Analyzing Web Service based Business Processes

Axel Martens

Humboldt-Universität zu Berlin	IBM T. J. Watson Research Center
Department of Computer Sciece	Component Systems Group
Berlin (Adlershof), Germany	Hawthorne (NY), USA
martens@informatik.hu-berlin.de	*amarten@us.ibm.com*

Abstract. This paper is concerned with the application of Web services to distributed, cross-organizational business processes. In this scenario, it is crucial to answer the following questions: Do two Web services fit together in a way such that the composed system is deadlock-free? – the question of *compatibility*. Can one Web service be replaced by another while the remaining components stay untouched? – the question of *equivalence*. Can we reason about the soundness of one given Web service without considering the actual environment it will by used in?
This paper defines the notion of *usability* – an intuitive and locally provable soundness criterion for a given Web services. Based on this notion, this paper demonstrates how the other questions could be answered. The presented method is based on Petri nets, because this formalism is widely used for modeling and analyzing business processes. Due to the existing Petri net semantics for BPEL4WS – a language that is in the very act of becoming the industrial standard for Web service based business processes – the results are directly applicable to real world examples.

Keywords Business Process Modeling, Web Service, BPEL4WS, Tool based Verification, Petri nets

1 Introduction

Over the past years, the Internet has evolved from just a communication media into a platform for B2B integration. Emerging technologies and industrial standards in the field of *Web services* enable a much faster and easier cooperation of distributed partners. This paper is concerned with the application of Web services to distributed, cross-organizational business processes.

The scenario A *Web service* [1] is a self-describing, self-contained modular application that can be published, located, and invoked over a network, e. g. the Internet. A Web service performs an encapsulated function and can be accessed via a standardized interface. In this paper, 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] to cover heterogeneity and distribution underneath a homogenous concept of components and composition. Among other

things, the language BPEL4WS [2] belongs to this stack. Due to this layered architecture, the presented analysis method can be focussed on the Web service's BPEL process model without losing generality or practical relevance.

The goal The Web service technologies define a technical framework to implement distributed business processes while a minimum of *syntactic* consistency is guaranteed. But as this paper will show, there is a need for more advanced analysis, and there exist effective methods that are able to support the development of Web services and Web service based business processes according to the *Service oriented architecture* (SOA [9]).

The service oriented architecture describes three roles: 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. For him, it is crucial to determine errors and weaknesses of his service prior to publication. Hence, this paper presents the notion *usability* – a locally provable soundness criterion for a given Web service – that prevents publication of erroneous services.

The *service requestor* is searching for a Web service that he could bind to his own components. For him, it is crucial to determine whether or not a given Web service does interact properly with his components. Hence, this paper defines the criterion of *semantic compatibility* and provides its verification.

Finally, the *service broker* manages a repository and allows the service requestor to find an adequate service. According to the *query-by-example* approach, he compares the actual Web service model (published by provider) with an abstract Web service model submitted by the requestor. Beside other use cases, the presented *equivalence* criterion provides a basis for the necessary *matchmaking*.

The method Many of the Web services technologies are still in the standardization process, and therefore some specifications will likely be changed several times until a consistent status is reached. Hence, the presented method refrains from the actual syntax of any proposed Web service modeling languages. Instead, it applies a generic formalism of *Petri nets* [16] to addresses the core problems of distributed business processes. This formal method is widely used for modeling and analyzing business processes and Web services [21,5,6]. Applying the rich theory of distributed systems, the presented method is able to define and verify *usability*, *compatibility* and *equivalence* of Web services. Moreover, the presented results can easily be adopted to almost any concrete modeling language (e. g. WS-CDL [7], OWL-S [25] or YAWL [23]). In particular, there exists already a Petri net semantics for BPEL4WS [18] – a language that is in the very act of becoming the industrial standard for modeling Web service based business processes. Hence, the method is directly applicable to real world examples.

The remaining paper is structured as follows: Section 2 gives a short introduction to Petri nets and describes the structure and composition of *workflow modules* – the formal model of a Web Sevice. Section 3, discusses and defines the notion of *usability* by help of examples, and derives the properties of *compatibility* and *equivalence*. Section 4 establishes the core section of this paper: Applied
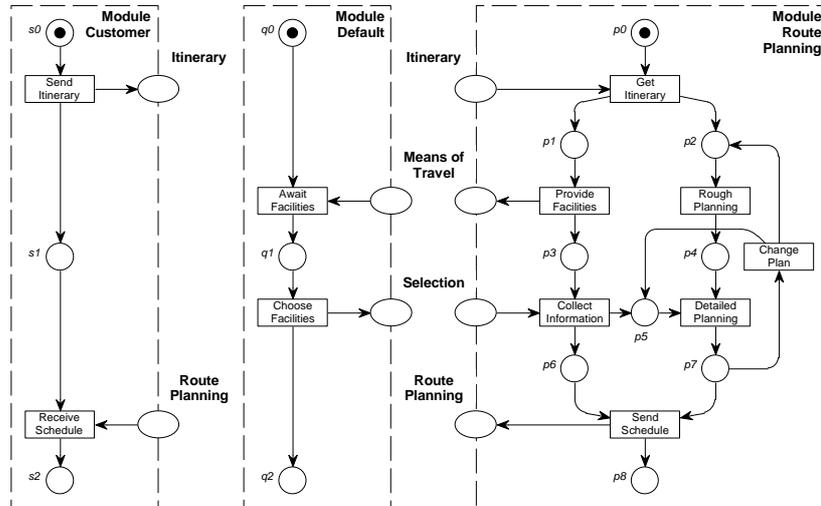
Fig. 1. Modeling with Petri nets

to an example, the verification algorithm is presented. Finally, Section 5 summarizes the results and discusses their correlation to other published approaches.

2 Modeling

The *Business Process Execution Language for Web Services* BPEL4WS [2] provides a syntax to describe Web service process models. But, its semantics so far is defined only by English prose or encoded into middleware components, more ore less accurately. To verify properties like properties compatibility or equivalence of two models a formal semantics of all its concepts is needed. The presented method is based on a Petri net semantics [18]. Petri nets are a well established method for modeling and analyzing (cross-organizational) business processes [20,21,8]. They possess an intuitive graphical representation as well as an algebraic foundation, and therefore Petri nets allow an effective analysis. Other recent research projects apply Petri nets to Web Services [6,15], too.

In contrast to the mentioned Petri net semantics, the current approach abstracts from data aspects. This is an 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. The mapping of a BPEL process model into an analyzable Petri net is explained in [13].

2.1 Modeling with Petri nets

Figure 1 shows Petri net models of three Web service: A route planning service, a services that acts as mediator and a customer's service. These examples will

be used to demonstrate the modeling method, and to visualize composition and compatibility of Web services. But first, some basic Petri net notions have to be introduced:

Petri nets A *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) [16]. A transition represents a dynamic element, i.e. an activity of a business process (e.g. Get Itinerary). A place represents a static element, i.e. the causality between activities or a message channel (e.g. Itinerary). The *marking* (i.e. state) of a Petri net is represented by black *tokens* distributed over the places (see places p0, q0 and s0). A transition t is enabled if on each place $p, (p, t) \in F$ there is at least one token. If an enabled transition t fires, t removes one token from each place $p_1, (p_1, t) \in F$ and produces one token on each place $p_2, (t, p_2) \in F$. Based on this *firing rule*, it is possible to reason about the behavior of a Petri net in term of *firing sequences*, *reachable states* and/or *concurrent runs* (cf. [17]).

Workflow modules A stateful Web service defines an internal process (i.e. activities building its 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* – a special Petri net that has two distinguished places ($\alpha, \omega \in P$) to denote the begin and the end of a process [21] – supplemented by a set of interface places – each of them representing one directed message channel. Such a model is called *workflow module*.

Definition 1 (Module). *A finite Petri net* $M = (P, T, F)$ *is called* workflow module *(or just* module*) if the following conditions hold:*

(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* $PM = (P^N, T, F^N)$ *is a workflow net.*
(iv) *No transitions is connected both to an input place and an output place.*

Within a workflow module M, the workflow net PM is called the *internal process* of M and the tuple $\mathcal{I}(M) = (P^I, P^O)$ is called its *interface*. Lets have a closer look on the module Route Planning shown in Figure 1. 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 Workflow modules

A distributed business process is realized by the composition of a set of Web services. The following section defines the pairwise composition of workflow modules. Because this yields another workflow module, composition of more than two modules is realized by recurrent application of pairwise composition.

(a) A composed Web service model (b) A composed business process model
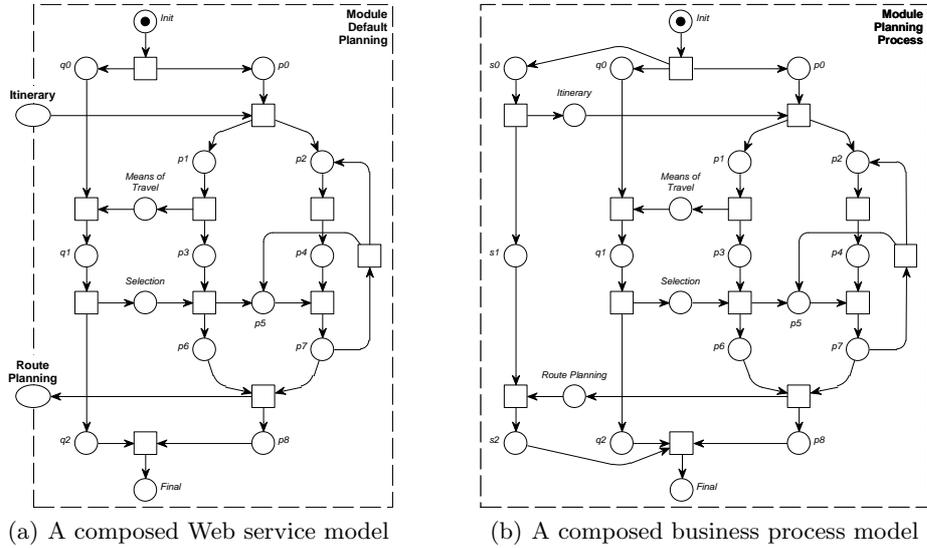
Fig. 2. Composition of workflow modules

Compatibility Figure 1 shows the module Default in the middle. The purpose of this service is to unburden the customer from making a selection: The module consumes the message on available Means of travel and returns a default Selection. Intuitively, it looks like the modules Default and Route Planning could be composed. Formally, we will define the property of *syntactic compatibility* as a precondition for composition of two modules.

Definition 2 (Syntactic compatibility). *Two workflow modules are called* syntactically compatible *if both internal processes are disjoint, and each common place is an output place of one module and an input place of the other.*

Referring to previous definition, the modules Default and Route Planning are syntactically compatible. Nevertheless, this is not a sufficient criterion for proper interaction between two partners. Lets consider a workflow module of an online shop and a workflow module of a customer: The customer sends the payment after he has received the ordered product, whereas the online shop waits for payment before sending the product. Both modules have a syntactically compatible interface, but the resulting distributed process leads to a deadlock. To avoid such errors, Section 3.2 will define the property of *semantic compatibility*.

Composition As the example has shown, two syntactically compatible modules do not need to have a completely matching interface. They might even have a completely disjoint interface. When two modules are composed, the common places are merged and the dangling input and output places become the new interface. To achieve a syntactically correct workflow module, it is necessary to add new components for initialization and termination.

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

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

It can be easily proven that the composition of two syntactically compatible workflow modules always yield a workflow module, too. This is because of the additional components for initialization and termination. If more than two modules are composed, it is important to guarantee *associativity* of pairwise composition (i.e. $(A \oplus B) \oplus C = A \oplus (B \oplus C)$). Hence, if there are already such syntactic components, they are merged while composition (cf. Figure 2(b)). In Figure 2(a), the workflow module Default Planning is shown – the model of the composed Web service Route Planning $\oplus$ Default. This service offers a simpler interface to the customer: He only has to submit the Itinerary to obtain the Route Planning information. To use this service, it is not relevant for a customer wether or not the actual service was deployed at once or was formed by composition.

Environment The composition of two workflow modules A and B yields a model of a distributed business process, if both sets of interface places completely match, i.e. the composed system $\Pi = A \oplus B$ has an empty interface. With other words, Π is also a workflow net. In that case, module A is called an *environment* of module B – obviously, this notion is symmetric.

The module Customer shown in Figure 1 is an environment of the (composed) module Default Planning and vice versa. Figure 2(b) shows the resulting workflow net. While composing both modules, the existing and the new added components for initialization and termination are merged, as already mentioned.

Given a workflow module and its environment, it is possible to reason about the *soundness* of the composed process model. This notion is an established quality criterion for workflow nets [21]. Basically, it requires each initiated process to reach eventually a proper final state. Additionally, each transition should be relevant, i.e. there should be at least one firing sequence of the process in which this transition participates. Although the second requirement was reasonable if a business process was modeled from scratch, in the Web service approach, a process arises from composition of several predefined components. Due to this, a workflow net is acceptable even if not all functionality of each specific component is used in that system. Hence, a slightly alleviated criterion is used in this paper.

Definition 4 (Weak soundness). *A workflow net (P, T, F) with the final place $\omega \in P$ is called* weak sound *if the following conditions hold:*

(i) For each reachable marking m holds: the final marking $[\omega]$ is reachable.
(ii) For each reachable marking m with $m \geq [\omega]$ holds: $m = [\omega]$.

The weak soundness of the module Planning Process shown in Figure 2(b) can be easily proven. Because of that, intuitively, the modules Default Planning and

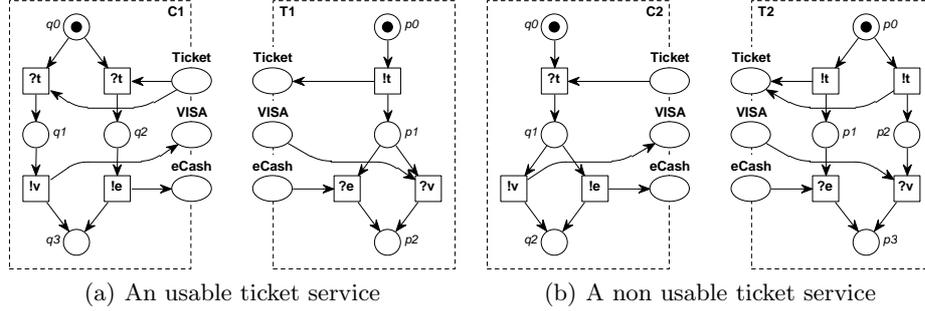(a) An usable ticket service   (b) A non usable ticket service

Fig. 3. Usability of workflow modules

Customer seem to be *semantically compatible*. In Section 3.1, the core notion of *usability* will be discuss and precisely defined. Based on this notion, definition of *semantic compatibility* can be derived that meets this intuition.

3 Properties

This section discusses the property of *usability* – the proposed soundness criterion for workflow modules – and derives the definitions of *compatibility* and *equivalence*. Obviously, the purpose of a Web service is to be bound to other components such that a proper realization of a distributed business process arises. Hence, the purpose of a workflow module is to be composed with an environment such that the resulting workflow net is at least weak sound. But, for a given module there might exist infinitely many possible environments. The question is: How many of these environments have to match to call the given module *usable*?

3.1 Usability

An example of a ticket service and a customer is used to address the question above. Figure 3(a) shows a workflow module C1 representing the customer and a module T1, which models the ticket service. The ticket service initiates the communication by sending a Ticket and waits for payment (either VISA or eCash). By receiving the Ticket, the customer solves an internal conflict and determines the kind of payment. The composed system C1 ⊕ T1 is weak sound, and therefore it seems reasonable to call these two modules *semantically compatible*. Is this enough to call both modules *usable*, as well?

Figure 3(b) shows two slightly modified workflow modules C2 and T2. The ticket service solves an internal conflict and sends the Ticket. Thereafter, module T2 is either in state p1 waiting for eCash only, or in state p2 waiting for VISA only. The customer receives the Ticket and has the choice between the two kinds of payment. But, he does not know the internal state of the ticket service module. Hence, he might choose the "wrong" payment, and the composed system C2 ⊕ T2 ends up in a deadlock, i.e. it is not weak sound.

Obviously, these two modules are *not* semantically compatible. To locate the modeling error, lets consider the other two possible combinations: The system C2 ⊕ T1 is also weak sound, whereas the system C1 ⊕ T2 may reach a deadlock, too. Hence, there are two compatible environments for module T1 and no compatible environment for module T2. By help of the developed analysis method (cf. Section 4), it can be proven that there can't be any environment that forms a weak sound composed system together with module T2. This is because of a severe error in module T2: An internal decision is made and not communicated properly to the environment. This type of errors is known in the literature as the *non local choice problem* [3]. Consequently, the module T2 is *not usable*.

For both modules C1 and C2, there is one compatible environment (T1) and one incompatible environment (T2). One might think of calling a module *usable* if *all* possible environments form a weak sound composed system together with that module. In that case, both modules C1 and C2 would be *not* usable – because of the environment T2. However, this definition is unfair: the error within the module T2 should not determine the quality of module C1. Moreover, for each given module it is possible to construct a *malicious environment* (cf. [12]). Hence, this paper proposes a more appropriate definition of usability:

Definition 5 (Usability). *Let M be a workflow module. An environment U utilizes* module M *if the composed system* $\Pi = M \oplus U$ *is weak sound. Module M is called* usable *if there exists at least one environment U that utilizes M.*

Thus, the modules C1, C2 and T1 are called usable. Section 4 presents the algorithm to decide usability of a workflow module by creating an utilizing environment (if possible). A further discussion on usability can be found in [11].

3.2 Compatibility

In the previous section, the notion of *semantic compatibility* has already be mentioned. Now, a definition of this notion shall be derived. There are two cases: If a workflow module and its environment are given, obviously, they are semantically compatible if the composed system is weak sound. But, if there are two arbitrary modules (e. g. modules Default and Route Planning shown in Figure 1), a different definition is required. Obviously, two modules are *not* compatible in case the composed system has an error, i. e. the resulting module is *not* usable. Hence, this paper proposes the following definition.

Definition 6 (Semantic compatibility). *Two syntactically compatible workflow modules A and B are called* semantically compatible *if the composed system $A \oplus B$ is usable.*

Thus, the modules Default and Route Planning are semantically compatible. Moreover, this definition also cover the composition of a module and its environment: Because the composition of those two modules yields workflow module with an empty interface (cf. Figure 2(b)), it is easy to find an utilizing environment – with an empty interface, too. Consequently, each weak sound composed system is usable. Hence, the modules Default Planning and Customer are semantically compatible. More details on compatibility can be found in [10].

3.3 Equivalence

There are many use cases, where it is crucial to decide whether two Web services behave similar, i.e. their models are *equivalent*. Beside the already mentioned problem of discovery (cf. Section 1), the problem arises if a Web service – being already part of a global business process – needs to be replaced, e.g. because of efficiency. Of course, all other participating components should stay untouched.

Intuitively, two Web services are equivalent if in a *comparable situation* one Web service *behaves like* the other and vice versa. Concerned with various formal methods, there are countless approaches published dealing with the comparison of behavior in terms of simulation or equivalence. This variety results from different interpretations of the terms "comparable situation" and "behave like" ([24] gives a substantial overview on equivalence notions). But, none of them seems to fit exactly to this field of application: The purpose of a Web service is to be used by an environment. Hence, an adequate notion of simulation or equivalence, first of all, should be derived semantically from the field of application.

Definition 7 (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 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 7.

In this approach, the verification of equivalence is based on a formal simulation relation between the *communication graphs* of both workflow modules – the explicit representation of the module's externally visible behavior, explain in the following section. Because of limited space a detailed discussion is omitted here, but some results are presented: The workflow modules C1 and C2 (shown in Figure 3) can be proven equivalent, whereas e.g. classical *bisimulation* does not yield this result. In contrast, the workflow modules T1 and T2 are proven to be not equivalent, whereas referred to *trace equivalence* those modules can't be distinguished. These examples, the comparison of selected notions of equivalence and the precise definition of the verification algorithm can be found in [13].

4 Analysis

The usability of a workflow module is defined through the existence of an utilizing environment. Hence, the definition does not describe how to disprove the usability of a given Web service. This section provides a different approach: First, an adequate representation of the Web service's external behavior is derived – called the *communication graph*. Second, the usable behavior of the Web service is determined – called the *u-graph*. Finally, the usability of a given workflow module and the algorithmic constructing its usability graph are related within the core theorem, such that usability can be decided effectively.
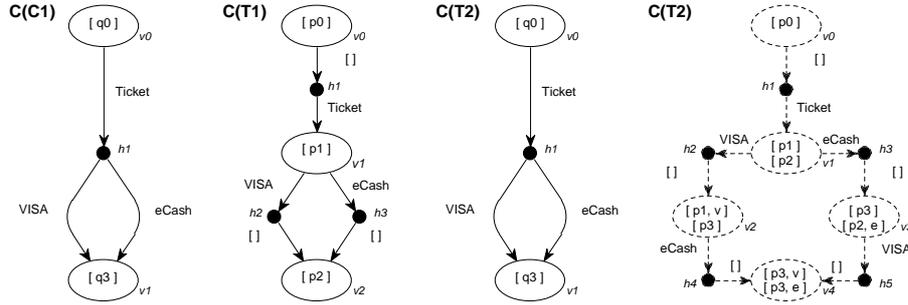
Fig. 4. Communication and usability graphs

4.1 Reflecting the behavior

A workflow module is a reactive system, it consumes messages from the environment and produces answers depending on its internal state. Problems may arise because an environment has no *explicit* information on the internal state of the module. But each environment can derive some information by considering the communication towards the module. Hence, an environment has some *implicit* information. We reflect exactly that kind of information within a data structure – called the *communication graph* (abbr. *c-graph*).

Definition 8 (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 4 presents the c-graphs of those workflow modules shown in Figure 3; Section 4.2 will explain why one graph is drawn with dashed lines. Lets have a closer look on the graph C(C1). The root node v0 is labeled with the initial state of the module [q0][1].

Each edge, starting at a visible node, is labeled with a bag of messages sent by the environment – called *input*. In the initial state, the module C1 is able to consume only the message Ticket. Each edge, starting at a hidden node, is labeled with a bag of messages sent by the module – called *output*. In the example, the customer replies to the Ticket by paying either with VISA or with eCash. An edge may be labeled with an empty bag as well, denoted by [ ]. Each path from the root to a leaf represents a complete communication sequence between the module an an environment. In general, a visible node may be labeled with more than one

---

[1] The actual labeling of the node $v0$ is $\{\,[q0]\,\}$ – the singleton containing the state $[q0]$. For reasons of simplicity, we omit these extra braces as well as the brackets around the bags of messages.

state (e. g. v1 in C(T2)), and an edge may be labeled with more than on message (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 *concurred run* [17]) 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.

The c-graph of a workflow module may contain cycles. That doesn't affect the presented analysis method as long as the graph is finite. But, workflow modules with an infinite c-graph always contain an severe modeling error. 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 [12]. Applying these notions, we are now able to present the construction of 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 the label of v_k 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 workflow module contains the maximal information an environment can derive. The environment always sends only those messages the module is able to consume (in at least one of the possible reached states), but enough to achieve an answer or to terminate the process in a proper state. By considering all reachable successor states (and all possible outputs), the choices within the module are not restricted.

4.2 Analyzing the behavior

In general, the c-graph may have several leaf nodes. But in each c-graph, there is at most one leaf node that 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 is a deadlock state of the module (e.g. v4 in C(T2)). Consequently, 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 directions for using the module – called the *usability graph* (abbr. *u-graph*). of that module.

Definition 9 (Usability graph/u-graph). *A finite, non-empty subgraph U of the c-graph C is called* usability graph *if the following conditions hold:*

- *U contains the root node of C and only that leaf node of C, which is labeled with the defined final state of the workflow module.*
- *For each hidden node of C that is in U, all outgoing edges are in U, as well.*
- *Each node within U lies on a path between the root and the defined leaf node.*

An u-graph arises by removing only those edges (and succeeding nodes) that start at a visible node. Hence, it restricts the behavior of an utilizing environment (some *inputs* are removed), but it does not restrict the behavior of the module (all *output* are still available). In that sense, an u-graph of a workflow module describes a *instruction manual* for a possible environment.

In Figure 4, those parts of the c-graphs that does not belong to any u-graph are drawn with dashed lines. The graph C(T2) does not contain any u-graphs. Hence, this module is not usable. Referring to modules C1 and C2, the whole c-graph is their only u-graph. But in general, a c-graph may contain several u-graphs. The whole graph C(T1) is an u-graph, and removing the hidden node *h2* (or *h3*, resp.) yields another u-graph, i.e. this ticket module can be used as well by a customer who exclusively pays with eCash (or VISA, resp.). To find the maximal u-graph, the algorithm walks backwards through the c-graph, and removes nodes according to Definition 9.

4.3 Theorem of usability

An u-graph U of a module M can easily be transformed into an environment of the M: Each node of the graph becomes a place, each edge starting at a visible node becomes a sending transition, and each remaining edge becomes a receiving transition of the environment. The resulting module is called the *constructed environment*, denoted by $\Gamma(U)$. Based on the restrictions of U, it is easy to prove that the composed workflow module $M \oplus \Gamma(U)$ does not contain any deadlock. If $M \oplus \Gamma(U)$ also does not contain any livelock (i.e. the composed module always can terminate), the constructed environment $\Gamma(U)$ is an utilizing environment of M. The following theorem formulates the correlation between the usability of a workflow module and the existence of an u-graph:

Theorem 1 (Usability). *Let M be a workflow module and let C be the c-graph of M. The module M is usable, if and only if C contains at least one u-graph U and the composed system $M \oplus \Gamma(U)$ always can terminate.*

The entire proof can be found in [12]. The following paragraph sketches its idea. *Implication*: As already mentioned, it can be proven easily that $M \oplus \Gamma(U)$ does not contain any deadlock. Hence, if the composed system terminates, it must have reached the desired final state, otherwise it would have reached a deadlock. Thus, $\Gamma(U)$ is an utilizing environment of M. In case M is an acyclic module, termination is granted, and the theorem only requires the existence of an u-graph. *Revers implication*: If the module M is usable, there is at least one utilizing environment E. Based on this property it is possible to project the all reachable states of the composed system $M \oplus E$ to the c-graph of M (without sticking to one specific environment). This yields a subgraph $C' \subseteq C$, which can be proven to meet the requirements of Definition 9. Hence, $\Gamma(C')$ is an utilizing environment of M.

The theorem on usability makes it possible to decide usability in many cases: An acyclic workflow module has a finite c-graph. Thus, we can search for an u-graph and decide usability. The most cyclic modules have a finite c-graph, too. Even if an usable cyclic module has an infinite c-graph, there exists a finite u-graph. Applying breadth-first-search, this graph will be found after finite time.

5 Summary

In this paper, a framework for modeling and analyzing Web service based business processes by help of Petri nets was presented [12]. Each Web service has an interface and an internal process structure. Hence, a Web service is modeled in terms of a *workflow module* – a workflow net with a set of interface places. Based on this formalism, the notion of *usability* was defined – an intuitive and locally provable soundness criterion for workflow modules. On top of usability, the questions *compatibility* and *equivalence* could be precisely addressed, and there are effective algorithms to verify all these properties. Due to the available Petri net semantic of BPEL4WS [18], the method is directly applicable to real world examples.

Of course, the current work was inspired by many other approaches, dealing with the problems of cross-organizational workflow and Web services. Some approaches also use Petri nets [20,6] and/or specify the global interaction by help of *Message Sequence Charts* (MSC) [22,8]. But, none of them presents such a focussed view on a components externally visible behavior as the *communication graph* does. Due to this representation, the comparison of behavior is more adequate w. r. t. the field of application than traces [24] or automaton [26].

All presented algorithms are implemented within the prototype Wombat4ws [27]. Currently, the work is focussed on improving the algorithms' efficiency by the application of *partial order reduction* techniques [19]. 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* [14]), 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. Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services.* Springer-Verlag, December 2002.
2. Andrews, Curbera, Dholakia, Goland, Klein, Leymann, Liu, Roller, Smith, Thatte, Trickovic, and Weerawarana. *BPEL4WS – Business Process Execution Language for Web Services.* OASIS, Standard proposal, Version 1.1, July 2002.
3. H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. LNCS 1217. Springer Verlag, 1997.
4. Karl Gottschalk. *Web Services architecture overview.* IBM developerWorks, Whitepaper, September 2000. `http://ibm.com/developerWorks`.
5. Paul W.P.J. Grefen and Samuil Angelov. *Three-Level Process Specification for Dynamic Service Outsourcing.* LNCS 2472. Springer-Verlag, 2003.
6. Rachid Hamadi and Boualem Benatallah. *A Petri Net based Model for Web Service Composition.* In *Proc. of ADC 2003*. Australian Computer Society, Inc., 2003.
7. Kavantzas, Burdett, Ritzinger, and Lafon. *Web Services Choreography Description Language.* W3C Working Draft, Version 1.0, October 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. Heather Kreger. *WSCA – Web Services Conceptual Architecture.* IBM Software Group, Whitepaper, 2001. `http://ibm.com/webservices/pdf/WSCA.pdf`.
10. Axel Martens. On compatibility of web services. *Petri Net Newsletter*, 65:12–20, October 2003.
11. Axel Martens. *On Usability of Web Services.* In Calero, Diaz, and Piattini, editors, *Proc. of Intl. Conference Workshop WQW 2003*, Rome, Italy, December 2003.
12. Axel Martens. *Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services.* PhD thesis, WiKu-Verlag Stuttgart, 2004.
13. Axel Martens. Simulation and equivalence between bpel process models. In *Proc. of Intl. Conference DASD'05*, San Diego, California, April 2005.
14. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
15. Alexander Norta. *Web Supported Enactment of Petri-Net Based Workflows with XRL/flower.* LNCS 3099. Springer-Verlag, June 2004.
16. W. Reisig. *Petri Nets.* Springer-Verlag, 1985.
17. W. Reisig. *Elements of Distributed Algorithms – Modeling and Analysis with Petri Nets.* Springer-Verlag, 1998.
18. K. Schmidt and Ch. Stahl. A Petri net semantic for BPEL. In Ekkart Kindler, editor, *Proc. of 11th Workshop AWPN*. Paderborn University, October 2004.
19. Karsten Schmidt. *Explicit State Space Verification.* Postdoctoral thesis, Humboldt-Universität zu Berlin, 2002.
20. W. M. P. van der Aalst. *Modeling and Analyzing Interorganizational Workflows.* In *Proc. of CSD'98*. IEEE Computer Society Press, 1998.
21. 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.

22. W. M. P. van der Aalst. Interorganizational Workflows – An Approach based on MSC and Petri Nets. *Systems Analysis - Modelling - Simulation*, 34(3), 1999.

23. W.M.P. van der Aalst and A.H.M. ter Hofstede. Yawl: Yet another workflow language. Qut technical report, fit-tr-2003-04, Queensland University of Technology, Brisbane, Australia, 2003.

24. Rob J. van Glabbeek. *The Linear Time - Branching Time Spectrum.* In Baeten and Klop, editors, *Proceedings CONCUR 90*, LNCS 458. Springer-Verlag, 1990.

25. Web-Ontology Working Group. *Ontology Web Language for Web Services.* OWL-S 1.1, November 2004.

26. A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. *Matchmaking for Business Processes.* In *Proc. of EEE-04*. IEEE Computer Society, 2004.

27. WOMBAT4WS. *Workflow Modeling and Business Analysis Toolkit for Web Services.* hompage, `http://www.informatik.hu-berlin.de/top/wombat/`.