event) is the potential creation of process instances, signaled with the BPEL attribute *createInstance*. In general, a process has one or more *start activ-*

ities, which can be formalized using replication. The process $INIT^{\langle\rangle} = !pLpTo$ activates a BPEL process and is therefore always inserted in a context hole at the beginning of a process. As it was already mentioned, we have to restrict the definition of process instantiation to inbound message activities that are not embedded in conditional constructs. Since conditional behavior is formalized as non-deterministic choice of branches starting with an internal action τ, the requirement of absence of process parts on the left side of the context hole is not fulfilled in these cases.

BPEL defines the *empty* activity, that does nothing. It is used to suppress faults or provide a synchronization point for control links. Moreover, a *wait* activity delays process execution for a certain time period or until a deadline is reached. Both activities, empty and wait, are irrelevant for control flow analysis and are formalized as follows:

$$W^{\langle\rangle} = \tau \qquad W_{SE}^{\langle\rangle} = (\mathbf{v}\ ex)\langle TAR\rangle.\tau.\overline{ex} \mid \langle SRC\rangle$$

5.4 Structured Activities

Owing to the idea of nesting activities through context replacement, all structured activities contain at least one context. As introduced in the previous part, we present two formalizations for every activity, one with standard elements and one without them.

A *sequence* contains several activities, that are performed in a sequential order according to their lexical occurrence ($n \geq 1$ as at least one activity is required):

$$S^{\langle\rangle} = \{\langle.\rangle\}_n \qquad S_{SE}^{\langle\rangle} = (\mathbf{v}\ ex)\langle TAR\rangle.\{\langle.\rangle\}_n.\overline{ex} \mid \langle SRC\rangle$$

Conditional behavior in BPEL is expressed via *if-elseif-else* constructs. We abstract from the explicit condition (which might consider data values) and use non-deterministic choices to link the different branches. With n as the total number of *if*, *elseif* and *else* branches, we formalize conditional behavior as follows:

$$I^{\langle\rangle} = (\mathbf{v}\ ex)\sum^{n}(\tau.\langle.\rangle.\overline{ex}) \mid ex$$

$$I_{SE}^{\langle\rangle} = (\mathbf{v}\ ex)\langle TAR\rangle.\sum^{n}(\tau.\langle.\rangle.\overline{ex}) \mid \langle SRC\rangle$$

BPEL provides two concepts to express repetitive execution. The *while* construct enables repeated execution as long as the provided condition evaluates to true at the beginning of each iteration. Again, we abstract from the data-based condition and use a non-deterministic choice instead (please note, that W_{loop} is a separate concurrent agent):

$$\begin{aligned} W^{\langle\rangle} &= (\mathbf{v}\ ex)W_{loop} \mid ex \\ W_{SE}^{\langle\rangle} &= (\mathbf{v}\ ex)\langle TAR\rangle.W_{loop} \mid \langle SRC\rangle \\ W_{loop} &= \tau.\overline{ex} + \tau.\langle.\rangle.W_{loop} \end{aligned}$$

The second concept to express repetition is the *repeat until* construct. In contrast to the *while* construct, the encapsulated activities are executed at least once as the condition is evaluated at the end of each iteration. This leads to a slightly different formalization of the separate loop agent, defined as $R_{loop} = \langle.\rangle.(\tau.\overline{executed} + \tau.R_{loop})$.

The *pick* activity enables selective event processing. It defines a set of branches of which one is selected by the occurrence of an event or by a timer-based alarm. Formalization of the event handling equals the definition of the receive activity presented above (to keep the definition short, only the simple formalization of the receive activity is used in the following). As timing is not of importance for our control flow analysis, the alarm branches are led in by internal activities, leading to the following processes (T and S comply TAR and SRC, respectively):

$$P^{\langle\rangle} = (\mathbf{v}\ ex)\sum_{i=1}^{n}(pLpTo_i.\langle.\rangle.\overline{ex}) + \sum^{m}(\tau.\langle.\rangle.\overline{ex}) \mid ex$$

$$P_{SE}^{\langle\rangle} = (\mathbf{v}\ ex)\langle T\rangle.(\sum_{i=1}^{n}(pLpTo_i.\langle.\rangle.\overline{ex}) + \sum^{m}(\tau.\langle.\rangle.\overline{ex}))\mid \langle S\rangle$$

The *flow* construct is an expressive concept enabling concurrency and synchronization. All encapsulated activities run in parallel, while control links can be used to establish synchronization relationships. While we already discussed the formalization of these links, the following processes enable concurrency:

$$F^{\langle\rangle} = (\mathbf{v}\ ex)\prod^{n}(\langle.\rangle.\overline{ex}) \mid \{ex\}_n$$

$$F_{SE}^{\langle\rangle} = (\mathbf{v}\ h, ex)\langle TAR\rangle.\prod^{n}(\langle.\rangle.\overline{h}) \mid \{h\}_n.\overline{ex} + \tau \mid \langle SRC\rangle$$

Please note, that there is a need to synchronize all concurrent subprocesses, as the completion of all nested activities indicates the completion of the flow activity. In $F^{\langle\rangle}$ the synchronization is done via the channel ex, while another name h is needed in $F_{SE}^{\langle\rangle}$. After all subprocesses have signaled their completion through h, an interaction on ex activates the process SRC. If the whole activity is skipped, the internal action τ enables the reduction of the according process part.

Processing multiple branches with a priori runtime knowledge is done with the *foreach* construct, which exists in a concurrent and a sequential variant. Although the number of branches is determined before the processing starts, a completion condition is evaluated after the termination of each branch. If it evaluates to true, some branches are not executed (in the sequential case) or terminated (in the concurrent case). The formalization of the sequential case equals the one of the *while* construct presented above.

As our focus is restricted to behavioral compatibility and consistency checking, we are able to abstract from several aspects in regard to the parallel case. On the hand, the maximum number of iterations is not of relevance, hence it is not formalized. On the other hand, we abstract from the possibility of explicit process termination, thus the formalization of the concurrent foreach activity equals the sequential case. Theoretically, such a sequentialization might lead to deadlocks due to links crossing the activity boundaries. According to the BPEL specification, however, links must not cross the boundaries of repeatable constructs. If sequentialization should be avoided for same reason, we advice the usage of the workflow pattern *multiple instances with a priori runtime knowledge* formalized by Puhlmann [18]. However, explicit process termination cannot be realized with this pattern.

The *scope* activity is used to define a certain execution context in BPEL. Besides the definition of variables (which are not formalized due to our focus on the process flow), correlation sets, partner links and message exchanges (which have already been treated above), a scope can contain a set of handlers. These handlers are applied for all basic and structural activities contained in the scope. In general, a scope without any handlers is formalized as follows:

$$SC^{\langle\rangle} = \langle . \rangle \quad SC^{\langle\rangle}_{SE} = (\mathbf{v}\ ex)\langle TAR\rangle.\langle . \rangle.\overline{ex} \mid \langle SRC\rangle$$

Additionally, a scope can contain a set of *event handlers*, which are triggered by events or time-based alarms. In contrast to the introduced *pick* construct, multiple event handlers can run concurrently. Therefore, our formalization resembles the one of the *pick* activity except for the usage of recursion to spawn new processes. Furthermore, the event handler runs concurrently to the scope context and is deactivated after scope completion, which requires a slight adaption of the $SRC^{\langle\rangle}$ process. Thus, $SRC^{\langle\rangle}_{SC}$ equals $SRC^{\langle\rangle}$ except of the first part, as a message is sent on c before the processing proceeds with the action on src ($sk.\overline{c}.\overline{s}+ex.\overline{c}.\overline{s}$). Consequently, the following processes define a scope with $n+m$ event handlers $E_{i/j} = \langle . \rangle$:

$$SC^{\langle\rangle} = \langle . \rangle.\overline{c} \qquad SC^{\langle\rangle}_{SE} = (\mathbf{v}\ ex)\langle TAR\rangle.\langle . \rangle.\overline{ex} \mid \langle SRC_{SC}\rangle$$

$$EH = \sum_{i=1}^{n}(pLpTo_i.(E_i \mid EH)) + \sum_{j=1}^{m}(\tau.(E_j \mid EH)) + c$$

Any BPEL process is bounded by the initial *process* construct, which is similar to a scope. However, the initial process must not contain any standard elements, nor any compensation or termination handler. Therefore the process $R = \langle . \rangle$ represents the root level of the BPEL process. In the case, that memory cells are used for dynamic binding or message exchange correlations R is enhanced with a set of restricted names, the process identifiers pid_i, and a sequence of output prefixes mem_{pl} (or mem_{me} respectively) requesting the creating of the according cells. Consequently, with n as the number of needed cells and $INIT$ as the initial start activity, the root level is specified as:

$$R = (\mathbf{v}\ pid_1, \ldots, pid_n)\langle INIT\rangle.\{\overline{mem}_{pl_i/me_i}\langle pid_i\rangle\}_{i=1}^{n}.\langle . \rangle$$

6 Compatibility and Consistency Checking

In this section, we sketch how our formalization is used in regard to compatibility and consistency checking. Both require congruence based on bisimulation. Thus, we used the Advanced Bisimulation Checker (ABC) [2] to validate our findings. Further on, the processes of the example scenario in their π-calculus representation can be found in [23].

In order to decide compatibility of BPEL processes mapped to the π-calculus, we apply *interaction soundness* as introduced by Puhlmann et al [19]. However, other compatibility notions, for instance the π-calculus based compatibility notion provided by Canal et al. [4] or *weak soundness* as defined by Martens [12], could also be applied. As the notion presented by Canal et al. is restricted to bi-lateral communication and *weak soundness* is defined for Petri nets and lacks support for multiple process instantiation, we focus on interaction soundness in the following.

Interaction soundness is defined for a process P_k of a process composition $SYS = \prod_{j=1}^{n} P_j$ in an environment $E_{P_k} = (\prod_{j=1}^{k-1} P_j | \prod_{j=k+1}^{n} P_j)$ (the environment contains all other subprocesses). It requires the unification, denoted as $P \uplus E$, to be lazy sound. The system consisting of P_k and E_{P_k}, denoted as $SYS_{P_k} = (P_k | E_{P_k})$, is enhanced with the free name i for the initial activity and the free name $\overline{o}$ for the final activity. The derived annotated system, denoted as $ASYS_{P_k}$, is then checked for weak open bisimulation equivalence with $S_{LAZY} = i.\tau.\overline{o}.\mathbf{0}$.

Regarding the example introduced in Section 2 (Figure 1), ABC decides weak open bisimulation equivalence on S_{LAZY} and the annotated system $ASYS_1$, thus deciding interaction soundness for the customer process $CUST$:

```
abc > agent S_LAZY(i,o)=i.t.'o.0
Agent S_LAZY is defined.
abc > weqd ASYS_1 SLAZY
The two agents are weakly related (916).
```

In the same manner, interaction soundness is decided for the other subprocesses, which proves the composition to be interaction sound and therefore compatible. In contrast, there is no weak open bisimulation equivalence, if the stockbroker process is replaced as discussed in Section 2, leading to a new definition $ASYS_2$ of the annotated system:

```
abc > weqd ASYS2_2 SLAZY
The two agents are not weakly related (3).
```

Concerning consistency between an abstract behavior description and a process implementation, we apply weak open bisimulation equivalence. Decker et al. [5] discussed the weaknesses of bisimulation equivalence as a consistency relation, however, an alternative is to be presented. Concerning our example, the process $CUST$ specifies the behavior of a customer, while $CUST_A$ and $CUST_B$ are different process implementations. ABC decides weak open bisimulation equivalence as follows:

```
abc > weqd CUST CUST_A
The two agents are not weakly related (3).
abc > weqd CUST CUST_B
The two agents are weakly related (11).
```

Consequently, $CUST_B$ is a valid process implementation of $CUST$, while $CUST_A$ is inconsistent regarding $CUST$. While weak open bisimulation can handle the addition of an internal action (in $CUST_B$), the parallelization of activities in $CUST_A$ violates the bisimulation equivalence, although $CUST_A$ is a consistent process implementation for $CUST$. That results from the above mentioned limitations of bisimulation equivalence in the field of consistency checking.

7 Complexity of Formalization Approaches

As we focus on compatibility and consistency checking, the introduced BPEL mapping abstracts from several aspects (as stated in Section 5) in order to enable a lean formalization. In this section we shortly discuss the consequences of simplification regarding the process state space. It was already mentioned that common formalization approaches often support a certain subset of the BPEL specification. For this reason, we consider the most simple process of our example, namely the customer process, in our complexity analysis.

Table 1 lists the number of states, the customer process traverses when deployed in a compatible environment. Regarding the three approaches applying process algebras (for which we calculated the state space manually), the small number of states with our approach derives from the neglect of internal activity states. Thus, we do not model the activity lifecycle, while Mazzara and Lucchi require every activity to signal at least its termination, leading to more than twice the number of process states. Moreover, Fadlisyah implements explicit activation and termination for each activity. Hence, two additional process states originate from every process activity. While these effects lead to a linearly increasing number of states in sequential settings, the impact is worse for concurrent activities. Parallelism of internal states results in an exponential dependency of the evolving state space from the number of activities, thus state explosion. In contrast, our approach activates and terminates activities implicitly, avoiding additional process states at the expense of missing error handling.

	FORMALISM	# STATES
Mazzara, Lucchi [13]	$web\pi_\infty$/π-calculus	17
Fadlisyah [6]	π-calculus	23
Stahl [21, 10] (init)	Petri nets	16
Stahl [21, 10] (comm.)	Petri nets	6
Stahl [21, 10] (reduced)	Petri nets	4
Our formalization	π-calculus	6

Table 1. State space of the customer process in different formalization approaches

Regarding the Petri net based approach presented by Stahl et al., the same effect can be determined. We used the tool BPEL2oWFN [11] to derive the Petri nets from the BPEL process of the customer. The initial formalization grounds on an extensive model capturing all contingencies, leading to a complex process representation. Nevertheless the state space of the evolving process is bounded to 16 process states (determined by the tool Integrated Net Analyzer [22]), as the initial net contains various dead transitions. The second representation, which is of the same complexity as our approach, abstracts from all data flow and error handling related constructs. Due to the definition of several heuristics to simplify the model, Stahl's third formalization has a once more reduced number of process states. Nevertheless, Stahl's approach cannot be applied in order to verify process consistency with dynamic binding as introduced in our example in Section 2. Thus, our formalization supports a wider field of application by offering the same level of efficiency.

Although the example process has a rather simple and sequential structure, the high relevance of an appropriate abstraction level is evidently even in this case. The prevention of unnecessary exponential multiplication (in concurrent settings) of the evolving state space is a prerequisite in order to enable process verification for real world scenarios.

8 Conclusion

This paper motivates the need for verification of BPEL processes, especially regarding compatibility of certain processes in a composition and consistency of process implementations to abstract behavior specifications. Therefore, we presented a formalization of BPEL 2.0 constructs based on the π-calculus and showed how it can be applied in process verification scenarios. We explicitly illustrated compatibility and consistency checking by means of an example and validated our findings using the Advanced Bisimulation Checker.

Due to our focus on behavioral analysis, we have been able to present a very compact and lean formalization. We achieved a reduced state space through abstracting from modeling the activity lifecycle. A comparison of our formalization with other approaches based on process algebras demonstrates, that we are able to reduce the state space significantly, resulting in decreased effort needed for process verification. Therefore, our formalization competes with the highly optimized Petri net based approach presented by Stahl et. al, which benefits from extensive research in the field of Petri net reduction. In contrast, the definition of structural transformation rules in regard to the optimization of π-processes (e.g. the transformation of multiple sequential τ activities to one τ activity), has not been covered in previous research and remains an open issue.

Further future work remains to extend our formalization concerning message correlation and the error handling framework. Owing to our neglect of internal activity states, any enhancement regarding fault and compensation handlers requires much effort. Nevertheless, a mechanism to encapsulate visible behavior by surrounding processes instead of explicit process termination could emerge as a concept to support the error handling framework without increasing the number of states drastically. Moreover, the application range of our formalization is to be extended, for instance regarding reachability analysis.

References

[1] J. Arias-Fisteus, L. S. Fernández, and C. D. Kloos. Formal verification of bpel4ws business collaborations. In *EC-Web*, volume 3182 of *LNCS*, pages 76–85. Springer, 2004.

[2] S. Briais. Advanced Bisimulation Checker (ABC). http://lamp.epfl.ch/∼sbriais/abc/abc.html, 2007.

[3] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal Semantics and Analysis of Control Flow in WS-BPEL. Technical report, BPM Center Report BPM-05-15, 2005.

[4] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and inheritance in software architectures. *Sci. Comput. Program.*, 41(2):105–138, 2001.

[5] G. Decker and M. Weske. Behavioral Consistency for B2B Process Integration. In *CAISE, Trondheim, Norway*, 2007.

[6] M. Fadlisyah. Using the π-Calculus for Modeling and Verifying Processes on Web Services. Master's thesis, Insitute for Theoretical Computer Science, Dresden University of Technology, 2004.

[7] D. Fahland. Formal Operational Semantics of BPEL4WS. Informatic Berichte 190, Humboldt-Universitt zu Berlin, 2005.

[8] D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative Control Flow. In *12th International Workshop on Abstract State Machines*. Lecture Notes in Computer Science. Springer-Verlag, March 2005.

[9] A. Ferrara. Web services: A process algebra approach. *CoRR*, cs.AI/0406055, 2004.

[10] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In W. M. P. v. d. Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *BPM*, volume 3649 of *LNCS*, pages 220–235, Nancy, France, Sept. 2005. Springer-Verlag.

[11] N. Lohmann, C. Gierds, and M. Znamirowski. BPEL2oWFN. http://www.gnu.org/software/bpel2owfn/, 2007.

[12] A. Martens. On Compatibility of Web Services. *Petri Net Newsletter*, 65:12–20, 2003.

[13] M. Mazzara and R. Lucchi. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 2006. To appear.

[14] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[15] OASIS. Web Services Business Process Execution Language Version 2.0, April 2007.

[16] OMG. Business Process Modeling Notation (BPMN) Specification Version 1.0, February 2006.

[17] F. Puhlmann. Why do we actually need the pi-calculus for business process management? In *BIS*, volume 85 of *LNI*, pages 77–89. GI, 2006.

[18] F. Puhlmann and M. Weske. Using the pi-calculus for formalizing workflow patterns. In *BPM*, volume 3649, pages 153–168, 2005.

[19] F. Puhlmann and M. Weske. Interaction Soundness for Service Orchestrations. In A. Dan and W. Lamersdorf, editors, *ICSOC*, volume 4294 of *LNCS*, pages 302–313. Springer Verlag, December 2006.

[20] D. Sangiorgi. A Theory of Bisimulation for the pi-Calculus. *Acta Informatica*, 16(33):69–97, 1996.

[21] C. Stahl. Transformation von BPEL4WS in Petrinetze. Diplomarbeit, Humboldt-Universitt zu Berlin, Apr. 2004.

[22] P. H. Starke and S. Roch. Integrated Net Analyser. http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/, April 1999.

[23] M. Weidlich and G. Decker. Efficient Analysis of BPEL 2.0 Processes using Pi-Calculus - Example Processes. http://bpt.hpi.uni-potsdam.de/twiki /pub/Public/GeroDecker/bpel2piprocesses.pdf, 2007.