Modeling and Model Checking Web Services

Holger Schlingloff

Institut für Informatik
Humboldt-Universität zu Berlin
Berlin, Germany,
and
Fraunhofer FIRST
Berlin
Germany

Axel Martens

Institut für Informatik
Humboldt-Universität zu Berlin
Berlin, Germany

Karsten Schmidt

Institut für Informatik
Humboldt-Universität zu Berlin
Berlin, Germany

Abstract

We give an overview on web services and the web service technology stack. We then show how to build Petri net models of web services formulated in the specification language BPEL4WS. We define an abstract correctness criterion for these models and study the automated verification according to this criterion. Finally, we relate correctness of web service models to the model checking problem for alternating temporal logics.

Key words: Web Service, Workflow module, Model Checking, Alternating Temporal Logic, BPEL4WS, Service Oriented Architecture.

[1] Email:

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

1 Introduction

Service-oriented architectures are a new paradigm in the development of communicating computational systems which are used in business organization. Today, more and more administrative and organizational tasks such as procurement, document handling, business transactions and management aspects are transferred to fully automated systems or at least supported by computing systems. Web service technologies are expected to fundamentally change the way such systems are constructed and how internal and external systems will interact.

For example, five years ago an information system for public administration which is used to collect and process forms containing personal data would be conceived as follows. The system is supposed to support up to 18.000 users on 10.000 clients with 120 backend servers. It allows stationary and mobile access via dedicated fiber optics and reserved radio bands and has an interface module to national and European networks. It consists of three main components, a large business transaction processing client comprising more than 350 forms and wizards, an information server accessing a very large data base, and a communication part allowing a distributed processing of forms and scheduling of activities. The client component keeps a local database of open transactions and connects to the PL/SQL server farm via SOAP/XML when online. The server component checks on the validity of incoming request and transfers them to the data base access programs according to a fair distribution strategy. The communication component attaches to the usual office- and email-applications and prepares the messages to be usable by the system.

Specification and documentation of the system is by approximately 300 “use cases”, which are textual descriptions of preconditions, possible user actions and system responses, resulting system states, and alternatives or exceptions. The use cases are the main contractual basis for the acceptance procedure with the ordering authority. Thus, verification and testing have been done by translating the use cases into formal descriptions, and then checking a model of these descriptions and constructing test cases from them. A major difficulty in this enterprise is the “black box” view onto a large distributed system: the use-cases describe the overall behaviour of the system, whereas for testing and verification activities only parts of it were available.

For the development of the system “from scratch” about 45 person-years were necessary; the design, implementation and placing into operation took approximately 5 years. With such a setting, service-oriented architectures promise a significant reduction in development cost and time. In particular, interoperability with external systems and existing components can be achieved much more easily. Moreover, verification and testing are greatly facilitated since they are performed on a component level.

Intuitively, a web service is “a web site for use by computers”. In a service-oriented architecture, a service is a function that is well-defined, self-contained,

and does not depend on the context or state of other services [2]. It is devised to be published, accessed and used via intra- or internet. A service provider is a component offering some service for (public or limited) use. A service broker maintains a catalogue of available services, which can be looked up and located. The service requestor searches a service from the service broker, and then attaches to the service provider by composing the offered service with its own components. If the service requestor thus establishes a new service, it may become a provider which again is registered with the service broker.

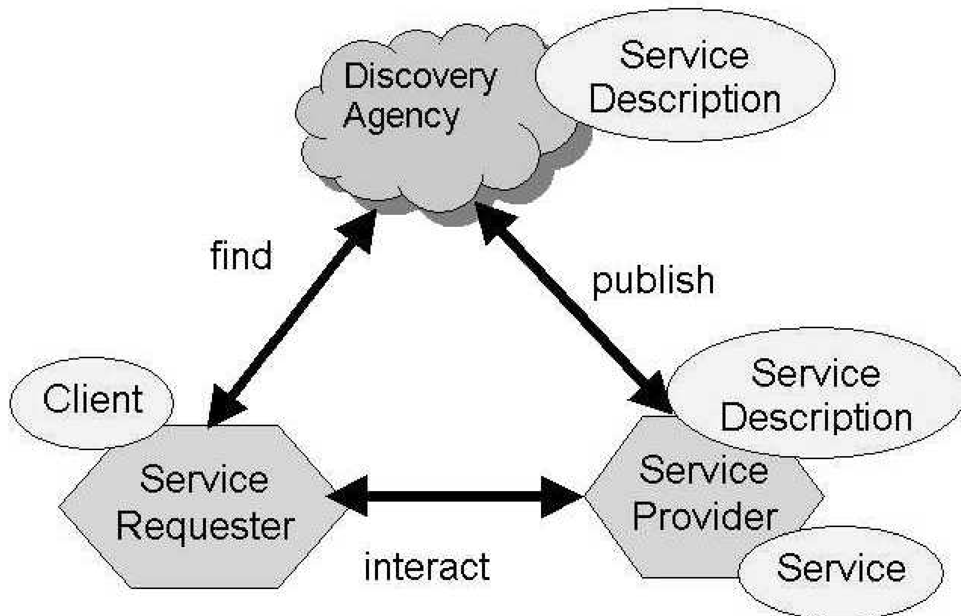

Fig. 1. service oriented architecture

A web service thus offers an arbitrarily complex functionality, which is described in a global directory structure. It can be used by end users or other web services and may be composed to form new web services. Web services communicate in the global internet, in an organization's intranet, and between different intranets. Examples for internet web services include information systems such as map or travel services, e-commerce systems such as web shops, travel agencies, stock brokers, auctioneering houses, and business grid appliances with a dynamic allocation of business partners. Whereas internet web services enjoy the greatest public visibility, the biggest current growth area for web services is in intranets: examples include the above transaction processing system, organisation of business activities, production scheduling and supervision, etc. Web services between intranets are not commonly used because of security concerns.

In contrast to prior service-oriented architectures like DCOM, CORBA and Enterprise Java Beans, web services connect to each other via XML mes-

sages. More specifically, the web service technology stack defines an upcoming standard for the communication in service oriented architectures. It is comparable to the ISO/OSI layered model for telecommunication and consists of the following layers: transport, messaging, description, quality of service, and business process modelling. The transport layer builds on classical computer network layers with the usual communication via TCP/IP and HTTP. On top of that is the messaging layer, where XML (the extended markup language) is used for the exchange of structured data items, and SOAP (the simple object access protocol [5]) describes remote procedure calls and return values with XML. Above these so-called core layers are the emerging layers which are currently being standardized: in the description layer, WSDL [7] is an XML-based language for specifying the syntactical interface of a web service (i.e., its operations and connection possibilities). A WSDL file consists of two parts: an abstract part defines language independent types, messages, operations and ports, and a concrete binding part maps abstract elements onto concrete data structures, protocols, and addresses. The quality of service layer consists of several optional items which can be used to enhance the connection: WS-Security is a language for syntactic coding of nonces and authentication information, as well as integrity and protection level of data. Protocol elements for tuning the communication between different web services can be given in the language WS-Coordination, and WS-Transaction is for supervising a running process and starting corrective measures during the run. There are several more suggestions for additional languages and protocol elements on this level. Presently, the top of the stack is formed by BPEL4WS, the business process execution language for web services [8], which is used to actually describe the sequence of interactions which comprise a web service. The Universal Description, Discovery and Integration language UDDI [4] allows to describe web services for lookup, similar to yellow pages in a telephone book.

Since the languages and interfaces in this technology stack are standardized, web services can be developed independently and distributed. Thus, it is possible to focus on the modelling and validation rather than on technical details of the communication. The situation is similar to the development of internet home pages, where the designer should focus on the content rather than on particular transport and rendering problems. In particular, for verification purposes it suffices to build an abstract version of a BPEL4WS description, given that the lower layers have been shown to perform correctly. BPEL4WS provides basic activities such as invoke, receive, and reply, which are used for communication with other web services, and basic activities which are elementary actions on XML files such as assign, copy and wait. Structured activities include the usual control flow elements from programming languages such as sequence, while, pick, switch, etc., and also flow and link constructs for parallelism and synchronisation. A particularity is the introduction of scopes which limit the range of tasks such that retraction of activities is possible via compensation handlers.

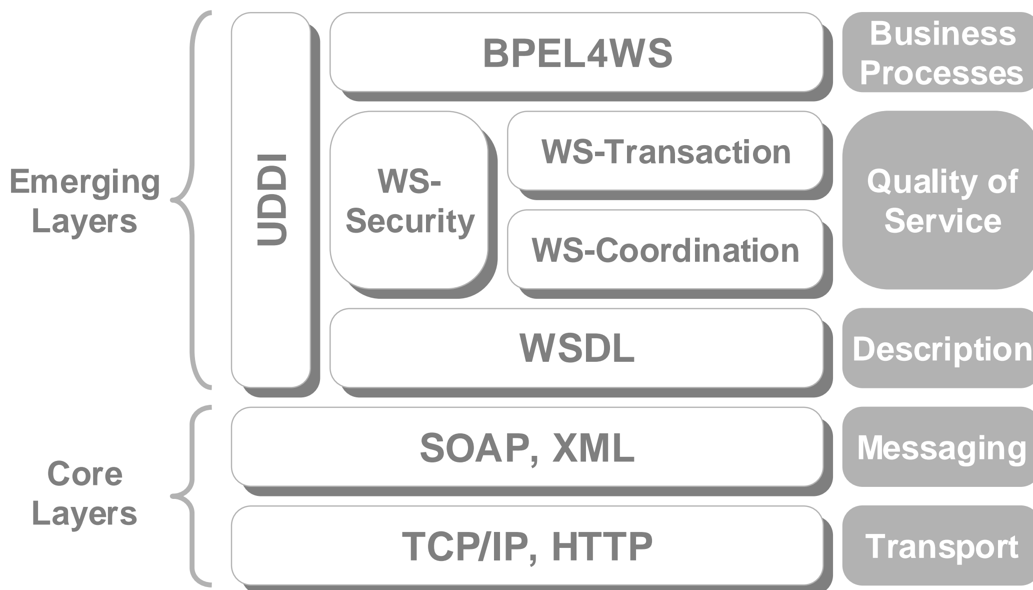

Fig. 2. The web service technology stack

2 Modelling BPEL4WS processes with Petri nets

The original goal of modelling BPEL processes with Petri nets was to give the language BPEL4WS a formal semantic, and to compare the applicability of several formalisms for this task (in parallel to a Petri net semantic [21], we are developing a semantic based on Abstract State Machines [9]). Through the formalistation of the informal semantic [8], we found several errors, ambiguities, and weaknesses, mostly originating from the conceptual differences between IBM's WSFL [12] and Microsoft's XLANG [23], the two "parents" of BPEL4WS. Some of our comments were included into subsequent working drafts.

The Petri net semantic for BPEL4WS consists of a set of Petri net patterns, one for each BPEL activity. The patterns are place bounded Petri nets. The boundary places form the interface to other patterns and have a distinguished meaning such as incoming and outgoing control flow, successful and unsuccessful termination as well has fault, compenstation, and event handling. Furthermore, global variables, modeled as Petri net places, can be accessed. Fig. 3 shows a particular pattern for a simple activity *invoke*, responsible for triggering another process via sending a message.

As an example for a structured activity, we show a pattern for BPELs flow construct, where sub-activities are executed in parallel.

For validating the semantic, we have started a student project for implementing a parser from BPEL4WS to Petri nets. As a first step, we built manually the Petri net corresponding to a simple purchase order business process (see Fig. 5) that is used as an example in the specification [8] of the language. It contains several invoke (sending a message), receive (receive a message), and reply (answer to a received message) activities. Some of the

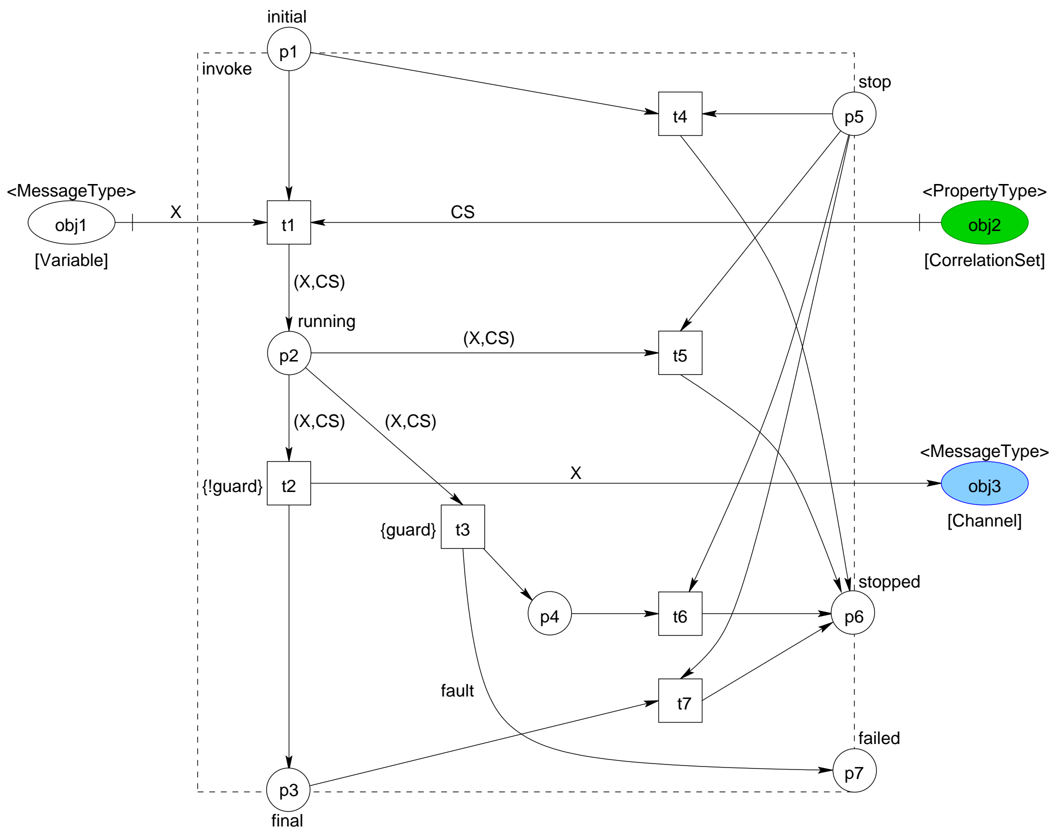

Fig. 3. Pattern for an (asynchronous) invoke activity. When started (p1 marked), it reads the content of a message from a variable and generates, some time later (p2), a message. This process may (t3) or may not (t2) lead to an error, for instance through a mismatch in BPELs correlation set mechanism which coordinates different instances of one and the same process description. At any time, the running activity may be interrupted (p5) as the result of an error occurring elsewhere. Successful termination (t4–t7) is acknowlegded (p7).

activities are arranged in sequences running in parallel. Additional links pose further causalities between activities that would otherwise run concurrently.

We resulted in a net with 158 places and 249 transitions. Thereby we abstracted from data values. In the validation process, executed with the help of the Petri net based model checking tool LoLA [17], we verified that the different patterns interact properly (e.g., stop leads to stopped, initial leads to final or failed, linked activities are executed in correct order, and so on). The whole state space consisted of 9991 states. On the machine used, LoLA can handle tens of millions of states. Furthermore, LoLA offers several state-of-the-art state space reduction techniques, including partial order reduction [24,15,18], the symmetry method [10,22,18], and the sweep-line method [19]. While the symmetry method was not applicable in this example, joint application of partial order reduction and the sweep-line method lead to substantial reduction, that, however, depends on the particular property. The best reduction, achieved when checking for termination, boiled the state space down to 1286

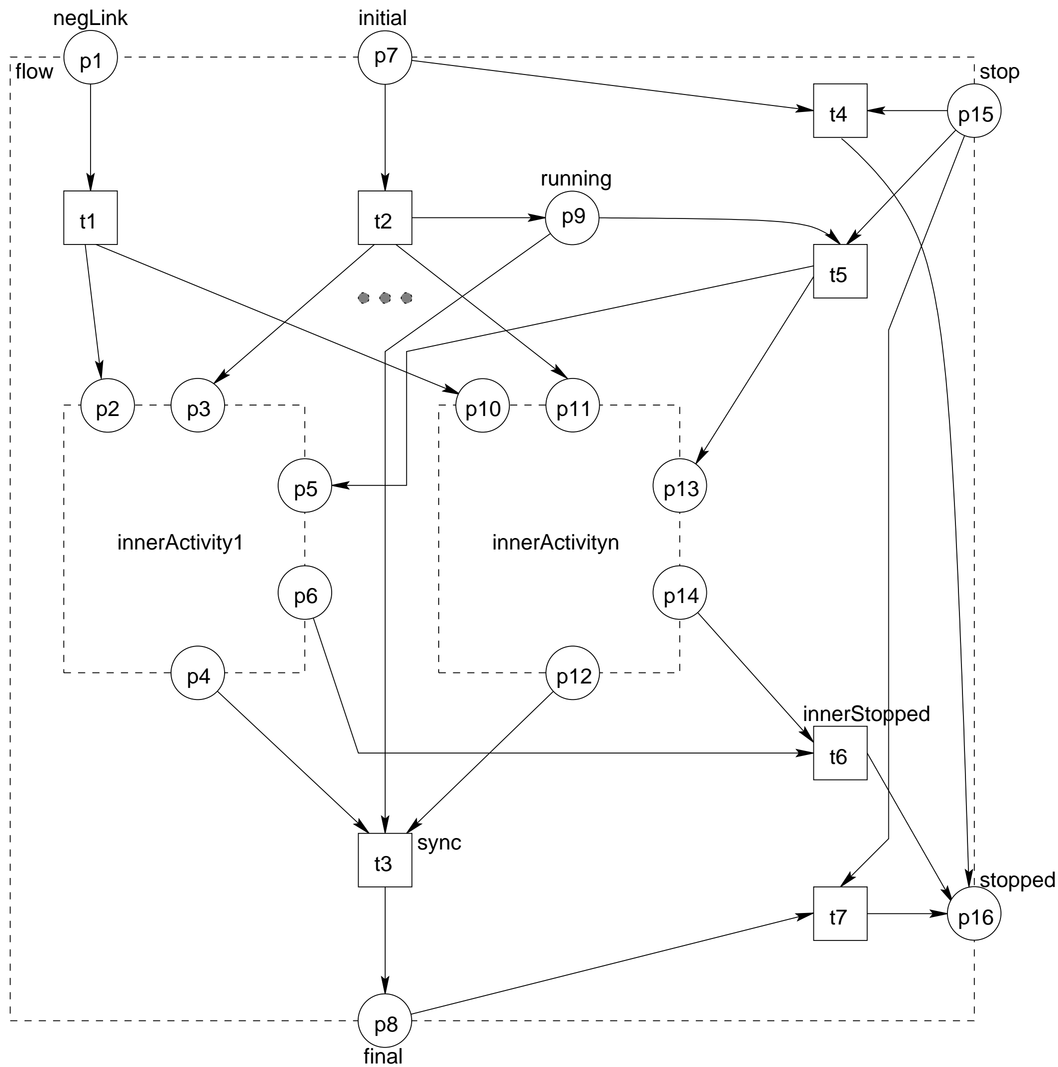

Fig. 4. Pattern for a flow activity. The dashed boxes carry the patterns for sub-activities. When started (p7), control flow forks (t2), and the sub-activities are started (p3, p11). After successful termination of both activities (p4,p12), the flow activity itself terminates (p8). When forced to stop (p15), the inner activities are forced to stop (p5,p13), and upon acknowledgment, flow itself acknowlegdes (p16). Places p1, p2, and p10 implement BPELs link concept that poses additional dependencies between concurrent activities. For better readability, we left out the mechanism for propagating faults thrown from the inner activities.

states. At this time, we cannot extrapolate these measures to larger examples. Nevertheless, we could show that there are chances for successful verification of larger processes. In particular, standard reduction techniques seem to work well on the generated Petri nets. Furthermore, we have additional ideas for further state space reduction. For instance, we are investigating the possibility to replace certain patterns by simpler patterns, when the property to be verified does not concern the pattern, or when static program analysis on the BPEL source code provides additional information (e.g., impossibility of

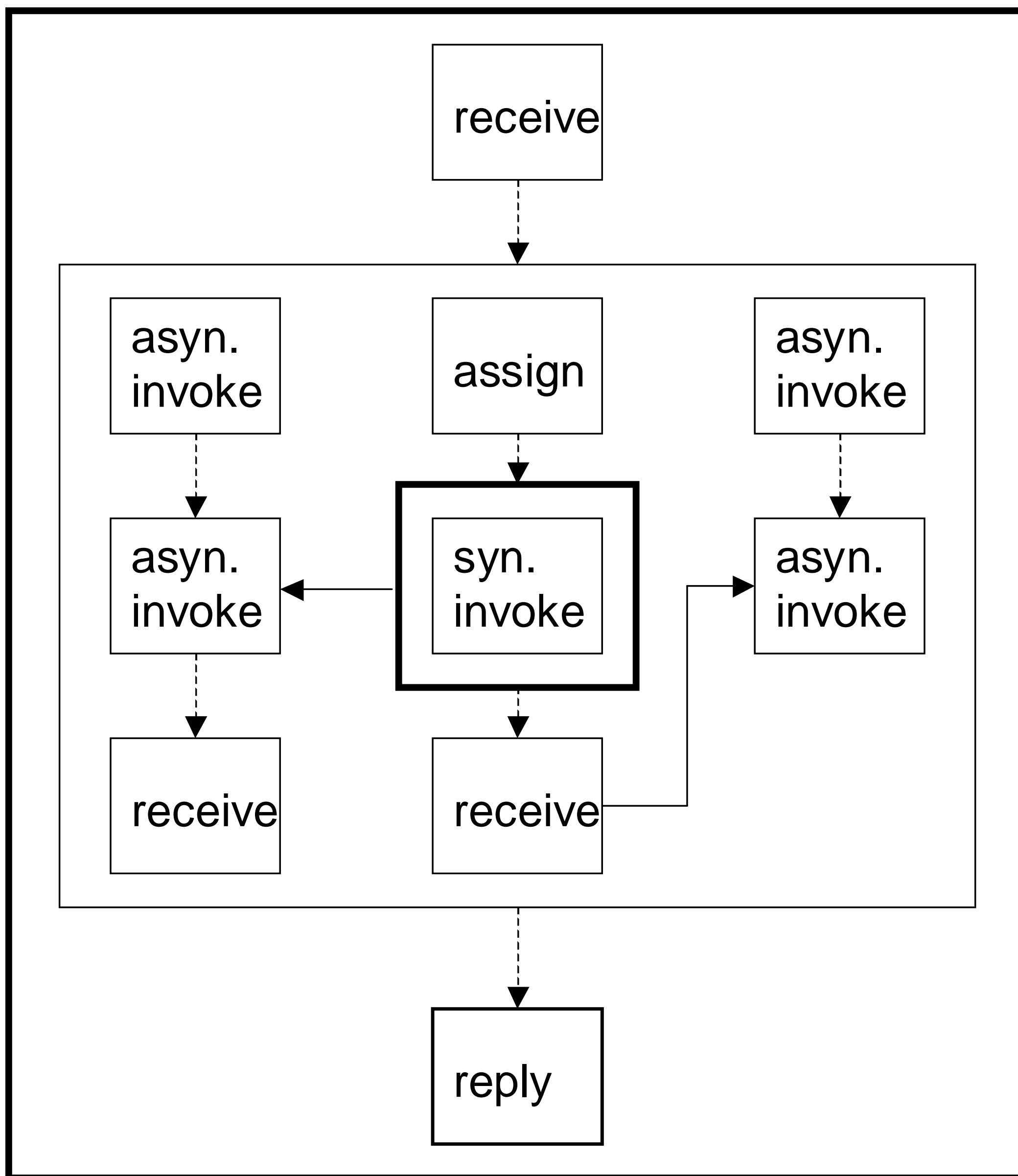

Fig. 5. This process is triggered by an incoming message. Then three activities run in parallel: calculating the final price (left), selecting a shipper (middle), and scheduling production and shipment (right). Edges between different threads symbolize links.

errors in inner activities).

3 Usability analysis

For more theoretical issues related to Petri net models of web services, it is convienient to study simpler models. For this purpose, we proposed *workflow modules* [13]. This concept allows us to study questions about well-behaviour of a service, such as usability, composability, and abstraction.

Usability is a criterion derived from the successful *soundness* criterion [25] for workflows: is it possible, from every reachable situation, to terminate properly, i.e., with no garbage left? For web services, proper termination depends on suitable interaction with the environment. So the new question is: is it possible for the environment to extract, from the description of the service itself, a strategy for sound interaction with the service? For a well-designed process, the answer should always be "yes", as web services are usually required to be self-explaining. Theoretically, this question turns out to be a special kind of controller synthesis problem, for which we have partial solutions. For the *composability problem*, two or more services are given and we ask whether they interact properly with each other. This is an important problem as practioneers intend to offer web services that are arranged just by a composition of third-party service (e.g., an online travel agency is expected to be little more than a composition of several online airline booking services, hotel reservation services, and car rental services). Services that interact properly should enjoy at least the property that their composition becomes a sound, or usable service. Finally, *abstraction* is important as enterprises do not like to publish their business processes. Instead, they would generate public views that hide internals but give sufficient information for proper interaction. The relation between private and public view of a service can be seen as an abstraction relation in the process algebraic sense. [13] contains some results on checking consistency between the views, and for automated public view generation. In the sequel, we present our results concerning the usability problem.

Definition 3.1 [Petri net] A Petri net $N = [P, T, F, m_0]$ consists of two disjoint sets P (places) and T (transitions) a relation $F \subseteq (P \times T) \cup (T \times P)$ (arcs), and a marking m_0 (the initial marking). A marking is a mapping $m : P \longrightarrow \mathbf{N} \cup \{0\}$.

Definition 3.2 [Behavior of Petri nets] Transition t is enabled in marking m if, for all places p, $[p, t] \in F$ implies $m(p) \geq 1$. Transition t can fire in marking m leading to marking m' ($m[t > m'$) if t is enabled in m and, for all p, $m'(p) = m(p) - W([p, t]) + W([t, p])$ where $W(f) = 1$ for $f \in F$ and $W(f) = 0$ for $f \notin F$. Denote the set of all reachable states with R_N.

Workflow modules [25] have a distinguished start place α and a distinguished end place ω. An interface is a set of places. Tokens on interface places are interpreted as messages sent via asynchronous channels. It is important to understand that the order in which messages are received may differ from the order in which they were sent.

Every interface place represents either messages from the service to a partner, or messages from a partner to the interface. That is, the service is connected to the interface place in only one direction. Furthermore, we assume that a service reads or writes only one message per transition. It may, however, perform transitions that do not interact with the interface at all. The concept of *module* formalizes our view on web services as workflow modules

equipped with an interface.

Definition 3.3 [Module] $M = [\alpha, \omega, P_M, P_I, P_O, T_M, T_I, T_O, F]$ is a module if (i) $\{\alpha, \omega\}$ (the start and end place, resp.), P_M (the set of internal places), P_I (the set of input ports), and P_O (the set of output ports) are pairwise disjoint, (ii) T_M (the set of internal transitions), T_I (the set of read transitions), T_O (the set of write transitions) are pairwise disjoint, (iii) $[\{\alpha, \omega\} \cup P_M \cup P_I \cup P_O, T_M \cup T_I \cup T_O, F, m_0]$ is a Petri net with $m_0(\alpha) = 1$ and $m_0(p) = 0$ for all other places p, (iv) for all places and transitions x, $[\alpha, x]$ and $[x, \omega]$ are in the reflexive and transitive closure of F, (v) every write transition is connected to exactly one output port and no input port, (vi) every read transition is connected to exactly one input port and no output port, (vii) every internal transition is connected to neither an input nor an output port.

Throughout the remainder of this section, we consider an arbitrary but fixed module $M = [\alpha, \omega, P_M, P_I, P_O, T_M, T_I, T_O, F]$.

Usability can be studied in a central as well as in a distributed setting. For distributed usability, we need to know which ports belong to one and the same partner. We restrict ourselves to a setting where every partner has exclusive access to some ports.

Definition 3.4 [Interface partition] An interface partition of M is a partition U of $\{P_I \cup P_O\}$.

Every class in such a partition is the set of ports one particular partner may access. In the setting of central usability, U contains just one class $P_I \cup P_O$. Fig. 6 depicts a module with internal places (with greek letters as names), input ports a,b,g,h, and output ports c,d,e,f. The dashed boxes represent the interface partition $\{\{a, c, d, g\}, \{b, e, f, h\}\}$.

Informally, we think of a partner as a system that interacts with its distinguished part of the module's interface (one particular class of U). We postulate that such a system has states that control the activation of certain activities, and that the activities include sending messages to, and receiving messages from the module. We further postulate that a partner cannot infer any knowledge about the module's internal state at a certain stage beyond the exploitation of the structure of the module (the Petri net) and the history of communication until that point. That is, we assume that different partners do not communicate with each other, that a partner does not have access to internal places of the module or interface places belonging to other partners, and that it is not sufficient for usability to "guess" a strategy. We further want to exclude settings where a strategy is successful only with a certain probability.

From these postulates it is apparant to think about a partner as an automaton where transitions trigger interaction with the module's interface.

Definition 3.5 [Partner] Let $L \subseteq P_I \cup P_O$. A partner of M serving L is an automaton $A = [Q, bags(L), \delta, q_0]$ with a set of states Q, multisets of ports as

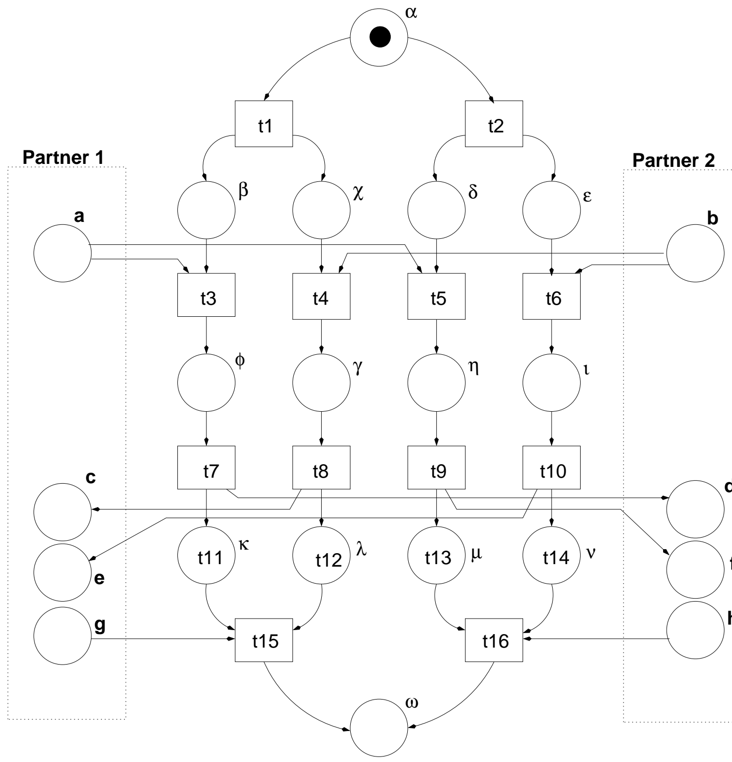

Fig. 6.

alphabet, a nondeterministic transition function $\delta : Q \times bags(L) \rightarrow \wp(Q)$, and an initial state q_0.

Fig. 7 depicts a partner serving $\{a, c, d, g\}$ (left), and a partner serving $\{b, e, f, h\}$ (right) of the module in Fig. 6. The interaction of partners with a module is formalized as a transition system. This system is basically the interleaved parallel composition of all components. In a step of a partner, the annotated multiset is interpreted as receiving and sending of appropriate messages (i.e., removal or production of tokens on the port places). We skip the formal definition, as it is obvious.

Most of our results hold for acyclic modules (i.e., the transitive closure of F is irreflexive). Some observations simplify our theory for acyclic modules. First, we may restrict ourselves to finite automata. We may assume a finite limit l_M for the length of the maximum reasonable number of communication steps. l_M can be determined from the workflow net. We may disregard silent moves and moves that perform more than one send or receive action at a time. We may further restrict ourselves to free (tree-like) automata. This has the advantage that a state may code the unique history of communication. Throughout the section, we write λ for the empty word and X^* for the set of finite words over an alphabet X.

Definition 3.6 [Partner of acyclic module] Let $L \subseteq P_I \cup P_O$. A partner of M serving L is a free automaton $A = [Q, L, \delta, \lambda]$ with a prefix-closed set Q

of words over L that all have length $\leq l_M$ as set of states, the set L as its alphabet, the partial deterministic transition function δ where for all $q \in Q$ and $p \in L$, $\delta(q,p) = \{qp\}$ if $qp \in Q$, and $\delta(q,p) = \emptyset$ if $qp \notin Q$, and the empty word λ as its initial state.

Consequently, a partner of an acyclic module is uniquely determined by its set of states Q.

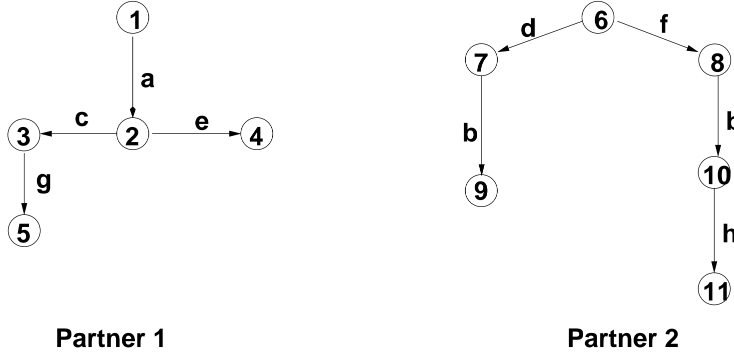

Fig. 7.

In [13], we proposed another, more condensed definition for partner behavior, called *communication graphs*. There, a partner would be represented as a bipartite graph where input and output phases alternate. For central usability, our approaches are similar, independent of the data structure used for representing partners. For distributed usability, however, the automata approach seems to be more successful. We therefore restrict ourselves to this approach for both central and distributed usability.

Successful control means to interact such that the module is brought from the initial state (only place α is marked) to a distinguished final state (only place ω is marked).

Definition 3.7 [Successful control] Let U be an interface partition $U = \{L_1, \ldots, L_k\}$. Let m_ω be the marking where $m_\omega(\omega) = 1$, and $m_\omega(p) = 0$ for all other places p. Let $A_1, \ldots, A_k$ be partners with A_i serving L_i. $A_1, \ldots, A_k$ control M successfully if from every state s it is possible to reach a state where the component belonging to M is m_ω.

For readers familiar with temporal logic, succesful control corresponds to the validity of the CTL formula $AGEFs|_M = m_\omega$ in the composed system (see next section).

The problem tackled here is: Given the (acyclic!) module M and a fixed interface partition U, do partners exist which control M sucessfully?

Intuitively, states of a partner represent knowlegde the automaton has about its environment. The following definition reveals this knowledge which is in this case knowledge about the possible states that the connected module can be in.

Definition 3.8 [Knowlegde function] Let A be a partner of module M serving L. Let $Q \subseteq L*$ be the set of states of A. The knowledge function $K : Q \longrightarrow \wp(R_M)$ is defined by: $K(q) = \{m \mid [m, q, \ldots] \in C\}$ where C is the union of the state sets of all composed systems that involve M, involve A as first partner, and arbitrary further partners.

$K(q)$ can be easily computed for arbitrary q, using the given workflow net. For several considerations in the sequel, we distinguish three kinds of deadlocks in the module.

Definition 3.9 [Deadlocks] A deadlock of a Petri net is a marking that does not activate any transition. In a module, a deadlock m_d is

(i) final iff $m_d = m_\omega$ (the latter marking as defined in Def. 3.7);

(ii) an internal deadlock iff it is not the final marking, no output port place is marked in m_d, and every transition t in M has an internal input place p in M (i.e. $p \in P_M$ and $[p, t] \in F$) that is unmarked in m_d;

(iii) an external deadlock iff there is a transition where the only unmarked places are port places (i.e. input port places), or an output port of m is marked.

Central usability is the question whether a module can be controlled by a single partner that accesses all ports of the module. We characterize successfully controlling partners and show the existence of a universal strategy, i.e. a partner that embeds all successfully controlling partners.

Theorem 3.10 (Successful control of acyclic modules) *Let $A = [Q, L, \delta, \lambda]$ be a partner (acc. to Def. 3.6) of the acyclic module M. Let A serve all ports of M. Then A controls M successfully if and only if all of the following is true for all states $q \in Q$: (i) $K(q)$ does not contain internal deadlocks. (ii) For every external deadlock m_d in $K(q)$, there is an active port place p such that $\delta(q, p) \neq \emptyset$. Thereby, an input port is always active while an output port is active in m_d if it carries a token.*

It can further be shown that, if M is usable, M has a unique universal partner. Thereby, universal means that every successfully controlling partner is embedded in the universal partner. The existence is proven by construction. We start with a partner that exhibits all behaviors permitted by Def. 3.6. Then we remove, step by step, everything that violates the characterization given in Thm. 3.10. It remains either nothing—then M is not usable—or the universal partner.

Fig. 9 illustrates the construction of a universal partner for the module in Fig. 8. The filled states are the universal partner. In the depicted values of the knowledge function, (e) marks external deadlocks, and (i) internal deadlocks. All states except a and aa are removed due to internal deadlocks. aa is removed since the external deadlock cannot be left. State a is removed since, after removal of ab, no successor is present to leave the external deadlock.

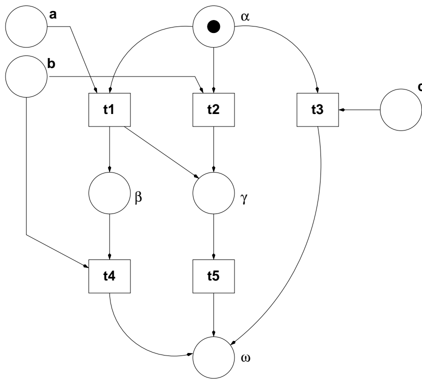

Fig. 8.

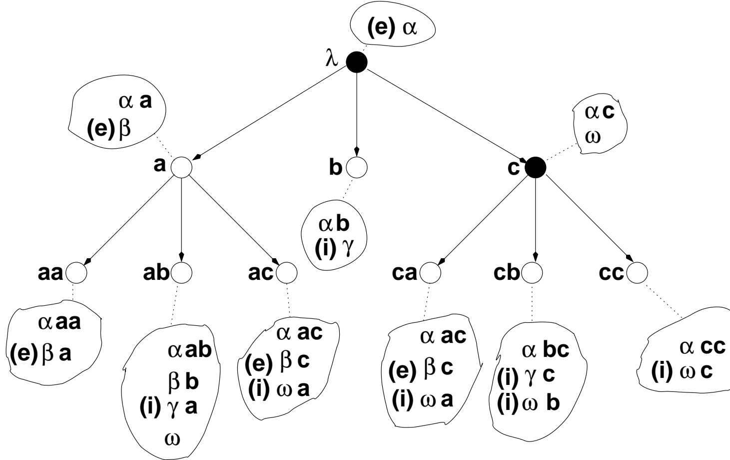

Fig. 9.

The successors of b are not drawn as all successors of states q with internal deadlocks in $K(q)$ contain internal deadlocks in their K-values, too. Fig. 10 shows another construction of a universal partner. The universal partner of the module depicted left has states with the empty set as K-values. These states satisfy trivially the conditions of Thm. 3.10. Empty K-values indicate that the partner can never be in such a state (here: b cannot be received without having sent a in advance). We might be tempted to remove such "dead code" from the universal partner. However, keeping these states in the universal partner turns out to be crucial for our results about coordinated distributed usability. There, we take up this example again.

Next we consider the capability of controlling a module that has an interface partition consisting of more than one class of ports. Assume, throughout the remainder of this section, an arbitrary but fixed partition $U = \{L_1, \ldots, L_k\}$ and let $L = L_1 \cup \ldots \cup L_k$.

It turns out that at least two different scenarios can be distinguished. In the first scenario, we have a set of partners that know how the remaining partner act. That is, they may know the algorithms according to which the other partners run. They do not, however, know the states of the remaining

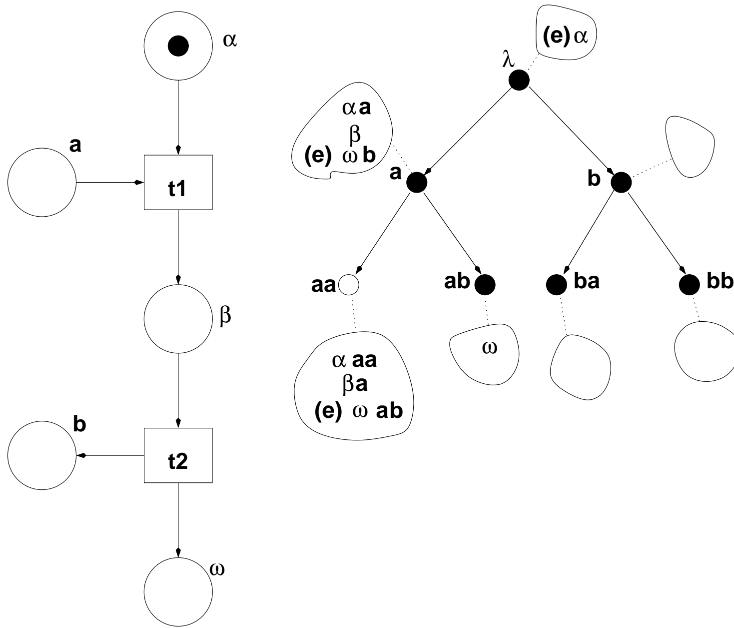

Fig. 10.

partners at run-time (otherwise, distributed usability would be equivalent to central usability).

In the second setting, the task is to interact with the module without knowing anything about the other partners beyond the general assumption that they would not willingly defect. We start with results for the first setting, called coordinated usability.

Definition 3.11 [Coordinated usability] Module M is coordinated usable if there exist partners $A_1, \ldots, A_k$ with A_i serving L_i ($i \in \{1, \ldots, k\}$) such that they control M successfully.

Every distributed strategy corresponds to a central strategy: the set of partner automata can be replaced by an equivalent product automaton. This interleaved product construction can be done easily for automata but not as easy for the bipartite partner model mentioned earlier in this section.

Techniques known from Petri net region theory [3,14] can be used to characterize "distributable" central strategies. Basically, we look for a central strategy that has "product shape", i.e. where transitions belonging to different parts of the interface do not disable or enable each other.

In the resulting algorithm, we start with the universal central strategy and then remove situations where transitions enable or disable transitions concerning other parts of the interface. We thereby may remove either the source or the target state of the influencing transition. This results in a nondeterministic algorithm which can, however, hardly be avoided, as the following consideration suggests.

Fig. 11 shows a possible result of the algorithm The two partners are the only two pairs of successfully controlling partners for the interface par-

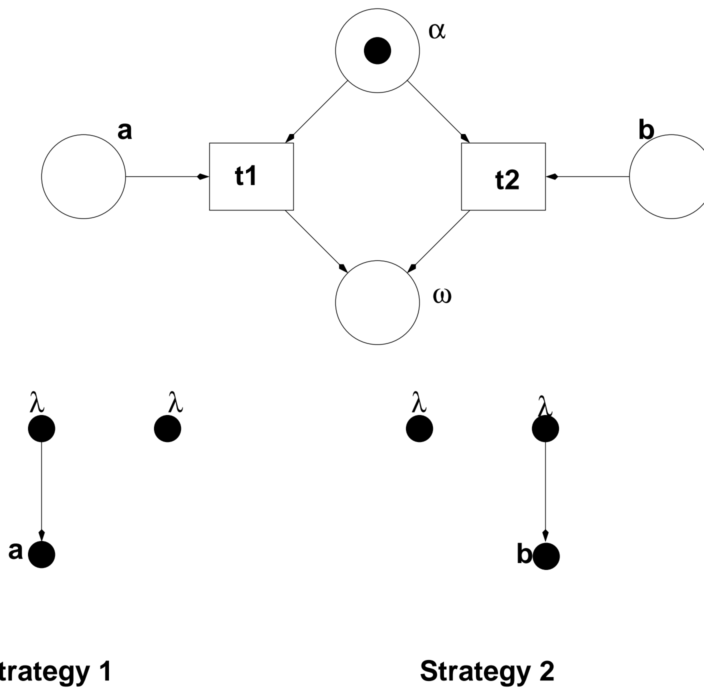

Fig. 11.

tition $\{\{a\},\{b\}\}$. Since, starting with a symmetrical module, we obtain two asymmetrical solutions (they are symmetric to each other, but not symmetric in themselves), we have some kind of indication that every algorithm that enumerates product partners explicitly needs to have some instrument for breaking symmetry, such as the choose-and-backtrack in our solution.

Now we consider the situation where a partner needs to act without knowledge about how the remaining partners act. We call this setting *uncoordinated usability*. In such a setting, the module in Fig. 11 cannot be usable. Remind that we fixed a module M and an interface partition U.

Though we want to study a partner's behavior in isolation, it is still necessary to pose some requirements on the remaining partners. Obviously, without *any* restriction on the other partners, almost every nontrivial module becomes unusable. Consider, for instance, the case where the remaining partners just do nothing or send unrequested messages. For a reasonable definition it is thus necessary to assume that the partners act reasonably, from their particular knowledge about the module. This "reasonable" behavior is formalized in the concept of *local consistency*. We can show that a module is controlled successfully whenever all partners act locally consistent. This justifies our definition of uncoordinated usability as the existence of locally consistent strategies for all involved interface classes.

In first approximation, we consider "reasonable" any partner that behaves as if it were the only partner, i.e. as if all other interface transitions of M were internal transitions. This includes avoidance of internal deadlocks as well as help in leaving external deadlocks. However, a few adjustments are necessary.

First, a partner cannot control ports belonging to other involved partners. That is, we cannot make a partner responsible for the marking on those ports. The objective of a partner must therefore be limited to bringing the module internally to its final state (to mark the final place ω while cleaning all other *internal* places), and to clean all its own port places. We cover this observation be a modified classification of deadlocks.

Definition 3.12 [New classification of deadlocks] Let $L_i \subseteq L$. A deadlock m of M is locally final w.r.t. L_i, if $m(\omega) = 1$ and $m(p) = 0$ for all $p \in P_M \cup L_i$. m is locally internal w.r.t. L_i if it is internal but not locally final.

Note that the concept of external deadlock remains unchanged.

A second modification concerns treatment of external deadlocks. In central usability (cf. Thm. 3.10) we required a partner to provide an action for every external deadlock that module M can be in. This is not reasonable for distributed usability, since there may be external deadlocks that need to be resolved by other partners. Consequently, we change the concept of active input ports to: an input port p is active in a marking m if $m(p) = 0$ and for least one transition t of M connected to p ($[p, t] \in F$), $p' \in P_M$ and $[p', t] \in F$ imply $m(p') > 0$ (i.e. all internal pre-places of t are marked). In other words, an input port is active in m if sending a message to that port can help to leave that deadlock. An output port remains active if it is marked. We are now ready to define local consistency. Remember the definition of function K (Def. 3.8).

Definition 3.13 [Local consistency] Let A_i be a partner serving L_i, and Q_i be its set of states. A_i is locally consistent if all of the following conditions hold for all $q \in Q_i$.

(i) $K(q)$ does not contain locally internal deadlocks;

(ii) for every external deadlock $m \in K(q)$ that has an active input port p in L_i, there is an active (input or output) port $p' \in L_i$ such that $qp' \in Q_i$.

Thereby, "active" concerns the conditions described above.

The conditions of local consistency are similar to the requirements of Thm. 3.10.

Theorem 3.14 *Let M be an acyclic module. If all of $A_1, \ldots, A_k$ with A_i serving L_i for all i are locally consistent then $A_1, \ldots, A_k$ control M successfully.*

With this theorem, it is justified to define uncoordinated usability as follows.

Definition 3.15 [Uncoordinated usability] Acyclic module M is uncoordinated usable if, for every class L_i of the given interface partition, there exists a locally consistent partner.

Consider the module in Fig. 6. Among the two partners depicted in Fig. 7, the left partner is locally consistent while the right one is not. For the module

in Fig. 11, it is easy to see that there are no locally consistent partners. Intuitively, it is impossible to control the module without an agreement between the partners about which of them is responsible for sending a message.

4 Usability and alternating-time logic

The notion of usability can be formulated in alternating-time temporal logic **ATL*** as defined in [1]. This logic was designed to formulate correctness properties for *open systems*, which are to be proved correct with respect to an arbitrary environment. However, in **ATL*** also the *controller synthesis problem* ([16], [11]) can be formulated: For a given system with controlled and uncontrolled states, construct a controller which always keeps the system within some safe set of states. As we will show, this problem is very close to the problem of constructing an environment which correctly uses a given workflow module.

For sake of presentational completeness, we recall the definition of game structures and the semantics of alternating-time logics. Conceptually, we are dealing with *two-player turn-based asynchronous game structures with incomplete information and perfect recall.* Such structures can be described as tuples $S = (\mathcal{M}, \mathcal{P}, \mathbf{v}, \delta, \mathit{turn})$ where

- $\mathcal{M}$ is a nonempty set of *states*,
- $\mathcal{P}$ is a nonempty set of *propositions*,
- $\mathbf{v} \subseteq \mathcal{M} \times \mathcal{P}$ is a *propositional valuation*,
- $\delta \subseteq \mathcal{M} \times \mathcal{M}$ is a *transition relation*, and
- $\mathit{turn}: \mathcal{M} \longrightarrow \{\mathit{sys}, \mathit{env}\}$ is a function indicating for each state whether the system or the environment may choose the next state.

Thus, game structures are just classical Kripke structures with an additional component conferring in each state the nondeterministic choice of the next state onto some "player" from $\Theta = \{\mathit{sys}, \mathit{env}\}$.

A *strategy with perfect recall* σ_θ for $\theta \in \Theta$ is a function which selects for each finite sequences $(m_0, ..., m_n)$ of states such that $\mathit{turn}(m_n) = \theta$ and $\delta(m_n) \neq \emptyset$ some successor state $m_{n+1} \in \mathcal{M}$ (that is, $(m_n, m_{n+1}) \in \delta$).

In other words, a strategy for the system selects, whenever it is the system's turn, one out of the various choices which the system could make to resolve nondeterminism. This selection may take into account the complete history of the computation so far. Similarly, a strategy for the environment determines the moves in states with $\mathit{turn}(m_n) = \mathit{env}$.

A *computation* of a game structure S from state $m_0 \in \mathcal{M}$ under the strategy σ_θ is a finite or infinite sequence $(m_0, m_1, m_2, ...)$ of states such that $(m_i, m_{i+1}) \in \delta$ for each i, and if $\mathit{turn}(m_i) = \theta$, then m_{i+1} is chosen according to σ_θ. Given a strategy for the environment and a strategy for the system, there is exactly one possible computation under both strategies.

In a distributed system, each global state is composed of several sub-states, one for each distributed component. Often, a player can observe and influence only particular sub-states. For example, the environment E of a web service W can observe only the outputs and influence the inputs of W. While E can choose its own state and outputs according to some strategy, it has no knowledge or control of the internal state of W.

Formally, a game structure with *incomplete information* contains for each player $\theta \in \Theta$ an equivalence partitioning $\simeq_\theta$ of the set of states which satisfies the following requirements:

- If $m_1 \simeq_\theta m_2$, then $turn(state_1) = \theta$ iff $turn(state_2) = \theta$.
- If $m_1 \simeq_\theta m_2$ and $turn(state_1) = \theta$, then for each m_1' such that $(m_1, m_1') \in \delta$ there exists an m_2' such that $(m_2, m_2') \in \delta$ and $m_1' \simeq_\theta m_2'$.

The first of these two conditions guarantees that in each equivalence class, the active player is uniquely determined. The second says that equivalent states have the same set of successor classes. (As usual, we call an equivalence class $[m']$ the *successor* of a state m, if there is some representative $m' \in [m']$ such that $(m, m') \in \delta$.)

A strategy with incomplete information σ_θ for $\theta \in \Theta$ is a function which selects for each finite sequences $([m_0], ..., [m_n])$ of equivalence classes of states such that $turn(m_n) = \theta$ and $\delta(m_n) \neq \emptyset$ some successor class $[m_{n+1}]$. Given a concrete state m_n, the second condition above guarantees that a concrete successor state in the chosen successor class exists.

Now we describe a logic to reason about game structures with incomplete information. The syntax of the temporal logic $\mathbf{ATL}^*$ is given by the following clause.

$$\mathbf{ATL}^* ::= \mathcal{P} \ \mid \ \bot \ \mid \ (\mathbf{ATL}^* \rightarrow \mathbf{ATL}^*) \ \mid \ (\mathbf{ATL}^* \, \mathcal{U} \, \mathbf{ATL}^*) \ \mid \ \langle\!\langle \Theta \rangle\!\rangle \mathbf{ATL}^*$$

That is, each proposition is a well-formed formula, the logic is closed under boolean operators, contains the temporal until-operator and an additional modality for quantification of strategies. As usual, $[\![\theta]\!]\varphi$ is short for $\neg\langle\!\langle\theta\rangle\!\rangle\neg\varphi$. Traditionally, $\mathbf{F}\,\varphi$ stands for $(\neg\bot)\,\mathcal{U}\,\varphi$ and $\mathbf{G}\,\varphi$ for $\neg\,\mathbf{F}\,\neg\varphi$.

The semantics is defined with respect to a particular state m and strategies σ_{env} and σ_{sys} of a game structure (with or without complete information) $S = (\mathcal{M}, \mathcal{P}, \mathbf{v}, \delta, turn)$.

- $(S, m, \sigma_{env}, \sigma_{sys}) \models p$ iff $(m, p) \in \mathbf{v}$ for $p \in \mathcal{P}$
- $(S, m, \sigma_{env}, \sigma_{sys}) \not\models \bot$
- $(S, m, \sigma_{env}, \sigma_{sys}) \models (\varphi_1 \rightarrow \varphi_2)$ iff
  $(S, m, \sigma_{env}, \sigma_{sys}) \models \varphi_1$ implies $(S, m, \sigma_{env}, \sigma_{sys}) \models \varphi_2$
- $(S, m, \sigma_{env}, \sigma_{sys}) \models (\varphi_1 \,\mathcal{U}\, \varphi_2)$ iff there exists a finite nonempty sequence of states $(m_0, m_1, ..., m_n)$ such that $m = m_0$, each m_{i+1} is selected according to the appropriate strategy of the player whose turn it is in m_i, $(S, m_i, \sigma_{env}, \sigma_{sys}) \models \varphi_1$ for all $0 < i < n$, and $(S, m_n, \sigma_{env}, \sigma_{sys}) \models \varphi_2$.

- $(S, m, \sigma_{env}, \sigma_{sys}) \models \langle\langle env \rangle\rangle \varphi$ iff there exists a strategy σ'_{env} such that $(S, m, \sigma'_{env}, \sigma_{sys}) \models \varphi$
- $(S, m, \sigma_{env}, \sigma_{sys}) \models \langle\langle sys \rangle\rangle \varphi$ is defined likewise

We say that $(S, m) \models \varphi$ iff $(S, m, \sigma_{env}, \sigma_{sys}) \models \varphi$ for all strategies σ_{env} and σ_{sys}.

Intuitively, the $\langle\langle \theta \rangle\rangle$ quantifier fixes one particular strategy or subtree for the evaluation of the formula. In a game structure with complete information where both the strategies of player and environment are fixed, there is only one possible computation path; therefore in this case $\langle\langle env \rangle\rangle \langle\langle sys \rangle\rangle \varphi$ is equivalent to the CTL* formula $\mathbf{E}\varphi$. The dual quantifier $[\![\theta]\!]$ "overrides" any previous binding of the strategy for θ: whenever it is θ's turn, any alternative can be taken. Thus, e.g. the CTL formula $\mathbf{A}\ \mathbf{F}\ \varphi$ can be expressed as $[\![env]\!][\![sys]\!]\ \mathbf{F}\ \varphi$.

The model checking problem is to determine whether $(S, m) \models \varphi$ for any given any S, m and φ. This problem is decidable for finite game structures with complete information in doubly exponential time, but undecidable in general for strategies with incomplete information ([1]). However, for specific instances and sublanguages, decidable efficient model checking algorithms exist.

Now we associate a game structure with every module M as described in section 3. Generally, a state in a Petri net is identified with a marking, i.e. a function from places into natural numbers. However, in our context, we also have to model partners and interfaces. Assume a module $M = [\alpha, \omega, P_M, P_I, P_O, T_M, T_I, T_O, F]$ and a partner which in any step can put a token into any input place or subtract a token from any nonempty output place. The set of states is $\{f : P \longrightarrow N\} \times \{sys, env\}$. The transition relation is given by the behavior of the Petri net and its partner, where $(m, (f, sys)) \in \delta$ iff $(m, (f, env)) \in \delta$. Of course, $turn((f, \theta)) = \theta$.

Now we give equivalence partinionings for both the environment and the system. The environment can observe only P_O. Hence, for the environment all states are equivalent which differ only in $P_M \cup P_I$. The system can observe its internal places and inputs. Thus, for the system all states are equivalent which differ only in P_O. With this definition, the two requirements on strategies with incomplete information are met.

To formulate properties of a module in **ATL***, we allow all formulas $(p = n)$ as atomic propositions, where p is a place in the net and n is a natural number. Thus,

$$\underline{\omega} = ((\omega = 1) \wedge \bigwedge_{p \neq \omega} (p = 0))$$

is a formula describing the final state in a succesfully controlled module.

With this definition, usability of a workflow module can be characterized as an **ATL*** formula. Let

$$\varphi = \langle\langle env \rangle\rangle\ \mathbf{G}\ \langle\langle sys \rangle\rangle\ \mathbf{F}\ \underline{\omega}$$

be the formula which states that there exists a strategy for the (unspecified)

environment which guarantees that for any state reachable under this strategy the system has a strategy to bring the module eventually to the distinguished final state. Then M is usable iff $(S, m_0) \models \varphi$, where S is the game structure associated with M and m_0 is the state associated with the initial marking of the net.

Other properties mentioned above such as coordinated usability can be characterized in a slightly extended framework with several players.

5 Conclusion

We described an approach to modeling web services specified in the language BPEL4WS with the help of Petri nets. We modeled an example process and were able to validate several features of the Petri net semantic using state space exploration techniques. We further were able to modelcheck correctness requirements of the process. We then proposed the notion of workflow module as a simple Petri net equivalent for web services and defined a notion of correctness for these modules, called usability. We presented results concerning computer aided usability analysis for acyclic workflow modules. The analysis covers scenarios of central as well as distributed usability. Finally, we related the usability problem to the model checking problem for alternating temporal logic.

Currently, we are implementing an automated translation from BPEL4WS to Petri nets, giving as access to larger examples. We are developing state space reduction techniques applicable to the decision procedure for central and distributed usability. Ultimately, we want to build up an integrated modeling, verification, and testing environment for web services.

On the theoretical side, we are investigating generalizations of our usability criteria to cyclic web services. This would enable us to verify a larger class of web services. Furthermore, we are developing methods for the automated generation of test cases from models of BPEL specifications. Finally, we are exploring the capabilities of alternating logics for the specification of relevant correctness criteria in the context of web service technology.

References

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time Temporal Logic. Journal of the ACM 49:672-713, 2002.

[2] D.K. Barry. Web Services and Service-Oriented Architectures. Morgan Kaufmann (2003).

[3] E. Badouel and P. Darondeau. Theory of regions. Lectures on Petri neets 1: basic models, pp. 529–258. LNCS 1491, 1998.

[4] T. Bellwood, L. Clement, and C. von Riegen. UDDI - Universal Discovery, Description, and Integration, Version 2.0. Standard UDDI.org, 2002. http://www.uddi.org/ipubs/uddi_v3.htm

[5] Box, Ehnebuske, Kakivaya,Layman,Mendelsohn,Nielsen, Thatte, Winer. SOAP - Simple Object Access Protocol. Version 1.1. Standard W3C. 2000. http://www.w3.org/TR/soap/

[6] C.G.Cassandras and S. Lafortune. Introduction to discrete event systems. Kluwer 1999.

[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. WSDL - Web Service Description Language, Version 1.1. Standard. W3C, 2001. http://www.w3.org/TR/wsdl/

[8] Curbera, Goland, Klein, Leymann, Roller, Thatte, and Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Technical report, BEA Systems, Interantional Business Machines Corporation, Microsoft Corporation, May 2003.

[9] D. Fahland. Ein Ansatz einer formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines. Studienarbeit. Humboldt-Universität zu Berlin, 2004.

[10] Huber, A. Jensen, Jepsen, and K. Jensen. Towards reachability trees for high–level petri nets. In *Advances in Petri Nets 1984, Lecture Notes on Computer Science 188*, pages 215–233, 1984.

[11] A. Ichikawa and K. Hiraishi. Analysis and control of discrete-event systems represented as Petri nets. In P. Varaiya and B. Kurzhanski, editors, Discrete Event Systems: Models and Applications, IIASA Conference, Soprpon Hungary, August 3-7, 1987, number 103 in Lecture Notes in Control and Information Sciences, pages 115134. Springer- Verlag, 1988.

[12] F. Leymann. WSFL – Web Service Flow Language. Whitepaper: IBM Software Group, 2001. http://ibm.com/webservices/pdf/WSFL.pdf

[13] A. Martens. Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services. Dissertation, Humboldt-Universität zu Berlin, 2003.

[14] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. Theoretical Computer Science 96(1992), pp. 3-33.

[15] D. Peled. All from one, one for all: on model–checking using representitives. *5th Int. Conf. Computer Aided Verification,Elounda, Greece, LNCS 697*, pages 409–423, 1993.

[16] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. SIAM J.Control and Optimization 25(1), pp. 206–230, 1987.

[17] K. Schmidt. LoLA - A Low Level Analyzer. Proc. Int. Conf. Application and Theory of Petri nets, LNS 1825, pp. 465ff, 2000.

[18] K. Schmidt. Explicit State Space Verification. Habilitation thesis. Humboldt-Universität zu Berlin, 2002.

[19] K. Schmidt. Automated generation of a progress measure for the sweep-line method. Proc. TACAS 2004, LNCS 2988, pp. 192ff, 2004. Submitted to a journal.

[20] K. Schmidt and C. Stahl. A Petri net semantics for BPEL4WS – validation and application. Workshop Algorithmen und Werkzeuge für Petrinetze, Paderborn, September 2004.

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

[22] P. Starke. Reachability analysis of Petri nets using symmetries. *J. Syst. Anal. Model. Simul.*, 8:294–303, 1991.

[23] S. Thatte. XLANG - Web services for Business Process Design. Initial public draft: Microsoft Corporation, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c

[24] A. Valmari. Error detection bu reduced reachability graph generation. *Proc. of the 9th European Workshop on Application and Theory of Petri Nets, Venice*, 1988.

[25] W.M.P. van der Aalst. The application of Petri nets to workflow management. Journal of circuits, systems, and computers 8(1) pp. 21–66, 1998.