Compatibility of Data-Centric Web Services

Benoît Masson[1], Loïc Hélouët[2], and Albert Benveniste[2]

[1] Epitech Rennes, 12 square Vercingtorix, 35000 Rennes, France
[2] INRIA Rennes, campus de Beaulieu, 35042 Rennes Cedex, France,
benoit.masson@epitech.eu,loic.helouet,albert.benveniste@inria.fr

Abstract. Before using a service in a composite framework, designers must ensure that it is compatible with the needs of the application. The inputs and outputs must comply with the intended ranges of data in the composite framework, and the service must eventually return a value. This paper addresses compatibility for modules described with document-based workflow nets, that can depict the semantics of active XML (AXML) systems, a language for Web Services design. The behavior of non-recursive AXML specifications with finite data can be represented as Docnets, i.e., finite labeled Petri nets carrying information on document types they transform. Compatibility of docnet modules is characterized in terms of a decidable reachability property in the underlying net. Finally, we show the distributivity of compatibility over composition, which allows a faster semi-decision algorithm to verify compatibility between sets of modules.

Keywords: Data-Centric Web Services; Composition; Petri Nets

1 Introduction

E-business and supply chain management involve a combination of widely distributed workflow systems and data/information management. According to [13], these systems can be viewed as workflows, or as information systems. In the workflow-based perspective, process is emphasized. Web services and their orchestrations are now considered an infrastructure of choice for managing business processes and workflow activities over the Web [12]. BPEL has become the industrial standard for specifying orchestrations, and formalisms such as Orc [10] have also been proposed. In the information-based perspective, processes are considered as operations that are triggered as a result of information changes. Information-centric systems typically rely on database-oriented technologies and the notion of (semi-structured) *document*.

Today, technologies in use for these two aspects are mostly separated. The WIDE approach [5] was a first attempt to combine them, and was further developed in [13] to consider process, information, and organization. The notion of "business artifact" has been proposed at IBM as a framework combining workflow and data management [11,8]. Active XML (AXML) [1,2] was proposed as a framework for *document-based workflows*. It consists of XML documents with embedded guarded service calls and offers mechanisms to store and query semi-structured data distributed over entities called *peers*. In [3,7] distribution was explicitly introduced in AXML, thus giving rise to the model *Distributed AXML*

(DAXML). In DAXML, guards are local to a peer, services can be local or distant, and in the latter case specified through an *interface*. This allows to reason about a DAXML system, either globally, or locally by representing distant service calls by their interfaces. However, the notion of interface proposed in [7] only specifies data exchanged during distant calls. Implementing an interface then means accepting and returning correct data, but does not guarantee that a distant call eventually returns a value.

This paper proposes a framework for document-based workflows, i.e., workflows that circulate documents and whose transitions are instantiated and guarded by documents. Transitions of the workflow are guarded service calls and returns (the pair (call; return) is not atomic). Documents are business processes comprising both data and references to the services needed to process the data. Typically, documents are meant to be semi-structured data of XML type obeying DTDs. However, unlike in the above-mentioned works where the mechanisms of querying documents were explicitly considered, we will abstract away from any XML-related pattern matching mechanism. We first propose a generic model called *Docnets*. These nets are finite Petri nets whose places are typed by document types, and whose transitions transform documents, or model calls to distant websites. Docnets can be equipped with a composition operator, which allows designing the evolution of documents owned by an agent of a system that offers services to its environment. Such pair of document processing plus environment will be called a *module*. Modules can then be assembled to model the fact that an agent provides services to another agent. However, such composition can only occur if the required services (interfaces) and the provided services are compatible. We show that assuming that the data exchanged between docnet modules can be typed by a finite set of documents, and that services are not recursive, compatibility of services and termination of distant calls is decidable, even for Docnets which may treat an unbounded number of documents.

The paper is organized as follows: Section 2 describes the concepts of active documents. Then Section 3 introduces Docnets, their properties and two composition operators. Sections 4 and 5 propose two notions of compatibility for docnet modules, and shows that they are decidable. Due to lack of space, proofs are omitted but can be found in an extended version of this work [9].

2 Distributed active documents

Web Services architectures based on active documents were introduced by [1,2]. The concepts of this framework can be illustrated by the following example. Consider a website that provides information on current weather in a panel of cities in the world. This site returns XML-like documents that carry the name of a city, current temperature and weather, plus a forecast for tomorrow. The main function of this site is to return current data for a city, so for many clients of the service, the forecast is useless. *Active documents* embed references to services that can be called to extend the data in the document. In our weather website, the returned value can embed a reference to a new service that can be called by a client to get the forecast, or simply ignored (this is illustrated in Figure 1).

```
<weather>
  <city> Paris </city>
  <current>
      <sky> sunny</sky>
      <temp>  24 </temp>
  </current>
  <forecast>
      <sky> sunny </sky>
      <temp>  20 </temp>
  </forecast>
</weather>
```

```
<weather>
  <city> Paris </city>
  <current>
      <sky> sunny</sky>
      <temp>  24 </temp>
  </current>
  <forecast>
      <service> www.meteofrance.fr/
                    getForecast?query=Paris
      </service>
  </forecast>
</weather>
```

Fig. 1. A document and an active document returned by a website

A system composed of active documents is distributed over a set of agents. Each agent possesses local services, and references to services provided by other agents, called external services. Local services are functions that transform documents in two phases: first the service is called, and then the service returns. Calls and returns are guarded — for example, in AXML the guards are expressed using a fragment of Xpath [2]. Calls and returns modify the document owned by an agent in a deterministic way. Due to the mechanism of guards, services do not return immediately after a call, thus, complex workflows can be designed. In addition to local services, agents possess references to services provided by other agents, and offer some of their services to the external world. When seen from a given agent, an external service is simply perceived as a way of mapping a possible set of input parameters to a possible set of output parameters. Calling an external service consists in sending data to another agent and waiting for its answer. Offering a service consists in receiving a new document, transforming it using some local services, and returning the transformed document to the caller.

In this work, we will abstract away from concrete documents and query languages and will only assume that checking that a guard is satisfied is an effective procedure. Our model of active documents is introduced next.

3 Modeling active documents workflows with Petri Nets

We begin with background material on Petri nets. A *labeled net* (LPN) is a tuple (P, T, F, L, ℓ) where P is the set of *places*, T is the set of *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation* seen as a set of (directed) *arcs* between places and transitions, L is the set of *labels*, and $\ell : T \to L$ is the *labeling function*. For any node $n \in P \cup T$, its *preset* and *postset* are defined as ${}^\bullet n = \{x \mid (x, n) \in F\}$ and $n^\bullet = \{x \mid (n, x) \in F\}$, respectively. The "state" of a labeled net is represented by a *marking*, i.e., a mapping $m : P \to \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers. For $p \in P$, the value $m(p)$ is the number of *tokens* in place p. For m, m' two markings and t a transition, say that firing t from marking m is possible and produces marking m', denoted $m[t\rangle m'$ if $\forall p \in {}^\bullet t, m(p) \geq 1$ and if $m'(p) = m(p) - 1$ when $p \in {}^\bullet t \setminus t^\bullet$, $m'(p) = m(p) + 1$ when $p \in t^\bullet \setminus {}^\bullet t$, and $m'(p) = m(p)$ otherwise. A *labeled Petri net (LPN)* is a tuple (P, T, F, L, ℓ, m_0) where (P, T, F, L, ℓ) is a labeled net and $m_0 : P \to \mathbb{N}$ is the *initial marking.* A *firing sequence* is a bipartite sequence $m_0, t_1, m_1, t_2, m_2, \ldots$ such that $m_{i-1}[t_i\rangle m_i$. Say that m' is reachable from m, if there exists a firing sequence starting from marking m and ending in marking m'.

3.1 Documents and services

Petri nets alone are not sufficient to describe document manipulation. They just consume and produce untyped tokens. A solution to increase the expressive power of Petri nets is to add colors to tokens and control flows, but this usually results in undecidability of many problems. In this paper, we adopt a different technique. We add types to places (that is places will represent types of documents), and consider transitions as actions that transform documents. Tokens in a place represent documents of a type defined by the place.

We assume a (non necessarily finite) set $\mathcal{S}$ of *services* and *interfaces* names, and a set of *service marks* of the form $f!$ or $f?$ for all $f \in \mathcal{S}$. In a document, a service mark $f!$ means that service f can be called. Similarly, a service mark $f?$ means that service f was called and a return is awaited. To simplify the model, we will assume that every document that our systems manipulate contains at most one occurrence of each service mark. Note however that the model can be easily extended to documents carrying a bounded number of service marks.

Let us denote by $\mathcal{D}$ all types of documents (this set is not necessarily finite). We denote by $\sigma : \mathcal{D} \mapsto 2^{\mathcal{S}\times\{!,?\}}$ the labeling map that associates to every document type $d \in \mathcal{D}$ the service marks that appear in this document, i.e., the service calls and returns that may occur from this document provided guards of the considered services hold. Additional (unspecified) marked services or information may also occur in this document. We assume that $\mathcal{D}$ is equipped with a partial order relation $\leq$ such that for all $d, d' \in \mathcal{D}$, $d \leq d'$ implies $\sigma(d) \subseteq \sigma(d')$. Two document types d and d' are called *comparable* if either $d \leq d'$ or $d' \leq d$ holds. We furthermore suppose that it is decidable whether $d \leq d'$. For two sets of document types D, D', we will say that $D \leq D'$ if for every $d \in D$, there exists $d' \in D'$ such that $d \leq d'$.

A *service* is a tuple (f^c, G^c_f, f^r, G^r_f), where $f \in \mathcal{S}$ is the name of the service, f^c and f^r are the service *call* and *return* functions, with G^c_f and G^r_f the corresponding *guards*. Mappings $f^c, f^r : \mathcal{D} \mapsto \mathcal{D}$ are partial and map document types to document types, and $G^c_f, G^r_f : 2^{\mathcal{D}} \mapsto \{\mathtt{T}, \mathtt{F}\}$ are boolean predicates over sets of document types. Write $D \models G^c_f$ (resp. $D \models G^r_f$) if any set of documents with types $D = \{d_1, \ldots, d_k\}$ satisfies G^c_f (resp. G^r_f). We also assume that guard satisfaction is monotonous w.r.t $\leq$, i.e., if $D \models G^c_f$ and $D \leq D'$, then $D' \models G^c_f$.

3.2 Docnets

Definition 1. *A* docnet *is a tuple* $\mathcal{N} = (P, T, F, L, \ell, m_0, S, D, \ell')$ *where* $S \subseteq \mathcal{S}$ *is a finite set of service and interface names,* $D \subseteq \mathcal{D}$, $L = \big(S \times \{c, r\}\big)$, $\ell' : P \mapsto D$ *associates a type to each place of* P *and* (P, T, F, L, ℓ, m_0) *is a LPN.*

Intuitively, places in docnets represent *types of documents* involving a finite set of service marks. This is captured by a chain of labeling maps $P \xrightarrow{\ell'} \mathcal{D} \xrightarrow{\sigma} 2^{\mathcal{S}\times\{!,?\}}$ A token in a place represents an instance of a document of the given type. In particular, this document must contain the service marks of the document type associated to the place (plus possibly some additional information). A place can hold any non-negative number of tokens of the referred type. Figure 2 represents a partial docnet. Places are represented by circles, and labeled by the service marks provided by the σ map. Transitions are represented by rectangles with labels at their side. The dashed arrows indicate that the net contains more places and transitions that are not represented. The initial marking is symbolized by dark circles located in the initially marked places. For the sake of readability, we will not discuss the role of interfaces immediately and introduce them only at the end of this section.

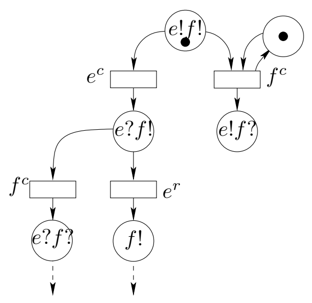

Fig. 2. A well-formed docnet.

As we want docnets to model guarded transformations of documents, we need to impose some consistency between transitions labeling, places labeling, and the transformations attached to transitions.

Definition 2 (well-formedness). *A docnet $\mathcal{N}$ is* well-formed *iff:*

1. *Every transition labeled f^c (resp. f^r) possesses in its preset a place having $f!$ (resp. $f?$) as part of its σ-label and in its postset a place whose σ-label has $f?$ substituted for $f!$ (resp. has $f?$ removed from it).*
2. *Any subset of places P' such that:*
   - *there exists a place $p \in P'$ with $f!$ (resp. $f?$) as part of its σ-label*
   - *$\ell'(P')$ satisfies the guard G^c_f (resp. G^r_f), and no subset $P'' \subset P'$ satisfies it.*

   possesses in its postset a transition t labeled f^c (resp. f^r). In addition, places in $P' \setminus \{p\}$ have t in their preset.
3. *Every transition labeled by f^c (resp. f^r) possesses a places p in its preset and p' in its postset such that $\ell'(p') = f^c(\ell'(p))$ (resp. $\ell'(p') = f^r(\ell'(p))$).*

Condition 1 means that a service can be activated only if it was mentioned in some document in the preset of a transition; activating this service results in a modification of the document. Condition 2 defines guards for docnets. Observe that well-formed docnets may specify services that return documents containing additional service marks of the form $f!$, thus allowing new service calls. Condition 3 indicates that a transition labeled by a service call or return transforms a document into another one according to the function labeling the transition. A consequence of these well-formedness rules is that transitions always have at least one place in their postset, and that subsets of places from which a service can be called (or can return) have a common transition in their postset. A well-formed docnet can thus be infinite. However, if no recursion among services occur, the docnet associated to the evaluation of a given document with local services (i.e., the replacement of all services by the actual data they represent in documents) is necessarily finite. It can be represented as the smallest well-formed docnet containing the initially marked places depicting the document under evaluation.

When a docnet is not well-formed, one can nevertheless define a closure operation to make it a well-formed docnet. For $\mathcal{N}$ a finite docnet defined over a set S of services, the closure operation $close(\mathcal{N})$ is defined as follows: for every service $f \in S$, for every place p such that $f! \in \sigma(\ell'(p))$ and every minimal subset P' of places of $\mathcal{N}$ such that $\ell'(P') \models G^c_f$, add a new transition t labeled by f^c, and a new place p' in the postset of t such that $\ell'(p') = f^c(\ell'(p))$ (if such transition/place does not already exist). Add t to the postset of each place in $\{p\} \cup P'$ and in the preset of each place in $\{p'\} \cup P'$. For every service $f \in S$, for every place p and minimal subset P' of places of $\mathcal{N}$ such that $f? \in \sigma(\ell'(p))$ and $\ell'(P') \models G^r_f$, add a new transition t labeled by f^r, and a new place p' in the postset of t such that $\ell'(p') = f^r(\ell'(p))$ (if such transition/place does not already exist). Add t to the postset of each place in $\{p\} \cup P'$ and in the preset of each place in $\{p'\} \cup P'$. We can now define the well-formed closure $\textit{wf-closure}(\mathcal{N})$ as the limit $\textit{wf-closure}(\mathcal{N}) = \lim_{n \mapsto \infty} \textit{wf-closure}^n(\mathcal{N})$, with:

$$\textit{wf-closure}^0(\mathcal{N}) = \mathcal{N} \text{ and } \textit{wf-closure}^n(\mathcal{N}) = close(\textit{wf-closure}^{n-1}(\mathcal{N}))$$

The *wf-closure* of a docnet $\mathcal{N}$ may not be finite if some services of S are recursively called. For N a finite docnet with finite set of places P, set of document types $D = \ell'(P)$ and set S of services, if there exists no $k \leq |S|$ and no f such that $f! \in (\sigma(D) \cap \sigma(D^k \setminus D))$ where D^k is the set of document types occurring in $\textit{wf-closure}^k(N)$, then the *wf-closure* of $\mathcal{N}$ is finite. Note that this is only a sufficient condition, and that due to markings, recursion might never occur. The closure operation for a net without transitions is a well-formed net.

Docnets describe the evolution of documents through embedded services evaluation. Of course an agent can only use local services that it implements. This is captured by the notion of *peer*: a peer is a pair $(\mathcal{N}, S)$, where S is the set of services that are accessible by the agent, and $\mathcal{N}$ is a well-formed docnet w.r.t. services of S, which services calls and returns all belong to S. Adding new documents to a system may allow new service calls or returns by making their guards true. Composing workflows of disjoint documents is then more than the simple union of their nets, and is defined by the parallel composition operation below.

Definition 3 (Parallel composition). *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two well-formed docnets. Their* parallel composition $\mathcal{N}_1 \parallel \mathcal{N}_2$ *is obtained as follows:*

1. *Compute the disjoint union of the underlying nets, seen as graphs, i.e., compute the disjoint union of places, transition, initial markings and flows.*
2. *Make the resulting docnet well-formed by wf-closure.*

The parallel composition preserves the behaviors of both nets. It also allows more behaviors, by synchronizing the filling of places, and then by performing

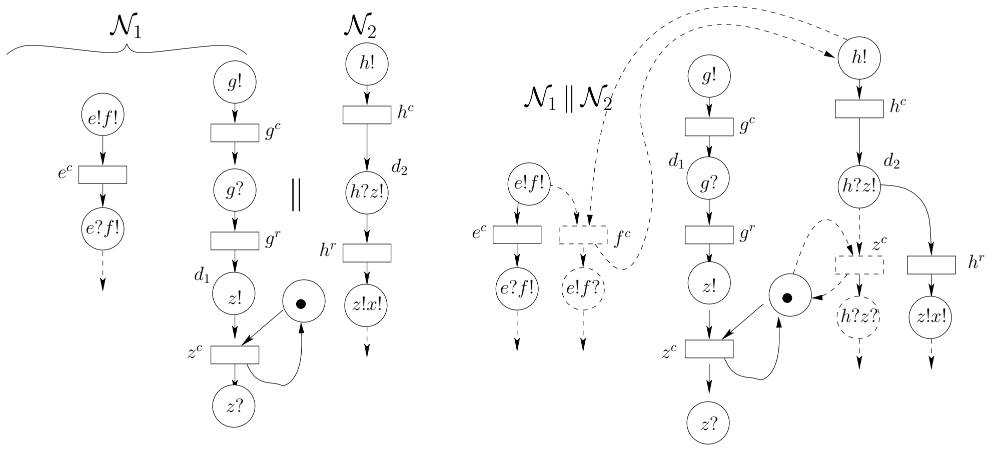

Fig. 3. Parallel composition of docnets.

the well-formed closure of the so obtained system. The main intuition behind parallel composition is that when a place with document type d_2 is filled in net $\mathcal{N}_2$, and when there exists a place with type $d_1 \leq d_2$ in net $\mathcal{N}_1$, then anything allowed due to the presence of a token of type d_1 in $\mathcal{N}_1$ should be allowed as soon as a new token arrives in place of type d_2. Figure 3 is an illustration of parallel composition. Note that in $\mathcal{N}_2$ alone, firing z^c is not possible, as no place in $\mathcal{N}_2$ satisfies the guard for z^c. Composing with $\mathcal{N}_1$ provides the needed guard. This hence results in appending a transition labeled by z^c. A similar situation holds for transition labeled by f^c. The places, transitions and flows added at step 2 are represented with dashed lines. Parallel composition is commutative, associative, but not idempotent (composing a docnet with itself results in a larger net).

3.3 Modules and interactions with the environment

A site in web service architectures is an open system: it accepts incoming service calls from its environment, and also expects the environment to provide services, known only through a web address (URI), accepted inputs and returned outputs. Even if a site never produces documents of some type, external calls or returns from distant services may involve documents of this type. Hence interacting with an environment may validate guards of services that would not be enabled otherwise. It is thus worth augmenting a docnet by a model of all demands coming from its environment, and all interactions it may have with distant services. This is achieved by adding a model for *interactions with the environment* to the considered docnet.

Consider a docnet $\mathcal{N}$ and a service g it provides to its environment. Let $D_g = \{d_1, \ldots, d_n\}$ be the set of (valid) document types that are allowed as parameters for a call to g. We assume D_g to be finite, and that each d_i embeds a call to g. Exposing the pair (g, D_g) to the environment is described as the parallel composition of $\mathcal{N}$ with the well-formed closure of a docnet $\mathcal{N}_{\text{env}}$ that contains n places $P^g_{\text{env}} = p^g_1, \ldots, p^g_n$ with respective types $d_1, \ldots, d_n$, plus a transition $t^{d_i}_{g,env}$ labeled by env^c_g for each allowed parameter d_i, with p^g_i in its postset. We also

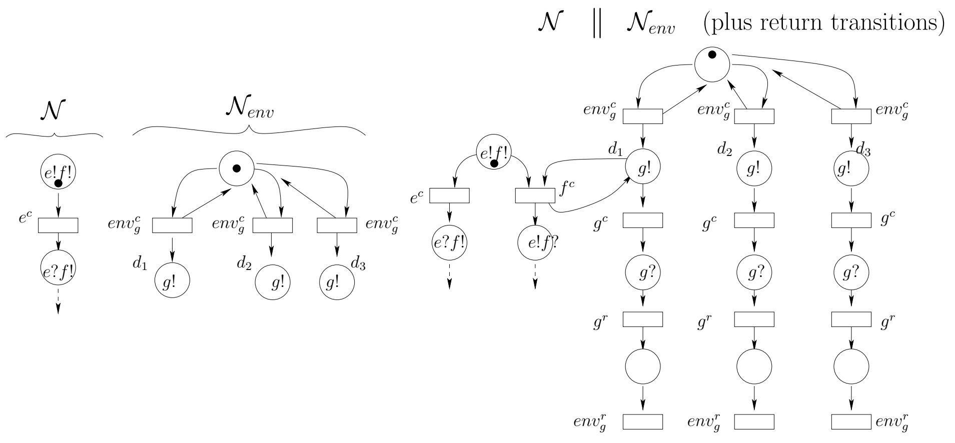

Fig. 4. A docnet providing service g with $D_g = \{d_1, d_2, d_3\}$ to its environment.

add to $\mathcal{N}_{\text{env}}$ a place p_{env} of type $init_{p_{env}}$, which has each $t^{d_i}_{g,env}$ in its postset and preset. Though this place does not change fundamentally the behavior of the environment net, it allows to control the environment if needed. As a result of step 2 of definition 3, constructing $\mathcal{N} \parallel \mathcal{N}_{\text{env}}$ unfolds both $\mathcal{N}$ and $\mathcal{N}_{\text{env}}$.

Arrival of new calls is symbolized by the set of transitions $t^{d_1}_{g,env}, \dots, t^{d_n}_{g,env}$, with respective postsets $p^g_1, \dots, p^g_n$. However, termination of calls is not yet modeled in $\mathcal{N} \parallel \mathcal{N}_{\text{env}}$. We symbolize this termination by adding a transition t_p labeled by env^r_g from every place p accessible from some p^g_i in P^g_{env} such that $\{g?, g!\} \cap \sigma(\ell'(p)) = \emptyset$ (which means that the call to service g was completed). This construction is illustrated in Figure 4, for a net $\mathcal{N}$ allowing environment calls to service g with parameters $\{d_1, d_2, d_3\}$. This modeling of environment can be extended to an arbitrary number of services in S with their call parameters.

Interfaces and their implementation. Web services are often orchestrations of local services, or services provided by other sites. At design time, these latter are usually known only as interfaces, that depict the parameters sent to a service that implements this interface, and the expected possible values returned by an implementation. In addition to usual service transitions, we allow for interface transitions. We will differentiate interfaces from services by writing their name in capital. Interface transitions will be simply labeled by I^c and I^r, denoting a call to an external service and a return. Like services, a call to an interface updates a document. It also sends parameters to the called site. The return from a distant call may be of several types, and hence distant call returns can not be modeled as for services. An *interface* is a pair $I = (params_I, D_I)$, where $params_I : \mathcal{D} \mapsto \mathcal{D}$ is a function that extracts parameters of a call from a document, and D_I is a *finite* set of document types depicting the expected returned values after a call to a distant service implementing interface I. At design stage, interfaces need not be implemented, and calls and returns can be represented as transition labeled by I^c and I^r that are fireable from any place with $I!$ in $\sigma(\ell'(p))$ (resp. $I?$ in $\sigma(\ell'(p))$). When no implementation is known, firing an interface changes a tag in a document from $I!$ to $I?$ indicating that an external service is being processed. Unlike services, when a return from an interface call occurs at a place with type d such that $I? \in \sigma(d)$, the effect of receiving an answer on d depends on the received value. Hence for every place with type d such that $I? \in \sigma(d)$, we will create one transition labeled I_i^r per document type d_i in D_I. As for services, we will assume that the effect of receiving an answer of type d_i from a document d is computable and deterministic, and results in a new document type $d +_I d_i$ (operation $+_I$ usually inserts document d_i into d at correct place). Hence, a transition labeled by I_i^r takes a token in a place of type d, and necessarily outputs a token in a place p' of type $d +_I d_i$. I^c and I^r transitions are not guarded: external service can always be called to enrich a document, and the answer can be returned at any moment after a distant call.

Figure 5 shows a docnet with a non-implemented interface I that accepts two return types. Non-implemented interfaces refer to functionalities provided by the environment. Defining the kind of document that can be returned is sometimes sufficient to study properties of a docnet in *any possible environment*: if a document type d is not reachable from an initial marking of a net with non-implemented interfaces, then this document type is not reachable either when interfaces are implemented by services returning only expected values. To keep well-formedness in presence of interfaces, we add a rule to definition 2:
(4) for every place p such that there exists an interface I with $I? \in \ell'(p)$ *then there exists a set* $t_1, ...t_{|D_I|}$ *of transitions labeled by* $I_i^r, i \in \{1, .., |D_I|\}$ *in the postset of p, and each* $t_i, i \in \{1, .., |D_I|\}$ *has a place of type* $\ell'(p) +_I d_i$ *in its postset.*

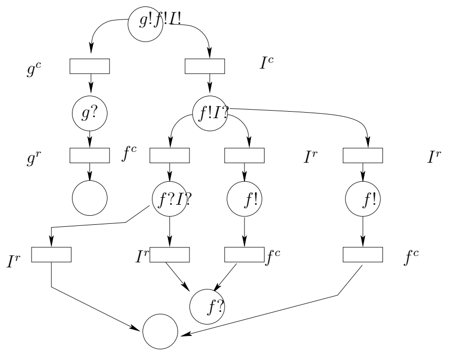

Fig. 5. A docnet with a non-implemented interface

A peer whose docnet models interactions with the environment and contains interface calls and returns can be seen as a module. Once modules are defined, the remaining task is to compose them, that is connect interfaces with services of other modules that implement them.

Definition 4. *A* docnet module *is a triple* $M = (\mathcal{N}, S, \mathcal{F}, \mathcal{I})$, *where* $(\mathcal{N}, S)$ *is a peer,* $\mathcal{F} \subseteq S$ *is a set of pairs of the form* (f, D_f) *depicting services proposed to the environment and their call parameters,* $\mathcal{I}$ *is a set of interfaces of the form* $(params_I, D_I)$. *We require that for every* $(f, D_f) \in \mathcal{F}$, *the environment part of* $\mathcal{N}$ *models external calls to every service* f *with parameters* D_f.

4 Composability

Composability of a service and an interface ensures that modules agree on the exchanged data during a call. Algorithms to check such property are important for distributed systems, since it allows component-based design and orchestration of services. A notion of composability for document-based workflow systems was already proposed in [3,7].

Definition 5. *Let $\mathcal{N}$ be a docnet, and p be one of its places. The* subnet *of $\mathcal{N}$ connected to p is the restriction of $\mathcal{N}$ to set of places P and transitions T, where P,T are the solutions of fixed point equation $(P^n, T^n) = (P^{n-1}, T^{n-1})$ where:*

$$\begin{array}{ll} P^0 = \{p\} & P^i = \{p \in P \mid \exists t \in T^{i-1}, p \in t^\bullet \textit{ and } p \notin {}^\bullet t\} \\ T^0 = \{t \in p^\bullet \mid p \in P^0\} & T^i = \{t \in p^\bullet \mid p \in P^{i-1}\} \end{array}$$

The subnet connected to a place depicts the successive evolutions of a document, forgetting details about guards needed to launch service calls or returns. We will use subnets to characterize control flows associated to external calls.

Definition 6. *Let $\mathcal{N}$ be a docnet, f be a service, and d be a document type such that $f! \in \sigma(d)$. Let $\mathcal{N}' = \mathcal{N}||\mathcal{N}_d$, where $\mathcal{N}_d$ is a net comprising a single place p_d of type d. The* execution *of f at $\mathcal{N}$ from d is the subnet connected to p_d in $\mathcal{N}'$. The* return places *of this execution are places p such that $f?, f! \notin \sigma(\ell'(p))$.*

Intuitively, the execution of f at $\mathcal{N}$ from d depicts the control flow followed by an external call to service f with parameters d arriving at a site that provides service f. Return places are the result of the execution of a request to f starting from parameter of type d.

Definition 7. *Let M_1, M_2 be docnet modules, and $I = (params_I, D_I)$ be an interface of M_1. We will say that service f of M_2 is* composable with interface I at place p *in M_1 if and only if:*

- *$I! \in \sigma(\ell'(p))$: an external call to a service depicted by interface I in $\mathcal{N}_1$ can be performed from place p.*
- *M_2 accepts call to service f with parameter $d = params_I(\ell'(p))$, i.e., $(f, D_f) \in \mathcal{F}_2$, with $d \in D_f$.*
- *Every return place connected to place p_d in the net $\mathcal{N}_2||\mathcal{N}_d$ has a type in D_I.*

We will say that service f of M_2 is composable *with interface I of M_1, and write $f \models I$, if it is composable with I at all places p of $\mathcal{N}_1$ such that $I! \in \sigma(\ell'(p))$.*

Note that composability of a service and its interface does not mean that the called service always terminates. It ensures that parameters of calls are accepted, that returned values are defined in the interface, but not that a return place is eventually filled, nor that such place exists (this might for instance be the case if executing f^c needs a guard that is never true at the called site). Section 5 addresses termination of external service calls. Composability of an interface

$I = (params_I, D_I)$ and of a service f that accepts input parameters D_f is decidable when the type of all places in the preset of transitions labeled by I^c is finite (this ensures finiteness of call possibilities as $params_I(\ell'(^\bullet(l^{-1}(I^c))))$ is finite) and the environment part of $\mathcal{N}_2$ is finite. In particular, if $\mathcal{N}_1$ and $\mathcal{N}_2$ are finite, composability of I and f is decidable. See [3] for a full proof in the DAXML context.

All AXML concepts can be mapped to Docnets. An AXML document is a finite XML document, depicted by a document type in a docnet. AXML works by calling services, and returning separately the results. Calls and returns are guarded queries, that is deterministic computations of a set of values from a finite XML database. Guards are defined with a fragment of Xpath, and their evaluation is also an effective procedure. However, negation of Xpath expressions to test absence of some data breaks monotony (i.e., we do no necessarily have $d \models g$ and $d \leq d' \Rightarrow d' \models g$). To preserve monotony, which is essential for parallel composition and modeling interactions with the environment, we have to restrict to *positive* guards that can only test the presence of some data or pattern on documents. Last, an AXML service can return references to other services to be called, hence yielding recursion. However, one can easily avoid recursion by forbidding cyclic dependencies among calls and returned values embedding services. Hence as far as positively guarded document transformations are concerned, docnets and AXML systems are equivalent models. The remaining question is then the modeling of an environment. In docnets, we assume that every service proposed to the environment can be called with a finite set of parameters, described as a finite set of document types. Such a restriction does not exist in AXML, but could be enforced using a DTD to filter external calls. We refer to [3] for a complete semantic mapping between AXML and Docnets.

5 Compatibility between Modules

In this section, we go beyond composability and address the termination of distant calls. We study two different notions of "behavioral" compatibility between modules, namely weak and strong compatibility. The weak notion allows the reception of particular environment calls (i.e., firing of transitions labeled env_f^c for some $f \in \mathcal{F}$) that may unblock the treatment of a distant call, while the strong one should complete requests in any environment. We then show that for finite modules, these properties are decidable (Theorem 2). We also show that compatibility "distributes" over the composition of modules (Theorem 4), which leads to a faster semi-algorithm to decide compatibility. The work in [4] considers a close notion for session types. Starting from a global specification, the problem is to ensure that a distribution on distant sites allows termination and correct typing of returned values. We emphasize that in our model, services distribution is already performed, which leverages a part of the problem addressed in [4].

Definition 8 (compatibility). *Consider a module M_1 accepting environment calls to $f \in \mathcal{F}$, and a module M_2 owning an interface I such that $f \Vdash I$. M_1 and M_2 are* weakly (I, f)-compatible *(denoted $M_1 \;{}_I\!\underset{\sim}{\lhd}_f\; M_2$) if and only if after*

some *environment calls to $\mathcal{F}$, any firing of a transition env^c_f with parameters allowed by I is eventually followed by a corresponding response env^r_f.*

M_1 and M_2 are (strongly) (I, f)-compatible *(denoted $M_1 \;{}_I{\lhd}_f\; M_2$) if and only if after* ***any*** *environment calls to $\mathcal{F}$, any firing of a transition env^c_f with parameters allowed by I is eventually followed by a corresponding response env^r_f.*

Clearly, $M_1 \;{}_I{\lhd}_f\; M_2$ implies $M_1 \;{}_I{\underset{\sim}{\lhd}}_f\; M_2$. Let us now prove that these notions of compatibility are decidable. The two notions mean that an external call to a service can terminate with or without the help of its environment. If we consider a marking of the docnet depicting behaviors of module M_1, and if we isolate a token in a parameter place p_d filled by a transition labeled by env^c_f, we should be able to find a reachable marking in which **this** token can be consumed by a transition labeled with env^r_f. "Isolating a token" can be modeled by adding to $\mathcal{N}_1$ (with the parallel composition operator) a copy of the net $\mathcal{N}_{params_I}$ associated to the processing of a the call parameters of I, which minimal place can be fed only once. We can then connect $\mathcal{N}_1$ to $\mathcal{N}_{params_I}$ in two different ways: in the first way, the first transitions (transitions labeled by env^c_f in $\mathcal{N}_{params_I}$) consume the token of place p_{env} in the environment part of $\mathcal{N}_1$. This modeling prevents any incoming call from the environment once a document is being processes in $\mathcal{N}_{params_I}$. Call this net $\mathcal{N}_{strong}$. The second solution is to let the $\mathcal{N}_1$ and $\mathcal{N}_{params_I}$ run in parallel, hence allowing environment calls while a document is being processes in $\mathcal{N}_{params_I}$. Call this net $\mathcal{N}_{weak}$.

Then, we can show that weak and strong (I, f)-compatiblity amounts to verifying some home-space property [6] respectively in $\mathcal{N}_{weak}$ and $\mathcal{N}_{strong}$ (complete proof is detailed in [9]). Note that an environment call that does not terminate in general, may terminate when restricted to parameters D_I. So, the fact that a particular environment call does not return in general does not imply that two modules are not (I, f)-compatible. Note also that deciding a home-space property relies on reachability in Petri nets, and can hence be a costly operation.

Theorem 1. *Consider two modules M_1 and M_2. Let I be an interface of M_2 and f a service of M_1 such that $f \Vdash I$. If $\mathcal{N}_1$ and $\mathcal{N}_2$ are finite, strong (I, f)-compatibility and weak (I, f)-compatibility are decidable.*

Usually, module composition involves more than one pair service/interface. A module M_1 can provide some services to M_2, but at the same time expect some functionalities (expressed as interfaces) that are implemented in M_2. Interfaces and services are paired via explicit mappings specified by the designer. A *pairing map* ξ is a mapping from some interfaces of a module to services provided by another module, with the constraint that $\xi(I) \Vdash I$ for all $I \in \mathrm{dom}(\xi)$.

We can now define compatibility of two modules with respect to a composition schema defined by a pairing map. The modules M_1 and M_2, with respective sets of external services $\mathcal{I}_1$ and $\mathcal{I}_2$, are *strongly* [resp., *weakly*] *compatible* with respect to a pairing map ξ if and only if for all $I \in \mathrm{dom}(\xi) \cap \mathcal{I}_1$, $M_1 \;{}_I{\lhd}_{\xi(I)}\; M_2$ [resp., $M_1 \;{}_I{\underset{\sim}{\lhd}}_{\xi(I)}\; M_2$], and for all $I \in \mathrm{dom}(\xi) \cap \mathcal{I}_2$, $M_2 \;{}_I{\lhd}_{\xi(I)}\; M_1$ [resp., $M_2 \;{}_I{\underset{\sim}{\lhd}}_{\xi(I)}\; M_1$]. Strong and weak compatibility are denoted respectively $M_1 \overset{\xi}{\bowtie} M_2$ and $M_1 \overset{\xi}{\underset{\sim}{\bowtie}} M_2$ (the symbol ξ may be omitted when it is clear from the context). The decidability result of compatibility of services and interfaces can be easily extended to modules and pairing maps (complete proof in [9]).

Theorem 2. *Let M_1, M_2 be two docnet modules, with finite docnets. Then, compatibility and weak compatibility of modules are decidable.*

5.1 Connecting interfaces and their implementations

Let us consider two docnet modules M_1 and M_2 such that M_1 comprises a non-implemented interface I and M_2 a service f with $f \mid\models I$. As M_1 and M_2 represent distinct sites that communicate through invocations, the document types manipulated in a module should not be used to satisfy guards in the other module. We will hence consider that distinct modules are defined over distinct and incomparable document types. The *composition* of M_1 and M_2 under mapping (f, I) is denoted by $M_1 \bigotimes_{I,f} M_2$ and consists in a new module $M = (\mathcal{N}', S_1 \cup S_2, \mathcal{F}_1 \cup \mathcal{F}_2, \mathcal{I}_1 \setminus \{(params_I, D_I)\} \cup \mathcal{I}_2)$, where $\mathcal{N}'$ is the docnet computed as follows:

- Compute $\mathcal{N} = \mathcal{N}_1 || \mathcal{N}_2$
- Compute $\mathcal{N} || \mathcal{N}_{d_1} || \dots || \mathcal{N}_{d_k}$ for every $d_i, i \in \{1, .., k\}$ such that there exists a transition t_i in $\mathcal{N}$ with label I^c, and a place $p \in {}^\bullet t_i$ with type d such that $params_I(d) = d_i$.
- Connect every transition t_i in $\mathcal{N}_1$ to place p_{d_i} in $\mathcal{N}_{d_i}$ (i.e., set $t^\bullet = t^\bullet \cup p_{d_i}$).
- Connect every return place p_j of type d_j in a net $\mathcal{N}_{d_i}$ to every transition t'_i labeled by I^r_j in the subnet connected to the transition t_i that feeds place p_{d_i} in $\mathcal{N}_1$ (i.e., set $t'_i \in p_j^\bullet$).
- Remove from $\mathcal{N}_1$ all transitions t'_i labeled by I^r_j in the subnet connected to the transition t_i that feeds place p_{d_i} such that no return place carries type d_j in the subnet connected to p_{d_i}.

Note that as $\mathcal{N}_2$ already accepts calls from the environment to service f, adding a request from $\mathcal{N}_1$ to execute service f does not add new document types to $\mathcal{N}_2$. Figure 6 below illustrates an implementation of an interface I by a service h. The interface can call h with a single parameter type, and accordingly, h can return two results. In this drawing, the added arcs are represented by dashed lines, return places by thick circles, and the removed part of the net by a gray zone. Transition I^r_3 cannot be fired anymore in the composition, as return type d_3 is never produced when executing h. This construction extends in an obvious way to an arbitrary number of modules and arbitrary pairing maps, and we will denote by $M_1 \bigotimes_\xi M_2$ the composition of two modules under pairing map ξ (the ξ symbol may be omitted when the map is clear from context). Slightly abusing the notation , we will also write $\bigotimes_{i \in K} M_i$ to denote the composition of a set of modules $\{M_i \mid i_i n K\}$ with appropriate pairing maps.

Theorem 3. *Let $(M_i)_{i \in K}$ be a finite family of modules, and let $M' = (M_1 \bigotimes_{\xi_1} M_2) \bigotimes_{\xi_2} \cdots \bigotimes_{\xi_{k-1}} M_k))$. Let m be a non-reachable marking of $\mathcal{N}_1$. Then for every reachable marking m' in the docnet of M', the restriction of m' to places of $\mathcal{N}_1$ differs from m.*

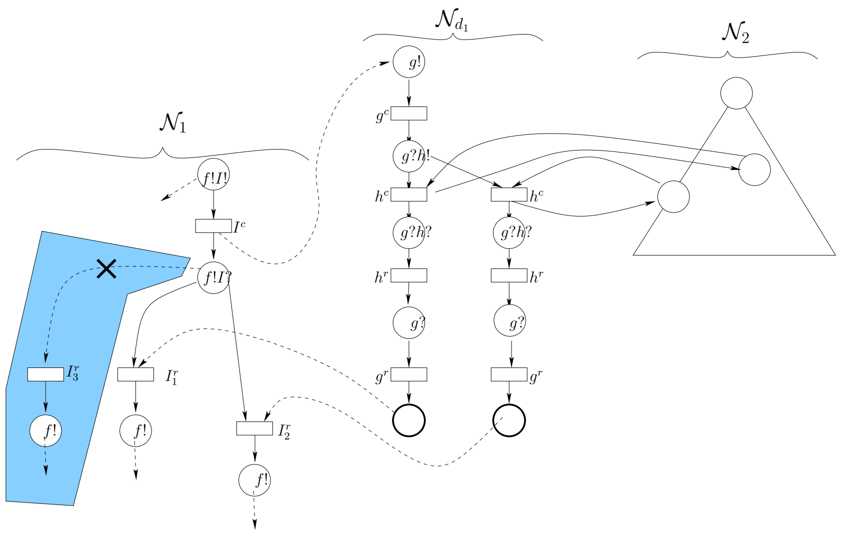

Fig. 6. Composition of modules

This property can be used to check for local safety properties of modules. The proof of this theorem is rather straightforward, as assembling modules does not create documents that were not already considered in the environment.

5.2 Distributivity of Compatibility

Compatibility has an interesting property: if several modules are pairwise-compatible, then any of their compositions are also compatible. This is useful because it allows a faster semi-algorithm to decide whether a large set of modules is compatible, by checking compatibility between pairs of modules only.

Theorem 4. *Let $(M_i)_{i\in K}$ be a finite family of modules. For any disjoint sets $K_1, K_2 \subseteq K$ and any pairing maps defined over disjoint domains, if all modules M_i are pairwise-compatible, then $(\bigotimes_{i\in K_1} M_i) \bowtie (\bigotimes_{j\in K_2} M_j)$*

Note that this theorem also holds for weak compatibility (a complete proof can be found in [9]). Also observe that the converse implication of Theorem 4 is not always true, i.e., $(\bigotimes_{i\in K_1} M_i) \bowtie (\bigotimes_{j\in K_2} M_j)$ does not imply that $M_i \bowtie M_j$ for all $i \in K_1$ and $j \in K_2$. Considering three modules M_i, M_j, M_k, with $i \in K_1$ and $j, k \in K_2$, the composition of M_j with M_k can restrict the possible behaviors of M_j. Hence, a value returned by a service of M_j that was not allowed by interface I of M_i may never be returned by the composition of M_j and M_k.

Theorem 4 provides a semi-algorithm to check compatibility of a set of modules without building a docnet involving all modules. The semi-algorithm checks compatibility for every pair of modules, and returns `true` when all the checks are positive, thus proving global compatibility. It returns `false` otherwise: it does not necessarily mean that the modules are not compatible, and finer checks can then be performed. Again, complete details and algorithms can be found in [9].

6 Conclusion and perspectives

We have proposed a Petri net model for document-based workflows called *docnets*. It encodes the semantics of a subset of Distributed Active XML. Composability and compatibility between modules with finite docnets are decidable, and semi-algorithms can be used for faster decision. A first extension of this work is to refine compatibility to cases where some environment calls needed to ensure progress of a service are guaranteed to occur by a contract. Even if recursion leads to undecidability [3,7], we also think that our results still hold if recursion does not create an unbounded number of service references in document types.

Compatibility is brought back to home-space problems, which use reachability checks (an $EXPSPACE$-hard problem). This may mean that our compatibility notion is not practical. However, Docnets have well-structured sets of configurations, and the markings considered in our compatibility definition contain only one token in return places. This may allow solving compatibility with efficient backward analysis techniques. We also need to improve drastically the size of the considered docnets, which grow rapidly during composition. A key issue is to avoid enumerating data values (for instance in calls parameters). Finally, we think that working with infinite but well-structured sets of document types, still allows decidability of compatibility.

References

1. S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: A data-centric perspective on web services. In *BDA 2002*, 2002.
2. S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of Active XML systems. In *PODS 2008*, pages 221–230, 2008.
3. A. Benveniste and L. Hélouët. Distributed Active XML and service interfaces. Technical Report 7082, INRIA, 2009.
4. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP*, volume 4421 of *LNCS*, pages 2–17, 2007.
5. S. Ceri, P. Grefen, and G. Sánchez. WIDE: A distributed architecture for workflow management. In *RIDE 1997*, pages 76–79, 1997.
6. D. de Frutos Escrig and C. Johnen. Decidability of home space property. Technical Report 503, LRI, 1989.
7. L. Hélouët and A. Benveniste. Document based modeling of web services choreographies using Active XML. In *ICWS 2010*, pages 291–298, 2010.
8. R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM 2008*, volume 5332 of *LNCS*, pages 1152–1163, 2008.
9. B. Masson, L. Hélouët, and A. Benveniste. Compatibility between DAXML schemas. Technical Report 7559, INRIA, 2011.
10. J. Misra and W.R. Cook. Computation orchestration. *Software and Systems Modeling*, 6(1):83–110, 2007.
11. Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.
12. Wil M. P. van der Aalst and Kees van Hee. *Workflow management: Models, Methods, and Systems.* MIT Press, 2002.
13. J. Wang and A. Kumar. A framework for document-driven workflow systems. In *BPM 2005*, volume 3649 of *LNCS*, pages 285–301, 2005.