

ANALYSIS AND RE-ENGINEERING OF WEB SERVICES

Axel Martens

*Humboldt-Universität zu Berlin,
Department of Computer Science*

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

Key words: Web Services, Business Processes, Analysis, Usability, Petri nets

Abstract: To an increasing extend software systems are integrated across the borders of individual enterprises. The Web Service approach provides group of technologies to describe components and their composition, based on well established protocols. Focused on business processes, one Web Service implements a local subprocess. A distributed business processes is implemented by the composition a set of communicating Web Services. At the moment, there are various modeling languages under development to describe the internal structure of one Web Service (e.g. *Business Process Execution Language for Web Services BPEL4WS* (BEA et al., 2002a)) and the choreography of a set of Web Services (e.g. *Web Service Choreography Interface WSCI* (BEA et al., 2002b)). Nevertheless, there is a need for methods for stepwise construction and verification of such components. This paper abstracts from concrete syntax of any proposed language definition. Instead, we apply Petri nets to model Web Services. Thus, we are able to reason about essential properties, e.g. *usability* of a Web Service – our notion of a quality criterion. Based on this framework, we present an algorithm to analyze a given Web Service and to transfer a complex process model into a appropriate model of a Web Service.

1 Introduction

In this paper, we focus on the application of Web Service technology to distributed business processes: One Web Service implements a local subprocess. Thus, we regard a Web Services as a stateful system. From composition of a set of Web Services there arises a system that implements the distributed business processes.

Within this setting, the quality of each single Web Service and the compatibility of a set of Web Services are questions of major interest. In this paper, we define the notion of *usability* – our quality criterion of a Web Service and present an algorithm to verify this property. Based on this analysis, we present an approach to restructure and simplify a given Web Service model.

1.1 Web Services

A *Web Service* is a self-describing, self-contained modular application that can be described, pub-

lished, located, and invoked over a network, e.g. the World Wide Web. A Web Service performs an encapsulated function ranging from a simple request-reply to a full business process.

A Web Service has a standardized interface and can be accessed via well established protocols. Thus, the Web Service approach provides a homogenous layer to address components upon a heterogenous infrastructure.

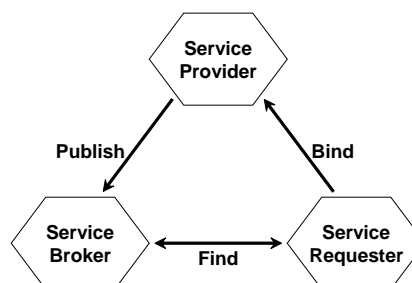


Figure 1: The service oriented architecture

Figure 1 sketches the usage of Web Services within the *Service oriented architecture* (Kreger, 2001). Basically, there are three roles: *service provider*, *service requester* and *service broker*.

The service provider implements the Web Service and *publishes* its description (the *Web Service model*) to one or more repositories for potential users to locate. The service broker manages the repository and allows the service requester to *find* an adequate service. Finally, the service requester can *bind* its own components to the selected Web Service. Our work aims to support the service provider. We propose analysis methods that can be applied to a Web Service in isolation, i. e. before publishing the model.

Instead of one new specific technology, the Web Service approach provides a group of closely related, established and emerging technologies to model, publish, find and bind Web Services – called the *Web Service technology stack* (Gottschalk, 2000). This paper is concerned with the application of Web Service approach towards the area of business processes. Thus, we focus on the behavior of a Web Service, defined by its internal structure.

1.2 Business Processes

A business process consists of a self-contained set of causally ordered activities. Each activity performs a certain functionality, produces and consumes data, requires or provides resources and is executed manually or automatically. A distributed business process consists of local subprocesses that are geographically distributed and/or organizationally independent. The communication between these subprocesses (via a standardized interface) realizes the coordination of the distributed process. Figure 2 sketches the mapping between the business terms and the Web Service technologies.

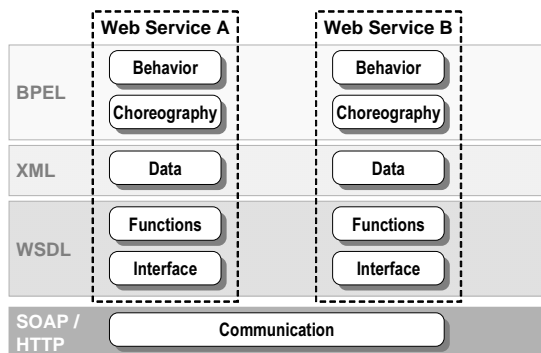


Figure 2: Business processes and Web Services

For each aspect of a distributed business process the Web Service technology stack provides an adequate technology, as shown in Figure 2. The *core layers* cover the technical aspects: data are represented by *XML documents*, the functionality and the interface are defined by help of the *Web Service Description Language WSDL* (Ariba et al., 2001) and the communication uses standardized protocols, e. g. the *Simple Object Access Protocol SOAP* (IBM et al., 2000).

The internal structure of a process and the organizational aspects are covered by the *emerging layers*. There are different proposals towards a specification language. We focus on the *Business Process Execution Language for Web Services BPEL4WS* (BEA et al., 2002a). In combination with the core layers, BPEL4WS allows to model a business process precisely, such that the model can be directly executed by the middleware. Moreover, an abstract model of the process can be expressed by help of BPEL4WS, too. Such an abstract model is published to the repository such that a potential service requestor can decide, whether or not that service is compatible to his own component.

The Web Service approach provides a technical framework to implement distributed business processes guarantee a minimum of *syntactical* consistency. But there is a need for more analysis methods, e. g. the proof of *semantic* compatibility. Let's consider a Web Service of an online shop and a Web Service of a customer: The online shop waits for payment before sending the product. The customer behaves in the opposite way. Both services have a syntactically compatible interface but the resulting distributed process leads to a deadlock.

To address those kinds of problems, we abstract from the technical aspects and use Petri nets to model the behavior of Web Services.

1.3 Solution

The current paper is part of a larger framework for modeling and analyzing distributed business processes by help of Web Services (Martens, 2003a). This framework is based on Petri nets. Petri nets are a well established method for distributed business processes (Kindler et al., 2000). As we will show, Petri nets are also an adequate modeling language for Web Services.

Based on this formalism, we are able to discuss and define *usability* of a Web Service – our notion of a quality criterion – and derive further properties: e. g. *compatibility* of two Web Services. Algorithms to verify these properties are

already implemented within an available prototype (Martens, 2003b).

Due to our abstract view on behavior and structure of Web Services, the results presented here can be adopted easily to every concrete modeling language, for instance BPEL4WS (Reisig and Martens, 2003).

The remaining paper is structured as follows: Section 2 presents our modeling approach. There, we give a short introduction to Petri nets, present the notion of workflow module and show how to compose modules. Afterwards, we discuss the already mentioned properties.

Section 3 establishes the core section of this paper. Applied to an example, we present the algorithm to verify usability. Thus, we derive an adequate representation of the behavior of a given Web Service – the *communication graph*.

Besides the verification of usability, the communication graph of a Web Service contains useful information for re-engineering. Section 4 presents our approach.

Finally, Section 5 gives an overview of the further methods that belong to our framework, e. g. guidelines of modeling and the notion of *equivalence* of two Web Services.

2 Modeling

Concerning business processes and Web Services, there are many modeling languages in use. Some of them come along with a specific tool and change with every new release. Only the fewest have a theoretical background and offer methods for analysis. To deal with the problems of distributed business processes in a generic manner, we use Petri nets.

The following section presents our modeling approach. Thus, we give a short introduction to Petri nets and show how to model a Web Service. Afterwards we deal with the composition of those models and discuss their essential properties.

2.1 Modeling Web Services

In this paper, a distributed business process comes into existence because of composition of Web Services. Each of these Web Services represents a local subprocess. Figure 3 shows the model of such a subprocess – the Web Service of a travel agency.

A business process consists of a self-contained set of activities which are causally ordered. A

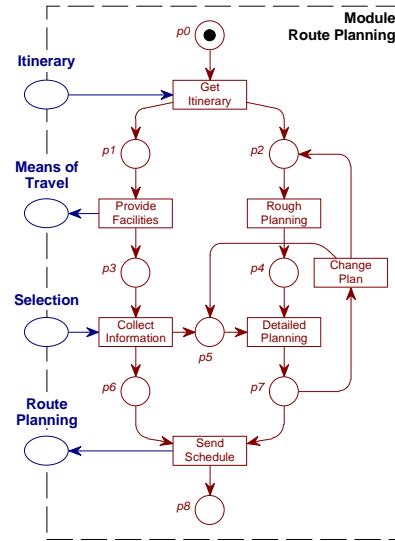


Figure 3: A workflow module

Petri net $N = (P, T, F)$ consists of a set of *transitions* T (boxes), a set of *places* P (ellipses) and a *flow relation* F (arcs) (Reisig, 1985). A transition represents an active element, i. e. an activity (e. g. Get Itinerary). A place represents a passive element, i. e. a state between activities, a resource or a message channel (e. g. Itinerary).

Unlike a lift controller, a business process has a defined beginning and terminates after the execution of a finite number of activities. A *workflow net* is a special Petri net, that has two distinguished places $\alpha, \omega \in P$ to denote the begin (α) and the end (ω) of a process (Aalst, 1995). Thus, we model a business process in terms of a workflow net.

A Web Service consists of internal structures that realize a local subprocess and an interface to communicate with its environment, i. e. other Web Services. Thus we model a Web Service by help of a workflow net supplemented by an interface, i. e. a set of places representing directed message channels. Such a model we call *workflow module*.

Definition 2.1 (Module).

A finite Petri net $M = (P, T, F)$ is called *workflow module* (*module* for short), iff:

- (i) The set of places is divided into three disjoint sets: *internal places* P^N , *input places* P^I and *output places* P^O .
- (ii) The flow relation is divided into *internal flow* $F^N \subseteq (P^N \times T) \cup (T \times P^N)$ and *communication flow* $F^C \subseteq (P^I \times T) \cup (T \times P^O)$.
- (iii) The net $\mathcal{N}(M) = (P^N, T, F^N)$ is a workflow net.

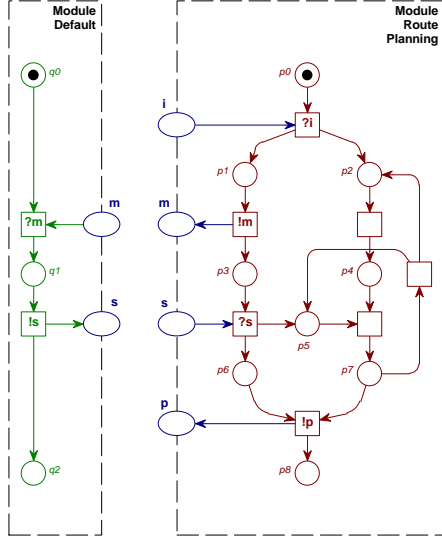


Figure 4: Syntactically compatible modules

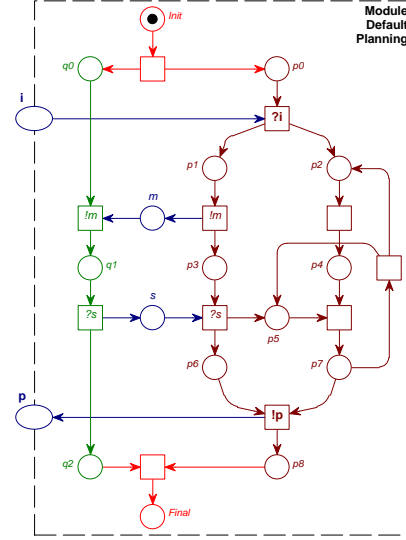


Figure 5: A composed workflow module

- (iv) Non of the transitions is connected both to an input place and an output place. *

Within a module, we call the workflow net $\mathcal{N}(M)$ the *internal process* and the tuple $\mathcal{I}(M) = (P^I, P^O)$ its *interface*. Figure 3 shows a workflow module. The internal process is triggered by an incoming *Itinerary*. Then the control flow splits into two concurrent threads. On the left side, an available *Means of travel* are offered to the customer and the service awaits his *Selection*. Meanwhile, on the right side, a *Rough Planning* may happen. The *Detailed Planning* requires information from the customer. Finally, the service sends a *Route Planning* to the customer.

2.2 Composing Web Services

A distributed business process is realized by the composition of a set of Web Services. We will now define the pairwise composition of workflow modules. Because this yields another workflow module, recurrent application of pairwise composition allows us to compose of more than two modules.

Figure 4 shows once more the workflow module *Route Planning*. Additionally, a module *Default* is presented. The purpose of this model is to unburden the customer from making a selection. Thus, the module *Default* consumes the message on available *Means of travel* and return a default *Selection*.

Obviously, both modules can be composed. As a precondition for composition, we will define the property of *syntactical compatibility* of two modules.

Definition 2.2 (Syntactical compatibility).

Two workflow modules A and B are called *syntactically compatible*, iff the internal processes of both modules are disjoint, and each common place is an output place of one module and an input place of the other. *

As shown in Figure 4, two syntactically compatible modules do not need to have a completely matching interface. They might even have a completely disjoint interface. In that case, the reason of composition is at least dubious.

When two modules are composed, the common places are merged and the dangling input and output places become the new interface. To achieve a correct module as the result of the composition, we need to add new components for initialization and termination. Figure 5 shows the composed module *Default Planning*.

Definition 2.3 (Composed system).

Let $A = (P_a, T_a, F_a)$ and $B = (P_b, T_b, F_b)$ be two syntactically compatible modules. Let $\alpha_s, \omega_s \notin (P_a \cup P_b)$ two *new* places and $t_\alpha, t_\omega \notin (T_a \cup T_b)$ two *new* transitions. The *composed system* $\Pi = A \oplus B$ is given by (P_s, T_s, F_s) , such that:

- $P_s = P_a \cup P_b \cup \{\alpha_s, \omega_s\}$
- $T_s = T_a \cup T_b \cup \{t_\alpha, t_\omega\}$
- $F_s = F_a \cup F_b \cup \{(\alpha_s, t_\alpha), (t_\alpha, \alpha_a), (t_\alpha, \alpha_b), (\omega_a, t_\omega), (\omega_b, t_\omega), (t_\omega, \omega_s)\}$

If the composed system contains more than one components for initialization and termination, the corresponding elements are merged. *

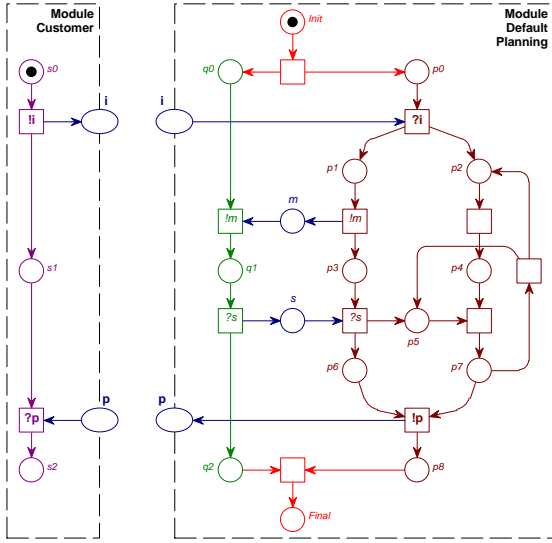


Figure 6: A module and its environment

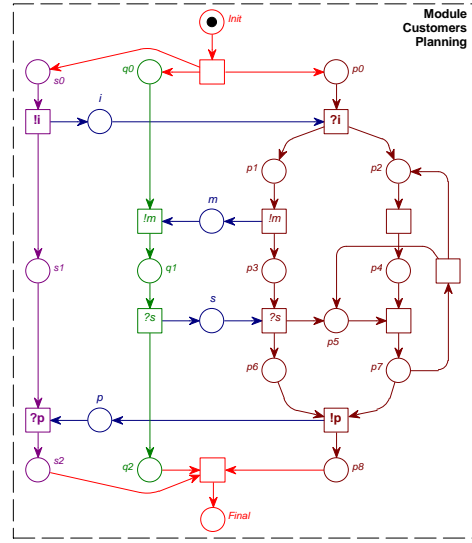


Figure 7: A composed workflow module

Figure 5 shows the model of a composed Web Service Route Planning \oplus Default. This service offers a simpler interface to the customer. Syntactically, the result of the composition is again a workflow module.

Corollary 2.1 (Composing modules):

Whenever A and B are two syntactically compatible workflow modules, the composed system $\Pi = A \oplus B$ is a workflow module too. *

This corollary can be easily proven. We therefore omit the proof here, it can be found in (Martens, 2003a). The next section deals with a special case: two syntactically compatible modules with completely matching interfaces.

2.2.1 Environments

This paper focusses on distributed business processes that come into existence by composition of Web Services. Thus, we aim at generating a *model* of the business process by composing the *models* of the Web Services. Thus, the composition of two workflow modules A and B represents a business process, if the composed system $\Pi = A \oplus B$ has an empty interface, i. e. Π is a workflow net. In that case, we call A an *environment* of B .

Definition 2.4 (Environment).

Let M and U be two syntactically compatible modules. U is called *environment* of M , iff each output place of U is an input place of M and vice versa. *

If a module U is an environment of M , obviously M is an environment of U too. Figure 6 shows once more the module Route Planning. Additionally, the module Customer is shown, which is an environment of the module Route Planning.

In the real world, the environment of a Web Service may consist of several other Web Services. If we want to reason about that specific Web Service, we don't have any assumption on its potential environment. Thus, without loss of generality, we may consider its environment as one, arbitrary structured Web Service.

2.3 Properties of Web Services

Given a workflow module and one environment, it is possible to reason about the quality of the composed system. The notion of *soundness* (Aalst, 1996) is an established quality criterion for workflow nets. Basically, soundness requires each initiated process to come to a proper final state. Additionally, each transition should be relevant, i. e. there should be at least one behavior of the process in which this transition participates.

The second requirement is reasonable, if a business process was modeled from scratch. In our approach a business process arises from composition. Thus, it might be reasonable to call a system "sound", although not all functionality of one specific component is used in that system. Hence, we use the slightly different notion.

Definition 2.5 (Weak soundness).

A workflow net $N = (P, T, F)$ is called *weak sound*, iff:

- (i) For each reachable marking (starting at $[\alpha]$) the final marking $[\omega]$ is reachable.
- (ii) For each reachable marking m such that $m \geq [\omega]$ holds $m = [\omega]$. *

The requirements of this definition are a subset of the requirements within the definition of soundness. Thus, each sound workflow net is weak sound. For more information, e.g. the precise definition of *reachable marking* see (Martens, 2003a).

Figure 7 shows a workflow net which is the composed system build from the modules shown in Figure 6. The soundness property of this net can be easily proven. Thus, this net is weak sound too. Because of that, intuitively, we want to call the modules *Default Planning* and *Customer* *semantically compatible*. In the following section, we will discuss the notion of *usability* and present a formal definition, such that we will call both modules *usable*. Based on this notion we can derive a definition of *semantic compatibility* that meets our stated intuition.

2.3.1 Usability

Obviously, the purpose of a workflow module is to be composed with an environment such that the resulting system is a proper workflow net, i. e. we require the resulting workflow net to be weak sound. For a given module there exist infinitely many environments. The question is: To call the given module *usable*, how many of these environments have to form a weak sound composed system together with this module?

One might think of calling a module *usable*, iff all possible environments form a weak sound composed system together with that module. But, this definition is not appropriate: Because of the following proposition, no module at all would be usable.

Proposition 2.2 (Malicious environment):

For each workflow module M there is a malicious environment U such that the composed system $M \oplus U$ is not weak sound. *

The proof together with a set of illustrating examples can be found in (Martens, 2003a). Because of that we give another, more appropriate definition of usability:

Definition 2.6 (Usability).

Let M be a workflow module.

- (i) An environment U *utilizes* the module M , iff the composed system $\Pi = M \oplus U$ is weak sound.

- (ii) The module M is called *usable*, iff there exists at least one environment U , such that U utilizes M . *

Concerning this definition, the module *Customer* utilizes the module *Default Planning* and vice versa. Thus, both modules are called usable.

2.3.2 Compatibility

Within the discussion of usability, we have already used the notion of semantic compatibility based on intuition. Now we want to give a formal definition. We like to call a module and its environment *semantically compatible*, iff the composed system is weak sound (e.g. modules *Customer* and *Default Planning*).

But, if we consider the two module shown in Figure 4, we want to call them *semantically compatible*, too, and we therefore need a different definition. Obviously, two modules are not compatible in case the composed system has an error, i. e. the resulting module is not usable. Thus, we propose the following definition.

Definition 2.7 (Semantic compatibility).

Let A and B be two syntactically compatible modules.

- A and B are called *semantically compatible*, iff the composed system $A \oplus B$ is usable. *

According to this definition, modules *Default* and *Route Planning* are called *semantically compatible*. One might wonder about this definition, but it does also cover the composition of a module and its environment. Lets have another a look at the composed system *Customers Planning* shown in Figure 7. The system is a workflow net, i. e. a workflow module with an empty interface. Thus, we can find a very simple environment with an empty interface, too – just one place. Although, this environment does not make any semantic sense, syntactically it utilizes the workflow module *Customers Planning*, therefore the modules *Customer* and *Default Planning* are proven to be *semantically compatible*.

Within (Martens, 2003a), both notions – usability and compatibility – are tighten in a way, that all behavior of a given Web Service should be used by an environment resp. a compatible module. In that case we call a workflow module *totally usable* resp. two workflow modules *totally compatible*.

3 Analysis

In the previous section we have introduced the notion of usability. Further essential properties of Web Services (e. g. compatibility) can be derived from this notion. The definition of usability itself is based on the existence of an environment. Thus, we cannot disprove the usability of a given Web Service, because we have to consider all its possible (infinitely many) environments.

Hence, this section provides a different approach: We derive an adequate representation of the behavior of a given Web Service – the *communication graph*. Illustrated by help of an example, we present the algorithm to *decide* usability. Besides the verification of usability, we use the communication graph of a Web Service for re-engineering in Section 4.

3.1 Example

To reason about usability, we first take a look on a example that is complex enough to reflect some interesting phenomenons, but small enough to be easily understood. Figure 8 shows this example.

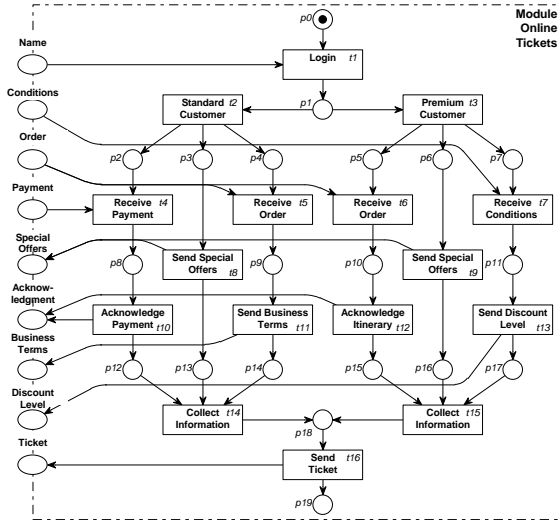


Figure 8: Module online tickets

In this paper, a Web Service implements a local business process. Thus, the model of a Web Service often is derived from an existing business process model and the structure of the model therefore reflects the organization structure within the process. As we will see later, such a Web Service model should be restructured before it is published.

Figure 8 shows a model of a Web Service selling online tickets. Basically, the workflow module

consists of two separated parts. After the module receives the **Name** of the customer, an internal decision is made: either the customer is classified as **Standard Customer** or as **Premium Customer**. In the first case, only the left part of the module is used, and the right part otherwise. At the end, both cases are join by sending the ordered **Ticket**.

Within each case, there are three independent threads of control – representing three independent departments: the *marketing dept.* sends **Special Offers**, the *canvassing dept.* receives the **Order** and answers by sending the **Business Terms** or an **Acknowledgment**, and the *accounts dept.* manages the financial part.

To prove the usability of the workflow module **Online Tickets**, we try to derive an environment that utilizes this module. Consider a customer who claims to be a **Premium Customer**. He might send his **Name** and waits for the **Special Offers**. Afterwards he sends his **Conditions** and expects the information on his current **Discount Level**.

Unfortunately, by some reasons he has lost his status and he is classified as a **Standard Customer**. In that case, the module ignores the **Conditions** and waits for the **Order** and for **Payment**. The composed system of such a customer and the module **Online Tickets** has reached a deadlock.

Our first approach to find an utilizing environment was based on the *structure* of a given module. Although the module **Online Tickets** is usable – as we will see later – this approach went wrong. Hence, our algorithm to decide the usability is based on the behavior of a given workflow module.

3.2 The communication graph

A workflow module is a reactive system, it consumes messages from the environment and produces answers depending on its internal state. The problem is, an environment has no explicit information on the internal state of the module. But each environment does know the structure of the module and can derive some information by considering the communication towards the module. Thus, an environment has implicit information. We reflect exactly that kind of information within a data structure – called the *communication graph*.

Definition 3.1 (communication graph).

A communication graph (V, H, E) is a directed, strongly connected, bipartite graph with some requirements:

- There are 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.
- There exists a labeling $m = (m_v, m_e)$, such that m_v yield an definite set of states for each visible node and m_e yields a bag of messages to each edge. *

For the precise, mathematical definition see (Martens, 2003a). Figure 9 shows the communication graph of module Online Tickets. Some parts of the graph a drawn with dashed lines – we will com to this later.

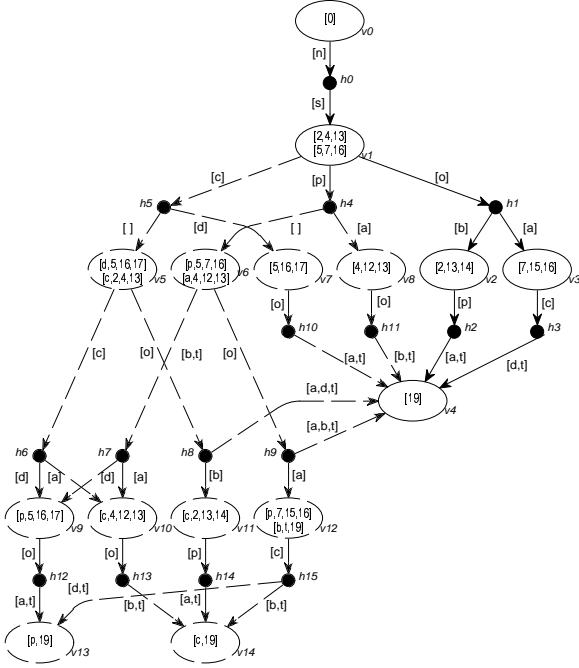


Figure 9: Communication graph

The root node v_0 is labeled with the initial state of the module ($[0]$ stands for one token on place p_0). An edge starting at a visible node is labeled with a bag of messages send by the environment – called *input*, an edge starting at a hidden node is labeled with a bag of messages send by the module – called *output*. Thus, a path within this graph represents a communication sequence between the module an an environment.

Some visible nodes are labeled with more than one state (e. g. v_1). In that case, after the communication along the path towards this node, the module has reached or will reach one of these states.

The communication graph of a given module is well defined and we present an algorithm for its calculation. But before we can do so, we have to introduce some functions on workflow modules:

Activated input Concerning a given state of a module, an activated input is a minimal bag of messages the module requires from an environment either to produce an output or to reach the final state. The function INP yields the set of activated inputs.

Successor state Concerning a given state of a module, a successor state is a reachable state that is maximal concerning one run of the module. The function NXT yields the set of successor states.

Possible output Concerning a given state of a module, a possible output is a maximal bag of messages the is send by the module while reaching a successor state. The function OUT yields the set of possible outputs.

Communication step The 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 for a given module M .

All notions mentioned above are well defined based on partial ordered run of the workflow module (see (Martens, 2003a)). Because of the limited space, we do not go into further details. Applying these notions, we are now able to present the construction of the communication graph. The algorithm starts with the root node v_0 labeled with the initial state:

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

The communication graph of a module contains that information, a “good natured” environment can derive. That means, the environment always sends as little messages as possible, but as much as necessary to achieve an answer resp. to terminate the process in a proper state. By considering all reachable successor states together with

all possible outputs, the choices within the module are not restricted.

3.3 The usability graph

By help of the communication graph we can decide the usability of a module. A communication graph may have several leaf nodes: none, finitely or infinitely many. Figure 9 shows a graph with three leaf nodes: v_4 , v_{13} and v_{14} . In each communication graph there is at most one leaf node labeled with the defined final state of the workflow module (v_4). All other leaf nodes contain at least one state, where there are messages left or which marks a deadlock within the module (v_{13} and v_{14}).

That means: If we build an environment that communicates with the module according to the labels along the path to such a leaf node, this environment does not utilize the module. Therefore, we call the communication sequence defined by such a path an erroneous sequence. Now we can try to eliminate all erroneous sequences. We call a subgraph of the communication graph that does not contain any erroneous sequences an *usability graph* of that module.

Definition 3.2 (Usability graph).

A subgraph U of the communication graph C is called *usability graph*, iff

- U contains the root node and the defined leaf node (labeled with the defined final state of the workflow module) of C .
- For each hidden node within U all outgoing edges are within U , too.
- Each node within U lies on a path between the root node and the defined leaf node. *

A usability graph of a module describes, how to use that module. For the precise, mathematical definition see (Martens, 2003a). A communication graph may contain several usability graphs.

Figure 9 shows the only usability graph of module *Online Tickets* drawn by solid lines. Now we can construct a more clever customer than we did at the beginning of this section: A customer send its name [n] and awaits the special offers [s]. Afterwards, he sends the order [o].

If he receives the business terms [b], he was classified as standard customer. Thus, he pays [p] and gets an acknowledgement and the ticket [a, t]. Otherwise, he is a premium customer and receives an acknowledgement [a]. In that case, he transmits his conditions [c] and receives finally the current discount level and the ticket [d, t].

If we look at Figure 9, there is a path from the node v_1 to the defined leaf node v_4 via h_5 ,

i. e. the module might serve properly the customer from beginning of this section. But, the decision whether or not the path to h_5 is continued towards the node v_4 is up to the module. An environment has no further influence. Hence, a utilizing environment must prevent to reach this node.

3.4 The theorem of usability

An usability graph U can easily be transformed into an environment of the workflow module – we call it the *constructed environment*, denoted by $\Gamma(U)$. The next section presents the generation of an abstract representation for a given workflow module (Figure 10). The construction of the environment takes place analogically, just by switching the directions of communication. We need the constructed environment to decide the usability of some cyclic workflow modules.

Now we can formulate the correlation between the usability of a workflow module and the existence of a usability graph:

Theorem 3.1 (Usability).

Let M be a workflow module and let C be the communication graph of M .

- The module M is *not* usable, if C contains no finite usability graph.
- An acyclic module M is usable, if and only if C contains at least one finite usability graph.
- An cyclic module M is usable, if C contains at least one finite usability graph U and the composed system $M \oplus \Gamma(U)$ is weak sound. *

The proof applies the precise definition and underlying structures of Petri net theory. Hence, we omit the proof here, it can be found in (Martens, 2003a).

Nevertheless, the theorem 3.1 gives us the possibility to decide usability in many cases: An acyclic workflow module has a finite communication graph. Thus, we can search for a usability graph and decide usability. The most cyclic modules have a finite communication graph, too. If a usable cyclic module has an infinite communication graph, than there exists a finite usability graph. Thus, by breath-first-search we will find this graph after a finite time. Information about the complexity of our algorithms and more detailed result on cyclic workflow modules can be found in (Martens, 2003a), too. The algorithms are also implemented within an available prototype (Martens, 2003b).

4 Re-engineering

As we have shown, the workflow module of the online ticket service is usable. Nevertheless, the representation is not adequate for publishing the service within a public repository. We already have address the problems of a customer who wants to use this service.

4.1 Views on Web Services

Anyhow, it is not correct to call the module shown in Figure 8 a “bad” model in general. The quality of a model always depends on its purpose. Concerning Web Services we can distinguish two purposes, that come along with totally different requirements.

On the one hand, a Web Service is modeled to describe the way it is executed. Such a model is useful for the provider of the service. Hence, it is called the *private view model* and needs to contain a lot of details on the internal structure. The module shown in Figure 8 is a good candidate for a *private view model*, because it reflects the organization structure (three independent departments).

On the other hand, a Web Service is modeled to describe how to use it. Such a model has to be easily understandable, because a potential requestor of the service wants to decide, whether or not that service is compatible to his own component. Hence, such a model is called the *public view model*. For that purpose the module Online Tickets is no adequate model of the services.

As a consequence thereof, we need another model of this services. Because of many reasons, it is not efficient to build a new public view model from scratch. Instead the public view model should be automatically derived from the private view model.

4.2 Transformation

Basically, the transformation of private view model into a public view model is an abstraction from details. Hence, a common approach focusses on elimination and fusion of elements within a given model. Concerning Petri nets, there are many of such rules of simplification (e. g. (Colom et al., 2002)).

In this paper, we present a different approach. A potential requestor of a Web Service ist not urgently interested in the (possibly simplified) structure of a process. For him, the behavior is of vital importance. As we have discussed, the usability graph is an adequate representation of the

usable behavior for a given Web Service. Thus, the public view model should be derived from the usability graph of the private view model.

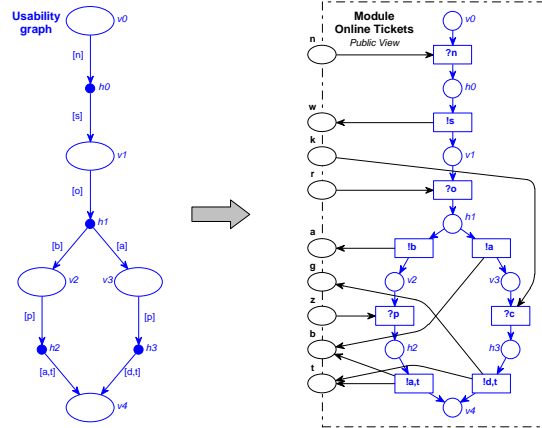


Figure 10: A module and its environment

Figure 10 shows on the left side the usability graph of the module Online Tickets. We have omit the labeling of the visible states, because this is of no interest for the transformation. On the right side, Figure 10 shows public view model of the online ticket service. The transformation is very simple: Each node of the usability graph is mapped to a place within the module and each edge of the graph is mapped to a transition that is put in between the two places representing the source and target nod of the edge. Finally, the interface places are added and each transition is connected to these places according to the label of the edge.

As the result, a customer can easily understand, how to use the service. The independency between the canvassing department and the accounts department was replaced be a causal order, because now utilizing environment of this module could communicate with both departments concurrently.

A public view model of a Web Service, that is generated by our algorithm contains only the usable behavior of the original model. Thus, the both views on the process are not equivalent. We require just a simulation relation: Each utilizing environment of the public view model has to be a utilizing environment of the private view model. In the very most cases this property holds per construction. There are a few abnormal cases, where we have to prove the simulation relation and to adjust the result in case. More details can be found in (Martens, 2003a).

4.3 Restructuring

The analysis of usability for a given Web Service does not only have an impact on the public view model. The communication graph resp. the usability graph offers a starting point for re-engineering of such components. As we have seen in the example, the independence of two department does not offer any benefit to the customer. Thus, these activities can be pooled in one department. Moreover, there are common activities of both classes of customers (Send Special Offers and Receive Order). These activities can be grouped ahead the internal decision.

In our example, each activity can be used by a customer. Of course there are modules where some transitions does not belong to any usable behavior or the whole module is not usable. In such case the communication graph points out the difficulties and offers an outside view on the Web Service to the provider. Within our framework we have identified syntactical indicators of weaknesses and guidelines of modeling Web Services. This covers pattern for the correction of certain frequent modeling errors (Richter, 2002).

5 Summary

In this paper, we have sketched a framework for modeling business processes and Web Services by help of Petri nets. This framework has enabled us to specify fundamental properties of such components, among them *usability* and *compatibility*. We have also presented algorithms to verify these properties locally. Moreover, the our approach yields a concrete example how to use a given Web Services.

Beside the results presented here, the notion of usability and the formalism of communication graphs are the basis for further investigations on Web Services. On the one hand, the analysis of usability offers a starting point for the simplification of Web Service models and for re-engineering of such components. On the other hand, the equivalence of two Web Services can be decided. This is exceedingly important for a dynamic exchange of components within a running system: Does the new component behave exactly the way the replaced component did?

All presented algorithms are implemented within a prototype. Currently, we try to improve the efficiency of the algorithms by the application of partial order reduction techniques. Due to this approach we will be able to handle much larger workflow modules which emerge by transforma-

tion of a real world modeling language into our framework, i.e. BPEL4WS (BEA et al., 2002a).

Last but not least, we work on structural criteria to indicate problems which prohibit usability, e.g. the *non local choice problem* (Ben-Abdallah and Leue, 1997). Thus, a service provider may get a hint on demand while modeling the services, without running an expensive analysis.

REFERENCES

- Aalst, W. M. P. v. d. (1995). A class of petri net for modeling and analyzing business processes. Computing science report 95/26, Eindhoven University of Technology.
- Aalst, W. M. P. v. d. (1996). Structural characterizations of sound workflow nets. Computing science report 96/23, Eindhoven University of Technology.
- Ariba, IBM, and Microsoft (2001). *WSDL – Web Services Description Language*. W3C Standard, Version 1.1. <http://www.w3.org/TR/wsdl>.
- BEA, IBM, Microsoft, and SAP (2002a). *BPEL4WS– Business Process Execution Language for Web Services*. Version 1.1.
- BEA, Intalio, SAP, and Sun (2002b). *Web Service Choreography Interface*. Proposal, Version 1.0. <http://www.w3.org/TR/wsci/>.
- Ben-Abdallah, H. and Leue, S. (1997). Syntactic detection of process divergence and non-local choice in message sequence charts. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Verlag, LNCS 1217.
- Colom, J., Teurel, E., Silva, M., and Haddad, S. (2002). Structural methods. In Girault, C. and Valk, R., editors, *Petri Nets for System Engineering.*, pages 277–317. Springer-Verlag Berlin Heidelberg New York.
- Gottschalk, K. (2000). *Web Services architecture overview*. IBM developerWorks, Whitepaper.
- IBM, Lotus, and Microsoft (2000). *SOAP – Simple Object Access Protocol*. W3C Standard, Version 1.1. <http://www.w3.org/TR/SOAP/>.
- Kindler, E., Martens, A., and Reisig, W. (2000). Inter-operability of workshop applications – local criteria for global soundness. In van der Aalst, W., Desel, J., and Oberweis, A., editors, *Business Process Management*. Springer-Verlag, LNCS 1806.

- Kreger, H. (2001). *WSCA – Web Services Conceptual Architecture*. IBM Software Group, Whitepaper.
- Martens, A. (2003a). *Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services*. Phd thesis, Humboldt-Universität zu Berlin.
- Martens, A. (2003b). *Workflow Module Toolkit*. Humboldt-Universität zu Berlin, Manual. <http://www.informatik.hu-berlin.de/~martens/WFMTK>.
- Reisig, W. (1985). *Petri Nets*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, eacts monographs on theoretical computer science edition.
- Reisig, W. and Martens, A. (2003). *Task Force – Geschäftsprozesssprache BPEL4WS*. Humboldt-Universität zu Berlin, Lehrstuhl Theorie der Programmierung.
- Richter, W. (2002). Spezifikation und Implementation organisationsübergreifender Geschäftsprozesse mit Petrinetzen. Master's thesis, Humboldt-Universität zu Berlin.