Services as a Paradigm of Computation

Wolfgang Reisig, Jan Bretschneider, Dirk Fahland, Niels Lohmann,
Peter Massuthe, and Christian Stahl

Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
{reisig,bretschn,fahland,nlohmann,massuthe,stahl}@informatik.hu-berlin.de

Abstract. The recent success of service-oriented architectures gives rise to some fundamental questions: To what extent do services constitute a new paradigm of computation? What are the elementary ingredients of this paradigm? What are adequate notions of semantics, composition, equivalence? How can services be modeled and analyzed? This paper addresses and answers those questions, thus preparing the ground for forthcoming software design techniques.

Key words:models of computation, services, SOA, open workflow nets

1 The Demand for a New Paradigm of Computation

1.1 Shortcomings of the Classical Paradigm

The classical paradigm of computation characterizes the behavior of an information processing system as a function, f: A user of the system supplies an argument x and expects to eventually receive the result $f(x)$ from the system.

Experience with distributed and reactive systems reveals the need for a more comprehensive paradigm. Among the many arguments for a new paradigm, the following may be the most striking one: A computation of a system does not necessarily receive all its input in the initial state, nor does it withheld all its output until it reaches a final state. Rather, a computation may start running with no or a first portion of input, and it may provide output whenever generated. In short, a computation may exchange messages with the systems' environment *during* its course. Examples of such systems include operating systems, any kind of technical control systems, and many forms of co-operating business processes. It is not too difficult to capture this kind of behavior in the framework of conventional programming, establishing communication of a program P with its environment by help of special input and output procedures, variables shared by P and its environment, and remote procedure calls. It took decades to acknowledge that this property of computations is not just a minor aspect, but that it affects (among other aspects, to be discussed elsewhere) our fundamental understanding of computation.

Wolfgang Reisig, Jan Bretschneider, Dirk Fahland, Niels Lohmann, Peter Massuthe, and Christian Stahl. **Services as a Paradigm of Computation**. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems, Essays in Honor of Dines Bjorner and Chaochen Zhou on the Occasion of Their 70th Birthdays, Papers presented at a Symposium held in Macao, China, September 24-25, 2007*, volume 4700 of Lecture Notes in Computer Science, pages 521-538, September 2007. Springer-Verlag.

1.2 Proposals to Adjust the Classical Paradigm

Church's thesis has dominated the discussion on the limits of computation since this kind of discussion has started in the 1950ies. It is overwhelmingly agreed that this thesis is most convincing when it comes to the computation of functions over sequences of symbols (for more details on this discussion we refer to [1]).

Nevertheless, it has frequently been argued that the classical paradigm of computable functions does not comprise all important aspects of the expressive power of information technology. We have discussed one such aspect above already. Here we survey a number of system models, all contributing to the non-standard aspects of communication, synchronization, and reactivity.

Petri nets: With his seminal Ph.D. thesis "Communication with Automata" [2], Carl Adam Petri pointed at the fundamental role of asynchronous, communicating processes, already in the early 1960ies. This led to the development of Petri nets as a technique to model concurrent behavior.

The decisive aspect of Petri nets in this context is the local character of its transitions. A behavior then is not a sequence of global states and steps, but a set of transition occurrences partially ordered by the relation of causality. This perception brought new insights into the fundamental notions of nondeterminism, fairness, scenarios, and others.

Omega-automata: ω-Automata [3] capture sequential, infinite computation in the late 1960ies already, laying ground for infinite, reactive computations.

Stream processing functions: The first proposals to model systems consisting of interacting components conceive a system component as a stream processing function (or, in the nondeterministic case, as a relation). This dates back to the early 1970ies already. As a typical example we refer to [4]. Streams (i.e. finite or infinite sequences) of data on the *input ports* of a stream processing function f are transformed into streams of data on the *output ports* of f. One system's output stream may be an other system's input stream. The *FOCUS* formalism [5] pursues this line of research. Stream processing functions are most adequate to describe a single system's semantics in isolation. The formalism properly reflects that a system's intermediate output can affect later input, via cooperation with the environment.

Process algebras: First suggested by Robin Milner in the late 1970ies [6] process algebras capture synchronous communication. The fundamental question of equivalence between system models gave rise to the notion of simulation and bisimulation. It has been a matter of surprise that those notions can not be simply captured in terms of formal language containment or equality.

Interface variables and remote calls: In the framework of programming languages, reactive behaviour can easily be represented by help of variables, shared by the program and its environment. This has been done since the late 1960ies. It has later been adapted by specification techniques such as Lamport's Temporal Logic of Actions [7], as well as Gurevich's Abstract State Machines [8]. Fairly more expressive than shared variables is the technique of remote procedure calls, which is a fundamental principle of middleware systems, in particular CORBA.

Misra and Cook in their *ORC* language recently generalized this principle, replacing the call of procedures by the call of services [9].

Interacting and communicating systems: Peter Wegner's contributions of the late 1990ies ([10,11]) boosted the awareness of a greater public, that interaction and communication was indeed a decisive argument to search for a new paradigm of computation. Many authors took up his arguments (as, e.g. in the volume [12]). Some authors extend the classical models, in particular Turing Machines (examples include [13] and [14]).

We follow this line in the sequel, too. To this end, we discuss some elementary aspects of the new paradigm in the next section, and pose some fundamental questions.

1.3 Aspects of the New Paradigm

The above described aspects models and representation techniques for information processing systems share a couple of aspects. Here, we focus just two of them.

Firstly, in the classical setting, non-termination of a computation denotes failure, as no output is generated in this case. Two different non-terminating computations cannot and need not to be distinguished in any respect. In contrast, in the new paradigm, a computation is in general not envisaged to terminate. Infinite computations are of utmost interest. Two different infinite computations in general very well exhibit different input/output behaviour. Interaction of services is usually split into finite slices: An instance of interaction is intended to terminate in a "reasonable" state. But a service is assumed "always on", capable to engage in interaction ad infinitum.

The second consequence of the new paradigm is related to the *composition* of systems. The classical setting offers *sequential composition* $A;B$ of two systems A and B as the only choice: A's output is B's input. In contrast, the new paradigm permits composed systems A and B to exchange data at any time during a computation.

Though we have identified only two aspects of the new paradigm, it is obvious that fundamentally new problems arise that cannot be identified, let alone be solved, in the framework of classical system models. Typical problems of systems that follow the new paradigm include:

- What kind of properties are important for such systems?
- Is there a canonical notion of *equivalence* for such systems?
- Can any two such systems be composed, at least syntactically, resulting in a (may be, futile) system?
- What, precisely, is refinement and abstraction, and which properties should refinement and abstraction preserve?
- How can the effect of such systems be abstractly described (in analogy to a function f in the classical case)?
- What kind of formal representations of such systems are reasonable?

– How can the expressive power of formal description techniques of such systems be compared?
– Is there a "most general" class of "representable" such systems, in analogy to the class of computable functions in the classical setting?

2 Service-Oriented Computing and Service-Oriented Architectures

The above discussion focused communicating agents as a basic construct of the new paradigm. We suggest *services* as an adequate concretion for such agents. Services provide viable means to implement communicating agents. Furthermore, there exists a rich theory to handle services, and to answer the above questions. Details will be given in the next sections.

2.1 The New Paradigm in Practical Applications

Governed by practical needs, and not caring too much about theoretical aspects, systems following the new paradigm have been implemented for decades. Examples include operating systems, technical control systems, workflows etc. But only nowadays such systems are conceived as following a new paradigm. This may be due to the emerging problems that arise in the course of automatic *composition* of such systems, as required for computer based interorganizational business processes and new software architectures, such as "programming-in-the-world" or "programming on demand".

In this context, systems that fit into the new paradigm are usually denoted as *services*. Their implementation in the framework of existing technologies evoked the concepts of *service-oriented computing* (SOC) and, as a principle of using SOC, the term of *service-oriented architectures* (SOA). We expand on those aspects in the sequel.

2.2 Service-Oriented Computing

Trying to identify what many different views, descriptions and definitions for services have in common, we can conclude that a service is a well defined, self-contained module that provides some concrete functionality to its environment. Consequently, the minimal requirements of a service include an *interface* and an *internal control*. An interface can usually be conceived as a set of *ports*, with each port capable to store *messages*. The internal control triggers the *actions* of the service. An action either sends a messages to a port or receives one from a port or operates locally. Hence, asynchronous communication is the usual communication mode of services. But other modes may be supported as well, such as synchronous (handshake) communication or lock step communication.

2.3 Web Services

Currently, the most prominent kind of services are *Web services*. A Web service is a functionality (e.g., a standard business function) provided at a unique network address, given as a URI. This functionality is described in a standard definition language, and available via various transport protocols, formats and profiles for quality of service. Today's implemented Web Services rely on highly distributable communication- and integration backbones (sometimes called "the Big Basic Bus", BBB). Each Web Service is addressed by its unique URI and is usable by other services along its – publically known – interface. A Web Service is assumed as "always on": A user does not have to create it, nor to care about destruction, etc. Several *instances* of a Web Service may exist concurrently, mimicking for each user exclusive access to the service. The language WS-BPEL established itself as a quasi-standard for the implementation of Web services.

2.4 Service-Oriented Architectures

Though independent of other services, a service is typically constructed with respect to other services: Purpose and use of a service is its *communication* with other services. Partners to communicate with may reside anywhere in the real world. For a Web service, any service on the web is a potential candidate to serve as a partner. A fundamental problem then is *service discovery*, i.e. for a service P the problem to identify proper communication partners, and to establish communication with those partners. A *service-oriented architecture* (SOA) solves this problem by help of a scenario that assumes

- agents called *providers*: A provider offers services to the public, to be used by (i.e., to be composed with) other services. To this end, a provider *publishes* information about the services he offers.
- agents called *requesters*: A requester requests services, i.e. wants to *find* services it can use, as they are published by providers.
- agents called *brokers*: A broker collects information about the services provided by providers and the services wanted by requesters. Upon detecting services that would properly fit, the broker informs the requester about the provider, such that they can directly *bind* their services. In more elaborated variants, a broker may itself compose two or more provider services, and offer the composed service to a requester. Even more, a broker may observe that a provider service only "almost" fits to a requester service. In this case the broker may construct an *adapter* service to bridge the gap. Figure 1 shows the conventional outline of SOA, indicating the three agents and their pairwise activities.

SOC and SOA can be conceived as *virtualization*, viz. abstraction, from technical implementation details of services. SOA is an architectural style to realize SOC. This can be conceived an analogy to the client/server architectural style that realizes distributed computing.

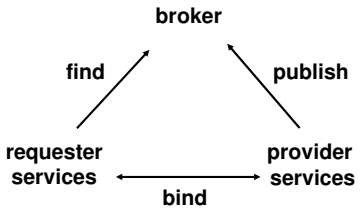

Fig. 1. the SOA triangle

The above described agents (provider, requester, and broker) may be likewise virtual. Usually there are no corresponding physical or implemented components in an SOA. It is the services themselves, that play the *role* of providers, requesters, and even brokers.

3 An Algebraic View on Services

Both, the foundational considerations of Sect. 1 and the applied aspects of Sect. 2 jointly establish principles of services. Here we introduce fundamental notions and their properties, as a nucleus for a rich conceptualization of "services as a paradigm of computation", to blossom in the sequel of this paper.

3.1 Composition of Services and "Reasonable" Services

As outlined in Sect. 2, a core aspect of services is their composition: Any two services P and R may be composed, resulting in a service $P \oplus R$. Of course, $P \oplus R$ is not always a very reasonable service, given "any" P and R. In particular, $P \oplus R$ may deadlock or livelock; sent messages may remain in a buffer forever, etc. Conceiving $P \oplus R$ as a transition system T with initial and final states, a typical requirement for $P \oplus R$ to be "reasonable" is *weak termination* of T: Each computation $s_0 s_1 \dots s_k$ starting from an initial state s_0, can be extended to a computation $s_0 \dots s_k \dots s_n$ with a *final* state s_n. A final state does not necessarily deadlock; it may just indicate that one "round" of computation is finished and the service is prepared to launch into a new round. Instead of weak termination, any other predicate may characterize "reasonable" services. Typical examples are fair termination or strong termination, i.e. each (fair) computation eventually will reach a terminal state.

Formulated in an abstract setting, on the set $\mathcal{S}$ of all services under consideration we assume a binary, symmetrical operator

$$\oplus : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$$

to compose services, and a distinguished predicate

$$\tau \subseteq \mathcal{S}$$

to discriminate "reasonable" (e.g., terminating) services. $\oplus$ and τ lay the ground for a canonical, rich theory of services, covering a wealth of important notions, questions and properties, as they are particularly important in the framework of SOA.

3.2 The Strategies of a Service

One of the central ideas of SOA is the assumption of a *provider* agent, offering a service P to the public. P is intended to be engaged by *requester* agents. To engage P means to follow a strategy to interact with P in order to reach the requester's goal. In technical terms, a *strategy of* P is an other service, R, such that the composition $P \oplus R$ of P and R is "reasonable", i.e. $P \oplus R \in \tau$. So, from the requester perspective, the most important aspect of P is the set

$$Strat(P) =_{def} \{R \in \mathcal{S} \mid P \oplus R \in \tau\}$$

of all strategies of P. This set may be conceived as the semantics of P.

3.3 Simulation and Equivalence

The strategies of a service P yield a canonical *generalization* relation

$$\leq \subseteq \mathcal{S} \times \mathcal{S}$$

on services: A service P' *generalizes* P if P' preserves all strategies of P:

$$P \leq P' \text{ iff } Strat(P) \subseteq Strat(P').$$

A typical scenario including this relation is the provider of P wanting to exchange P by a service P' without bothering the so-far users of P.

Consequently, two services P and P' are *equivalent* iff they generalize each other:

$$P \sim P' \text{ iff } Strat(P) = Strat(P').$$

This equivalence is in fact the canonical counterpart of functional equivalence in the classical setting: Two systems are equivalent iff their environment cannot distinguish them.

3.4 Brokering of Services

The handling of services includes ways to find an adequate provider services P for given requester services, R. This basic problem gives rise to a lot of derived questions, including efficient decision procedures and construction algorithms. Many of them can be posed in the general framework as developed so far. More precisely, a given provider service P rises the quest of

- *Controllability*: Does P have a strategy at all, i.e. $Strat(P) \neq \emptyset$?

- *Compatibility*: For a given service R, is R a strategy for P, i.e. $R \in Strat(P)$?
- *Public view*: The provider of the service P may want to hide internal details of P, and describe only communication capabilities of P, i.e. an abstract version of P. Technically, we search to derive a canonical P' such that $Strat(P') = Strat(P)$.
- *Operating guideline*: A requester of P is usually interested in all potential strategies of P. Technically we search for a concise description of $Strat(P)$.

A given requester service R rises a couple of questions at the broker's job, including

- *Efficient search*: The broker may administer a large provider repository $\mathcal{P}$ of provider services. Efficient algorithms are mandatory to find a service $P \in \mathcal{P}$ that is compatible with R. Such algorithms of course depend on the kind of information available to the broker about both R and P, as well as on the organization of $\mathcal{P}$.
- *Adaption*: For a given provider service P, the composition $P \oplus R$ may only be "almost reasonable". The broker may (automatically) construct a service A ("adapter") such that R is a strategy for $P \oplus A$. In technical terms we are behind a solution X of the problem

  $$R \in Strat(P \oplus X).$$

  This is equivalent to the problem

  $$(R \oplus X) \in Strat(P).$$

- *Composed adaption*: As a special case of the general adaption problem, an adapter may be just some other service in the depository. Even more, the broker may compose two or more services $P_1, \ldots, P_n$ such that $P_1 \oplus \ldots \oplus P_n$ is compatible to R. Formulated differently, the repository $\mathcal{R}$ may be virtually extended by the n-fold composition of all its services.

A concrete modeling technique for services should provide efficient algorithms to answer the above questions and to construct corresponding services.

4 Service Nets

Above we compiled a number of requirements at a proper framework for the notion of services. Here we strive for a more concrete, operational model that would meet those requirements. To this end we suggest *open workflow nets* (*oWFN*) as a staring point to model services. Before launching into details, we justify and motivate this approach.

4.1 The Motivation for Open Workflow Nets

As outlined in Sect. 2.2 already, the essential components of a service are its interface and its internal control. Here we stick to those two aspects:

Modelling interfaces with oWFN: Open workflow nets represent asynchronous communication, along message ports, with the order of sent messages not necessarily preserved upon their arrival: A message port contains an unordered set of messages (just as your home letter box). Even more, two different messages may have identical content, i.e., cannot be distinguished in any respect. The messages in a port thus constitute a *bag* (i.e. a finite multiset). This is the most liberal and general form of asynchronous communication, and the most common form in the world of business processes.

Modeling internal control with oWFN: An open workflow net is not confined to sequential control, but may very well exhibit *concurrent* control flow. This is most useful: Firstly, composition of two oWFNs results in an oWFN again, with the previous components' two flows of control merged into internal concurrent control of flow of the composed system.

Secondly, languages such as WS-BPEL anyway exhibit concurrent control flow. This can adequately be modeled in the framework of oWFN.

4.2 The Formal Framework of Open Workflow Nets

Technically, an open workflow net N is a conventional Petri net with distinguished input and output places to store input and output messages during computation. Consequently, an input place has no ingoing arcs in N, and an output place no outgoing arcs. Furthermore, an oWFN has an initial marking m_0 and a set Ω of final markings. Summing up, an oWFN can be written as

$$N = (P, T, F, in, out, m_0, \Omega),$$

with $in, out \subseteq P$, ${}^\bullet in = out^\bullet = \emptyset$, a marking m_0 and a set Ω of markings.[1]

Graphically we extend the classical Petri net representation by an encompassing dotted line. The input and output places are located on the line's surface. The initial marking is explicitly represented. The final markings have to be described elsewhere. Figure 2 shows an example.

According to the usual notions of Petri nets, a step

$$m \xrightarrow{t} m'$$

of an oWFN N transforms a marking m into a marking m', following the well-known occurrence rule for transitions t. A *run* of N is a sequence

$$m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} \ldots \xrightarrow{t_k} m_k$$

[1] We assume the reader's familiarity with elementary notions of Petri nets. The appendix provides formal details.

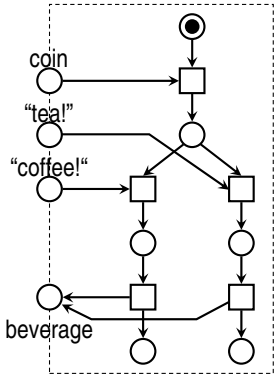

Fig. 2. Example of an oWFN: A beverage service A.

with m_0 the initial marking, m_k a final marking, and $m_{i-1} \xrightarrow{t_i} m_i$ a step of N $(i = 1, \ldots, k)$.

The pragmatic idea of oWFNs is obvious: An oWFN N describes a set of runs, starting at the initial marking.

According to the above described pragmatic idea of oWFNs, *termination* is a crucial issue. An isolated oWFN rarely terminates; usually a partner is required, such that the composed system would terminate. We consider the weakest version of termination in the sequel: An oWFN N is *weakly terminating* if each sequence $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} \ldots \xrightarrow{t_k} m_k$ of steps $m_{i-1} \xrightarrow{t_i} m_i$ $(i = 1, \ldots, k)$ is a prefix of a run of N (i.e. can be extended to eventually reach a final state).

A sequence of steps may fail to be extensible to a run due to a wrong number of tokens at input or output places.

We occasionally want to abstract from input and output and concentrate on the *inner subnet*:

For an oWFN N, the set

$$I =_{def} in \cup out$$

is the *interface of* N, and

$$J =_{def} P \setminus I$$

is the set of *inner places of* N. Furthermore,

$$inner(N) = (J, T, F', m_0', \Omega')$$

is the *inner subnet of* N, generated by the restriction to the inner places of N, the transitions of N, and the corresponding restriction of F, m_0 and Ω to J and T. As an example, Fig. 3 shows the inner subnet of the oWFN in Fig. 2.

N is apparently ill designed in case $inner(N)$ is not weakly terminating.

Historically, the term “open Workflow Nets” has been derived from workflow nets [15]. A workflow net is a formal model of the process logic of a workflow.

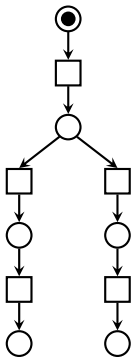

Fig. 3. $inner(A)$

Conceptually, a service extends workflows with an explicit interface to enable communication with other services. Technically, the inner subnet $inner(N)$ can be conceived as a representation of workflows. A workflow net N additionally requires special properties of initial and final markings.

4.3 Composition of Open Workflow Nets

As outlined in the introduction, composition of services is a major concern of SOC. Consequently, composition of service models should be as general and as simple as possible.

We can always assume that two oWFNs M and N don't share inner elements, but only interface places. Then the composition $M \oplus N$ of two oWFNs M and N is just the (sorted) union of their elements. Figure 4 shows an example. Appendix A2 provides formal details.

In particular, if p is an output place of M as well as an input place of N, then p turns into an inner place of $M \oplus N$. It is important to observe for two oWFNs M and N:

$$M \oplus N \text{ is an oWFN.}$$

The above definition of oWFN immediately implies for two oWFNs M and N

$$M \oplus N = N \oplus M.$$

Furthermore, for three oWFNs L, M, and N that all three have no element in common, composition is associative, viz.

$$L \oplus (M \oplus N) = (L \oplus M) \oplus N.$$

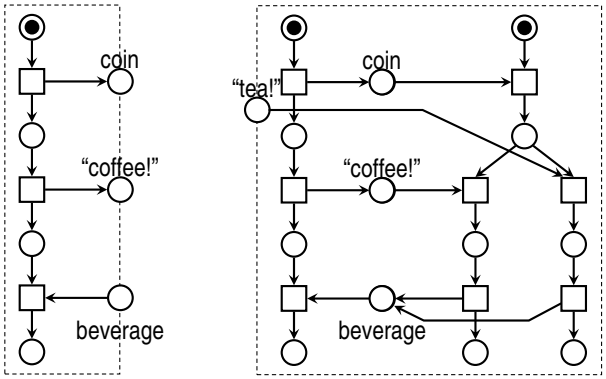

Fig. 4. Composition of services.

4.4 Partners and Fellows

In real applications, two oWFNs M and N to be composed are mostly *partners*, i.e. they communicate along interface places, but do not share input or output places:

$$in_M \cap in_N = out_M \cap out_N = \emptyset.$$

The left oWFN,B, of Fig. 4 shows a partner, B, to the oWFN A of Fig. 2. The two oWFNs depicted in Fig. 5 are no partners.

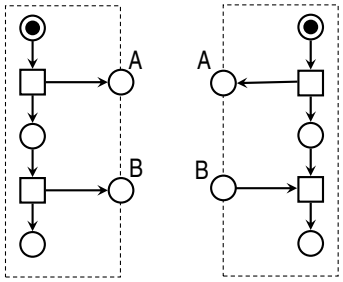

Fig. 5. No partners.

Likewise interesting is the special case of M and N joining input or output places, and not communicating at all: M and N are *fellows* iff

$$in_M \cap out_N = out_M \cap in_N = \emptyset. \tag{1}$$

The interface of the composition $M \oplus N$ of two fellows M and N is the union of the interfaces of M and N. Figure 6 shows an example. The two oWFNs of Fig. 5 are neither partners nor fellows.

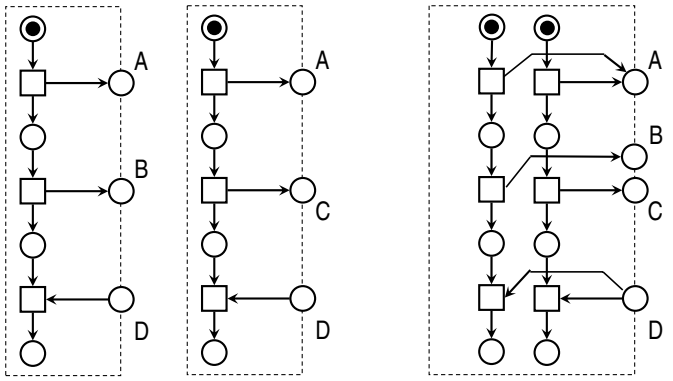

Fig. 6. Two fellows and their composition.

If L, M and N are pairwise partners, then $L \oplus M$ is a partner of N and $\oplus$ is associative i.e. $(L \oplus M) \oplus N = L \oplus (M \oplus N)$.

Likewise, if L, M and N are pairwise fellows, then $L \oplus M$ is a fellow of N, and $\oplus$ is associative, as described above.

4.5 Open Workflow Nets with Ports

Experience shows that the composition of services requires more flexibility than offered by oWFN as defined above.

As an example, the composition $A \oplus B$ of the beverage service A and its strategy B of Fig. 4 remains with a fairly unintuitive input place, *tea!*. Intuitively, A and B fit perfectly and consequently their composition $A \oplus B$ should be a "closed" net, i.e. a net with empty interface. More flexibility is also required when the issue of *refinement* and *abstraction* is taken into account in the sequel.

A fairly simple idea suffices to provide oWFNs with the required degree of flexible composition: The interface places are grouped into *ports* such that each interface place belongs to exactly one port. The ports are decorated with (pairwise different) *names*. As an example, Fig. 7 equips the beverage service A of Fig. 2 with three ports. One of them, "select", contains two input places "coffee" and "tea". The other two, "pay" and "offer", contain one element each. The graphical representation is obvious. Correspondingly, Fig. 7 identifies three ports for the strategy B of Fig. 4 one for each place.

Composition of two oWFNs with ports, M and N, say, then follows a simple rule: Just glue ports of M and N with identical names. Gluing the ports of M and N with name α then means to identify a place p of the α-port of M with

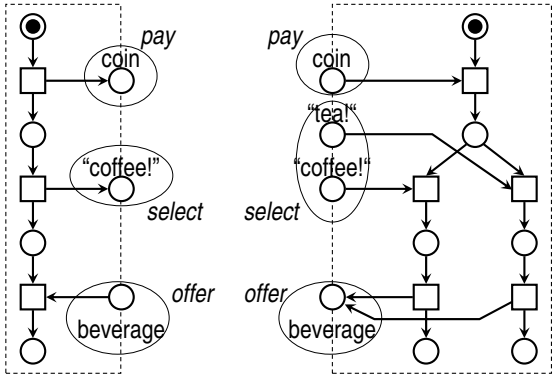

Fig. 7. Beverage service and a strategy, equipped with ports.

a place q of the α-port of N if and only if $p = q$, described in Sect. 4.3. As an example, Fig. 8 shows the composition of the port equipped oWFN of Fig. 7.

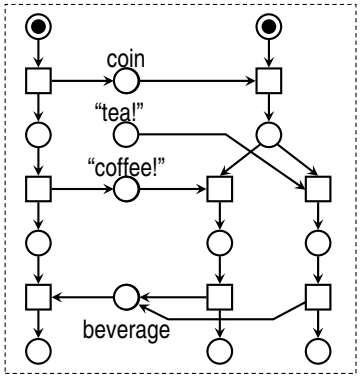

Fig. 8. Composition of the port equipped oWFNs of Fig. 7.

4.6 Hierarchical Open Workflow Nets

Abstraction and refinement are fundamental construction principles for complex systems: Only a hierarchical design process makes complexity tractable. oWFN allow for a simple, canonical notion of refinement. The basic idea is the replacement in N of a transition t by an other oWFN M, written (as usual for replacement operators)

$$N[M \setminus t]$$

("in N, replace M for t"). This is possible whenever the interface of M coincides with the environment ${}^\bullet t \cup t^\bullet$ of t.

As an additional technicality, the addition $m + m'$ of markings m and m' of oWFN N and N', respectively, is defined for each $p \in P_N \cup P_{N'}$, by

$$(m + m')(p) = m(p) + m'(p),$$

with $m(p) = m'(p') = 0$ if $p \in P' \setminus P$ and $p' \in P \setminus P'$. Furthermore, for two sets Ω and Ω' of markings, let

$$(\Omega + \Omega') = \{m + m' \mid m \in \Omega \text{ and } m' \in \Omega'\}.$$

With this in mind, we define refinements of two oWFNs $N = (P, T, F, in, out, m_0, \Omega)$ and $N' = (P', T', F', in', out', m'_0, \Omega')$ as follows:

i A transition $t \in T$ *coincides with the interface of* N' iff ${}^\bullet t = in'$ and $t^\bullet = out'$.
ii Let $arcs(t) =_{\text{def}} ({}^\bullet t \times \{t\}) \cup (\{t\} \times t^\bullet)$ ("the *arcs* of t").
iii Let t coincide with the interface of N. Then $N[N' \setminus t] =_{\text{def}}$ $(P \cup P', (T \setminus \{t\} \cup T', (F \setminus arcs(t)) \cup F', in, out, m_0 + m'_0, \Omega + \Omega')$ is the *refinement of* t *in* N *by* N'.

Refinement in fact meets all properties one would expect. Firstly, refinement of t by N' in N is independent of the context of N:

$$(N \oplus M)[N' \setminus t] = N[N' \setminus t] \oplus M.$$

Secondly, the order of refining t by N and t' by N' in M is irrelevant:

$$(M[N \setminus t])[N' \setminus t'] = (M[N' \setminus t'])[N \setminus t].$$

Finally, weak termination is preserved: If N and N' weakly terminate, then $N[N' \setminus t]$ weakly terminates, too.

4.7 Analysis Techniques for Open Workflow Nets

Open workflow nets are generell enough to capture decisive properties of services, and are simple enough to allow for formal analysis techniques. Due to the structure of oWFN it should come without surprise that most analysis techniques are based on variants of automata.

The most important kind of automata to analyze an oWFN N is its operating guideline, $OG(N)$. This automaton describes $Strat(N)$, viz. the set of all strategies of N. $OG(N)$ is essentially a finite state automaton, inscribed by Boolean expressions, built from the interface of N. Details are given in [16]. Most of the questions discussed in Sect. 3.4 can efficiently answered for any oWFN N by help of $OG(N)$. Corresponding algorithms have been implemented in the Petri net analysis tool Fiona, and successfully been used to analyze Petri net models of realistic BPEL processes. Details can be found in [17].

Conclusion

Models of interactive computation have been constructed since the early days of computing. In recent years, started as a smart combination of existing middleware techniques, *services* achieved prominence as a general, albeit variable software architecture model. We suggest to establish this model as a standard model of asynchronous interactive computation, completing process algebras as a standard model for synchronous computation. Open workflow nets are a most useful starting point as a representation techniques for asynchronous interactive computation, as they provide a lot of techniques to effectively analyze the most important properties of asynchronous interactive systems. In their given form, open workflow nets stick to control flow of services. Data and data development behavior can easily be incorporated, extending the formalism as usual in high level Petri nets.

References

1. Gandy, R.: Church's thesis and principles for mechanisms. The Kleene Symposium, North-Holland, Amsterdam (1980) 123–148
2. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2 (1962) also: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Suppl. 1, English translation.
3. Thomas, W.: Automata on Infinite Objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). (1990) 133–192
4. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress. (1974) 471–475
5. Broy, M., Stoelen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer (2001)
6. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer (1980)
7. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
8. Gurevich, Y.: Interactive Algorithms 2005. In Jedrzejowicz, J., Szepietowski, A., eds.: Mathematical Foundations of Computer Science 2005. Volume 3618 of Lecture Notes in Computer Science., Springer (2005) 26–38
9. Misra, J., Cook, W.R.: Computation Orchestration: A Basis for Wide-Area Computing. Journal of Software and Systems Modeling (May 2006) 83–110
10. Wegner, P.: Why Interaction Is More Powerful Than Algorithms. Commun. ACM **40**(5) (1997) 80–91
11. Wegner, P.: Interactive Foundations of Computing. Theor. Comput. Sci. **192**(2) (1998) 315–351
12. Goldin, D., Smolka, S.A., Wegner, P.: Interactive Computation - The New Paradigm. Springer (2006)
13. Goldin, D.Q.: Persistent Turing Machines as a Model of Interactive Computation. In Schewe, K.D., Thalheim, B., eds.: Foundations of Information and Knowledge Systems. Volume 1762 of Lecture Notes in Computer Science., Springer (2000) 116–135

14. Leeuwen, J.v., Wiedermann, J.: On Algorithms and Interaction. In Nielsen, M., Rovan, B., eds.: Mathematical Foundations of Computer Science 2000. Volume 1893 of Lecture Notes in Computer Science., Springer (2000) 99–113
15. Aalst, W.M.P.v.d.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
16. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. In Kleijn, J., Yakovlev, A., eds.: International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007. Volume 4546 of Lecture Notes in Computer Science., Springer-Verlag (2007) to appear.
17. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation. Data Knowl. Eng. (2007) accepted for special issue of BPM 2006.

A General notions and notations

Definition 1 (Net).
Let P and T be finite, disjoint sets.
Let $F \subseteq (P \times T) \cup (T \times P)$.
Then $N = (P, T, F)$ is a net*.*

The elements of P, T and F are *places*, *transitions* and *arcs*, graphically depicted as circles, boxes and arrows, respectively.

In the rest of this Appendix A we assume a net $N = (P, T, F)$.

Definition 2 (Pre-set, Post-set).
For $x \in P \cup T$, let
$^\bullet x =_{def} \{y \mid (y, x) \in F\}$ *is the* pre-set *of x*
$x^\bullet =_{def} \{y \mid (x, y) \in F\}$ *is the* post-set *of x.*

Definition 3 (Marking).
A marking of N *is a mapping $m : P \to \mathbb{N}$.*

Graphically, a marking m is depicted by $m(p)$ black dots ("*tokens*") at each place $p \in P$.

For two markings m_1 and m_2 of N, let $m_1 + m_2$ be the marking of N, defined for each $p \in P$ by $(m_1 + m_2)(p) =_{def} m_1(p) + m_2(p)$.

For a marking m of N and a set $Q \supseteq P$, extend m canonically to $m : Q \to \mathbb{N}$ for each $q \in Q \setminus P$ by $m(q) = 0$

Definition 4 (Enabling, Step).
Let $t \in T$,
and let m be a marking of N.

1. *m* enables *t if for each $p \in {}^\bullet t$ holds: $m(p) \geq 1$.*
2. *Let m enable t and let the marking n be defined by*
   $n(p) =_{def} m(p) - 1$ if $p \in {}^\bullet t \setminus t^\bullet$
   $n(p) =_{def} m(p) + 1$ if $p \in t^\bullet \setminus {}^\bullet t$
   $n(p) =_{def} m(p)$, otherwise.
   Then (m, t, n) is a step *of N, frequently written $m \xrightarrow{t} n$.*

Definition 5 (Run).
A finite or infinite sequence $m_0 t_1 m_1 t_2 \ldots$ *is a* run of N *if* (m_{i-1}, t_i, m_i) *is a step of* N *for* $i = 1, 2, \ldots$.

B Open Workflow Nets

Definition 6 (Open Workflow Net, oWFN).
Let (P, T, F) *be a net,*
let $in, out \subseteq P$ *with* ${}^\bullet in = out^\bullet = \emptyset$,
let m_0 *be a marking of* N,
let Ω *be a set of markings of* N.
Then $N = (P, T, F, in, out, m_0, \Omega)$ *is an* open workflow net *(oWFN for short).*

in and *out* contain the *input* and *output places*, $I =_{\text{def}} in \cup out$ is the *interface*, $J =_{def} P \backslash I$ contains the *inner places* of N respectively. Whenever N is not obvious from the context, we affix the index N, as in $P_N, T_N, F_N, m_{0_N}, \Omega_N, in_N, out_N, I_N, J_N$.

Definition 7 (Inner(N)).
Let N *be an oWFN. Then*
$inner(N) =_{def} (J_N, T_N, F_N \cap ((J_N \times T_N) \cup (T_N \times J_N)))$,
is the inner subnet *of* N.

Definition 8 (Internally disjoint oWFNs).
Two oWFNs M *and* N *are* internally disjoint *iff* $(P_M \cup T_M) \cap (P_N \cup T_N) \subseteq (I_M \cap I_N)$.

Remark 1. Two oWFNs can canonically be made internally disjoint: Each shared internal element is replicated.

General assumption: Two oWFNs M and N will always be assumed as internally disjoint.

Definition 9 (Composition of oWFNs).
The composition $M \oplus N$ *of two (internally disjoint) oWFNs* M *and* N *is the oWFN*
$M \oplus N =_{def} (P_M \cup P_N, T_M \cup T_N, F_M \cup F_N)$, *with*
$in_{M \oplus N} =_{def} (in_m \setminus out_N) \cup (in_N \setminus out_M)$,
$out_{M \oplus N} =_{def} (out_m \setminus in_N) \cup (out_N \setminus in_M)$,
$m_{0_{M \oplus N}} =_{def} m_{0_M} + m_{0_N}$,
$\Omega_{M \oplus N} =_{def} \{m + n \mid m \in \Omega_M \text{ and } n \in \Omega_N\}$.

Definition 10 (Partners).
Two oWFNs M *and* N *are* partners *iff* $out_M \cap out_N = in_M \cap in_N = \emptyset$.

Definition 11 (Fellows).
Two oWFNs M *and* N *are* fellows *iff* $out_M \cap in_N = out_N \cap in_M = \emptyset$.