Modeling- and Analysis Techniques for Web Services and Business Processes

Wolfgang Reisig

Humboldt-Universität zu Berlin

Abstract. Open distributed systems include in particular Web services and business processes. There is a need of techniques to model such systems formally, and to derive decisive properties from such models. Three such techniques are presented in this paper, exemplified by help of realistic examples, and mutually related w.r.t. their respective expressive power and the availability of analysis techniques.

1 Web Services and Business Processes

The term *Web service* describes a wide range of software architectures, and has no entirely clear-cut definition. But most experts in the field would agree that the following two aspects are essential for Web services:

Firstly, a Web service has a *technological basis*, which is a systematic combination of conventional middleware components for transport (e.g. TCP/ IP), messaging (e.g. SOAP, XML), description (WSDL), quality of service (e.g. WS-coordination, WS-transaction) and integration (UDDI). This combination of technologies has occasionally been denoted as the "technology stack" of Web services. Business processes as well as other distributed services can then be implemented on top of this. Hence, the technological basis of Web services is a combination of existing middleware. The essential idea of Web services is however not merely the middleware components and their combination in the technology stack, but the second aspect of Web services, its *abstraction* from its technological basis.

Web services are a prominent example for the paradigm of service oriented architectures. They in turn are intended to overcome well known problems of updating or replacing single components of conventional, monolithic IT systems. Abstracting from their technological basis, Web services themselves provide the ground for further abstractions, in particular abstractions from the technological foundation of business processes. Such abstractions turn high-level objects and operations into elementary notions. Typical examples include objects like "client" and "message to client", and operations such as "to answer a client's recent request". This kind of objects and operations are elementary in the world of business process. Their implementation in the technological basis of Web services remains irrelevant for the user of business processes.

M. Steffen and G. Zavattaro (Eds.): FMOODS 2005, LNCS 3535, pp. 243–258, 2005.
© IFIP International Federation for Information Processing 2005

2 Modeling- and Analysis Techniques

The definition of a Web service must be communicated among its designers, implementers, users, etc. This requires a language, i.e. a meta-model, capable to represent Web services intuitively and uniquely. A commonly accepted meta-model does not exist, however. Instead, notions and notations in the area of Web services emerged quickly and, occasionally, with little mutual recognition. Quoting the W3C consortium, and specifically the group involved in the Web service activity [17], a web service is "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered by XML artefacts. A Web service supports direct interactions with other software agents, using XML-based messages exchanged via Internet-based protocols." Hence, a meta-model for Web services must in particular be capable of describing interfaces and bridging of services, in addition to abstract objects as described in Section 1.

Specification languages for Web services and business processes differ fundamentally from conventional programming languages: The semantical basis of a conventional programming language essentially consists of objects such as symbols, sequences of symbols, binary integer representations, and operations such as composing and comparing symbol sequences. The corresponding theoretical framework is the world of computable functions. This framework can however not be employed as the semantical basis of specification languages for Web services and business processes, because elementary objects can be *any* items, and elementary operations can be *any* operations. We consider two approaches to tackle this problem: The first one starts out with the observation that many questions can be stated and answered without detailed semantical aspects, focusing only onto the *control structure* of services. Typical examples of such questions include necessary conditions for proper termination, usability and equivalence. Low-level Petri nets turned out to be particularly useful for this purpose. This line of research has mainly been started by [15], [16], and continued by e.g. [6], [7], [9], [8], [13], and [14]. The second solution to the above stated problem applies a kind of mathematics that has been designed to cope with any kind of items and operations on then: General Algebra and first order logic. This kind of mathematics, however, describes *static* structures, whereas we have to tackle dynamic behavior. This goal is achieved by two formalisms, *high-level Petri nets* and Gurevich's *Abstract State Machines*. Each of the three modeling techniques trades expressivity for analysis techniques: The more expressive the modeling technique, the less it offers specific analysis techniques.

Various versions of automata and process algebras have been suggested to model web services and business processes. They do not decisively contribute to the aspects considered in this paper.

3 Low-Level Petri Nets for Business Processes

We start out with the quite elementary technique of *business process nets (BP nets)*, a special class of elementary Petri nets. BP nets model the structure of

control within a single business process, as well as control of communication among processes.

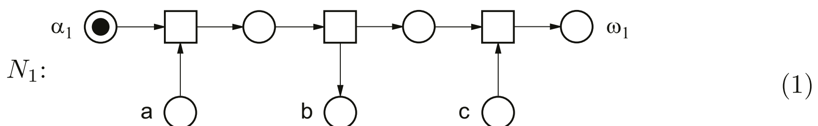
Elementary Petri nets have frequently been advocated to model control aspects of communicating business processes. (e.g. [15], [16], [9]) We suggest a variant that is technically simpler, and slightly more general.

A reasonable well structured business process exhibits a number of regularities and important properties: It can properly terminate in combination with any "serving" environment, it may exhibit a "most liberal" serving environment and a most abstract "public view". One business process may simulate or be equivalent to an other business process. Business processes may be composed to larger business processes, thereby systematically transferring important properties of the component processes to the composed process. It should be possible to decide those properties and to derive those processes from a representation of given processes. A number of reasons favors *low-level Petri nets* as an adequate technique for many of those questions:

- Many of those questions depend essentially on the *control structure* only, i.e. are independent of concrete data and operations.
- The paradigm of message passing of business processes ignores concrete delays among processes. In particular, the order of sent messages may swap upon their arrival. This corresponds naturally to the behavior of tokens in the places of Petri nets.
- Composition of business processes correlates exactly with gluing interface places of the corresponding Petri nets.
- In business processes, in particular in cooperating, distributed processes, actions occur locally and causally independently. Petri nets support and describe this kind of behavior by help of *distributed runs*.
- Criteria for selecting an activity out of a set of alternative in a business process, are frequently not fully characterized. Nondeterministic choice of conflicting transitions of a Petri net adequately simulate this kind of behavior.
- There exits a number of specific analysis tools for Petri nets, well applicable to Petri net models of business processes.

3.1 Business Process Nets

As mentioned above, a business process is intended to begin its activities in a definite *start* state and to terminate in a definite *stop* state. Activities include message exchange with an appropriately cooperating *environment*. This fixes the structure of Petri net models of the control structure of business processes. A simple example is



(1)

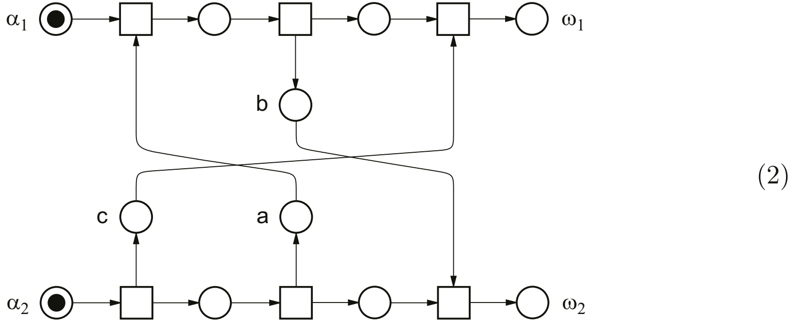
The places a and c are *input places*, and b is an *output place* of the net. Furthermore, the net has a *start marking*, with one token at place α_1 and no tokens elsewhere. Furthermore, the *stop marking* of this net has a token on place ω_1, and no tokens elsewhere. We assume the reader be familiar with the basics of elementary Petri nets, and define the general pattern of a *business process net* (a BP net, for short) N as a Petri net structure, consisting as usual of places (circles), transitions (squares) and arcs (arrows) together with

- a distinguished subset of its places where no arcs end, called the *input places of N*
- a distinguished subset of its places where no arcs start, called the *output places on N*
- a *start* marking, $start_N$, and a *stop* marking, $stop_N$, both with empty input- and output places.

The input- and output-places form together the *environment places* (also called the *channels*) of N; all other places are *inner* places. $start_N$ often has tokens only on a set of inner places without ingoing arcs, usually denoted by the (indexed) symbol "α". $stop_N$ often has tokens only on a set of inner places without outgoing arcs, usually denoted by the (indexed) symbol "ω". We follow the convention to draw α and ω at the left and right margin of graphical representations, respectively. This implies control flowing from left to right. (1) follows this convention. This definition of BP nets, as well as the forthcoming definition of their composition, is more liberal than corresponding definitions of [15], [7] and [13]. It is technically simpler and intuitively more natural, while preserving all relevant properties.

3.2 Closed Business Processes

Business process nets without input- and output places are useful for a number of purposes. Such a net is *closed*; here an example:



(2)

Its start marking has tokens on α_1 and α_2, its stop marking has tokens on ω_1 and ω_2.

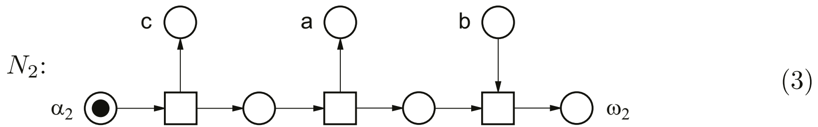
We will see later on that a closed net may result from composing two business process nets. By construction, a closed net remains if we skip the environment places of a BP net N, retaining the *inner subnet of* N, written $inner(N)$.

3.3 Composition

Cooperation of business processes is properly reflected by the composition of business process nets. Without loss of generality we assume for any two BP nets M and N that a place or a transition of M does not belong to inner(N) and vice versa: Otherwise one may construct *two* instances. Formulated differently, M and N share only places in their environments.

Composition $M \cdot N$ of M and N is then a BP net again, defined by identifying shared places. This way, an input place of M that coincidently is an output place of N, evolves into an inner place of $M \cdot N$.

As an example, one may compose (1) with the BP net



(3)

The resulting net $N_1 \cdot N_2$ is the closed BP net (2). Fig. 1 shows a further example.

Composition of BP nets is commutative, i.e. for any BP nets M and N holds

$$M \cdot N = N \cdot M \tag{4}$$

Furthermore, for any three BP nets L, M and N with no interface place shared by all three processes, the product is also *associative*, i.e.

$$(L \cdot M) \cdot N = L \cdot (M \cdot N) \tag{5}$$

3.4 Well-Formed Business Process Nets

We focused the *static structure* of BP nets so far, in particular their input- and output-places, their start- and stop marking, and their composition. Now we consider aspects of *dynamic behavior*, i.e. reachable states, runs, termination etc.

The most important property of a BP net, concerning its dynamic behavior, is *termination*: A BP net N *can terminate* if for each marking m reachable from $start_N$, the marking $stop_N$ is reachable from m. This definition reflects potential loops of N.

Occasionally we require each component of a BP net to be "useful": A BP net N is *covered* in case each transition t of N occurs at least in one occurrence sequence

$$start_N \longrightarrow \ldots \stackrel{t}{\longrightarrow} \ldots \longrightarrow stop_N. \tag{6}$$

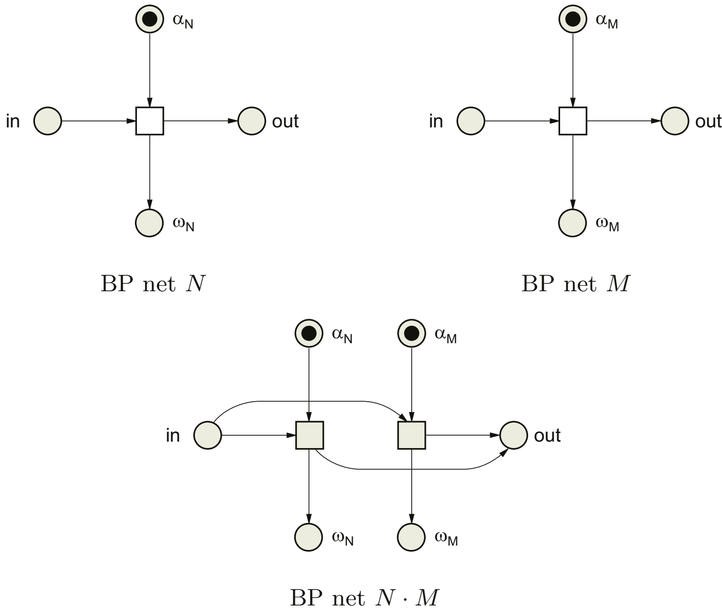


Fig. 1. Composition of BP nets.

It is furthermore reasonable to assume unambiguous start- and stop markings: A BP net N is *unambiguous* if there exist sets α and ω of inner places of N such that

– $start_N$ is the only reachable marking with tokens on all places of α
– $stop_N$ is the only reachable marking with tokens on all places of ω.

The above three conditions define the important class of well formed BP nets: A BP net N is *well formed* if N can terminate, is covered, and is unambiguous.

Any reasonable, closed business process has a well formed model. It remains to decide whether or not a given BP net is well formed. [15] reduces this problem to classical problems of Petri nets: Given a BP net N, he suggests to construct a Petri net N^* from N, by an additional transition t that leads the stop marking ω back to the start marking α. N is then shown to be well formed iff N^* is live and safe. (As a technicality, [15] and others restrict α and ω to one place. Our definition appears technically simpler, in particular the definition of composition, while all analysis techniques are retained).

3.5 Usable Business Process Nets

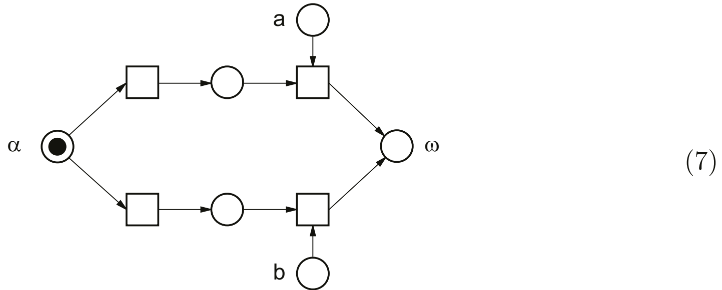
A well-formed bp exhibits a reasonable inner structure. In this section we ask for "reasonable" behavior w.r.t. the partners in the environment of a BP net.

Two BP nets are *partners* if they together can reach their joint *stop* state. As an example, (1) and (3) are partners.

More precisely formulated, two BP nets M and N are *partners* if $M \cdot N$ is unambiguous and can terminate (as defined in 3.4). We do not expect M and N together can employ all alternatives of M and of N. Hence we do not require $M \cdot N$ be covered.

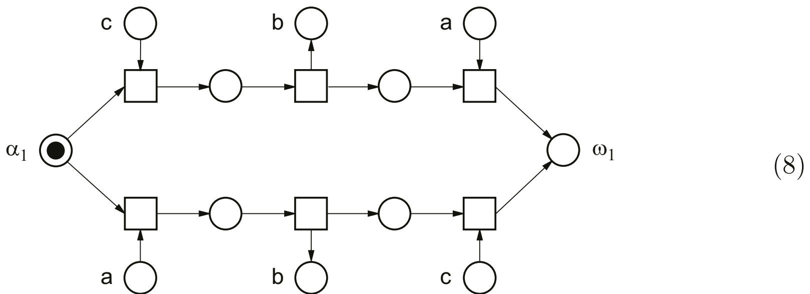
Based on the above notion of partners, we can now define the central notion of *usability*: A BP net is *usable* if there exists at least one partner of N. As an example, (1) is usable due to its partner (3).

Here an example of a BP net that is *not* usable:



(7)

Intuitively formulated, this business process decides whether to expect a or b from its environment. The process fails to propagate this decision to its environment. But the environment needs this information to act accordingly. In contrast, the following BP net is very well usable, e.g. by (3):



(8)

This rises the quest for an algorithm to decide whether or not a BP net is usable. In fact, such algorithms have been constructed (e.g. in [7]).

3.6 Further Properties

Usability is a fundamental property; but a number of other non-trivial properties and derived artefacts are likewise important, including *equivalence*, *abstract views*, *operating guidelines*, *fault handling*, and *transactions*. Various algorithms to decided those properties and to generate those artefacts have been published, including [4], [9], [10], and [14].

4 High-Level Petri Nets for BPEL

Here we suggest *schematic high-level Petri nets* as a modeling technique that is expressive enough to model quite complex behavior, such as essentials of the semantics of the *business process execution language*, BPEL. The core concept of schematic high-level Petri nets are symbols to be interpreted by *any* item or operation, not confined to conventional data structures. In analogy to low-level Petri nets as considered above, this technique fits perfectly to model business processes.

A number of analysis techniques are available for schematic high-level Petri nets, quite useful for (but not particularly confined to) models of business processes.

Efficient and reliable implementation of business processes is a tedious task. Different local business processes, running on different hardware on different software platforms, must correctly co-operate.

The business process execution language for web services, BPEL [3] has risen to a quasi-standard to describe and to run distributed business processes on an abstract level.

The semantics of BPEL has been presented in plain English, with some ambiguities, in particular when it comes to the *compensation* of activities.

By help of a small example we will show in the sequel, why high-level Petri nets provide adequate means to formulate the semantics of BPEL.

4.1 The BPEL Language

A BPEL program describes the structure of a business process as a particular Web service, and specifies the interaction of a business process with partner processes in its environment.

A central problem of business processes is *compensation* of already executed sub-activities (e.g. canceling an already booked flight) whenever it turns out later on that the overall goal fails (e.g. no hotel room was available).

BPEL consequently distinguishes *positive* control flow of a business process, formulating the intended activities to achieve its goal, and *negative* control flow, managing the case of faults, in particular the problems of compensation.

4.2 Activities

A core construct of BPEL are *activities*: An activity may be *elementary* (e.g. it may receive a message from its environment), or it may be *composed* from elementary activities. There are different ways to compose activities. They essentially correspond to control structures of conventional programming languages, i.e. sequences, loops and conditional alternative. In the next section we will consider one of them, called *scope*. An activity may be *executed*. Execution of an elementary activity strongly resembles conventional programming languages, and will not be considered in detail here. Executing a composed activity means to

iteratively select the next component activity to be executed, in accordance with the activity's control structure, and governed by actual values and predicates.

An execution of an activity can come to an end in three different manners:

- it terminates successfully
- it causes a fault
- it is canceled by a stop signal from its environment

The activity would signal to its environment the manner of ending. Actually, it is not activities but *instances of activities* that are executed. Various instances of an activity may co-exist and be executed concurrently. The phrase "to execute an activity" stands for "to execute one of the activity's instances". As it is intuitive and convenient, we will apply the shorthand whenever confusion can be ruled out.

4.3 Scopes

As mentioned above already, a set of activities may be combined in a *scope*. In addition to its "ordinary" activities, a scope includes a *fault handler* managing fault signals sent from the scope's ordinary activities. In particular, the fault handler may cancel all activities of the scope. Consequently, each activity must be prepared to accept a stop signal, and to process it accordingly.

A message m, as controlled by a scope, has two components: Its *contents* which is irrelevant in the sequel, and its *correlation set*, $cor(m)$. Details of the correlation set will not be relevant in the sequel. We only must be able to decide whether or not two correlation sets are equal. So, it suffices to represent each correlation set as a symbol.

4.4 The Activity receive

A typical activity is receive. Its instances access three components of the receive activity, provided by the overall process : an *input channel*, carrying messages to be processed by receive, a *correlation set* to direct incoming messages to the corresponding instance, and a *variable* to store the last accepted message.

Each instance i of receive processes an incoming message, m. If the correlation set $cor(m)$ of m and the correlation set of i coincide, the variable of receive is updated and given the message m as its new value. Otherwise the message is extinguished and a "failed" message is generated. As described above, there is always a chance for the fault handler to stop running instances of receive.

We are now prepared to state the problem to be solved: How can activities such as receive be described? This includes in particular

- to properly administer the various instances of receive
- to provide a *composition* technique for descriptions, that reflect the cooperation of activities.

In the next chapter we show that *high-level Petri nets* provide a more than adequate technique to model such systems.

4.5 A Model for Receive

Fig. 2 shows a high-level Petri net model for the *receive* activity. The reader familiar with high-level Petri nets will easily grasp this model. Other readers are helped by the following explanations. One may conceive those explanations as coincidentally providing an introduction to high-level Petri nets.

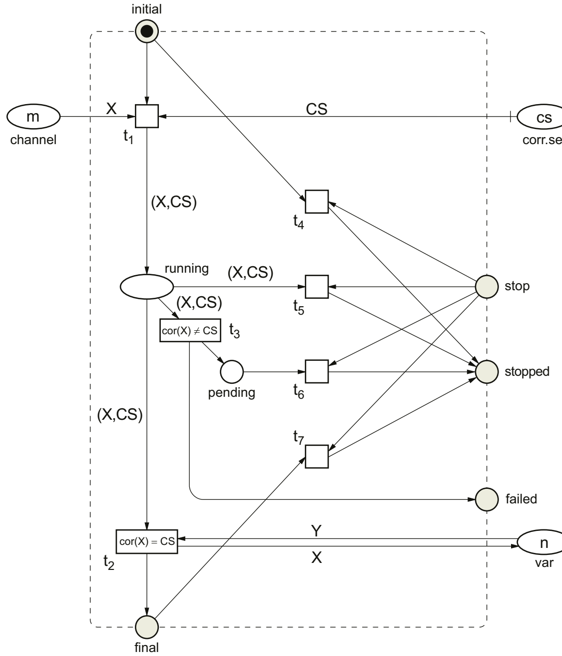


Fig. 2. The receive activity.

In Fig. 2, the dotted frame separates the inner components of the model from its surface and its environment. The five circles on the frame's surface (initial, stop, stopped, failed, final) are places, intended to exchange black dot tokens with other activities. Tokens on these places represent control signals to trigger activities, as discussed above. The three ellipses outside the frame (channel, corr.set, var) are places that model the activities' communication with the overall process: The activity may receive a message along the place channel. The process furthermore provides an initial correlation set at the place corr.set, and an initial value at var. This place represents a variable that always carries the last acceptable message.

Fig. 2 shows a typical state where the activity is ready to act: Some other activity has triggered receive (black dot token on initial) and the scope has sent a message, (token m on channel) A correlation set *cs* is anyway assumed at place corr.set, as well as some value, n, at the place var.

Describing the behavior of the net, we start with the intended, positive control flow. The state shown in Fig. 2 enables the transition t_1, provided the variables X and CS are properly valuated: X by m, and CS by cs, respectively. Occurrence of t_1 then

- removes the black dot token from initial, and the m token from channel
- produces the pair (m, cs) as a token at place running
- retains the cs token at corr.set, as $\longleftrightarrow$ is a *read arc*.

In this situation, to continue one has to evaluate the predicates inscribed in t_2 and t_3, again with $X = m$ and $CS = cs$. If the correlation set $cor(m)$ of the message m coincides with the correlation set cs provided by the environment, the transition t_2 is enabled. Occurrence of t_2 then updates the value at the place var. With the fresh value $X = n$, given the old value $Y = m$, the execution terminates (black dot token at final). If the predicate inscribed in t_2 fails with $X = m$ and $CS = cs$, t_3 is enabled and "throws a fault", i.e. triggers some other activity (black dot token on failed). This token will eventually, via the fault handler activity, cause a token on stop, thus enabling t_6. The activity then terminates with a black dot token at stopped. This completes description of the positive control flow.

A stop token may arrive any time. Hence at any state, the activity may leave its positive control flow by one of the transitions t_4, t_5 or t_7, resulting in a stopped token.

During execution of receive, a fresh message, l, may arrive and another activity may provide a fresh start signal to the *receive* activity. This is the situation where a new instance of *receive* must be created, executing concurrently to the existing one.

In the Petri net of Fig. 2, this may be modeled by a token "l" on the channel place, and another black dot token at the place initial. Concurrent execution of the two instances is then properly modeled by the Petri net, due to the definition of *distributed runs*, not considered here.

Schematic Petri nets come with a number of useful analysis techniques. For example, for the model of the *receive* activity as given in Fig. 2, one may prove that the token m, initially at the channel, eventually reaches the place var, or the system fails. Technically, this is represented by the temporal logic formula

$$(\mathsf{channel}.X \wedge \mathsf{initial} \wedge |\mathsf{corr.set}| \geq 1) \mapsto (\mathsf{var}.X \vee \mathsf{stopped})$$

with "$\mapsto$" denoting the "leads-to" operator. This formula can be proven to be valid in Fig. 2, by the help of techniques described e.g. in [11].

4.6 Lessons Learned

The above example provides a first glimpse at schematic high-level Petri nets, and the motivation to model business processes with this kind of Petri nets. Here the most important aspects:

- Elementary objects and operations of business processes are fairly abstract. Examples are “message”, “correlation set of an activity”, or “correlation set of a message”, but also “reply to a quest” or “cancel an order”. All these objects and operations, though elementary in the given context, come without any fixed or agreed representation in conventional data structures. High-level Petri nets, in their *schematic* setting as applied in Fig. 2, support this approach.
- The paradigm of business processes ignores delays of messages passing between processes. In particular, messages may “overtake”. The semantics of Petri nets, with tokens residing at a place without any order, correspond naturally to this paradigm.
- Single business processes cooperate along message channels: An output channel of one process serves as an input channel of another process. This is mimicked in Petri nets by glueing (identifying) the corresponding places.
- Activities of different business processes operate locally and independently. This is reflected in Petri nets by the occurrence rule for transitions: The behavior of a transition depends only on and only affects the adjacent places. No notion of global state or global time is required.
- Generation of instances of a business process, to occur concurrently, are perfectly modeled in Petri nets by more than one initiating tuple of transitions, and the notion of distributed runs. We refrain from details here.
- Analysis techniques as available for schematic high-level nets, are useful to verify business processes.

5 ASMs for Web Services

As a most universal modeling technique we suggest Gurevich’s Abstract State Machines (ASM). The demand of ASM to be “most universal” has been justified in [5]. We refrain from details here. But we show that ASM in fact are expressive enough to model a wide range of Web Service oriented Systems in a natural way.

5.1 The Abstract Basis of Web Services

The monograph [1] provides a comprehensive view on Web services. Concepts are presented in plain English, supported by various kinds of graphical representations, in about 140 Figures. About half of the figures show static, mainly hierarchical structures. The rest of the figures show dynamic behavior, both of middleware components (i.e. the technological basis of Web services) and of abstract components (in particular, components for business processes).

Web services are usually implemented on top of some middleware. Any formal description of the semantics of Web services hence must rely on a formal semantics of those middleware components. Fortunately, the semantics of Web services requires only quite abstract aspects of middleware semantics. What is needed, however, is a formalism to adequately represent those aspects of middleware. This comes in addition, of course, to an adequate representation of the Web service components themselves.

5.2 The Core Idea of ASM

The above problem can be solved by help of an idea that we applied in the context of high-level Petri nets already: Items and operations are symbolically represented, leaving their semantical aspect to be defined elsewhere. For example, in Fig. 2, m is a constant symbol and $cor(m)$ is a term. We only informally specified what a "message" is assumed to be; we only assume that a message has a correlation set, and that the term $cor(m)$ represents the correlation set of m. This idea of "pseudo code" used to represent dynamic behavior, given meaning only up to the interpretation of the involved symbols. It is central to the specification technique of *Abstract State Machines (ASM)*. An ASM is essentially a set of conditional assignment statements, to be executed in parallel. Condition, left side, and right side of each assignment statement are terms over a signature (i.e. a set of symbols, each with an arity). An ASM can be executed for *any* arity respecting interpretation of its symbols. Details on the ASM method can be found e.g. in [2].

ASM provide in fact an adequate framework to formally represent Web services.

5.3 A Small Case Study

Here we consider a small example, taken from [1]. This case study assumes a scenario where a *customer* and a *supplier* communicate along the web. The customer starts an interaction, sending a request for an offer (a *quote request*) to a supplier. After receiving a quote from the supplier, the customer returns and order, and after receipt of the ordered goods, submits his payment. Fig. 3 outlines this behavior, as given in [1], page 198.

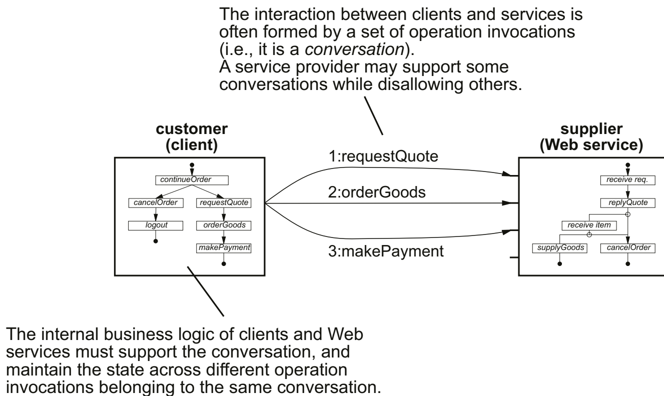


Fig. 3. A sample conversation between a customer (client) and a supplier (Web service).

The supplier Web service offers three operations, symbolically represented as *requestQuote*, *orderGoods* and *makePayment* to the customer. The customer

is allowed to envoke the operations only in the order as fixed by the supplier. This order is denoted as *conversation*. To demonstrate the ASM method, we just model the – admittedly quite simple – customer behavior.

We first consider the items we speak about. Each time the customer starts a conversation, he newly chooses the goods he wants to order, as well as the supplier for those goods. In the ASM formalism, this is modeled by the two constant symbols Goods and SupplierMsg. One may have expected variables at this point. But the idea is that the value of a constant symbol is fixed upon the start of an ASM program. Each time the forthcoming ASM *conversation* is started, it starts in a different *initial state*. Each initial state interprets the above mentioned two constant symbols by a newly chosen set of goods, and a newly chosen supplier. We assume corresponding messages to be sent from the client to the supplier Web service. Technically, Messages represents the set of potential messages symbolically. The *undefined* element, symbolically undef, is also assumed as a message. Furthermore, we assume the two constant symbols, true and false, to be always interpreted as expected. Finally, we assume two further constant symbols, GoodsOrdered and GoodsPayed, each to be initially valuated by a truth value.

The next issue to be tackled are the necessary functions. The first function is, symbolically,

$$\mathsf{RequestQuote} : \mathsf{Messages} \times \mathsf{Messages} \rightarrow \mathsf{Messages}$$

Semantically, the parameters should be the supplier to offer a quote, and the goods. The function should return the quote as given by the supplier.

The second function is

$$\mathsf{OrderGoods} : \mathsf{Messages} \times \mathsf{Messages} \rightarrow \mathsf{Boolean}$$

This is merely a predicate, expecting a supplier and an order, and declares whether goods have been ordered already.

Finally,

$$\mathsf{MakePayment} : \mathsf{Messages} \times \mathsf{Messages} \rightarrow \mathsf{Boolean}$$

is again a predicate and declares whether goods have been payed.

We are now ready to formulate the ASM. This is a program, i.e. a text, using the above introduced constant- and function symbols, together with the keywords **par**, **endpar**, **if**, **and**, $\neq$:

```
par
  if (Goods ≠ undef) and (SupplierMsg ≠ undef) and (Quote = undef)
    Quote := RequestQuote(SupplierMsg, Goods)
  if (Quote ≠ undef) and (GoodsOrdered ≠ true)
    GoodsOrdered := OrderGoods(SupplierMsg, Quote)
  if (GoodsOrdered = true) and (GoodsPayed ≠ true)
    GoodsPayed := MakePayment(SupplierMsg, Quote)
endpar
```

Constant symbols play the role of variables of programming languages here.

In general, an ASM may employ *any* kind of terms also on the left side of an assignment statement. Variables are used in ASM as bounded by a quantifier, as usual in logic.

Further examples of ASM models for Web services business processes, and the language BPEL can be found in [12], [4].

6 Conclusion

Service orientation is a principle to organize software architectures, independent of platforms, programming languages, and any other implementation oriented aspect. Service oriented architectures nevertheless deserve a unique representation, i.e. a formal model. This raises the quest for adequate techniques to formulate such models.

In this paper we advocate three such techniques, spanning from a very specific class of low-level Petri nets up to the most universal technique of Abstract State Machines. High-level Petri nets are located somewhere in the middle of the spectrum.

Each modeling technique trades expressivity for analysis techniques. This implies the following rule of thumb: To cover a specific problem, choose a modeling technique expressive enough to represent all relevant aspects intuitively and comprehensively. Coincidently, the chosen modeling technique should be as restrictive as possible, thus exploiting particular structures and regularities for verification issues.

Business process nets have been defined as a special class of elementary Petri nets. Consequently, their distinguished structure is exploited in the definition of derived notions such as well formedness, usability, etc. This structure has furthermore been exploited in analysis algorithms.

One my define corresponding classes of schematic high-level Petri nets and Abstract State Machines, together with corresponding analysis algorithms.

The above outlined spectrum of modeling techniques may cover *operational* models. One my wonder what other models may be useful. An example may be *logic based models*, with Lamport's *Temporal Logic of Actions* as a typical representative. It was particularly useful in this context, would composition of specifications just turn out as conjunction, and implementation as implication. These principles have been advocated by Abadi and Lamport in the early 1990ies already.

References

1. G. Alonso, C. Casati, H. Kuno, and V. Machirajv. *Web Services*. Springer Verlag, 2004.
2. E. Börger and R. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

3. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1. Specification, BEA Systems, IBM, Microsoft, SAP, Siebel, 05 May 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbiz2k2/html/bpel1-1.asp.
4. D. Fahland and W. Reisig. ASM based semantics of Web services: the negative control flow. International conference, ASM05, Paris, March 2005.
5. Y. Gurevich. Sequential Abstract-State Machines Capture Sequential Algorithms. *ACM Tranactions on Computational Logic*, Vol.1 No.1:77–111, July 2000.
6. Ekkart Kindler, Axel Martens, and Wolfgang Reisig. Inter-operability of Workflow Applications: Local Criteria for Global Soundness. In *Business Process Management*, LNCS 1806, pages 235–253, 2000.
7. Axel Martens. On Usability of Web Services. In *Proceedings of WQW 2003*, Rome, Italy, 2003. IEEE Computer Society Press.
8. Axel Martens. Analyzing Web Service based Business Processes. In *Proceedings of Intl. Conference on Fundamental Approaches to Software Engineering (FASE'05)*, Edinburgh, Scotland, April 2005. Springer-Verlag.
9. Axel Martens. Consistency between Executable and Abstract Processes. In *Proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05)*, Hong Kong, China, March 2005. IEEE Computer Society Press.
10. P. Massuthe and K. Schmidt. Operating guidelines – an alternative to public view. Internal report, Humboldt-Universität zu Berlin, 2005.
11. W. Reisig. *Elements of Distributed Systems.* Springer Verlag, 1997.
12. W. Reisig and A. Brade. ASM models of Web Services. Technical report, Humboldt-Universität zu Berlin, Computer Science Institute, December 2004. No. 181.
13. W. Reisig, K. Schmidt, and Chr. Stahl. Geschäftsprozesse modellieren und analysieren auf der Basis von Petri-Netzen. Technical report, Humboldt-Universität zu Berlin, Computer Science Institute, December 2004. No. 182.
14. Karsten Schmidt and Christian Stahl. A Petri net Semantic for BPEL4WS - Validation and Application. In Ekkart Kindler, editor, *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04)*, pages 1–6. Universität Paderborn, October 2004.
15. W.M.P. van der Aalst. Structural Characterization of Sound Workflow Nets. Technical report, Eindhoven University of Technology, Dept. of Mathematics and Computing Science, 1996. Computing Science Report 96/23.
16. 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.
17. W3C. Web Services Architecture Requirements. Working group note, W3C, October 2002. http://www.w3.org/TR/wsa-reqs/.