

Simulation and Equivalence between BPEL Process Models

Axel Martens
IBM T. J. Watson Research Center,
Component Systems Group
E-mail: amarten@us.ibm.com

Abstract

The integration of business process across the boundaries of individual enterprises or business units is becoming increasingly important. In this scenario, process models play an all-important role. On the one hand, the interaction between the participating companies often is specified globally, for example by means of multiple abstract process models – one for each partner. On the other hand, each partner defines its local process autonomously in terms of an executable process model. The important question is whether such an executable model is consistent to the predefined abstract model. This paper defines a simulation relation between BPEL process models, and presents a method to verify consistency automatically, on top of it.

Emerging technologies and industrial standards in the field of *Web services* enable a much faster and easier cooperation of distributed partners. A *Web service* [1] is a self-describing, self-contained modular application that can be described, published, located, and invoked over a network. One self-evident application of Web services is the support of distributed business processes: Each local sub-process of each participating company is realized through one Web service. The composition of all Web services of all participating companies realizes the global business process.

Instead of *one* new specific technology, the Web service approach provides a stack of closely related technologies [4]. Among other things, the *Business Process Execution Language for Web Services* BPEL4WS [2] belongs to this stack. Due to this layered architecture, the presented analysis method can be focussed on the Web service's process model in terms of BPEL4WS – without losing generality or practical relevance.

The goal

Process modeling is one of the most crucial tasks while developing or integrating enterprise applications. In practice, there are various levels of abstraction to model a business process: A coarse grained model vi-

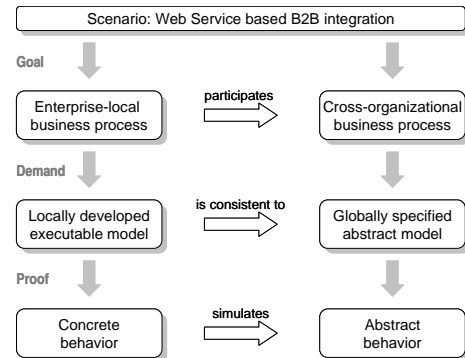


Figure 1. Field of application

ualizes the overall dependencies between the activities. Usually, it is refined in several steps during the development process. Finally, a very precise model, covering all necessary aspects, can be executed directly by help of a workflow engine. Hence, such a model is called *executable* model, whereas each model of a prior stage can be called “*abstract*” model in some sense. BPEL4WS can be used to express both: the abstract process model as well as the executable process model.

Referring to Web service based business processes, for each local sub-process there is already one executable BPEL process model. But, the distributed global process itself has to be modeled, as well. Such a *global model* is required, especially, to ensure the compatibility or to specify the interaction in terms of a contract between the partners. And, although the global process is realized through composition of multiple local sub-processes, to establish the global model, however, it is not adequate to simply combine the local *executable* models of the participating partners (e.g. to keep business secrets). Instead, the global model should aggregate one *abstract* model for each partner.

If both models are given for a certain partner, an important question is whether the executable model is *consistent* to the abstract model. Consider for example Web portal, which is offering various online shopping

services to potential customers. This portal has defined an abstract process model such that each customer can structure his own process accordingly to this model. If a new shop wants to join this portal, its process should interact with each customer properly, i.e. the shop's executable process model *has to be* consistent to the predefined abstract process model – Figure 1 shows this correlation. This paper describes an approach to verify the consistency between process models automatically.

The method

As this paper will show, very differently structured process models might be called consistent. Hence, the presented method focusses on the *behavior* rather than the structure: An executable process model is called to be *consistent* to an abstract process model if and only if the concrete behavior *simulates* the abstract behavior. By this means, each component that was built accordingly to the abstract model can't see any difference while interacting with the executable process.

The analysis of consistency runs in three steps. First, a precisely and explicitly defined semantics of the modeling language is needed. This approach is based on *Petri nets* [18], because this formal method is widely used for modeling and analyzing business processes and Web services [25, 5, 6]. The second step extracts the relevant information and generates the *communication graph* [13] – an explicit representation of the process' external visible behavior. Rest upon this necessary abstraction, the third step verifies the *simulation* between concrete and abstract behavior by comparing the corresponding graphs. All three steps are performed automatically by the tool WOMBAT4WS [30]. Hence, the user only has to deal with BPEL process models. Moreover, the method could easily be extended to other modeling languages (e.g. WS-CDL [7], OWL-S [28] or YAWL [23]), by adjusting the first step only.

The remaining paper is structured according to the presented method: Section 1 delves into the already mentioned use case of the Web portal, and presents *abstract* and the *executable* BPEL process models. Section 2 gives a very brief introduction in some Petri net notions, and sketches the mapping of BPEL4WS into this formalism. Section 3 describes the extraction of the *behavior* of a process model and discusses its comparison. The proposed definition of *simulation* is presented in Section 4. Finally, Section 5 summarizes the results and relates them to other published approaches.

1 Examples

A Web portal offering customers access to various online is used as a running example through this paper. The language BPEL4WS allows to define abstract pro-

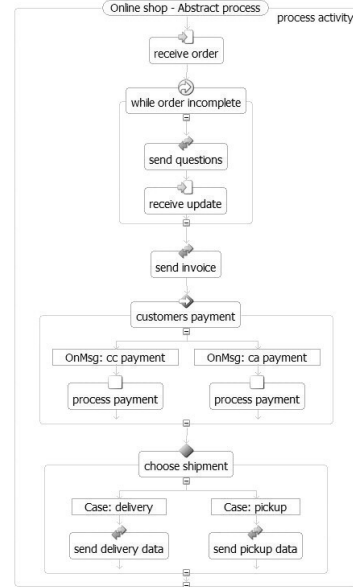


Figure 2. Abstract BPEL process model

cess models as well as executable models. Basically, an executable model is required to be completely specified, e.g. all variables have to be declared and used properly. In contrast, an abstract model may leave out some of those aspects (cf. [10]). Although this paper refers to these notions, it focusses on the semantic distinction rather than the syntactic difference: An executable models contains more details than the abstract model does. Hence, the notion 'executable model' could be replaced by the notion 'refined model', if preferred.

1.1 The abstract process model

The business strategy of the Web portal has two aspects: On one hand, the portal is open to all online shops. On the other hand, the portal requires the participating shops to build their services according to a standardized protocol, specified in terms of an abstract BPEL process model. Figure 2 shows this model.

Basically, a BPEL model consists of two different kinds of activities: *Basic activities* are used to communicate to the outside (e.g. *receive order*, *send questions* in Figure 2), to perform internal steps (e.g. *process payment*) or to interfere with the control flow (e.g. by signaling fault and process termination – not present in the example). *Structured activities* aggregate other activities. Therefore, they are used to build the control structures of the process, i.e. alternative branches – either based on incoming messages (*customers payment*) or based on data (*choose shipment*) – loops (*while order incomplete*), sequential concatenation (*process activity*), and parallel branches (*process activity* of Figure 3).

The process shown in Figure 2 models the follow-

ing behavior: First, the customer should place his order. Then, the shop may send zero or more questions to the customer concerning his order and await his update. Eventually, the shop sends the invoice and requires the customer to pay, either with credit card (cc payment) or out of his checking account (ca payment). Finally, the shop finishes the process by sending the delivery data or pickup data accordingly to its business strategy. The formats and the channels of messages being exchanged are defined in the *Web service description WSDL* [1].

1.2 The executable process model

Figure 3 shows the executable process model of an online shop that wants to participate in the Web portal. Compared to the abstract model, this process follows a different strategy. First, it offers to the customers a product query that may precede the actual order, and second, no pickup of ordered products is available. Due to this business strategy, the structure is quite different compared with the abstract process. Nevertheless, this process model can be proven to be consistent.

The executable process is structured into two concurrent activities: *customers initial choice* and *order processing*. But, these activities are synchronized by two *links*. Hence, the *order processing* is started only if the customer send an *order* right at the beginning or he sends a *request*, the product is available, and he decides to order after he got the confirmation.

Assuming the customer has ordered a product, he either gets the invoice or he is asked questions concerning his order exactly once. In contrast, the abstract process model allows an arbitrary number of callbacks. The payment is handled the same way in both processes. Finally, the executable process sends the *delivery data*.

In spite of the many differences, it is not difficult to see: Each customer, who behaves properly according to the abstract process (i. e. he behaves like an *compatible environment* of this Web service, cf. Section 2.1), has no problems to cooperate with the executable process (i. e. he behaves like an compatible environment of that Web service, too). But there is no obvious way to prove this property just by syntactical matching.

In fact, the executable process follows a common business strategy for smaller companies: On the one hand, it respects the rules made by a company with larger market power, but on the other hand, it offers additional services to attract more customers. Hence, it is worth supporting to this use case methodically.

2 Formalization

The *Business Process Execution Language for Web Services* BPEL4WS [2] is in the very act of becoming the industrial standard for modeling Web service based

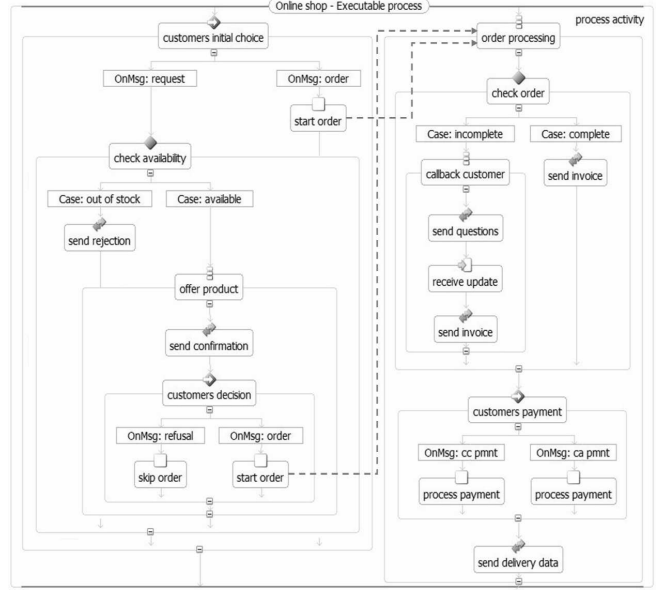


Figure 3. Executable BPEL process model

business processes. But, its semantics so far is defined only by English prose or encoded into middleware components, more ore less accurately. To verify properties like the consistency of two models, a formal semantics of all its concepts is needed. The presented method is based on a Petri net semantics [21].

2.1 Modeling with Petri nets

Petri nets are a well established method for modeling and analyzing (cross-organizational) business processes [24, 26, 8]. Moreover, beside the already mentioned semantics for BPEL4WS, other recent research projects apply Petri nets to Web Services [6].

Figure 4 shows the Petri net model of the abstract order process that was generated automatically by the tool WOMBAT4WS [30]. A *Petri net* (P, T, F) consists of a set of *transitions* T (boxes), a set of *places* P (ellipses), and a *flow relation* F (arcs) [18]. A transition represents a dynamic element, i. e. an activity of a business process. A place represents a static element, i. e. the causality between activities or a message channel. The state of a Petri net is represented by a set of black *tokens* distributed over the places, cf. place p1.

A Web service possesses an internal structure and an interface to communicate with other Web services. Hence, the Petri net model of such a Web service consists of a *workflow net* [25] supplemented by set of places representing directed message channels. Such a model is called a *workflow module* [13]. Two modules are composed by merging all common interface places (matching by name). The dangling input and output places become the new interface of the composed mod-

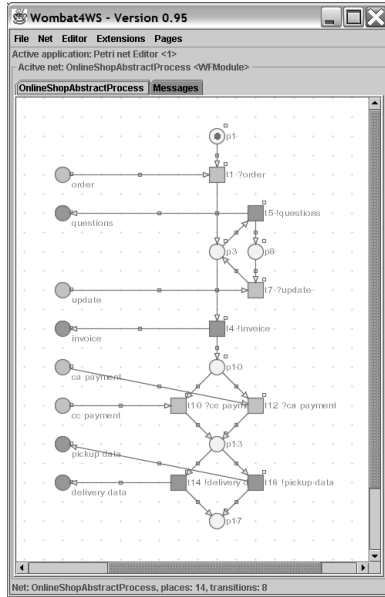


Figure 4. Petri net model of the abstract process

ule. Additionally, some net elements for initialization and termination are added. If both sets of interface places completely match, one module is called *environment* of the other (obviously, this notion is symmetric). A Module M is called *usable* if there exists at least one environment U such that the composed system $M \oplus U$ is a sound process model (i.e. the Petri net must be *weak-sound*). Such an environment is called an *utilizing environment* (cf. [13]). Additionally, an utilizing environment must never send a message towards the module that can't be accepted in the current state (referring to the information the environment can derive, cf. Section 3.2). This requirement is reasonable: In practice, no customer would send money to a shop before the availability of the ordered product was confirmed. Moreover, the algorithm presented in [13] generates such an environment automatically and the proof of simulation will become much simpler.

Based on the notion of a workflow module, several analysis methods are now applicable to BPEL processes models: the verification of *usability* of one Web service [13], the verification of *compatibility* of two Web services [12], the *automatic generation* of an abstract process model for a given Web service [14], and the verification of *consistency* – presented in this paper.

2.2 Transforming BPEL4WS into Petri nets

The transformation of a BPEL process into a Petri net is based on the semantic definition described in [21]. Each element of the source language is represented by a modular Petri net pattern. Hence, the structure of a BPEL process is still visible in the resulting Petri net.

The approach presented in this paper (and implemented in WOMBAT4WS) differs from the complete semantic definition through its abstraction from data aspects, i.e. each data driven decision is mapped into a non-deterministic choice. This is a usual procedure in the field of computer aided verification. On the one hand, a model without data has the disadvantage of less precision (i.e. a more general behavior). But on the other hand, this enables analysis methods yielding important results that are not applicable to models with full expressive power of arbitrary data objects. Because of the possible loss of information due to abstraction, WOMBAT4WS provides the feature of user interaction. On specific points that are crucial for the process' behavior, the user may add structure to the Petri net, for example by translating join and transition conditions (cf. *link semantics*). Thereby, data dependencies are transformed into control dependencies.

Another feature of WOMBAT4WS is syntactical reduction. By transforming a BPEL process step by step, the resulting Petri net may contain many elements that have no direct impact on the externally visible behavior (e.g. sequences of internal activities). Those structures are needed to understand the concepts of BPEL4WS, but their presence in the Petri net impedes an efficient analysis. Hence, those structures are eliminated by application of reduction rules that are based on the syntax of Petri nets (cf. [3]). Figure 4 shows the resulting Petri net model of the abstract order process. The net contains only eight transitions (the BPEL process model contains 12 activities). Still, its structure can easily be mapped to the original process model, shown in Figure 2. The next paragraphs take a closer look to the transformation, especially on the link semantics.

Basic activities: Each communicating activity is represented by one transition (or two transition in case of a synchronous invoke-activity) connected to the corresponding interface place (e.g. `receive order` \rightarrow `?order`). All other activities (e.g. `process payment`) are transformed into an internal transition. In most cases, they are eliminated applying the reduction rules. The handling of failures and compensation is not covered by the current implementation. In future work, those activities, which interfere with the control flow (e.g. `throw`), will be transformed in a more detailed manner.

Structured activities: An alternative branch is transformed into a set of transitions that share at least one preceding place – called *conflicting transitions*. The conflict is either solved by external messages (`customers payment` \rightarrow `?cc payment`, `?ca payment`), or by non-deterministic choice (`chose shipment` \rightarrow `!delivery data`, `!pickup data`). A loop (`while order incomplete`) is transformed by merging the preceding place of the loop's

Listing 1. BPEL code of a linked activity

```

...
<flow>
  <link name="link1"/>
  <link name="link2"/>
  <link name="link3"/>
  <link name="link4"/>
  ...
  <switch ...>
    <case ...>
      <sequence name="linkedActivity"
        joinCondition="link1 AND link2">
        <target linkName="link1"/>
        <target linkName="link2"/>
        <source linkName="link3"
          transitionCondition="p(...)/>
        <source linkName="link4"
          transitionCondition="NOT p(...)/>
        ...
      </sequence>
    </case>
  </switch>
  ...
</flow>
...

```

first activity with the succeeding place of the loop's last activity. Finally, a sequence of activities is represented by the sequential concatenation of one Petri net pattern for each of the activities. The parallel branch is a more difficult concept of BPEL, because there might be *links* that define dependencies between concurrent activities. Besides the links, two transitions are needed, to split the control flow into concurrent threads at the beginning and to join them again at the end.

Link semantics: Each activity within a flow can be source and/or target of several links. Listing 1 shows a piece of a BPEL process model. The structured activity `linkedActivity` has two incoming and two outgoing links. Figure 5 shows the corresponding Petri net pattern for this activity. Each link is transformed into two places reflecting the boolean value of that link. Before the activity `linkedActivity` (activity pattern) can be executed, all incoming links have to be evaluated with respect to the *join condition*. In Listing 1, the join condition *AND* is defined, which is modeled by the *join pattern*. In general, each boolean condition over n links could be expressed by 2^n transitions. Instead of reflecting the individual *transition condition* of each outgoing link, in this approach, a global split behavior is assigned to the activity. In Listing 1, the two transition conditions are mutually exclusive. Hence, the *split pattern* in Figure 5 models *XOR*. Again, by help of up to 2^n transitions, each boolean combination can be expressed. If join or transition condition impede the automatic construction, the user is ask to select the desired pattern.

In case the join condition was evaluated to *FALSE*, all outgoing links have to be set to *FALSE*, too. Additionally, all outgoing links have to be set to *FALSE* if the

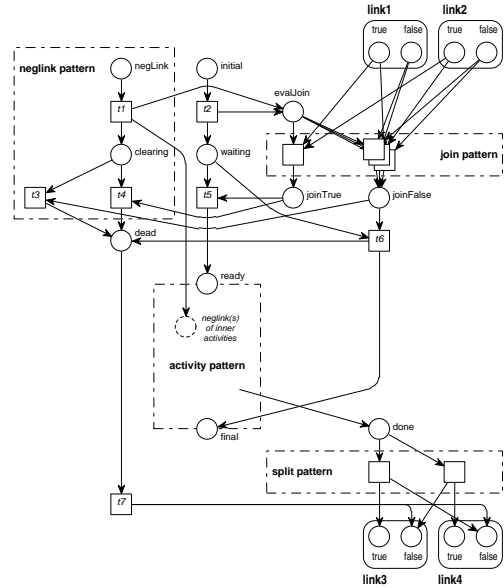


Figure 5. Transformation of a linked activity

activity lies on a path of an alternative branch that was not chosen. In that case, the activity will not be activated. Instead, it will get a token on the place `negLink` (abbr. for: *propagate negative link values*). The *neglink pattern* consumes the incoming link information, sets all outgoing links to *FALSE* and propagates the `negLink`-token (if necessary) towards the embedded activities. This concept is called *death-path-elimination* (cf. [9]).

The current implementation of WOMBAT4WS does not yet consider *event handling* and *fault handling*, but these concepts will be added soon. In contrast, the integration of transactional behavior and *compensation handling* is not indented until the standardization of all related Web service technologies is completed.

3 Behavior

Based on the underlying Petri net formalism, it is now possible to reason on the behavior of a BPEL process with well defined notions of current state, enabled activities and occurrence of transitions. Intuitively, one process simulates another process if in a *comparable situation* the first process *behaves like* the second process. There are countless approaches published dealing with simulation or equivalence, respectively. This variety results from different interpretations of the terms “comparable situation” and “behave like” ([27] gives a substantial overview). But, none of them fits exactly to this field of application, i. e. none of them allows an executable process to accept more messages and send less messages (w. r. t. the abstract process). Figure 6 shows four workflow modules – inspired by [16] – to

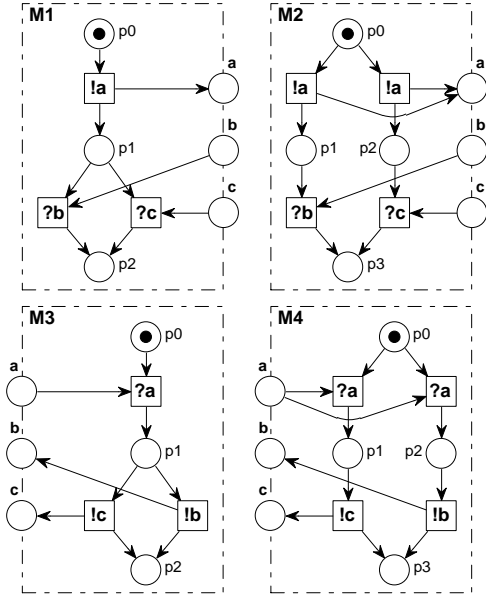


Figure 6. Examples of (non-)equivalent modules

demonstrate the difference between various notions of simulation or equivalence. The following paragraphs compare two of them, and presents the preferred definition at the end.

3.1 Comparing behavior

Trace equivalence: Two communicating systems could be compared by looking from the outside only at the sequences of messages being exchanged – called *traces*. From this point of view, the modules M1 and M2 both simulate each other, because they have identical traces a,b and a,c. But, module M3 is an utilizing environment of module M1, whereas the composition of M3 and module M2 yields a deadlocking system. Moreover, it is possible to prove the non existence of any utilizing environment for module M2. Hence, the requirement of trace equivalence is too weak for our purpose.

Bisimulation equivalence: The opposite approach compares the internal states and enabled transitions of both modules: A state z_s simulates a state z if each transition (identified by its label), which is enabled in z , also is enabled in z_s , and the reached state z'_s simulates the reached state z' . Two modules are equivalent if their initial states simulate each other. In that sense, module M4 does not simulate module M3, because neither state [p1] nor state [p2] of module M4 simulates the state [p1] of M3. But, module M1 is an utilizing environment of module M3 and an utilizing environment of M4 as well. Moreover, it is possible to prove that each environment utilizes either both modules or none. Hence, the requirement of bisimulation equivalence is a too restrictive for our purpose.

This short discussion shows that an adequate notion of simulation (or equivalence, resp.), first of all, should be derived semantically from the field of application. In a second step, a proving method can be developed.

Definition 3.1 (Simulation/Equivalence).

A workflow module A *simulates* a workflow module B if each utilizing environment of module B is an utilizing environment of module A , too. Two workflow modules A and B are called *equivalent*, if module A simulates module B and module B simulates module A . *

This definition exactly meets the requirement of the cross-organizational business process scenario: Lets consider a workflow module M and an utilizing environment E . For each module M' simulating M and for each module E' simulating E holds: E' is an utilizing environment of M' and vice versa. This property follows directly from Definition 3.1. The remainder of this paper describes the verification of simulation.

3.2 Reflecting behavior

As discussed in the previous paragraph, a level of abstraction is needed that still combines the internal and the external views on a workflow module. A Web service is a reactive system; it consumes messages from the environment and produces answers depending on its internal state. But, the environment has no *explicit* information on the service's internal state. Instead, each environment can derive some *implicit* information by considering the history of communication. Exactly that kind of information is reflected within a data structure – called the *communication graph* (abbr. *c-graph*).

Definition 3.2 (communication graph/c-graph).

A communication graph $((V, H, E), m)$ is a directed, strongly connected, labeled, bipartite graph such that:

- The graph has two kinds of nodes: *visible* nodes V and *hidden* nodes H . Each edge $e \in E$ connects two nodes of different kinds.
- The graph has a definite root node $v_0 \in V$, each leaf is a visible node, too.
- The labeling m maps each visible node to a set of states of the corresponding Petri net, and each edge to a bag of messages. *

Figure 7 shows the c-graph of the abstract BPEL process shown in Figure 2. The labeling of visible nodes was omitted, because it is not needed to verify simulation. Each edge, starting at a visible node, is labeled with a bag of messages sent by the environment – called *input*. In the root node a1, the module is able to consume only the message *order*. Each edge, starting at a hidden node, is labeled with a bag of messages sent by

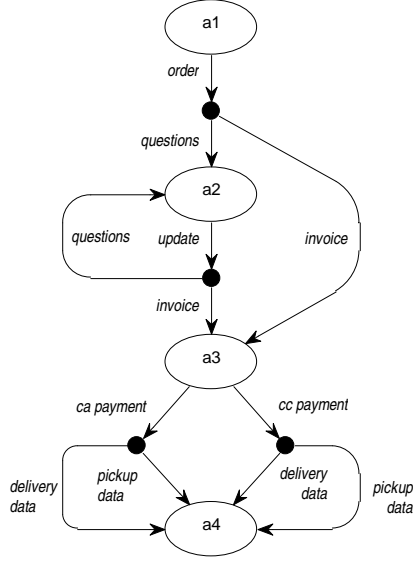


Figure 7. C-Graph of the abstract process

the module – called *output*. In the example, the abstract process answers the *order* by sending either the *questions* message or the *invoice*. Each path from the root to a leaf represents a complete communication sequence between module and environment. In general, an edge may be labeled with more than one message, but this is not present in the chosen examples. The c-graph of a given workflow module is well defined, and it can be calculated based on the following notions:

Activated input: Referring to a given state, an activated input is a bag of input messages that are consumed by the module along a firing sequence, whereas no output message was produced and the firing sequence ends either in the module’s final state or in a state that enables a transition that can produce an output message. The function INP yields the set of activated inputs.

Successor state: Referring to a given state, a successor state is a maximal reachable state w. r. t. one possible behavior (i. e. one *concurrent run* [19]) of the module. The function NXT yields the set of successor states.

Possible output: Referring to a given state, a possible output is the bag of output message that was produced while reaching one successor state. The function OUT yields the set of possible outputs.

Communication step: A four-tuple (z, i, o, z') is called communication step if z, z' are states of a module, i is an input and o is an output, and $(z' + o)$ is a successor state of $(z + i)$. $\mathcal{S}(M)$ denotes the set of all communication steps of a module M .

As shown in Figure 7, the c-graph may contain cycles. That doesn’t affect the presented analysis method as long as the graph is finite, which is the case for all

reasonable workflow modules. The precise, mathematical definition of all notions mentioned above, a discussion on the complexity of the algorithm and possible optimizations, and the problem of infinite graphs is discussed in [15]. Applying these notions, the following algorithm constructs the c-graph. The algorithm starts with the root node v_0 labeled with the initial state:

1. For each state within the label of v_k calculate the set of activated inputs: $\bigcup_{z \in m(v_k)} \text{INP}(z)$.
2. For each activated input i within this set:
 - a) Add a hidden node h , add a new edge (v_k, h) with the label i .
 - b) For each state within v_k ’s label calculate the set of possible outputs: $\bigcup_{z \in m(v_k)} \text{OUT}(z + i)$.
 - c) For each possible output o within this set:
 - i. Add a visible node v_{k+1} , add a new edge (h, v_{k+1}) with the label o .
 - ii. For each state $z \in m(v_k)$ and for each communication step $(z, i, o, z') \in \mathcal{S}(M)$ add z' to the label of v_{k+1} .
 - iii. If there exists a visible node v such that $m(v_{k+1}) = m(v)$ then merge v and v_{k+1} . Otherwise, goto step 1 with node v_{k+1} .

The c-graph of a module contains the maximal information an environment can derive. In general, this graph may contain further leaf nodes. But in each c-graph there is at most one leaf node, which is labeled with the defined final state of the workflow module. All other leaf nodes contain at least one state, where there are messages left or which marks a deadlock within the module. That means that if an environment was communicating with the module according to the labels along the path towards such a leaf node, this environment would not be an utilizing environment. The elimination of all such erroneous sequences yields a (possibly empty) subgraph that can be regarded as the user manual of the module – called the *usability graph* (abbr. *u-graph*). A workflow module is usable if and only if it has a non-empty u-graph (cf. [13]). In both examples, the whole c-graph is an u-graph, too.

4 Simulation

Figure 8 presents the c-graph of the executable process model (cf. Figure 3). Comparing it manually with the graph shown in Figure 7, it is easy to figure out many similarities. In the initial node of both graphs, the customer may send the message *order*, and he will receive either the message *questions* or the message *invoice*. In case of the message *questions*, the customer should send the message *update* and may receive the message *invoice*.

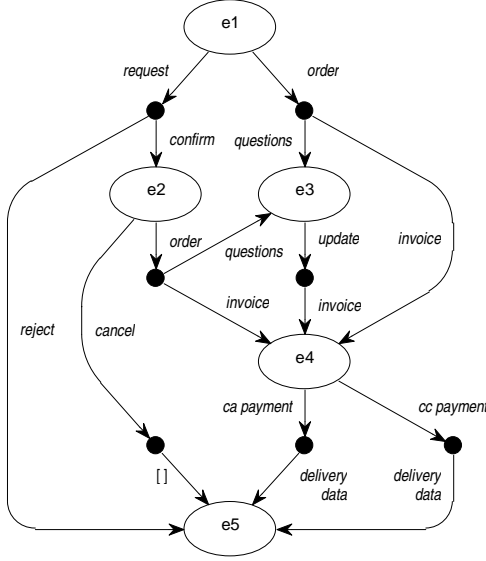


Figure 8. C-Graph of the executable process

Finally, the customer has to pay, and the process may answer by sending the message *delivery data*.

But, there are of course many differences, too. For example, the executable process additionally accepts the message *request* at the very beginning, and it never sends the message *pickup data* to the customer. If the customer sends a *request* and cancels the process after he got the message *confirm*, the process terminates without further feedback (denoted by the empty bag []).

Hence, the executable process does not behave exactly like the abstract process. But, all differences fit a common pattern: Either a visible node has more outgoing edges than the other graph (possibly followed by structures that are not present in the other graph), or a hidden node has less outgoing edges than the other graph. Because of that, at least all those messages a utilizing environment would send to the abstract process could also be accepted by the executable process. The possible answers of the executable process are at most those, the abstract process would send to the environment. Consequently, such an environment can't see any difference between the two processes.

4.1 Definition of simulation

To define simulation between two c-graphs precisely, some functions to navigate through those graphs are needed. Let $((V, H, E), m)$ be a c-graph of a workflow module, and let $(V_u, H_u, E_u) \subseteq (V, H, E)$ be a u-graph.

Path: A sequence of visible nodes $(v_0, \dots, v_n) \in V^*$ is called a *path* if for each two nodes v_k, v_{k+1} there is a hidden node $h \in H$, such that $(v_{k-1}, h), (h, v_k) \in E$. Considering a path, the labels of the edges are addressed as follows: $i_k = m(v_{k-1}, h)$ and $o_k = m(h, v_k)$.

Acceptable inputs: Given a node $v \in V_u$, an *acceptable input* is a bag of input messages that can be consumed along a path in the *usability graph*, starting at v and generating no output message in between. $AI(v)$ denotes the set of acceptable inputs in v .

$$AI(v) = \left\{ i = \sum_{k=1}^n i_k \text{ where } (v, v_0, \dots, v_n) \in V_u^* \right. \\ \left. \text{is a maximal path w. r. t. } [] = \sum_{k=1}^{n-1} o_k \right\}$$

Maximal paths: Given a node $v \in V_u$ and an acceptable input $i \in AI(v)$, a *maximal path* within the c-graph consumes at most those messages being member of i . $MP(v, i)$ denotes the set of maximal paths.

$$MP(v, i) = \left\{ (v, v_1, \dots, v_n) \in V^* \text{ such that } \right. \\ \left. (v, v_1, \dots, v_n) \text{ is a max. path w. r. t. } i \geq \sum_{k=1}^n i_k \right\}$$

Definition 4.1 (Simulation).

A c-graph $C(E)$ *simulates* a c-graph $C(A)$ if both graphs contain non-empty u-graphs, and the root node of $C(E)$ simulates the root node of $C(A)$. A visible node e of $C(E)$ *simulates* a visible node a of $C(A)$ if the following requirements are fulfilled:

- (i) For each input $i \in AI(a)$, for each path $(e, \dots, e_n) \in MP(e, i)$ there is a path $(a, \dots, a_m) \in MP(a, i)$ such that $\sum o_k^e = \sum o_k^a$ and e_n simulates a_m .
- (ii) If $AI(a)$ is empty, then $AI(e)$ is empty, too.
- (iii) For each input $i \in AI(e)$ if there is an input $i' \in AI(a)$ such that $i \leq i'$, then $i \in AI(a)$. *

Requirement (i) has been motivated, already: If a module E shall simulate a module A , then E has to accept *at least* those messages A accepts, and E has to produce *at most* those messages A produces. The proper simulation is guaranteed by requirement (ii). Finally, the module E must not respond to a smaller bag of messages than A does, e. g. by sending a product without receiving the complete payment. Such behavior can yield to a deadlock, e. g. a customers may refuse to complete the payment after receiving the product.

4.2 Theorem of simulation

The following theorem correlates the simulation of c-graphs and the simulation of workflow modules.

Theorem 4.1 (Simulation).

A workflow module E simulates a workflow module A iff the c-graph $C(E)$ simulates the c-graph $C(A)$. *

Proof (Theorem 4.1): Let E and A two arbitrary workflow modules, and let $C(E)$ and $C(A)$ be the corresponding c-graphs.

Assume, $C(E)$ simulates $C(A)$. To prove the simulation between E and A , we consider an arbitrary utilizing environment U of A . To utilize module A , the environment has to send a bag of messages out of $\text{AI}(v_a^0)$, and it has to accept all possible answers of A . Because v_r^0 simulates v_a^0 , the module E consumes the input properly, and sends one of A possible answers. Because the resulting node of $C(E)$ again simulates the resulting node of $C(A)$, a proper communication between the environment and module E could happen again – until the final node is reached. Hence, module U is an utilizing environment of E , i. e. E simulates A .

Assume, E simulates A . To prove the simulation between $C(E)$ and $C(A)$, we again consider an arbitrary utilizing environment U of A , which also has to be an utilizing environment E . Now, we assume $C(E)$ does not simulate $C(A)$ and derive a contradiction. In that case, after some interaction with the environment, in both c-graphs a node is reached, such that this pair of nodes contravenes one requirement of Definition 4.1. Without loss of generality, let v_r and v_a be these nodes.

Requirement (i): Given an input $i \in \text{AI}(v_a)$, and module A answers by sending o . If E could not accept i and module U was sending i towards module E , the system would deadlock. Otherwise, module E answers by sending o' , $o' \neq o$. The environment U awaits the messages o . In case $o' > o$, at the end of the conversation the messages $o' - o$ will remain in the communication channels. Otherwise, module U will deadlock. In all cases, module U is no utilizing environment of E .

Requirement (ii): Because of $\text{AI}(v_a) = \emptyset$, module U does not send further messages. If $\text{AI}(v_r) \neq \emptyset$, module U is no utilizing environment of E .

Requirement (iii): Assume, module E sends the output o after it has received the input $i \in \text{AI}(v_r)$, whereas A sends the output o not until it has received the input $i' \in \text{AI}(v_a)$, $i' > i$. Assume, module U sends first the messages i and waits for a while. In case it gets the messages o yet, module U starts sending counterproductive messages. Otherwise, it completes the input i' . In that case, module U still utilizes A , but it is no utilizing environment of E .

Because, each violation of any requirement yields to a contradiction to our assumptions, all requirements have to be fulfilled, i. e. $C(E)$ simulates $C(A)$. \square

5 Summary

In the field of cross-organizational business processes a crucial question is to decide whether a locally defined *executable* process model is *consistent* to a globally specified *abstract* process model. Based on a Petri net semantics for BPEL4WS [21], this paper presents

a method to verify consistency automatically. The method is part of a larger framework supporting the development of Web service based business processes [15].

Additional use cases

Beside the chosen scenario of this paper, the presented method is useful in several other use cases, such as:

Process publication: If an executable process should be offered to the outside as a Web service, the abstract model (to be published) has to hide all unnecessary or confidential details of the process, and still represent the same behavior. Of course, there are approaches of syntax-based abstraction that yield a less detailed process model while preserving the consistency to the original process model (e.g. [15]). But in practice, the abstract model is often designed manually to some degree. Hence, there is a need to verify consistency between the process models at the end.

Process replacement: Assume an executable process – being already part of a cross-organizational process – needs to be replaced by another process, e.g. because of efficiency. Obviously, all other participating processes should stay untouched. In that case, the new executable process has to behave like the old one. The presented method could also be applied to verify consistency between two executable process models.

Process discovery: Assume a company is searching for a partner that provides a certain service. This company may have already a clear idea of the behavior the partner should provide. Hence, the company defines an abstract process model and wants a directory service to find a partner process, which behaves like this. This approach – called *query-by-example* in the application domain of databases – also can be supported by the the consistency check presented in this paper.

Related research

The current work was inspired by many other approaches, dealing with consistency and matchmaking process models. Some approaches also use Petri nets [24, 6] and/or specify the global interaction by help of *Message Sequence Charts* (MSC) [26, 8]. But, none of them presents such a focussed view on a components externally visible behavior as the *communication graph* does. Concerning matchmaking of Web services, a recently published approach employs finite state automata to solve a similar problem [29]. It seems to yield similar theoretical results, whereas the application to a real world modeling language is not provided, yet.

Finally, there are many papers dealing with ontology and semantic Web based approaches to compare Web services and business processes (e.g. [11]). While those

methods try to match different notion with the same meaning, or to filter candidates out of a huge number of available services, the presented paper compares two given process models while having a clear idea of all used notions. Hence, semantic Web based approaches may supplement the presented method or vice versa.

Open questions

Currently, the simulation between process models is based on the assumption that there is one abstract process model for one executable process. But, an executable process modeled by help of BPEL may interact with several partners, and therefore it is reasonable to have several abstract models of the same process, whereby each abstract model only emphasizes those interactions one specific partner has to perform. In the next step, the notion of the communication graph and the definition of simulation will be adopted to meet the requirement of this scenario.

All presented algorithms are implemented within the prototype WOMBAT4WS [30]. Currently, the work is focussed on improving the algorithms' efficiency by the application of *partial order reduction* techniques [22, 20]. Moreover, up to a certain degree the integration of data aspects into the formalism is planned. Especially the dependencies between the content of incoming message and internal decisions made by the process are the focussed target. Applying technologies of static program analysis (e.g. *slicing* [17]), it seems possible to achieve a higher level of precision in mapping a given process model into a Petri net, without loosing the possibility of efficient analysis.

References

- [1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer-Verlag, Dec. 2002.
- [2] Andrews et al. *BPEL- Business Process Execution Language for Web Services*. OASIS, Standard proposal, Version 1.1, July 2002.
- [3] Colom, Teurel, Silva, and Haddad. *Structural Methods*. In Girault and Valk, editors, *Petri Nets for System Engineering*. Springer-Verlag, 2002.
- [4] K. Gottschalk. *Web Services architecture overview*. IBM developerWorks, Whitepaper, Sept. 2000.
- [5] P. W. Grefen and S. Angelov. *Three-Level Process Specification for Dynamic Service Outsourcing*. LNCS 2472. Springer-Verlag, 2003.
- [6] R. Hamadi and B. Benatallah. *A Petri Net based Model for Web Service Composition*. In *Proc. of ADC 2003*. Australian Computer Society, Inc., 2003.
- [7] Kavantzaz, Burdett, Ritzinger, and Lafon. *Web Services Choreography Description Language*. W3C Working Draft, Version 1.0, Oct. 2004.
- [8] E. Kindler, A. Martens, and W. Reisig. *Inter-operability of Workshop Applications – Local Criteria for Global Soundness*. LNCS 1806. Springer-Verlag, 2000.
- [9] F. Leymann and D. Roller. *Production Workflow – Concepts and Techniques*. Prentice Hall, 1999.
- [10] F. Leymann and D. Roller. *Modeling Business Processes with BPEL4WS*. In *Proc. of Modellierung 2004*, LNI 45. GI, 2004.
- [11] S. A. Ludwig and P. van Santen. *A Grid Service Discovery Matchmaker Based on Ontology Description*. In *EuroWeb 2002*. British Computer Society, 2002.
- [12] A. Martens. On compatibility of web services. *Petri Net Newsletter*, 65:12–20, Oct. 2003.
- [13] A. Martens. *On Usability of Web Services*. In Calero, Diaz, and Piattini, editors, *Proceedings of WQW 2003*, Rome, Italy, Dec. 2003. IEEE Computer Society Press.
- [14] A. Martens. *Analysis and Re-engineering of Web Services*. In S. Hammoudi and J. Cordeiro, editors, *Proceedings of ICEIS 2004*, Porto, Portugal, 2004.
- [15] A. Martens. *Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services*. PhD thesis, WiKu-Verlag Stuttgart, 2004.
- [16] R. Milner. A modal characterisation of observable machine-behaviour. In G. Astesiano and C. Böhm, editors, *Proc. CAAP 81*, LNCS 112. Springer, 1981.
- [17] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
- [18] W. Reisig. *Petri Nets*. Springer-Verlag, 1985.
- [19] W. Reisig. *Elements of Distributed Algorithms – Modeling and Analysis with Petri Nets*. Springer, 1998.
- [20] K. Schmidt. *Explicit State Space Verification*. Habilitation, Humboldt-Universität zu Berlin, 2002.
- [21] K. Schmidt and C. Stahl. A Petri net semantic for BPEL. In E. Kindler, editor, *Proc. of 11th Workshop AWPN*. Paderborn University, Oct. 2004.
- [22] A. Valmari. *State Space Generation – Efficiency and Practicality*. PhD thesis, Tampere University, 1988.
- [23] W. van der Aalst and A. ter Hofstede. *YAWL: Yet Another Workflow Language*. Brisbane, Australia, 2003.
- [24] W. M. P. van der Aalst. *Modeling and Analyzing Interorganizational Workflows*. In *Proc. of CSD'98*. IEEE Computer Society Press, 1998.
- [25] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
- [26] W. M. P. van der Aalst. Interorganizational Workflows – An Approach based on MSC and Petri Nets. *Systems Analysis - Modelling - Simulation*, 34(3), 1999.
- [27] R. J. van Glabbeek. *The Linear Time - Branching Time Spectrum*. LNCS 458. Springer-Verlag, 1990.
- [28] Web-Ontology Working Group. *Ontology Web Language for Web Services*. OWL-S 1.1, Nov. 2004.
- [29] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. *Matchmaking for Business Processes*. In *Proc. of EEE-04*. IEEE Computer Society, 2004.
- [30] WOMBAT4WS. *Workflow Modeling and Business Analysis Toolkit for Web Services*. homepage, <http://www.informatik.hu-berlin.de/top/wombat/>.