On Usability of Web Services

Axel Martens
Humboldt-Universität zu Berlin,
Department of Computer Science,
10099 Berlin, Germany,
E-mail: martens@informatik.hu-berlin.de

Abstract

This paper is concerned with the application of Web services to distributed, cross-organizational business processes. Web services provide a platform independent concept of components and composition. Thus, they seem to be a proper technology to cover the heterogenous structures within distributed business processes: One Web service realizes a local subprocess. A distributed business process is realized by the composition of a set of Web services.

Although the technological basement is given, there is a lot of open questions: Do two Web services fit together in a way, that the composition yields a deadlock-free system? – the question of compatibility*. Can one Web service be exchanged by another within a composed system without running into problems? – the question of* equivalence*. Can we reason about the quality of one given Web service without considering the environment it will by used in? In this paper we present the notion of* usability *– our quality criterion of a Web service. This criterion is intuitive and can be easily proven locally. Moreover, this notion allows to answer the other questions mentioned above.*

The approach and the results presented in this paper are taken from a larger framework for modeling and analyzing business processes by help of Web services published in my PhD thesis [9].

1 Introduction

In this paper, we regard Web services as components of distributed business processes. The aim is to establish locally provable quality criteria for Web services, that guarantee global properties of the business processes. Thus, we focus on structure and behavior rather than technical aspects.

1.1 Business Processes

For a couple of years, business processes have become the center of enterprise information systems. To an increasing extent, business processes run across the borders of individual enterprises. The distribution by space and the organizational independence of participating companies prohibit the application of traditional workflow management. Instead, the cross-organizational processes are divided into self-contained components with a defined interface. The dynamic composition of these components forms business processes, whereas each component can be implemented autonomously.

To hide the heterogeneous structures within the participating companies, there is a need for standardized concepts of components and composition. The *Web service approach* provides a group of technologies to meet this requirement [1]. One Web service realizes a local subprocess. A distributed business process is realized by the composition of a set of Web services.

1.2 Web services

A *Web service* is a self-describing, self-contained, modular application that can be described, published, 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.

The *service oriented architecture* [8] describes 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. Thus, the service requester can bind its own components to the selected Web service. In this paper,

one Web service represents a local process. We introduce an analysis method to support a service provider while modeling the Web service.

Instead of one new specific technology, the Web service approach provides group of closely related, established and emerging technologies to model, publish, find, and bind Web services – the *Web service technology stack* [6].

These technologies guarantee a minimum of *syntactical* compatibility. But there is a need for *semantic* compatibility. Lets 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 process leads to a deadlock.

Most of the Web services technologies are still in the standardization process. Because there are inconsistencies and ambiguities left in the initial drafts, specification will be changed several times until a consistent status is reached. To address the core problems of distributed business processes, we prescind from the technical aspects and use Petri nets to model the behavior of Web services.

1.3 Modeling

At the moment, there are various languages in use for modeling business processes and Web services. Some of them come along with a formal semantics. For most of them, no analysis is directly applicable.

In this paper, we apply Petri nets to model business processes and Web services. Thus, we can adopt the rich theory of distributed systems to solve the problems described above. Petri nets are a well established method for modeling and analyzing distributed business processes [7]. As we will show, Petri nets are also an adequate modeling language for Web services.

Each Web service has an interface and an internal structure. Hence, we introduce the notion of a *workflow module* – a workflow Petri net with an interface – to model a Web services. Based on this formalism, we discuss and define the criterion of *usability* and derive further properties: e. g. *compatibility* and *equivalence* of two modules. Algorithms to verify these properties are implemented within an available prototype [10].

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: A first approach to map the *Business Process Execution Language for Web services BPEL4WS* [2] to Petri nets can be found in [9] – the whole story will be published in [12].

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

Section 3 establishes the core section of this paper. We discuss the notion of *usability* by help of examples and give a formal definition. Thereafter we strengthen the criterion and define *total usability*. Based on these two notions, we derive the property of semantic compatibility of two modules.

Because of limited space, verification of usability is just sketched in Section 4. A more detailed consideration can be found in [9]. Finally, Section 5 presents further applications of usability, e. g. to reason about equivalence.

2 Modeling

Our modeling approach is based on Petri nets. Besides the intuitive graphical representation, Petri nets allow an effective analysis. By help of an example, we introduce *workflow modules* and demonstrate their composition. Thereby we give references to the Petri net theory. A basic introduction into the application of Petri nets to business processes can be found in [19].

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 1 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 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) [14]. 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 [17]. Thus, we model a business process in terms of a workflow net.

A Web service consists of internal structures that realize a local subprocess (modeled e. g. by help of BPEL4WS) and an interface to communicate with its environment, i. e. other Web services (specified by help of WSDL [5]). Thus we model a Web service by help of

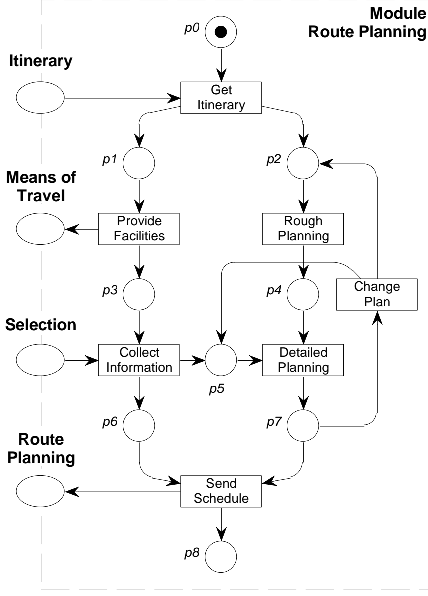

Figure 1. A workflow module

a workflow net supplemented by an interface, i. e. a set of places representing directed message channels. Such a model we call *workflow module*. A workflow module can be a model both for an *executable process* as well as for an *abstract process* (cf. [1]).

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.

(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 1 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. In general, there are two ways to specify the composition: On the one hand, a global model specifies the *orchestration* of a whole set of web services [3]. On the other hand, each Web service specifies the interaction with its *partners* [2]. We will follow the second approach and therefore 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 2 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 2, 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 3 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:

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

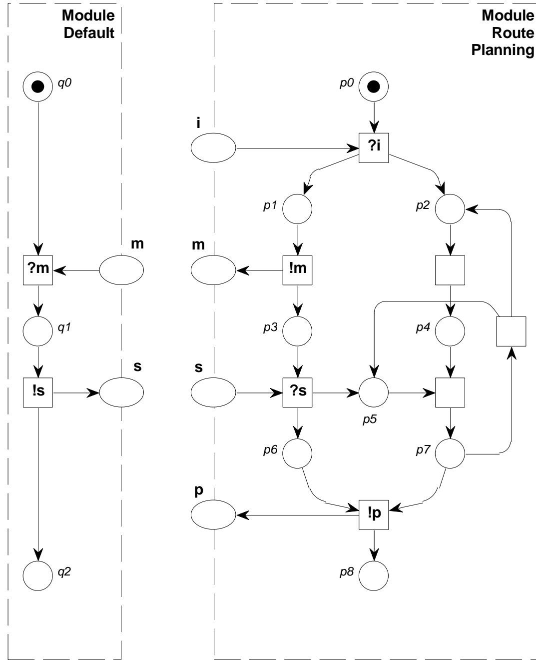

Figure 2. Syntactically compatible modules

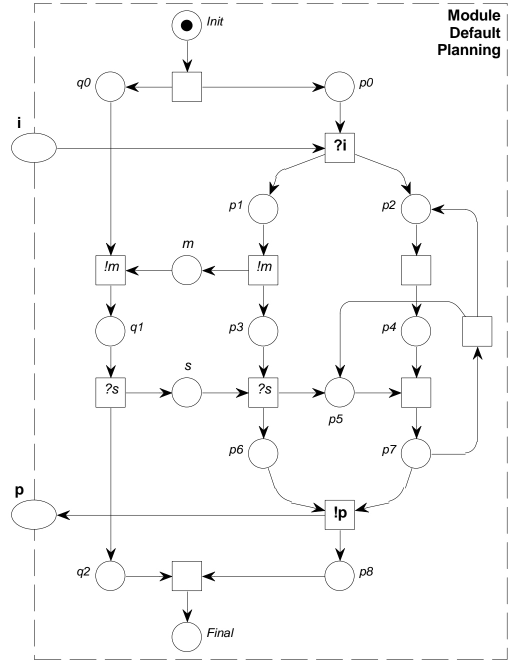

Figure 3. A composed workflow module

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

Figure 3 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 compatible 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 [9]. The components for initialization and termination are added to an composed workflow module for reasons of syntactical correctness. If we compose more than to modules, it is important to guarantee associativity of pairwise composition. Hence, if one module already contains such syntactic components, we merge the corresponding elements while composition. Section 3.3 presents an example. The next section deals with a special case: two syntactically compatible modules with completely matching interfaces.

2.3 Environments of Web services

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 4 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.

In some cases we want two workflow modules to form composed system with an empty interface, although one module is not an environment of the other module. In such cases we expand the modules syntactically, i. e. we add transitions that are never reached such that each module is an environment of the other. An exam-

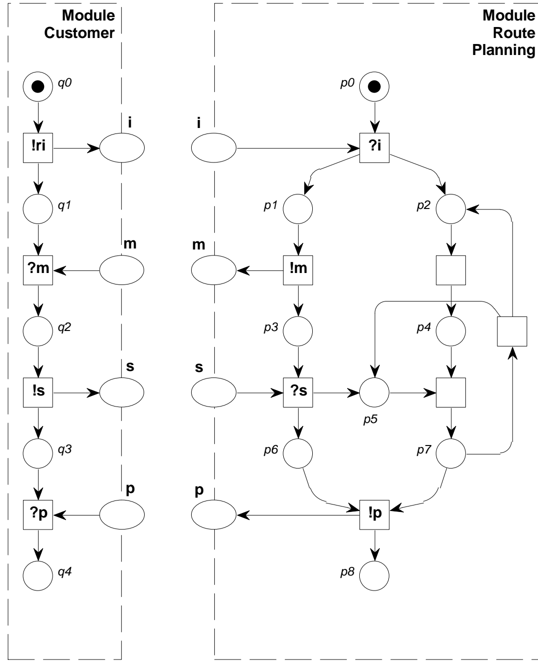

Figure 4. A module and its environment

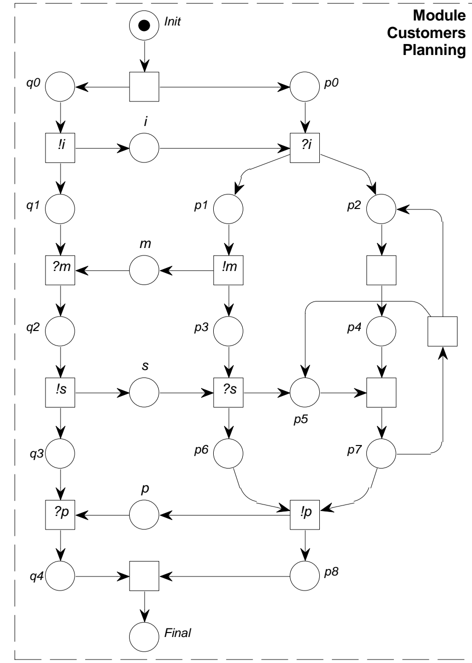

Figure 5. A composed workflow module

ple is shown in Figure 8.

Given a workflow module and one environment, it is possible to reason about the quality of the composed system. The notion of *soundness* [18] 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. Thus, we use the slightly different notion of *weak soundness*.

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 $m = [\omega]$. $\star$

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 [9].

Figure 5 shows a workflow net which is the composed system build from the modules shown in Figure 4. 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 Route Planning and Customer *semantically compatible*. In the next 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.

3 Usability

In this section, we discuss our quality criterion. Concerning workflow nets, the soundness property is a well established criterion because of the simple, intuitive definition and the algorithmic provability. To reason about the behavior of a workflow module, we always have to consider the influence of a concrete environment. Nevertheless, our aim is to establish a quality criterion for a workflow module, which is as intuitive as soundness and can be proven locally.

3.1 Discussion

Obviously, the purpose of a workflow module is to be composed with an environment such that the resulting

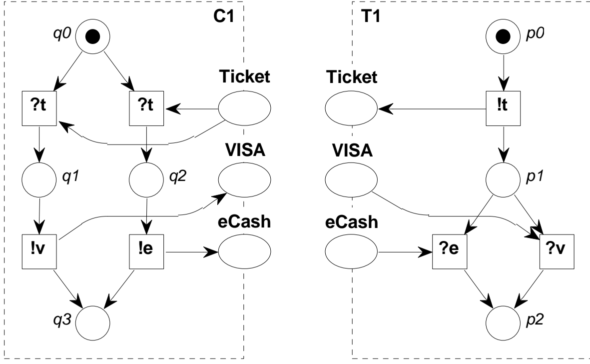

Figure 6. Two compatible modules

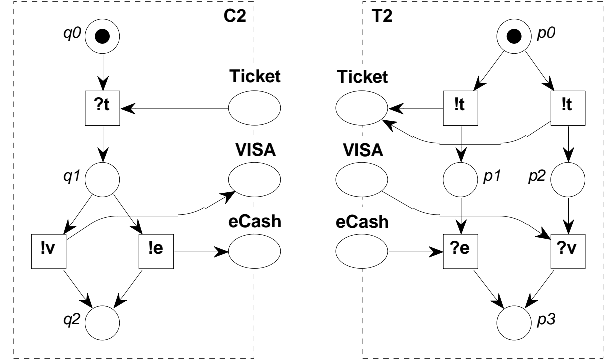

Figure 7. Two incompatible modules

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?

Lets have a look at some examples. We consider a ticket service and a customer and present two alternative models of each.

Figure 6 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 fixes the kind of payment. It can be easily proven: The composed system C1 $\oplus$ T1 is weak sound. Thus, we call both modules semantically compatible.

Figure 7 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 or in state p2 waiting for VISA.

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. Thus, he might choose the "wrong" alternative. The composed system C2 $\oplus$ T2 may end in a deadlock, thus it is not weak sound. The two modules are not semantically compatible.

To locate the modeling error, we look at two other combinations: C2 $\oplus$ T1 and C1 $\oplus$ T2. The first system is also weak sound, whereas the system C1 $\oplus$ T2 may end in a deadlock too. As result, we found two compatible environments for module T1 and no compatible environment for module T2. Concerning modules C1 and C2, we have found one compatible and one incompatible environment for each.

3.2 Definition

Within the module T2 we can locate an error: An internal decision is made and not communicated properly to the environment. This error is known in the literature as the *non local choice problem* [4]. Thus, there is no environment at all such that the composed system is weak sound – we will give a proof in Section 4. Thus, we call the module T2 *non usable*.

One might think of calling a module *usable*, iff all possible environments form a weak sound composed system together with that module. In that case, both modules C1 and C2 would not be 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. Because of the following proposition, no module at all would be usable.

Proposition 3.1 (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. $\star$

The proof can be found in [9]. Because of that we give another, more appropriate definition of usability:

Definition 3.1 (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 that utilizes M. $\star$

Concerning this definition, the modules C1, C2 and T1 are called usable. We take a look at another variation of the ticket service and the customer.

Figure 8 shows a module T3 which models a ticket service that either receives payment per VISA and sends the Ticket afterwards or start by sending the Ticket. In

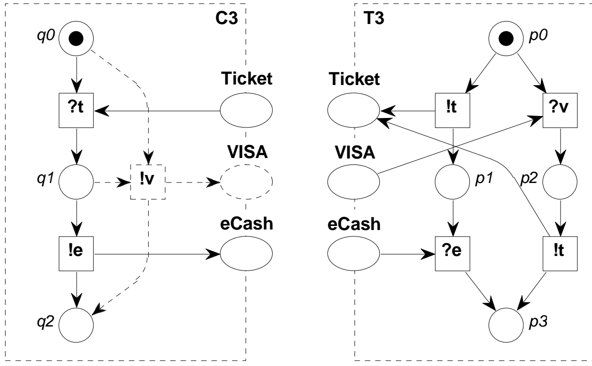

Figure 8. A partially usable ticket service

that case, the module reaches state p1 and waits for eCash. Consider a customer who pays first with VISA and awaits the Ticket. The service might have sent the Ticket meanwhile and does not accept VISA anymore. Such a customer would not utilize module T3.

Instead, the customer modeled by module C3 awaits the Ticket before sending eCash. The composed system C3 $\oplus$ T3 is weak sound, thus module T3 is usable. The transition !v within module C3 has just be added for syntactical reasons but will never ever be reached. Without this transition, module C3 would not be an environment of T3.

Although the module T3 is usable, no customer who wants to pay with VISA can utilize this service as we have explained above. Thus, we like to establish a stronger quality criterion that points out this shortcoming of module T3.

First we need a brief statement on runs of Petri nets: The behavior of a Petri net can be annotated by a set of partial ordered runs (see [15]). Each run is again a special Petri net. Thus, we talk about the behavior of a module that is used by a given environment.

Definition 3.2 (Coverage).
Let M be a workflow module and U an environment. U *covers* M, iff for each run ρ_m of the internal processes of M there is a run ρ_s of the composed system $\Pi = M \oplus U$ such that ρ_m is a subnet of ρ_s. ⋆

For a deeper understanding of the theory behind see [9]. Now we can define *total usability*:

Definition 3.3 (Total usability).
Let M be a workflow module.

(i) An environment U *utilizes* the module M *totally*, iff U utilizes M and U *covers* M.

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

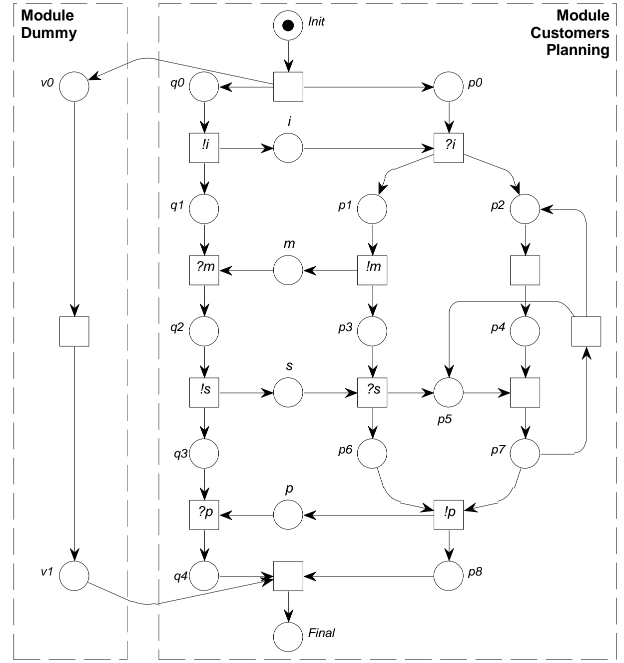

Figure 9. A workflow net with environment

Concerning this definition, modules C1, C2 and T1 are called totally usable. Because the definition is based on the existence of an environment, we can't directly prove module T3 not to be totally usable. In Section 4 we will provide the framework for such a proof.

3.3 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. But if we consider the two module shown in Figure 2, 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 3.4 (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.
- A and B are called *totally semantically compatible*, iff there exists an module C such that $C \oplus A$ *utilizes* B *totally* and $C \oplus B$ *utilizes* A *totally.* ⋆

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 modules Route Planning and Customer presented in Section 2.3. The composed system of these two modules is shown in Figure 5. There, we can see a workflow net that is also a workflow module with an empty interface. Thus, we can find an environment with an empty interface, too.

Figure 9 shows the composed system Customers Planning $\oplus$ Dummy. For reasons of simplicity, we have merged the components for initialization and termination. Because the module Customers Planning already is a workflow module with an empty interface, the environment consists of just one transition. Nevertheless, this environment (totally) utilizes the composed system Customers Planning. Thus, concerning Definition 3.4, the modules Route Planning and Customer are called (totally) semantically compatible.

4 Verification

Concerning the definition, usability of a module can be proven by help of an utilizing environment. In this section, we briefly expose a method to decide whether or not a module is usable. Thus, we introduce a new formalism – the *communication graph* – and present the algorithms of it construction and analysis. A more detailed characterization including the proofs of all theorems can be found in [9].

4.1 Communication graph

To decide the property of usability of a module, we want to look at this module in isolation. As the examples of the previous section have shown, the analysis of its structure is not sufficient. Thus, we have to consider the behavior of a module.

Basically, a module reacts on an input from its environment depending on its current internal state. On the one hand, an utilizing environment has to consider this state, but, on the other hand, the environment has no explicit knowledge of this state. It can only deduce the state by help of the communication trace. The *communication graph* is the appropriate data structure that reflects the knowledge an environment has.

Figure 10 presents the communication graph of module T1 shown in Figure 6. A communication graph (*c-graph* for short) is a directed bipartite graph. A visible node (drawn as an ellipse, e. g. v0) represents a set of states of the module. A hidden node (drawn as filled circle, e. g. h1) divides the interaction between module an environment: an arc between a visible and a hidden

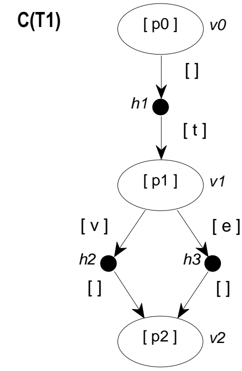

Figure 10. A communication graph

node is labeled with an input to the module and analogously an arc between a hidden and a visible node is labeled with an output of the module. States and labels are multisets of tokens and therefore written in squared brackets.

The root of a c-graph represents the initial state of the module. Starting in [p0], module T1 can produce output [t] without any input from the environment ([] stands for empty multiset). Afterwards, the environment has a choice between [v] and [e]. In both cases, the module does not answer but reaches its final state. In the following section, we will explain the construction of the c-graph.

4.2 Construction

The c-graph stands for the knowledge a *most clever* environment can deduce about the module. That means, the environment tries to avoid errors by sending as little input as possible yet, but as much as necessary. For a given state z, the function $\text{INP}(z)$ yields a set of minimal inputs such that all behavior of the module still is possible. A precise definition of this function and the following is given in [9].

To reason about the module's reaction towards a given input, we just have to add tokens on the input places and fire transitions according to the Petri net semantics. For a given state z and an input i, the function $\text{NXT}(z + i)$ yields a minimal set of reachable states that cover all behavior of the module. Remark: $z + i$ again is a state of the module.

To build the c-graph, last but not least we need a function that selects the produced output while the module fires. For a given state z and an input i, the function $\text{OUT}(z + i)$ yields a set of maximal outputs such that $o \in \text{OUT}(z + i)$ is an output if there is a reachable state $(z' + o) \in \text{OUT}(z + i)$.

Now we can describe the algorithm. It starts with

the root node which only contains the initial state of the module. Each visible node is a set of states that does exist at most once in the graph. If a set is calculated twice, these nodes are merged. For reason of simplicity, we omit this case:

For each state $z \in v$, calculate the set $\text{INP}(z)$. For each element i of the unification of all these sets:

1. add a new hidden node h; the arc gets label i.
2. For each state $z \in v$, calculate the set $\text{OUT}(z+i)$. For each element o of the unification of all these sets:
   a) add a new visible node v'; the arc gets label o.
   b) For each state $z \in v$, calculate the set $\{z' \mid (z'+o) \in \text{OUT}(z+i)\}$. Assign the unification of all these sets to v' and goto start again.

Within the communication graph of the module T1 shown in Figure 10, each visible node contains only one state. As we will see in Figure 11, such a node has often more than one state. Moreover, a c-graph may contain cycles and it might be infinite, but this does not affect our approach.

4.3 Analysis

Each path within the c-graph starting at the root node represents one *communication sequence* between the module and its environment. A maximal sequence ends in a leaf node. Since we require a composed system to be weak sound, we can distinguish good leaf nodes (containing only the final state of the module) from bad leaf nodes.

The c-graph of module T1 has only one, good leaf node. Figure 11 shows the c-graphs of modules T2 and T3. Later, we will refer to the dotted lines. As we remember, we claimed module T2 not to be usable. The c-graph of this module has only one, bad leaf node. According to theorem 4.1, this proves module T2 not to be usable.

The c-graph C(T2) also points out the problem of the module T2. After the ticket [t] was send to the customer, the module might be in two different states. Regardless of which payment the customer chooses ([v] or [e]), there is the possibility of a wrong decision.

A c-graph might contain good leaf nodes as well as bad leaf nodes. We take a closer look at the c-graph C(T3). If the customer waits for the ticket (sending nothing = [ ]), the communication sequence leads to a good end. In contrast, paying directly with VISA (sending [v]) leads to two different states of the module

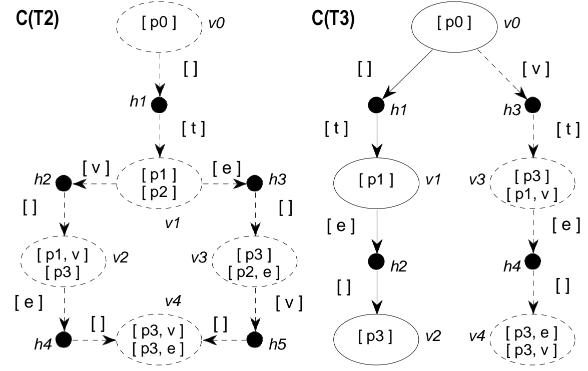

Figure 11. The usability graphs

([p3] and [p1, v]). The customer has no opportunity to distinguish between them. Thus, this communication sequence leads to a bad end. Nevertheless, there is at least one good sequence and therefore the module T3 is usable.

In general, to prove usability we need to find a finite subgraph within the communication graph of the module, such that the subgraph contains the root node and exactly one good leaf node, and it covers all reactions of the module, i. e. all outgoing edges of a hidden node are still present within the subgraph. We call such a subgraph a *utilization graph* (*u-graph* for short) of the module.

A communication graph may contain several utilization graphs. For example, the whole c-graph of module T1 is an u-graph. Moreover, the subgraph without node h2 (resp. h2) is an u-graph, too. The c-graph of module T2 has no u-graph at all and finally, the c-graph of module T3 has exactly one u-graph – shown in Figure 12. Every communication graph contains at most one maximal utilization graph. In our examples, the maximal u-graph has been drawn with a solid line, the remainder has been drawn with a dashed line.

Roughly spoken, an u-graph of a model can be found by walking backwards through the c-graph, starting at a good leaf node. By application of breadth-first search strategy it is possible to prove usability even if the c-graph in infinite. For more details see [9].

The utilization graph contains those communication sequences, that yield to a proper final state. Thus, the u-graph can be easily transformed into an utilizing environment. Figure 12 shows on the right side U(T3) – the maximal utilization graph of module T3 and on the left side, the constructed environment Γ(U(T3)).

For a given u-graph U the constructed environment is denoted with $\Gamma(U)$. The construction of an environment runs strait forward: Every node becomes a place, every arc becomes a transition which sends or receives

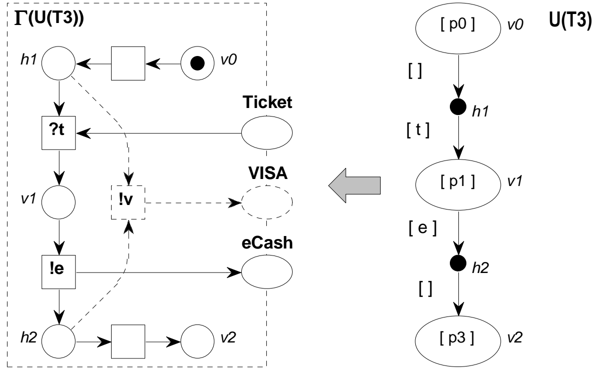

Figure 12. The constructed environment

messages w. r. t. its label. Thus, the module shown in Figure 12 has five internal places, and four transitions are arranged to a sequence. For syntactical reasons, we have to add a dummy transition !v (cf. Section 3.2) subsequently. Because of the construction, the module Γ(U(T3)) is an utilizing environment of module T3.

The construction of the communication graph, as well as the search for the utilization graph and the transformation into an environment, has been implemented within a prototype [10]. The complexity of the c-graph construction is unfortunately exponential in the number of conflicts within the module. Its analysis and transformation, however, has a linear complexity.

4.4 Theorems

Now we have all fundamentals available to formulate the relations between the notion of usability and the communication graph resp. utilization graph. We omit the proofs in here and refer to [9].

The main result of our research is summarized in the first theorem. The algorithms explained above yield an environment of the module which either utilizes this module or the module is not usable at all.

Theorem 4.1 (Usability).
A workflow module M is usable, iff its communication graph contains an usability graph C and the constructed environment $\Gamma(C)$ utilizes the module M. ⋆

Because of the limited space, we cannot present the proof in here – it can be found in [9]. The idea is the following: If we have an usability graph, then we can construct an environment. Due to of the motivation of the usability graph, obviously, the composed system of the module and this environment has no deadlock. But in case the module is cyclic, the composed system may run into an endless loop. Thus, we have to check for termination and usability is proven. If a module is usable, there exists at least one environment that utilizes this module. Without specifying a concrete environment, we can classify reachable states of the module and guarantee the existence of an usability graph.

In case of acyclic workflow modules, we do not have to check for termination, because every run of such a module is finite.

Corollary 4.2 (Usability): An acyclic workflow module M is usable, iff its communication graph contains an usability graph C. ⋆

By help of this corollary, we can finally prove our supposition: Module T2 is not usable, because there is no usability graph within its communication graph (cf. Figure 11). Analogously, we can formulate the relationship between the maximal usability graph of a module and its property of total usability.

Theorem 4.3 (Total usability).
A workflow module M is totally usable, iff its communication graph contains a maximal usability graph C_{max} and the environment $\Gamma(C_{max})$ utilizes the module M totally. ⋆

One direction of Theorem 4.3 follows by definition. We sketch the argumentation of the opposite direction: The maximal usability graph contains all communication sequences that lead to a proper final state. If an environment, that covers all these sequences does not utilize the given module totally, there can be no other environment, which does so.

Thus, we can also prove our second supposition: Module T3 is not totally usable. The provider of this Web service should either restructure the module to make it totally usable or omit the non-usable part within the published model.

5 Summary

In this paper, we have sketched a framework for modeling business processes and Web services by help of Petri nets. This framework enables 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 *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? On the other hand, the analysis of usability offers a starting point for *re-engineering* of components. Both applications are described in [9].

All presented algorithms are implemented within a prototype [10]. 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 transformation of a real world modeling language into our framework – at the moment, our group is working on a Petri net semantics of BPEL4WS [11]. On the other hand, we want to expand or notion of composition towards the orchestration of a whole set of web services. This work is related to [13].

Last but not least, we work on structural criteria to indicate problems which prohibit usability, e. g. the already mentioned non local choice problem [4]. Thus, a service provider may get a hint on demand while modeling the services, without running an expensive analysis [16].

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, Dec. 2002.

[2] Andrews, Curbera, Dholakia, Goland, Klein, Leymann, Liu, Roller, Smith, Thatte, Trickovic, and Weerawarana. *BPEL4WS– Business Process Execution Language for Web Services.* Version 1.1, July 2002. http://ibm.com/developerworks/webservices/library/ws-bpel/.

[3] Arkin, Askary, Fordin, Jekeli, Kawaguchi, Orchard, Pogliani, Riemer, Struble, Takacsi-Nagy, Trickovic, and Zimek. *Web Service Choreography Interface.* Version 1.0, May 2000. http://www.w3.org/TR/SOAP/.

[4] H. Ben-Abdallah and S. Leue. 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.*, pages 259–274, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.

[5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *WSDL – Web Services Description Language.* W3C, Standard, version 1.1 edition, Mar. 2001. http://www.w3.org/TR/wsdl.

[6] K. Gottschalk. *Web Services architecture overview.* IBM developerWorks, Whitepaper, Sept. 2000. http://ibm.com/developerWorks/web/library/w-ovr/.

[7] E. Kindler, A. Martens, and W. Reisig. Interoperability of workshop applications – local criteria for global soundness. In W. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management.*, LNCS 1806, pages 235–253. Springer-Verlag, 2000.

[8] H. Kreger. *WSCA – Web Services Conceptual Architecture.* IBM Software Group, Whitepaper, May 2001. http://ibm.com/webservices/pdf/WSCA.pdf.

[9] A. Martens. *Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services.* PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin, 2003.

[10] A. Martens. WOMBAT4WS– *Workflow Modeling and Business Analysis Toolkit for Web Services.* Humboldt-Universität zu Berlin, Institut für Informatik, 2003. http://www.informatik.hu-berlin.de/top/wombat.

[11] A. Martens, W. Reisig, and M. Weber. *Task Force – Geschäftsprozesssprache BPEL4WS.* Lehrstuhl Theorie der Programmierung, Humboldt-Universität zu Berlin, Projektbeschreibung, 2003. http://www.informatik.hu-berlin.de/top/forschung/.

[12] A. Martens, C. Stahl, D. Weinberg, D. Fahland, and T. Heidinger. Business process execution language for web services – semantik, analyse und visualisierung. Informatik-Bericht, Institut für Informatik, Humboldt-Universität zu Berlin, to appear 2004.

[13] M. Mecella, F. P. Presicce, and B. Pernici. Modeling e-service orchestration through petri nets. In Buchmann et al., editor, *Technologies for E-Services: Proceedings of TES'02*, LNCS 2444, pages 38–47. Springer-Verlag Heidelberg, 2002.

[14] W. Reisig. *Petri Nets.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, eatcs monographs on theoretical computer science edition, 1985.

[15] W. Reisig. *Elements of Distributed Algorithms – Modeling and Analysis with Petri Nets.* Springer-Verlag, 1998.

[16] W. Richter. Spezifikation und Implementation organisationsübergreifender Geschäftsprozesse mit Petrinetzen. Master's thesis, Humboldt-Universität zu Berlin, 2002.

[17] W. M. P. van der Aalst. A class of petri net for modeling and analyzing business processes. Computing science report 95/26, Eindhoven University of Technology, 1995.

[18] W. M. P. van der Aalst. Structural characterizations of sound workflow nets. Computing science report 96/23, Eindhoven University of Technology, 1996.

[19] 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.