Compatibility between DAXML Schemas

Benoît Masson, Loïc Hélouët, Albert Benveniste

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Compatibility between DAXML Schemas

Benoît Masson — Loïc Hélouët — Albert Benveniste

N° 7559

March 2011

Networks, Systems and Services, Distributed Computing

ISSN 0249-6399 ISRN INRIA/RR--7559--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Compatibility between DAXML Schemas

Benoît Masson, Loïc Hélouët, Albert Benveniste

Domain : Networks, Systems and Services, Distributed Computing
Équipe-Projet DistribCom

Rapport de recherche n° 7559 — March 2011 — 36 pages

Abstract: A challenge in Web Services architectures is to compose basic services to obtain larger applications. Before using a service, a designer must ensure that it is compatible with the needs of the application. This means that inputs and outputs of the service comply with the intended ranges of data in the composite framework, but also that the service eventually returns a value. In this paper, we address the compatibility problem for modules described with Distributed Active XML (DAXML), a language for Web Services design. We first show that the behavior of non-recursive DAXML specifications with finite data can be represented as bounded labeled Petri nets. We then define compatibility of a DAXML service with some needs in terms of a home space property in the underlying Petri net. We use this result to define compatibility between DAXML modules, and prove its decidability. Finally, we give a faster semi-decision algorithm to verify compatibility between arbitrary sets of modules, without building the Petri net depicting the behavior of all modules.

Key-words: Web Services; Compatibility; Termination; Petri Nets

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Compatibilité entre schémas DAXML

Résumé : Dans le domaine des architectures de services web, la composition de services est un défi majeur devant permettre d'obtenir des applications à grande échelle. Avant d'utiliser un service, un créateur d'application doit s'assurer que ce service est compatible avec ses besoins. Cela implique non seulement que les plages d'entrée et de sortie du service satisfont à certains critères, mais également que le service garantisse un retour. Dans cet article, nous nous intéressons à la compatibilité entre modules décrits à l'aide de *Distributed Active XML* (DAXML), un langage pour la mise en oeuvre de services web. Nous montrons dans un premier temps que le comportement de spécifications DAXML non-récursives et à données finies peut être représenté par un réseau de Petri borné. Nous exprimons alors la compatibilité entre services DAXML en termes d'espace d'accueil du réseau de Petri sous-jacent. Nous utilisons ces constructions pour définir la compatibilité entre modules DAXML, et montrer sa décidabilité. Nous terminons en proposant un semi-algorithme permettant de vérifier plus rapidement la compatibilité entre des ensembles quelconques de modules, sans pour cela avoir besoin de construire l'intégralité du réseau de Petri décrivant leur comportement global.

Mots-clés : Services web ; Compatibilité ; Terminaison ; Réseaux de Petri

1 Introduction

Web Services composition and discovery is a challenging topic: distinct services from several providers must be assembled to build a web application from the orchestration of smaller bricks. These bricks are not only computation tasks, that could be performed locally, but can also involve access to information owned by the service provider. Services are very often described as an interface, that specifies how to call a service, interact with it, and the range of values that shall be returned. Discovering a service then amounts to finding in a repository the service that complies with the needs of the orchestration. Attempts such as WSDL [16] have been proposed to collect information about services, but it is now admitted that finding the right service for some need is not a completely automatizable task, as the service description, which remains informal, is the main element that helps selecting services in a repository. The automatizable part hence mainly concerns the verification of whether the interface of a service matches the needs of an user. In particular, it means that a service called with correct parameters terminates, and return results within the range specified by the service interface. Once a pool of building service bricks have been chosen, the orchestration invokes them and collects answers using a middleware (CORBA, DCOM), communication protocols over the web (e.g., SOAP), or simply the standard HTTP protocol (as is done in REST).

Several languages have been proposed for services orchestrations. Without being exhaustive, one can cite BPEL [14], ORC [11], XML nets [12, 5], or more recently Active XML (AXML) [2]. BPEL, ORC, and XML nets explicitly describe the workflows in Web Service compositions, while AXML makes them explicit at runtime. In this work, we mainly focus on a variant called Distributed AXML (or DAXML for short) [3], that takes distribution of data and services into account. One advantage in using DAXML is that it is at the same time a formal model and a running implementation [2]. Once a Web Service composition is proved correct, it should run as designed if adequate and reliable communication means between components are provided.

In this work, we propose a new notion of compatibility between services, described with DAXML, where needs are described as input/output interfaces. This notion of compatibility holds when a service terminates and returns correct outputs. We show that this compatibility relation is decidable for a subset of DAXML that bounds recursion and deals with finite data. The decidability of compatibility relies on a translation from the considered fragment of DAXML to Petri nets. Compatibility consists in syntactic criteria (that ensure that services are called with correct data and return expected values) as in previous work [3], plus an additional termination constraint that can be brought back to a home space property of the underlying net. We then build on this compatibility relation to define compatibility between DAXML modules with respect to a map that pairs needs and services of the composed peers. As checking a home space property is a costly operation, we then provide a faster semi-decision algorithm that checks compatibility of DAXML modules without building the whole associated net.

The translation from DAXML to Petri nets uses two new operations that merge two labeled nets, or conversely split them into separated components. These two operations emphasize distribution of actions over a set of processes. Furthermore, the merging operation allows to identify recurrent behavioral pat-

terns within a behavior. Thanks to these two operations, our translation from DAXML to labeled Petri nets becomes more than a simple operational conversion of DAXML semantics, and allows to work with finite nets.

This paper is organized as follows: Section 2 defines syntax and semantics of DAXML. Section 3 defines the translation from DAXML documents to labeled Petri nets. Section 4 builds on this translation to show that termination of a service is decidable for a fragment of DAXML. Section 5 studies the distributivity of compatibility, and proposes a semi-decision algorithm to check compatibility of DAXML modules, before conclusion.

2 Distributed Active XML

Active XML (AXML) is a declarative language that was proposed by [1]. It mainly consists in XML documents with embedded service calls that transform these documents. This model has been extended to include guards [2], and distribution [3, 9]. In this latter version, one can consider a Distributed AXML (DAXML) model as a set of semi-structured documents (trees) and services located on distinct machines called *peers*. Services are programs, that run locally to a peer and transform a document. They modify locally the trees owned by their peer, but can also be called by a distant peer. A peer can hence call distant services that are known only via an interface, that depicts inputs and outputs to a distant program located on another machine.

Services are described as guarded rewriting rules, plus return conditions. Guards can then be used to model complex workflows as in ORC or BPEL. DAXML is then a powerful model, and can simulate Turing machines [2]. The model itself is similar to XML nets [12], in that a service queries a document whenever a guard holds. However, XML nets use a high-level Petri net representation of the workflow, while DAXML is defined by formal declarative rules which do not make the workflow explicit. Moreover, the definition of DAXML allows "lazy evaluation" of services, i.e., a service can return the address to a new service, letting the orchestrator choose whether he wants to call it or not.

In DAXML, distant calls are performed by a peer when no local service can fulfill the operation provided by the called peer. The needs of a peer are given under the form of a *requirements interface* (or simply interface), i.e., a pair of patterns that describes the range of inputs and the range of outputs that a service implementing the needed functionality should accept at calling time, and return after completion. A natural question is then whether a service f and an interface I are compatible. A static notion of compatibility was proposed in [3]: f is compatible with I if it accepts all input parameters described in I and only returns output values allowed by I. However, this definition is purely *static*: it does not guarantee that a call will eventually return. In the rest of this section, we formalize the notions of DAXML schemas, services, interfaces.

2.1 Trees, Patterns, and Services

We consider DAXML documents as unordered and unranked (arbitrary number of children) trees, labeled with tags, data values, and service names. A *DAXML document* is a labeled tree $T = (N, E, \mu)$, where N is a set of nodes, E is the set of directed edges, and μ is a labeling function. Internal nodes of a tree are

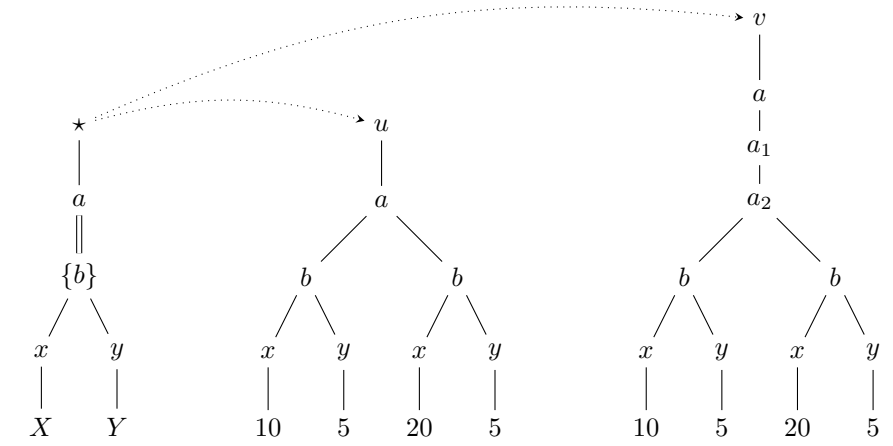
labeled with tags from a set $\mathcal{T}$, and leaves can be labeled with tags $\mathcal{T}$, data values $\mathcal{V}$, services $\overline{\mathcal{F}} = \{!\alpha, ?\alpha, \alpha \mid \alpha \in \mathcal{F}\}$ where $\mathcal{F}$ is the set of services. The symbol $!\alpha$ indicates that service α is waiting to be called, $?\alpha$ that it is currently being called, and α that it has been completed. We will consider that all these sets are disjoint, that $\mathcal{T}$ and $\mathcal{F}$ are finite sets, but that $\mathcal{V}$ can be infinite.

Within this setting, a *forest* is a set of labeled trees, often expressed as a disjoint union $F = \uplus_i T_i$, and T can also be used as a synonym for forest $\{T\}$. A *subtree* of a tree T is a tree rooted at some internal node n of T that contains all successors and edges of T starting from node n. All trees are assumed to be reduced, i.e., they do not contain isomorphic subtrees rooted at the same node.

DAXML documents are semi-structured data, that can be queried. Several query mechanisms have been proposed in the literature (Xquery, Xpath, tree patterns, etc.). In the sequel, we will use tree patterns for simple boolean queries, and build on these patterns to define queries returning semi-structured data. A *tree pattern* is a tuple $G = (M, H, c, \mu_M, \mu_H)$ in which (M, H, μ_M) is a tree labeled with elements of $\mathcal{T} \uplus \mathcal{V} \uplus \mathcal{F}$ plus a wildcard $\star$ (or possibly with variables from a finite disjoint set $\mathcal{X}$ at the leaves), c is a node of M called the *constructor node*, and μ_H is a function that labels edges as child edges or descendant edges. A tree pattern G *holds* at tree T, written $T \models G$, when there is at least one mapping h from nodes of G to nodes of T, such that h respects labeling ($\mu_M(n) = \mu(h(n))$ whenever $\mu_M(n) \neq \star$), child and descendant relations (the image by h of two nodes (n, n') such that $\mu_H(n, n') = \mathit{child}$ is an edge of T, and the image by h of two nodes (n, n') such that $\mu_H(n, n') = \mathit{descendant}$ are in the closure of the edge relation), and such that nodes labeled with variables are sent onto nodes with data values in T. This way, a matching h assigns values to the variables of a tree pattern. A *pattern* is a tuple $\mathbf{G} = (\{G_1, \ldots, G_k\}, \mathit{cond})$, where $G_1, \ldots G_k$ are tree patterns, and *cond* is a boolean condition on the values taken by variables in $G_1, \ldots, G_k$. We will say that $\mathbf{G}$ holds in a forest F, denoted $F \models \mathbf{G}$, iff there exists a matching for each tree pattern to a tree of F such that the variable assignment satisfies condition *cond*. Patterns only indicate whether a given shape appears in a forest, and are hence boolean queries returning `yes/no` answers. They will in particular be used to guard against service calls and returns, thus a pattern may be called a *guard*. We now detail how to extract data from documents.

An *expansion* of a pattern $\mathbf{G}$ is a forest F such that for every tree pattern G of $\mathbf{G}$, there is a tree T in F such that G holds in T, and G does not hold in subtrees of T. Note that there can be infinitely many expansions of a pattern (essentially because of wildcards and descendant edges). Expansions will be used to define forests accepted as parameters of a service, or returned as results. The set of all expansions of $\mathbf{G}$ is denoted $[\mathbf{G}]$. Examples of expansions are shown on Fig. 1, we refer interested readers to Appendix A for a more formal definition.

A *query* is a pair $Q = (\mathbf{B}, \mathbf{H})$ such that $\mathbf{B}$ and $\mathbf{H}$ are patterns. $\mathbf{B}$ is called the *body* of the query and $\mathbf{H}$ the *head*. In addition, $\mathbf{H}$ does not contain descendant edges nor wildcards ($\star$). The semi-structured document (forest) returned by a query on a forest F is an expansion of $\mathbf{H}$ that takes as valuations the valuations found from all possible matchings of $\mathbf{B}$ in F. We denote this forest by $Q(F)$. Intuitively, $\mathbf{B}$ is used to collect data from a forest, and $\mathbf{H}$ to return them as a semi-structured document. Queries are used to define services in DAXML. Each peer in an AXML system will distinguish between internal services, invoked



(a) Tree pattern. (b) Two possible expansions.

Figure 1: Two different expansions of a tree pattern, for the set of valuations $V = \{v_1, v_2\}$, with $v_1(X) = 10$, $v_2(X) = 20$, $v_1(Y) = v_2(Y) = 5$. The dotted arrows indicate how a wildcard can be turned into an arbitrary tag, while the descendant edge is expanded by 0 or 2 nodes (a_1 and a_2).

locally to query local documents, and external services, which are only known as a set of requirements interfaces. We will denote by $\mathcal{F} = \mathcal{F}_{int} \uplus \mathcal{F}_{ext}$ the set of all services, and differentiate internal services $\mathcal{F}_{int}$ and external ones $\mathcal{F}_{ext}$.

An *internal service* (or function) is a tuple $f = (\mathbf{G}^c, Q^c, \{(\mathbf{G}^r_i, Q^r_i)\}_{i\in\{1,\ldots,K_f\}})$. An internal service of a peer p can be called when its *call guard* $\mathbf{G}^c$ holds on local document owned by p. Calling it results in application of the call query Q^c to this document. The result of the query is kept temporarily in memory, in a tree called the *workspace* of the call. The service returns when one of its *return guards* $\mathbf{G}^r_i$ holds on the workspace, and the returned value is computed by application of Q^r_i to the workspace; service processing is described more precisely in the rest of this section. An *external service* (or interface) is a pair $I = (\mathbf{P}^c, \{\mathbf{P}^r_i\}_{i\in\{1,\ldots,K_I\}})$, where $\mathbf{P}^c$ and each $\mathbf{P}^r_i$ are patterns. Intuitively, $\mathbf{P}^c$ describes possible values that a service implementing I should accept, and $\{\mathbf{P}^r_i\}_{i\in\{1,\ldots,K_I\}}$ the set of expected legal returns. External services can be implemented by internal services provided by another peer, or left unimplemented. We will say that an internal service f *implements* an external service I, and write $f \Vdash I$ if all expansions of I can be accepted as input of f, and if all values returned by f (the expansions of each $\mathbf{H}^r_i$ in return queries of f) satisfy one of the return patterns in $\{\mathbf{P}^r_i\}_{i\in\{1,\ldots,K_I\}}$. The property $f \Vdash I$ is decidable when variables range over finite domains [3].

Proposition 2.1 ([3]). *Let I be an interface which variables range over finite domains, and $f = (\mathbf{G}^c, Q^c, \{(\mathbf{G}^r_i, Q^r_i)\}_{i\in\{1,\ldots,K_f\}})$ be a service with variables in each $\mathbf{H}^r_i, i \in \{1, \ldots, K_f\}$, ranging over finite domains. Then it is decidable whether $f \Vdash I$.*

This implementation relation is purely static: even if all values that can be returned by internal services meet some requirements, return queries are guarded, and hence nothing guarantees that an internal service will ever return

a value. In general, it is undecidable whether some guard can be eventually satisfied within a DAXML system. However, we will show later that we can impose restrictions on DAXML to ensure decidability of services termination.

2.2 DAXML Schemas and Instances

The dynamics of DAXML models is seen as a sequence of service calls and returns. We first formalize *DAXML schemas* to describe a system, i.e., a set of peers, each of them "owning" local data (DAXML documents) and a set of internal or external services. Each peer accesses only its data, and invokes its local services, or distant services offered by another peer via a call to one of its external services.

Definition 2.1 (DAXML schema)**.** A *DAXML schema* (*schema* for short) is a tuple $S = (\mathcal{P}, \mathcal{F}_{int}, \mathcal{F}_{ext}, \nu, \gamma)$, where

- $\mathcal{P}$ is a finite set of *peers*, denoted $p, q, \ldots$;
- $\mathcal{F}_{int}$ is a finite set of internal services *(functions)*, denoted $f, g, \ldots$; and $\mathcal{F}_{ext}$ is a finite disjoint set of external services *(interfaces)*, denoted $I, J, \ldots$;
- $\nu : \mathcal{F}_{int} \uplus \mathcal{F}_{ext} \to \mathcal{P}$ is the *localization map*, it localizes each internal or external service on a peer;
- $\gamma : \mathcal{F}_{ext} \to \mathcal{F}_{int}$ is a partial function, called *implementation map*, that maps some interfaces to functions, with the additional constraints that for all $I \in \mathrm{dom}(\gamma)$, $\nu(I) \neq \nu(\gamma(I))$ and $\gamma(I) \not\models I$.

If $\mathrm{dom}(\gamma) = \mathcal{F}_{ext}$, the schema S is said to be *closed*. When $f = \gamma(I)$, we say that f *implements* interface I in S.

DAXML schemas define data and services distributed over a set of peers. We can easily compose two schemas S_1 and S_2, by doing the disjoint union of their documents and services. In addition to this union, we parameterize composition by a partial function called a *pairing map* that pairs external services of a schema with internal services located on different peers that implement them. Let $\xi \subseteq \mathcal{F}_{ext_1} \cup \mathcal{F}_{ext_2} \to \mathcal{F}_{int_1} \cup \mathcal{F}_{int_2}$ be a pairing map between S_1 and S_2, then we necessarily have that, for every interface such that $\xi(I)$ is defined, $\xi(I) \not\models I$ and $\nu(\xi(I)) \neq \nu(I)$. The *composition* $S^1 \parallel_\xi S^2$ is a schema $S = \big(\mathcal{P}^1 \uplus \mathcal{P}^2, \mathcal{F}^1_{int} \uplus \mathcal{F}^2_{int}, \mathcal{F}^1_{ext} \uplus \mathcal{F}^2_{ext}, \nu^1 \uplus \nu^2, \gamma\big)$, where $\gamma = \gamma^1 \uplus \gamma^2 \uplus \xi$. We furthermore require that $\xi(\mathcal{F}^1_{ext}) \subseteq \mathcal{F}^2_{int}$ and $\xi(\mathcal{F}^2_{ext}) \subseteq \mathcal{F}^1_{int}$. We can also define the union $\xi = \xi_1 \cup \xi_2$ of two pairing maps ξ_1, ξ_2 such that $\mathrm{dom}(\xi_1) \cap \mathrm{dom}(\xi_2) = \emptyset$, by $\xi(I) = \xi_1(I)$ if $I \in \mathrm{dom}(\xi_1)$ and $\xi(I) = \xi_2(I)$ if $I \in \mathrm{dom}(\xi_2)$.

Note that services may return trees that embed new service calls. Hence, DAXML schemas can encode recursive behaviors. Most of undecidable issues in DAXML occur due to this recursion. It is however possible to trace dependencies between service calls to detect if such recursion can occur. The *call graph* of a schema S is an oriented graph $CG_S = (\mathcal{F} = \mathcal{F}_{int} \cup \mathcal{F}_{ext}, EC)$ where $(f, g) \in EC$ if:

- $f \in \mathcal{F}_{int}$, $g \in \mathcal{F}$ and there exists a node labeled with $!g$ in a head of a call or return query of f, or
- $f \in \mathcal{F}_{ext}$, $g \in \mathcal{F}$, and there exists a node labeled with $!g$ in one of the return patterns of f, or
- g implements f in S.

A DAXML schema is called *recursive* if its call graph contains cycles.

The "state" of a schema at some time t is defined as follows. Each peer owns local data and memorizes the status of local or distant services invocations (that may eventually return a value). This memorized information is represented as DAXML trees: some of them store the current data of a peer, the others contain "active" computations and are called *workspaces* of a service. A workspace is simply a tree that is created by the owner of the service when it is called, modified by its execution, and deleted when the service returns. Hence, the computation of services is done within a delimited area for each invocation. As several instances of the same service might be running at the same moment, workspaces are connected to their calling node. This is formalized by the notion of *DAXML instance*.

Definition 2.2 (DAXML instance)**.** A *DAXML instance* (*instance* for short) over a schema $S = (\mathcal{P}, \mathcal{F}_{int}, \mathcal{F}_{ext}, \nu, \gamma)$ is a tuple $D = (F, eval, \ell)$, where

- F is a forest. We denote by $F_a \subseteq F$ the subset of trees from F whose root is labeled with $a_\alpha \in \mathcal{T}$ for some $\alpha \in \mathcal{F}_{int} \cup \mathrm{dom}(\gamma)$ (i.e., α is either a function or an implemented interface), and let N be the set of nodes of F labeled with $?\alpha$ for some $\alpha \in \mathcal{F}_{int} \cup \mathrm{dom}(\gamma)$, called *active* nodes;
- $eval : N \to F_a$ is a bijection which maps an active node n labeled with $?\alpha$ to a tree of F whose root node is labeled with a_α, this tree $eval(n)$ is called the *workspace* of n; in addition, if α is an interface in $\mathrm{dom}(\gamma)$, then the root node of $eval(n)$ has a child labeled with f, $?f$ or $!f$, with $f = \gamma(\alpha)$;
- $\ell : F \to \mathcal{P}$ maps every tree of F to a peer, such that for all nodes n (in the following, $T(n)$ denotes the tree containing n):
  - if n is labeled with $?f$, $f \in \mathcal{F}_{int}$, then $\ell(T(n)) = \ell(eval(n)) = \nu(f)$ [for internal functions, the workspace is created on the peer on which the function is localized];
  - if n is labeled with $?I$, $I \in \mathrm{dom}(\gamma)$, then $\ell(eval(n)) = \nu(\gamma(I)) \neq \nu(I) = \ell(T(n))$ [for implemented interfaces, the workspace is created on the peer on which the function implementing the interface is localized, which has to be different from the one making the call, because of the definition of γ in Definition 2.1].

Intuitively, in an instance $D = (F, eval, \ell)$, F represents the memory used by each peer, *eval* maps service references to the workspaces that have been created to evaluate them, and ℓ localizes all trees on a peer. An instance may be represented as on Fig. 2: peers are delimited by dashed circles, they own a document which is a forest of trees, some of which being workspaces. The *eval* function is represented by the dotted arrows, and depict relation from a calling node to a workspace ($?I$ to tree a_I and node $?g$ to tree a_g).

The set of all instances D over a schema S is denoted $\mathcal{D}_S$, it contains all triples $(F, eval, \ell)$ which satisfy the conditions of Definition 2.2, for S fixed. We will say that two instances $D = (F, eval, \ell)$ and $D' = (F', eval', \ell')$ are *isomorphic* if there exists a bijective function $h : F \to F'$ that preserves labeling and edges, locality of trees (i.e., $\ell(n) = \ell'(h(n))$) and *eval* functions (i.e., $eval(n)$ is defined iff $eval'(h(n))$ is also defined, and then, $h(eval(n)) = eval'(h(n))$). The existence of such an isomorphism between two instances D and D' is denoted $D \equiv D'$.

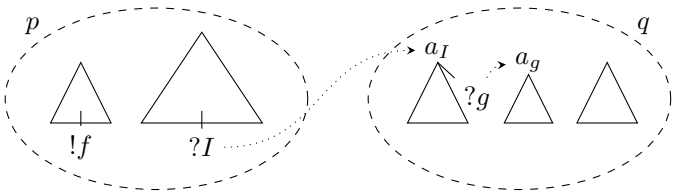


Figure 2: Illustration of a DAXML instance over a schema with two peers p and q, where peer p owns the service f and the interface I, implemented by service g on peer q.

The *union* of two instances $D \uplus D'$ is the disjoint union $(F \uplus F', eval \uplus eval', \ell \uplus \ell')$ of their forests, locality and evaluation functions. It is defined even when D and D' are not over the same schema. The *projection* of an instance on a subset of peers $\mathcal{P}' \subseteq \mathcal{P}$ is the restriction of the forest, *eval* and ℓ functions to trees located on a peer of $\mathcal{P}'$ and is denoted by $D^{\mathcal{P}'}$.

Remark 2.1. If $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$, it may not always be the case that $D = D^{\mathcal{P}_1} \uplus D^{\mathcal{P}_2}$. Indeed the *eval* function is not preserved by projection in general. For example, suppose the schema contains an interface I and a function f such that $f = \gamma(I)$, $\nu(f) \in \mathcal{P}_1$ and $\nu(I) \in \mathcal{P}_2$. If there is a running call to I in D, projecting puts the workspace in a forest of $D^{\mathcal{P}_1}$, while the node responsible for the ongoing call is in a forest of $D^{\mathcal{P}_2}$. Thus, the link between them provided by *eval* is destroyed during the projection and can not be restored by the union. In all other cases (interfaces implemented in the same set of peers, or no such ongoing interface call in D), the equality holds.

2.3 Events

The possible evolutions of a schema are described by *events*, which transform an instance D into an instance D'. The set of events is denoted $\mathcal{E}$, then, a move by event $e \in \mathcal{E}$ is written $D \xrightarrow{e} D'$. In the sequel, $D = (F, eval, \ell)$ and $D' = (F', eval', \ell')$ are two instances over the same schema $S = (\mathcal{P}, \mathcal{F}_{int}, \mathcal{F}_{ext}, \nu, \gamma)$. Moreover, $f = \left(\mathbf{G}^c, Q^c, \{(\mathbf{G}^r_i, Q^r_i)\}_{i \in \{1,\ldots,K_f\}}\right) \in \mathcal{F}_{int}$ denotes a function and $I = \left(\mathbf{P}^c, \{\mathbf{P}^r_i\}_{i \in \{1,\ldots,K_I\}}\right) \in \mathcal{F}_{ext}$ an interface.

Let us define the different event types: internal service calls and returns, external service calls and returns when an interface is implemented, and internal service calls and returns when an interface is not yet implemented. Despite the apparent complexity of the rules, one should only keep in mind that a move is a guarded action that represents either a call or a return, and simply transforms a document instance into another one. For the sake of clarity, we will not provide more formal details on DAXML semantics, and refer interested readers to Appendix B or [3].

An **internal service call** f^c occurs only when F contains a node n labeled with a label of the form $!f$, where f is a local service name, and when the call guard of f holds on the part of the document owned by the peer $\nu(f)$ that owns f. The result of this call is to change the label of n to $?f$ (indicating that service f is currently processed), to create a new tree (the workspace) computed

by application of the call query Q^c to the part of F owned by $\nu(f)$, and to link it to n by updating function *eval*.

The **return of an internal service** f^r is allowed when one of the return guards of the service holds on the workspace associated to the considered call. Then, the corresponding return query is applied to the workspace, and the computed result appended as a sibling of the calling node. Finally, the label of the calling node is changed to f, and the workspace is deleted.

A **call to an implemented external service** I_f^c is allowed when a node n labeled with $!I$ exists in F, and I is an external service owned by a peer p, that is implemented by a service f, owned by another peer q. Furthermore, a matching for the input pattern of I can be found in the siblings of node n (i.e., external service has correct parameters). The result of the call is to change the label of n to $?I$, create a new workspace on peer q that contains all parameters of the call plus a node labeled with $!f$, and connect it to the calling node by updating function *eval*. In other words, p sends to q a demand to execute f.

A **return from an implemented external service** I_f^r can occur only when the implementation f has returned its value. It consists in copying the results that have been appended to the original workspace as siblings of the calling node, and delete the workspace.

A **call to a non-implemented external service** I^c is allowed when a node n labeled with $!I$ exists in F, and I has no implementation. Furthermore, a matching for the input pattern of I can be found in the siblings of node n (that is this external service has correct parameters). This call simply changes the label of n to $?I$. As there is no implementation for I, the call is is supposed processed by a peer of the environment, and no workspace is created.

A **return from a non-implemented external service** $\overline{I^r}$ is not guarded, and consists in changing the label of a node from $?I$ to I, and appending as a sibling of this node an expansion of one of the return patterns of the external service I, chosen non-deterministically (descendant edges are expanded and variables are replaced by a value in their domain, see Appendix A for details).

In Section 4, we will also allow DAXML peers to receive calls from their environment, but for the moment, this possibility is ignored. We extend the move relation to sequences of events $\sigma = e_1 e_2 \dots e_i$, with $D \xrightarrow{\sigma} D'$ if and only if there exist instances $D_1, \dots, D_{i-1}$ such that $D \xrightarrow{e_1} D_1 \xrightarrow{e_2} \cdots \xrightarrow{e_i} D'$. If there is a (possibly empty) sequence σ such that $D \xrightarrow{\sigma} D'$, D' is *reachable* from D. The set of instances reachable from D is denoted $\mathcal{RD}(D)$.

3 Petri Net Representation of DAXML Behavior

Here we show how to represent the behavior of DAXML schemas with labeled Petri nets. We first recall basic notions on labeled Petri nets, and describe a simple representation. Then, we introduce an operation that *merges* two nets, while highlighting concurrency of events.

3.1 Basic Definitions

Let us start by a small reminder of (labeled) Petri nets. For more information on Petri nets and their properties, see for example [7].

Definition 3.1. A *labeled net* is a tuple $(\Sigma, \Theta, \Phi, \Lambda, \lambda)$ where Σ is the set of *places*, Θ is the set of *transitions*, $\Phi \subseteq (\Sigma \times \Theta) \cup (\Theta \times \Sigma)$ is the *flow relation* seen as a set of (directed) *arcs* between places and transitions, Λ is the set of *labels*, and $\lambda : \Theta \to \Lambda$ is the *labeling function*. A *subnet* of $(\Sigma, \Theta, \Phi, \Lambda, \lambda)$ is a labeled net $(\Sigma', \Theta', \Phi', \Lambda, \lambda')$, where $\Sigma' \subseteq \Sigma$, $\Theta' \subseteq \Theta$, $\Phi' \subseteq \big((\Sigma' \times \Theta') \cup (\Theta' \times \Sigma')\big) \cap \Phi$, and $\lambda' = \lambda_{|\Theta'}$ is the restriction of function λ to the domain Θ'.

The "state" of a labeled net is represented by a *marking*, i.e., a mapping $m : \Sigma \to \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers. For $s \in \Sigma$, the value $m(s)$ is the number of *tokens* in place s.

Definition 3.2 (LPN)**.** A *labeled Petri net (LPN)* is a tuple $(\Sigma, \Theta, \Phi, \Lambda, \lambda, m_0)$ where $(\Sigma, \Theta, \Phi, \Lambda, \lambda)$ is a labeled net and $m_0 : \Sigma \to \mathbb{N}$ is the *initial marking*.

If $N = (\Sigma, \Theta, \Phi, \Lambda, \lambda)$ is a labeled net and m a marking, we also denote by (N, m) the LPN $(\Sigma, \Theta, \Phi, \Lambda, \lambda, m)$. A labeled net, or an LPN, is *finite* if the sets of places and transitions are finite.

Consider an LPN $P = (\Sigma, \Theta, \Phi, \Lambda, \lambda, m)$. A transition t is *enabled* in P if all the places preceeding it contain at least one token, i.e., $t \in \Theta$ is enabled if and only if for all $s \in \Sigma$ such that $(s, t) \in \Phi$, $m(s) \geq 1$; this is denoted by $m[t\rangle$. If transition t is enabled, it may *fire* and generate a new marking m' by removing a token in each predecessor place of t, and adding a token in each successor place: this is denoted $m\,[t\rangle\,m'$, where the new marking m' is such that $m'(s) = m(s) - 1$ for all $s \in \Theta$ such that $(s, t) \in \Phi$, $m'(s) = m(s) + 1$ for all $s \in \Theta$ such that $(t, s) \in \Phi$, and $m'(s) = m(s)$ otherwise (including the case where $(s, t), (t, s) \in \Phi$).

A sequence of transitions wich can be fired one after the other is called an *occurrence sequence* (denoted σ in general), we naturally extend the firing notation to (potentially empty) occurrence sequences by $m\,[\sigma\rangle\,m'$. Then, we say that m' is a *reachable* marking from m, and we denote $\mathcal{RM}(m)$ the set of markings which can be reached from m. The LPN P is *bounded* if there exists $k \in \mathbb{N}$ such that for all markings $m' \in \mathcal{RM}(m)$, for all places s of P, $m'(s) < k$. Equivalently, a finite LPN P is bounded if and only if $\mathcal{RM}(m)$ is finite.

Definition 3.3 (bisimulation equivalent LPNs)**.** Consider two LPNs $P = (N, m_0)$ and $P' = (N', m'_0)$, and a relation $\mathcal{R} \subseteq \mathcal{RM}(m_0) \times \mathcal{RM}(m'_0)$ between markings of P and P'. The LPNs P and P' are *bisimulation equivalent* with respect to $\mathcal{R}$ (denoted $P \equiv_{\mathcal{R}} P'$) if the reachability graphs of their markings are bisimilar [13, 15], i.e., if:

(i) $(m_0, m'_0) \in \mathcal{R}$;

(ii) for any $m \in \mathcal{RM}(m_0)$, $m' \in \mathcal{RM}(m'_0)$ such that $(m, m') \in \mathcal{R}$:

- for any transition t and marking $m_1 \in \mathcal{RM}(m_0)$ such that $m\,[t\rangle\,m_1$, there exist a transition t' and a marking $m'_1 \in \mathcal{RM}(m'_0)$ such that $m'\,[t'\rangle\,m'_1$, $\lambda(t) = \lambda'(t')$, and $(m_1, m'_1) \in \mathcal{R}$;
- for any transition t' and marking $m'_1 \in \mathcal{RM}(m'_0)$ such that $m'[t'\rangle m'_1$, there exist a transition t and a marking $m_1 \in \mathcal{RM}(m_0)$ such that $m\,[t\rangle\,m_1$, $\lambda(t) = \lambda'(t')$, and $(m_1, m'_1) \in \mathcal{R}$.

In the rest of the paper, we use LPNs to define the behavior of systems described with DAXML. Intuitively, the places of such Petri nets will represent the status of all actors in the system, that is the memorized data and the state of all started processes.

3.2 Basic Petri Net Representation

The behavior of a DAXML schema S, starting from the initial instance D_0, can be conveniently described by the LPN $P_{S,D_0} = (\Sigma, \Theta, \Phi, \Lambda, \lambda, m_0)$, where:

- $\Sigma = \mathcal{RD}(D_0)$, the places are the instances reachable from D_0;
- $\Lambda = \mathcal{E}$, the labels are the events, which can be function or interface calls or returns;
- $\Theta \subseteq \Sigma \times \Sigma$, Φ, and λ are such that for any $D, D' \in \mathcal{RD}(D_0)$, if there exists an event $e \in \mathcal{E}$ with $D \xrightarrow{e} D'$, then $t = (D, D') \in \Theta$, $(D, t), (t, D') \in \Phi$, and $\lambda(t) = e$;
- $m_0(D_0) = 1$, and for all $D \in \mathcal{RD}(D_0) \setminus D_0$, $m_0(D) = 0$.

Let N_{S,D_0} be the labeled net such that $P_{S,D_0} = (N_{S,D_0}, m_0)$. Note that the forest of the initial instance D_0 will only contain service requests $!\alpha$ and no running or completed calls, so that the domain of *eval* is initially empty. Figure 3 illustrates this construction. As usual, places are represented by circles, transitions by rectangles and their labels, and the markings by tokens inside the places. In this figure, each place represents a document. Transitions are labeled with events from the set $\mathcal{E}$, which represent respectively calls to services, or returns from invocations. Note from this drawing that all actions are not necessarily allowed from any state. For instance, calling service g from document D_0 prevents calling service f afterwards.

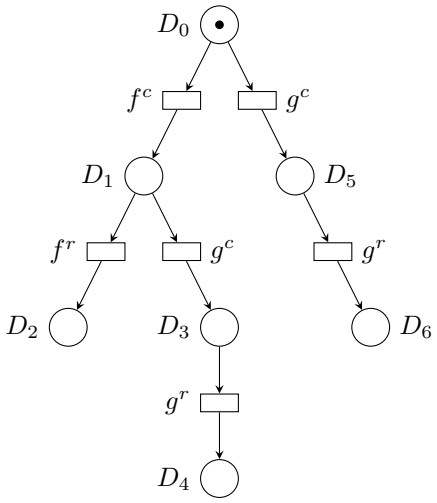


Figure 3: LPN illustrating the behavior of a simple DAXML schema.

The tokens (here, there is only one, initially in place D_0) indicate the current state of the modeled DAXML system. This mapping from tokens to instances is formalized in the following definition.

Definition 3.4. The instance *generated* by a marking m is the disjoint union of instances $D(m) = \biguplus_{D \in \Sigma} \biguplus_{k=1}^{m(D)} D_D^k$, where for all D, for all k, D_D^k is an instance isomorphic to D (one for each token in D).

Basically, $D(m)$ is the union of all the places containing tokens, with as many copies as there are tokens in each place. It represents the current global "state" of the DAXML schema.

Note that in general, an LPN P_{S,D_0} is not deterministic. Indeed, events are simply labels: call event abstracts away the exact reference (node of the document) that leads to the call, a return event abstracts away the service calls that is completed and the returned parameters, etc. Hence different places (representing different instances) can be reached from a single place by several transitions having the same label. It is not necessarily confluent either, in the sense that starting from a given place, following two transitions labeled with e_1 and e_2 respectively may not lead to the same place as if we had followed two transitions labeled with e_2 and e_1. This is mainly because services may reuse the nodes returned by other services, thus creating different workspaces and return instances, which depend on the order of their call and return.

Also note that such an LPN is not necessarily finite. Indeed, the transitions labeled with $\overline{I^r}$ depict non-implemented services returns, that add returned values (an expansion of a pattern) to an existing document. If an external service has a return pattern with an infinite number of expansions, this automatically leads to infinite branching in the LPN, so that it may not be constructed or analysed automatically. However, particular (realistic) restrictions on the model lead to interesting simplifications. If we assume finite domains for the variables, as well as query and interface patterns of bounded height (in particular this means that services always return trees of bounded height), it is sufficient to consider expansions of return patterns of bounded height when a non-implemented service returns. With this restriction, there is always a finite number of possible $\overline{I^r}$ transitions leaving a given instance. Besides, some occurrence sequences can be infinite due to services calling one another. However, if we assume that the call graph of a DAXML schema has no cycles, then a call to a service returns after a finite number of computation steps. As there is only a finite number of service requests ($!\alpha$) in an initial instance, and no unbounded recursion, all occurrence sequences are finite. If all these assumptions hold, then the LPN is finite and bounded; we call these the *finiteness hypotheses* and denote them $\mathfrak{F}$.

3.3 Merging Nets

The "internal" behavior of a schema is well described by such a labeled Petri net, but at this stage, it is no more than a labeled transition system, i.e., an interleaved behavior. In order to make concurrency visible, we split an instance into several documents. In that case, the behavior of one schema would be described by the union of several Petri nets acting in parallel, independently from each other. The only form of interaction comes from the fact that for internal service calls, the guards have to be checked among all documents located on the peer initiating the call, thus adding new arcs in the flow relation between some transitions (the function call or return) and places (the part of the document which satisfies the call or return guard). This is defined by a *merge* operator $\oplus$.

Notation. Let S^* be schema S where all guards evaluate to true, and $D \in \mathcal{D}_S$. The labeled net $N_{S,D}$ is a subnet of $N_{S^*,D} = (\Sigma \uplus \Sigma^*, \Theta \uplus \Theta^*, \Phi \uplus \Phi^*, \Lambda, \lambda^*)$. All extra data added (new places, transitions and flow relation) is marked with a star, and the labeling function is such that $\lambda = \lambda^*{}_{|\Theta}$.

Definition 3.5 (merge)**.** For a given schema S and two instances $D_1, D_2 \in \mathcal{D}_S$, define the labeled nets $N_{S,D_i} = (\Sigma_i, \Theta_i, \Phi_i, \Lambda_i, \lambda_i)$, for $i \in \{1,2\}$. Their *merge* is the labeled net $N_{S,D_1} \oplus N_{S,D_2} = (\Sigma'_1 \uplus \Sigma'_2, \Theta'_1 \uplus \Theta'_2, \Phi', \Lambda_1 \uplus \Lambda_2, \lambda)$. The elements of Σ'_i, Θ'_i, for $i \in \{1,2\}$, of Φ', and the function λ are constructed inductively as follows.

(*i*) $\Sigma_i \subseteq \Sigma'_i$ and $\Theta_i \subseteq \Theta'_i$, for $i \in \{1,2\}$; $\Phi_1, \Phi_2 \subseteq \Phi'$.

(*ii*) $\lambda(t) = \lambda_i(t)$ if $t \in \Theta_i$, for $i \in \{1,2\}$.

(*iii*) For all instances $D \in \Sigma'_i$, $D' \in \Sigma'^*_i$, $D_l \in \Sigma'_{j_l}$, $1 \leq l \leq k$ for some k, for each transition $t \in \Theta^*_i$, with $i, j_1, \ldots, j_k \in \{1,2\}$, if there exist a node n in a forest F of the instance D, a forest F_l in each D_l, and a function $f \in \mathcal{F}_{int}$, which are such that
- n is labeled with $!f$,
- $\lambda^*_i(t) = f^c$,
- $\nu(f) = \ell(F) = \ell(F_1) = \cdots = \ell(F_k)$,
- $(\cup_{l=1}^k F_l \cup F) \models \mathbf{G}^c$,
- for all $F' \subsetneq \cup_{l=1}^k F_l$, $(F' \cup F) \not\models \mathbf{G}^c$;

[in simple words, all the items above mean that the event $D \xrightarrow{f^c} D'$ is made possible by forests inside documents $D_1, \ldots, D_k$, and that each of these forests is necessary for the guard to hold;]
then we create a fresh transition $t' \in \Theta'_i$ labeled with $\lambda(t') = f^c$, we add $D' \in \Sigma'_i$, and add to the flow relation the arcs (D_l, t'), (t', D_l), (D, t'), $(t', D') \in \Phi'$, for $1 \leq l \leq k$.
We finally add all the newly reachable instances $\mathcal{RD}(D')$ to Σ'_i, as well as the corresponding transitions and arcs to Θ'_i and Φ' respectively.

Note that guards can be slightly more complex than presented in this paper. We kept them as simple as possible for readability reasons. However, implementing the whole complexity of guards as presented in [3] would not impose any change to our LPN model.

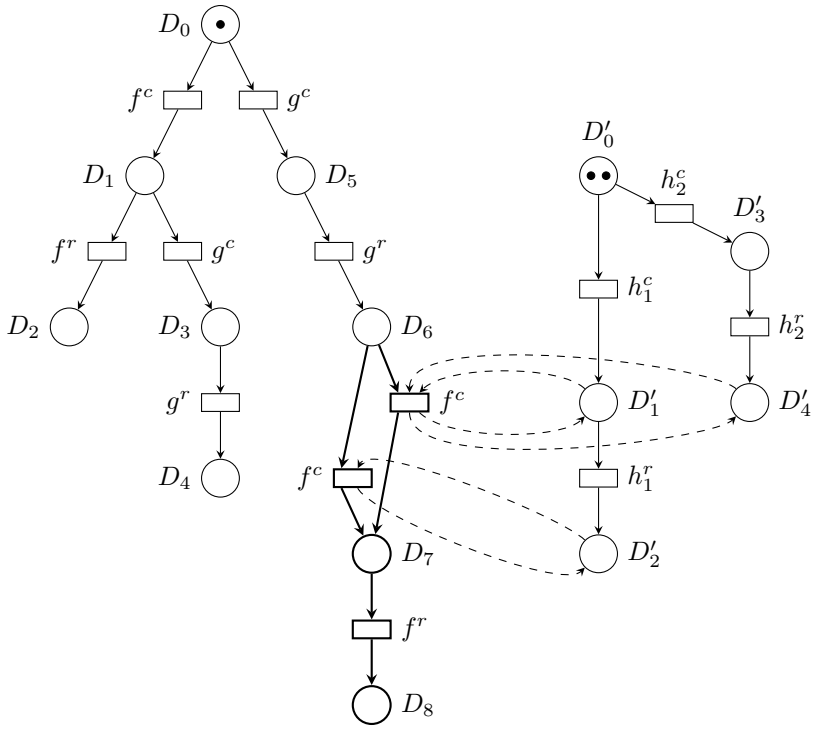
Figure 3.3 illustrates how merging can enhance the behavior of an LPN associated with a DAXML instance. It preserves the respective behaviors of both of the merged LPNs, adding new possibilities allowed by the presence of the documents of the other component.

The following proposition allows to use the notation $\bigoplus_{D \in \mathcal{D}} N_{S,D}$, for a set of instances $\mathcal{D}$ with more than 2 elements.

Proposition 3.1. *$\oplus$ is associative and commutative.*

Proof. Commutativity follows from the definition of $\oplus$ (symmetric roles of Σ_1 and Σ_2, etc.).

Associativity is less obvious. Take 3 instances D_1, D_2, D_3 over the same schema S, denote $N_i = N_{S,D_i}$ and $N_{ij} = N_i \oplus N_j$ for $i, j \in \{1,2,3\}$. We show that $N_{12} \oplus N_3 = N_1 \oplus N_{23}$. Let D be some place in $N_{12} \oplus N_3$. If D was already in N_1, N_2, or N_3, then D is in $N_1 \oplus N_{23}$. If D was added while merging N_1 and N_2, step *(iii)* of Definition 3.5 tells us that there are places in N_1 and N_2 which permitted its creation: these places are also present in N_1 and in N_{23}, so D will also be added in $N_1 \oplus N_{23}$. A similar reasoning, iterated once more, gives the same result if D is introduced when merging N_{12} and N_3. This works as well for transitions and flow relation, thus proving that $N_{12} \oplus N_3$ is a subnet



of $N_1 \oplus N_{23}$. Finally, the other direction is done similarly and we obtain the equality. □

So far, we have considered behavioral equivalence of nets (Definition 3.3). However, this equivalence is not sufficient to compare systems that manipulate documents. We need, in addition to behaviors, to ensure that the manipulated documents are isomorphic.

Definition 3.6 (equivalent LPNs)**.** Two LPNs P and P' whose places are DAXML instances are *equivalent*, denoted $P \equiv P'$, if they are bisimulation equivalent for the relation $\mathcal{R}_{\equiv}$ defined by $(m, m') \in \mathcal{R}_{\equiv}$ if and only if $D(m) \equiv D(m')$.

The following theorem is a first step towards "splitting" instances, to simplify the behavioral study. It states that given several instances, the merged LPN of these instances has the same behavior as the LPN of their union.

Theorem 3.2. *Given a schema S and a finite set K, consider $D = \biguplus_{k\in K} D_k \in \mathcal{D}_S$. Denote $P_{S,D} = (N_{S,D}, m_0)$ as usual, and $P'_{S,D} = \left(\bigoplus_{k\in K} N_{S,D_k}, m'_0\right)$, with $m'_0(D') = 1$ if $D' = D_k$ for some $k \in K$, $m'_0(D') = 0$ otherwise. Then, the LPNs $P_{S,D}$ and $P'_{S,D}$ are equivalent.*

Proof. Since $D = \biguplus_{k\in K} D_k$, it holds that $D(m_0) \equiv D(m'_0)$.

Consider some markings $m, m_1 \in \mathcal{RM}(m_0)$, $m' \in \mathcal{RM}(m'_0)$, and a transition t such that $D(m) \equiv D(m')$ and $m\,[t\rangle\, m_1$ in $P_{S,D}$. We show that there exist a marking m'_1 and a transition t' such that $m'\,[t'\rangle\, m'_1$ in $P'_{S,D}$, with $\lambda(t) = \lambda'(t')$ and $D(m) \equiv D(m')$.

If $\lambda(t) \notin \{f^c \mid f \in \mathcal{F}_{int}\}$, then by definition, the event is not guarded (case of interface returns), or its guard has to hold in the same tree as the node responsible for the call (interface calls), or it holds in a tree in the same instance as this node (function returns, the guard has to hold in the workspace, which is necessarily in the same instance as the caller node, to maintain the *eval* link). So, a transition t' labeled with $\lambda(t)$ can be fired in $P'_{S,D}$ without using step *(iii)* of Definition 3.5. Thus, there exists m'_1 such that $m' [t'\rangle m'_1$ in $P'_{S,D}$. Moreover, because $D(m) \equiv D(m')$, it is possible to fire the transition with the same result as in $P_{S,D}$, hence to choose t' and m'_1 such that $D(m_1) \equiv D(m'_1)$.

If $\lambda(t) = f^c$ for some $f \in \mathcal{F}_{int}$, then since $D(m) \equiv D(m')$ and the call guard of f is satisfied in $D(m)$, it is also satisfied using a forest of trees F from $D(m')$. Recall that $D = \biguplus_{k \in K} D_k$, let D_k be the instance which contains the tree from D containing the node responsible of the firing of t in $P_{S,D}$. If all trees of F belong to D_k, we reach the same conclusion as previously, a transition t' with the same label is enabled in $P'_{S,D}$ and can lead to an instance isomorphic to $D(m_1)$. Otherwise, because of step *(iii)* of Definition 3.5, a transition labeled with f^c should also be enabled in $P'_{S,D}$ because the trees of F can all be found in places with tokens, which by construction are predecessors of all the transitions labeled with f^c. It suffices to fire the one which gives the same result as in $P_{S,D}$, and with the tokens from the guard returning to their place we obtain a marking m'_1 and a transition t' such that $m' [t'\rangle m'_1$, $\lambda(t) = \lambda'(t')$, and $D(m_1) \equiv D(m'_1)$.

The converse direction can be solved with a similar case study over the enabled transitions labels, if there are $m \in \mathcal{RM}(m_0)$, $m', m'_1 \in \mathcal{RM}(m'_0)$ and t' such that $D(m) \equiv D(m')$ and $m' [t'\rangle m'_1$ in $P'_{S,D}$, then there exist $m_1 \in \mathcal{RM}(m_0)$ and t such that $m [\sigma\rangle m_1$ in $P_{S,D}$, with $\lambda(t) = \lambda'(t')$ and $D(m_1) \equiv D(m'_1)$. □

A first consequence of this result is that in order to reduce the number of places of an LPN $P_{S,D}$, which could grow exponentially in the number of service calls, one can merge all the LPNs obtained by splitting the instance D instead. This technique makes explicit the concurrency which can be found in the DAXML framework, and provides a better view over the sequences of related events than the simple interleaving found in $P_{S,D}$.

The following proposition shows that merging preserves equivalence of LPNs.

Proposition 3.3. *Given two labeled nets N_1 and N_2 and markings over these nets m_1 and m_2 such that $(N_1, m_1) \equiv (N_2, m_2)$, then $(N_1 \oplus N, m'_1) \equiv (N_2 \oplus N, m'_2)$, for any labeled net N and any markings m'_1, m'_2 such that for $i \in \{1, 2\}$, $m'_i(D) = m_i(D)$ if D is a place of N_i, $m'_1(D) = m'_2(D)$ if D is a place of N, and $m'_1(D) = m'_2(D) = 0$ otherwise.*

Proof. Initially, $D(m'_1) \equiv D(m'_2)$ because $D(m_1) \equiv D(m_2)$ (since $(N_1, m_1) \equiv (N_2, m_2)$), and the other places are either empty, or are shared between the two nets and contain the same number of tokens.

Then, reasoning by case-study over the label of the transitions that can be fired on the two LPNs, in the same way as it was done in the proof of Theorem 3.2 above, one gets the result. □

4 Compatibility between Schemas

In this section we define and study two notions of "behavioral" compatibility between two DAXML schemas, where the first schema provides a service for

which the other schema only knows the interface. This notion of compatibility is important for a distributed system, since it allows for component-based design and orchestration of services. It is stronger than the implementation relation [3], which is only a kind of "static" compatibility notion, in the sense that it does not take into account the possible internal behaviors of both schemas.

First, we introduce definitions which are useful to model the treatment of incoming calls. Then, we define weak and strong compatibility, and show that with the assumptions $\mathfrak{F}$ ensuring a finite LPN representation, these properties are decidable (Theorem 4.2).

4.1 Environment Calls and Fairness

We introduce two more events, corresponding to the reception of an external call from the environment, and to its return once the service is completed. Note that they are complementary to non-implemented interface calls and returns: they create (and delete) the workspace on the proper peer, for a non-implemented interface on a peer outside the current schema. Consider a schema $S = (\mathcal{P}, \mathcal{F}_{int}, \mathcal{F}_{ext}, \nu, \gamma)$ and an internal service $f = (\mathbf{G}^c, (\mathbf{B}^c, \mathbf{H}^c), \{(\mathbf{G}^r_i, Q^r_i)\}_{i\in\{1,\ldots,K_f\}}) \in \mathcal{F}_{int}$.

A **reception of environment call** to a local service f is an event $\overline{E^c_f}$. Such an environment call can occur at any time, and simply consists in adding to the existing document a new tree (workspace) containing a reference to the called service ($!f$) and the parameters of the call (a forest F). We furthermore suppose that the call parameters are compatible with the called services, that is, F is an arbitrary element of $\left[\mathbf{B}^c\right]$. Recall that $\left[\mathbf{B}^c\right]$ is the set of expansions (Section 2, or Definition A.3), of $\mathbf{B}^c$, i.e., the set of admitted parameters of f.

A **return to the environment** E^r_f can occur only when the called service has completed its task. The effect of this return is to delete the whole tree that was originally created for this service call.

We give a formal semantics of environment calls and returns in Appendix C. Allowing environment calls/returns slightly modifies the definition of instances, as in this settings, the workspace delimiting the computation of a service f is not related to any node by function *eval*. If we want to close the schema, that is compose it with another schema that makes explicit all interfaces that refer to f, the joint behavior of both descriptions adds the appropriate links in *eval*, merge I^c and $\overline{E^c_f}$ into I^c_f, and $\overline{I^r}$ and E^r_f into I^r_f for every I such that $\gamma(I) = f$.

These events easily integrate into our LPN representation.

Notation. For a forest F, we denote by $T_f(F)$ a tree whose root is labeled with a_{E_f}, such that the children of the root are all the trees from F and a node labeled with $!f$. It represents the workspace W defined above for the reception of an environment call associated with f, where the parameters consist of the trees of F. Furthermore, let $D_f(F)$ be the instance $(T_f(F), eval, \ell)$ where *eval* is undefined, and $\ell(T_f(F)) = \nu(f)$.

In the labeled net $N_{S,D_f(F)}$, denote by $\mathcal{D}^r_f(F)$ all the places representing a successful computation of the environment call, i.e., $D \in \mathcal{D}^r_f(F)$ if it respects all conditions described above in the return to environment event.

Define the labeled net $N^f_{S,D}$ as follows, where D is an instance over S.

1. First, consider $N_1 = N_{S,D} \oplus \bigoplus_{F \in [\mathbf{B}^c]} N_{S,D_f(F)}$. The nets $N_{S,D_f(F)}$ represent the processing of an environment call to f with parameters F.
2. Add an additional place, named s. For all $F \in \left[\mathbf{B}^c\right]$, add a transition in N_1 labeled with $\overline{E_f^c}$, whose unique predecessor is s and whose successors are s and $D_f(F)$. This allows to arbitrarily add tokens (provided there was initially a token in s) to the place $D_f(F)$, representing the reception of a new environment call. Denote N_2 the resulting net.
3. For all $F \in \left[\mathbf{B}^c\right]$, for all places D in $\mathcal{D}_f^r(F)$, add a transition in N_2 labeled with E_f^r, with no successors and the place D as unique predecessor. This removes a token when the environment call is complete, representing the return to environment. The resulting net is $N_{S,D}^f$.

It is clear that the LPN $P_{S,D}^f = (N_{S,D}^f, m')$, where $m'_0(D) = m'_0(s) = 1$ and m'_0 equals 0 for any other place, represents the behavior of the DAXML schema S accepting environment calls to the local function f, starting from the instance D. Indeed, $m\ [t\rangle\ m'$ in $P_{S,D}^f$ if and only if schema S allows a move $D(m) \xrightarrow{\lambda(t)} D(m')$. Figure 4 illustrates such an LPN by showing the extra place and transitions.

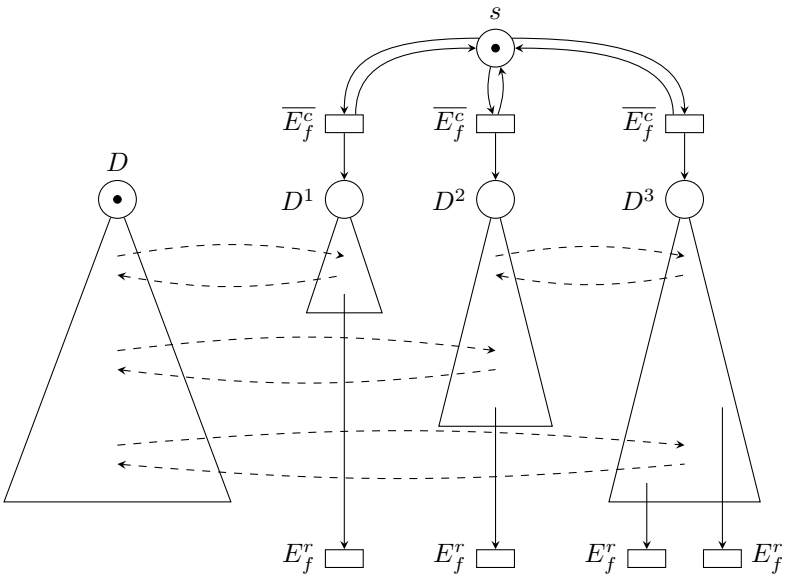


Figure 4: LPN describing the behavior of a schema receiving environment calls to a function f. To simplify, here $\left[\mathbf{B}^c\right] = \left\{F^1, F^2, F^3\right\}$ and $D^i = D_f(F^i)$, for $1 \leq i \leq 3$. The triangles abstract the LPNs initiated by the instance on top of them, the dashed arrows represent the arcs introduced by the mergings.

These labeled nets can be easily generalized to any set $\mathcal{F}_{env} \subseteq \mathcal{F}_{int}$ of internal functions which can be triggered by an environment call. In this case, the construction leads to the labeled net $N_{S,D}^{\mathcal{F}_{env}}$. In particular, $N_{S,D}^{\mathcal{F}_{int}}$ is the labeled net accepting any environment call. We use similar notation for the LPNs. The set $\mathcal{F}_{env}$ can be seen as a "prototype" of the public functionalities that a schema

provides to the environment. It is fixed when the schema is defined, and can thus be seen as an additional element of a schema.

With the finiteness hypotheses $\mathfrak{F}$, the sets $\left[\mathbf{B}^c\right]$ are finite for all functions f, hence only a finite number of nets $N_{S,D_f(F)}$ are merged to the initial net $N_{S,D}$. Each of these merged nets are finite, so $N_{S,D}^{\mathcal{F}_{env}}$ is also finite, for any (finite) set $\mathcal{F}_{env} \subseteq \mathcal{F}_{int}$. The main difference with the initial LPN $P_{S,D}$ is that $P_{S,D}^{\mathcal{F}_{env}}$ is not bounded: since the transitions labeled with $\overline{E_f^c}$, for some $f \in \mathcal{F}_{env}$, can occur at any time, there can be an arbitrary number of tokens on any of the $D_f(F)$ places. The structure used to model the behavior of an extended schema is finite, but it admits infinite behaviors. As a consequence, a token might remain in a place forever even in infinite occurrence sequences of $P_{S,D}^{\mathcal{F}_{env}}$, because some other transitions may keep firing.

To deal with these infinite occurrence sequences, we introduce a fairness notion [4, 8], and more particularly the *weak fairness* assumption [4, 10] (sometimes also called *finite delay property*).

Definition 4.1 (Weak fairness)**.** For an LPN with set of transitions Θ and initial marking m_0, an infinite occurrence sequence $\sigma = t_1 t_2 \cdots$ is *weakly fair* if for all $t \in \Theta$, if there exists $i \in \mathbb{N}$ such that for all $j \geq i$, $m_0 \left[t_1 \cdots t_j\right\rangle m$ and $m[t\rangle$, then there exist infinitely many positions $k \in \mathbb{N}$ such that $t_k = t$.

In other words, if starting from a given time a transition is continuously enabled, then weak fairness guarantees that this transition is fired infinitely often. In particular, this fairness assumption prevents an LPN $P_{S,D}^{\mathcal{F}_{env}}$ from continuously firing one of the transitions labeled with $\overline{E_f^c}$, unless no other transition is enabled, thus ensuring that requests progress.

4.2 Decidability of Compatibility

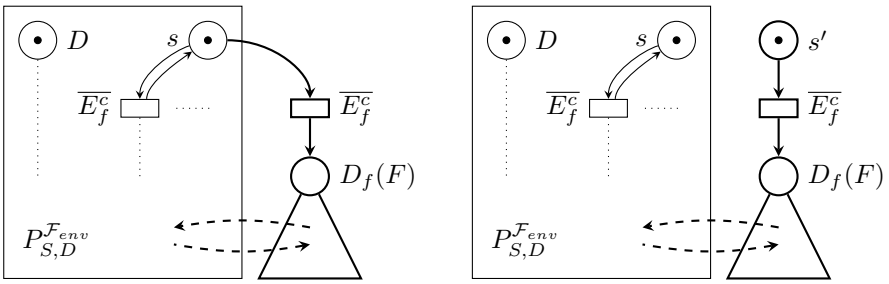
Intuitively, a compatibility notion should ensure reliable termination of internal and external services. While the implementation relation guarantees that the parameters and returns of an interface call have a correct "type", compatibility should guarantee that an environment call, sent through an interface, will always be treated and completed.

The following definition builds the labeled nets and LPNs that we use to define compatibility. The basic idea is to start from a net $N_{S,D}^{\mathcal{F}_{env}}$ and add places and transitions to "select" a single request, to check whether it is processed. One LPN allows further environment calls to be received once the selected request has been initiated, while the other does not. Figure 5 shows the general structure of the LPNs we are defining.

Definition 4.2. For a schema S, an instance $D \in \mathcal{D}_S$, a function $f \in \mathcal{F}_{env} \subseteq \mathcal{F}_{int}$ and a forest $F \in \left[\mathbf{B}^c\right]$, where $\mathbf{B}^c$ is the body of the call query of f, we define the labeled nets $N_{S,D}^{\mathcal{F}_{env}}(F)$ and $N_{S,D}^{\mathcal{F}_{env}}[F]$. For both nets, we start by constructing $N_{S,D}^{\mathcal{F}_{env}} \oplus N_{S,D_f(F)}$. That is, we add to $N_{S,D}^{\mathcal{F}_{env}}$ a copy of the net associated to the execution of f with parameters F. From now on, $D_f(F)$ (abusively) denotes the merged copy of the existing place $D_f(F)$. Add a new transition t labeled with $\overline{E_f^c}$.

- In $N^{\mathcal{F}_{env}}_{S,D}(F)$, add arcs (s,t) and $(t, D_f(F))$ to the flow relation: the token inside s is used to select an environment call associated with f, preventing any further incoming calls to be received.
- In $N^{\mathcal{F}_{env}}_{S,D}[F]$, add a new place s' and two arcs (s',t) and $(t, D_f(F))$: the copied places have no effect on the rest of the net.

Moreover, define the LPNs $P^{\mathcal{F}_{env}}_{S,D}(F) = (N^{\mathcal{F}_{env}}_{S,D}(F), m''_0)$ and $P^{\mathcal{F}_{env}}_{S,D}[F] = (N^{\mathcal{F}_{env}}_{S,D}[F], m''_0)$, with $m''_0(D) = m''_0(s) = 1$ (and $m''_0(s') = 1$ in $P_{S,D}[F]$), and $m''_0(D') = 0$ for all other places D'.



(a) The LPN $P^{\mathcal{F}_{env}}_{S,D}(F)$. (b) The LPN $P^{\mathcal{F}_{env}}_{S,D}[F]$.

Figure 5: Example of LPNs $P^{\mathcal{F}_{env}}_{S,D}(F)$ (left) and $P^{\mathcal{F}_{env}}_{S,D}[F]$ (right). The rectangle symbolizes the LPN $P^{\mathcal{F}_{env}}_{S,D}$, as represented on Fig. 4. New elements are thickened, they are used to select one environment call and process it separately from the rest. Some interactions due to the merge may still appear, they are indicated by dashed arrows.

We introduce two versions of compatibility, a weak and a strong one, depending on whether receptions of environment calls are allowed during the treatment of an ongoing environment call.

Definition 4.3 ((Strong) compatibility)**.** For two schemas S^1 and S^2 with $I \in \mathcal{F}^1_{ext}$, $f \in \mathcal{F}_{env} \subseteq \mathcal{F}^2_{int}$, and $f \not\models I$, then S^1 and S^2 are *(strongly)* (I,f)*-compatible* for an initial instance D (denoted $S^1 \; {}_I\!\lhd^f_D \; S^2$) iff after any sequence of receptions of environment calls from $\mathcal{F}_{env}$, every event $\overline{E^c_f}$ in $P^{\mathcal{F}_{env}}_{S,D}$ induced by a call to I is eventually followed by a corresponding response E^r_f.

Formally, let $\mathbf{P}^c$ represent the call pattern of I. Then, $S^1 \; {}_I\!\lhd^f_D \; S^2$ if and only if for all $F \in [\mathbf{P}^c]$, for all $m \in \mathcal{RM}(m''_0)$ in the LPN $P^{\mathcal{F}_{env}}_{S,D}(F)$, there exists $m' \in \mathcal{RM}(m)$ such that $m'(D') = 1$ for some place $D' \in \mathcal{D}^r_f(F)$.

Definition 4.4 (Weak compatibility)**.** For two schemas S^1 and S^2 with $I \in \mathcal{F}^1_{ext}$, $f \in \mathcal{F}_{env} \subseteq \mathcal{F}^2_{int}$, and $f \not\models I$, then S^1 and S^2 are *weakly* (I,f)*-compatible* for an initial instance D (denoted $S^1 \; {}_I\!\lhd^f_D \; S^2$) iff there is a sequence of receptions of environment calls from $\mathcal{F}_{env}$ such that every event $\overline{E^c_f}$ in $P^{\mathcal{F}_{env}}_{S,D}$ induced by a call to I is eventually followed by a corresponding response E^r_f.

Formally, let $\mathbf{P}^c$ represent the call pattern of I. Then, $S^1 \; {}_I\!\lhd^f_D \; S^2$ if and only if for all $F \in [\mathbf{P}^c]$, for all $m \in \mathcal{RM}(m''_0)$ in the LPN $P^{\mathcal{F}_{env}}_{S,D}[F]$, there exists $m' \in \mathcal{RM}(m)$ such that $m'(D') = 1$ for some place $D' \in \mathcal{D}^r_f(F)$.

Note that an environment call may not terminate in general, but it may if we restrict to forests $F \in [\mathbf{P}^c]$. So, the fact that a particular environment call does not return does not imply that the two schemas are not (I, f)-compatible.

The difference between the two notions resides in the fact that once the selected $\overline{E_f^c}$ has been received, the LPN used for strong compatibility refuses any further environment calls (the context is fixed), while the one used for weak compatibility may accept them if they help processing the selected environment call. As a consequence, $S^1 \ {}_I\!\lhd_D^f \ S^2$ implies $S^1 \ {}_I\!\lhd\!\!\sim_D^f \ S^2$.

The schemas S^1 and S^2 are *(strongly)* [resp., *weakly*] *compatible* for the instances D^1 and D^2 with respect to a pairing map ξ (refer to Sect. 2.2 for definition) if and only if for all $I \in \mathrm{dom}(\xi) \cap \mathcal{F}^1_{ext}$, $\xi(I) \Vdash I$ and $S^1 \ {}_I\!\lhd_{D^2}^{\xi(I)} \ S^2$ [resp., $S^1 \ {}_I\!\lhd\!\!\sim_{D^2}^{\xi(I)} \ S^2$], and for all $I \in \mathrm{dom}(\xi) \cap \mathcal{F}^2_{ext}$, $\xi(I) \Vdash I$ and $S^2 \ {}_I\!\lhd_{D^1}^{\xi(I)} \ S^1$ [resp., $S^2 \ {}_I\!\lhd\!\!\sim_{D^1}^{\xi(I)} \ S^1$]. Strong and weak compatibility are denoted respectively $S^1 \ {}_{D^1}\!\overset{\xi}{\bowtie}_{D^2} \ S^2$ and $S^1 \ {}_{D^1}\!\overset{\xi}{\bowtie\!\!\sim}_{D^2} \ S^2$, and as for schema composition, ξ may be omitted when it is given in the context.

Compatibility together with fairness (Definition 4.1) ensure that any environment call will be answered in finite time. For weak compatibility, this is also true, provided the necessary environment calls (if any) have been received. We can prove the decidability of these important properties in our settings.

Theorem 4.1. *Given two schemas S^1 and S^2, an interface $I \in \mathcal{F}^1_{ext}$ and a function $f \in \mathcal{F}_{env} \subseteq \mathcal{F}^2_{int}$ such that $f \Vdash I$, with the finiteness hypotheses $\mathfrak{F}$, (I, f)-compatibility and weak (I, f)-compatibility are decidable.*

Proof. We state both (I, f)-compatibility properties in terms of home spaces. A *home space* [6] is a set of markings $\mathcal{M}$ which can be reached from any reachable marking, i.e., $\mathcal{M}$ is a home space of an LPN (N, m) if for all $m' \in \mathcal{RM}(m)$, $\mathcal{RM}(m') \cap \mathcal{M} \neq \emptyset$.

Assume that the set of functions which can be triggered by an environment call is $\mathcal{F}_{env} \subseteq \mathcal{F}^2_{int}$, with $f \in \mathcal{F}_{env}$. Denote by $\mathbf{B}^c$ the body of the call query of f, and by $\mathbf{P}^c$ the call pattern of I. Consider an instance D over S^2 and a forest $F \in [\mathbf{P}^c]$. Note that since $f \Vdash I$, it also holds that $F \in [\mathbf{B}^c]$, so $P^{\mathcal{F}_{env}}_{S^2,D}(F)$ is well defined. Let $\mathcal{M}^r(F)$ be the set of markings of $P^{\mathcal{F}_{env}}_{S^2,D}(F)$ in which the environment call $\overline{E_f^c}$ with parameters F can return, i.e., $\mathcal{M}^r(F) = \{m \mid \exists D \in \mathcal{D}^r_f(F), m(D) = 1\}$. Also define the sets of markings $\mathcal{M}_D$ with at least one token in place D, i.e., $\mathcal{M}_D = \{m \mid m = m_D + \sum_{D' \in \mathcal{D}} k_{D'} \cdot m_{D'} \quad \text{with } k_{D'} \in \mathbb{N}\}$, where $\mathcal{D}$ denotes the set of places of $P_{S^2,D}(F)$, and for any $D \in \mathcal{D}$, m_D is the marking such that $m_D(D) = 1$ and $m_D(D') = 0$ if $D' \neq D$. The sets $\mathcal{M}_D$ are *linear sets*, of *base* m_D and of *period* $\{m_{D'} \mid D' \in \mathcal{D}\}$. Observe that the set of markings $\mathcal{M}^r(F)$ can also be seen as $\mathcal{M}^r(F) = \bigcup_{D \in \mathcal{D}^r_f(F)} \mathcal{M}_D$.

Strong (I, f)-compatibility is equivalent to showing that for the external service I of schema S^1, for a given set $\mathcal{F}_{env} \subseteq \mathcal{F}^2_{int}$ containing f, for all $F \in [\mathbf{P}^c]$, $\mathcal{M}^r(F)$ is a home space of $P^{\mathcal{F}_{env}}_{S^2,D}(F)$. It was proved in [6] that this property is decidable for finite unions of linear sets with the same period, which we have seen is the case here for $\mathcal{M}^r(F)$. Thus, given some forest $F \in [\mathbf{P}^c]$, the home space property of $\mathcal{M}^r(F)$ in $P^{\mathcal{F}_{env}}_{S^2,D}(F)$ is decidable. Besides, since the assumptions $\mathfrak{F}$ hold, there are finitely many values for F in $[\mathbf{P}^c]$, hence the decidability result.

Weak (I, f)-compatibility is decided similarly, by testing home spaces of the LPNs $P^{\mathcal{F}_{env}}_{S^2,D}[F]$, $F \in [\mathbf{P}^c]$. □

Theorem 4.2. *With the finiteness hypotheses $\mathfrak{F}$, compatibility and weak compatibility are decidable.*

Proof. Consider a pairing map ξ between two schemas S^1 and S^2, two instances D^1 and D^2 over S^1 and S^2 respectively, and two set of functions $\mathcal{F}^i_{env} \subseteq \mathcal{F}^i_{int}$, $i \in \{1,2\}$, which can be triggered by environment calls. It suffices to check that for all $I \in \mathrm{dom}(\xi)$, $\xi(I) \not\models I$, and $S^1 \ {}_I\triangleleft^{\xi(I)}_{D^2} \ S^2$ if $I \in \mathcal{F}^2_{ext}$ or $S^2 \ {}_I\triangleleft^{\xi(I)}_{D^1} \ S^1$ if $I \in \mathcal{F}^1_{ext}$ (or the respective weak $(I,\xi(I))$-compatibility equations if weak compatibility is considered).

For a given pair of services (I,f), with finite variable domains assumption, implementation relation is decidable (Proposition 2.1), as well as strong or weak (I,f)-compatibility (Theorem 4.1). Therefore, since $\mathrm{dom}(\xi) \subseteq \left(\mathcal{F}^1_{ext} \cup \mathcal{F}^2_{ext}\right)$ is finite, weak and strong compatibility can be decided. □

5 Distributivity of Compatibility

We now study an interesting property of compatibility. We prove in Theorem 5.4 that if several schemas are pairwise-compatible, then any of their compositions are also compatible. This is useful because it allows a faster semi-algorithm to decide whether two schemas are compatible, by checking compatibility between smaller sets of services. In practice, this can also be used by a company which provides services to another, and wants to hide the fact that it delegates part of its tasks to another service provider.

5.1 Splitting Nets

We introduce a second operation on labeled nets, called *splitting*. It will be used to shorten the proofs in this section, but it is also an interesting concept: this operation "splits" the places of a net to obtain several interacting subnets. Each of these places is the restriction of the global instance to a set of peers. Before defining it, we need to make explicit the projection of a labeled net over a subset of peers.

Definition 5.1 (projection)**.** Given a schema S, an instance $D \in \mathcal{D}_S$ and a subset $\mathcal{P}' \subseteq \mathcal{P}$ of the peers of S, the *projection* of $N_{S,D}$ over $\mathcal{P}'$ is the labeled net denoted by $N^{\mathcal{P}'}_{S,D}$ which is defined as follows.

(*i*) First, project all the places (instances) of $N_{S,D}$ over $\mathcal{P}'$ as defined at the end of Section 2.2;

(*ii*) The rest corresponds to the projection of a transition system over the non-empty places:

- if any projected instance ${D'}^{\mathcal{P}'}$ is empty, remove it and all its successors (places and transitions);
- inductively merge the isomorphic instances separated by a single transition, removing this transition;
- merge the transitions which have the same predecessor and successor, and the same label.

Let us now define the split operation. Intuitively, splitting a net extracts external service calls and computation from a net. This operation allows to

isolate the computation steps of a service in a subnet involving places (instances) localized on a given set of peers, hence emphasizing the distribution of services. The construction is illustrated in Fig. 6, a formal description follows.

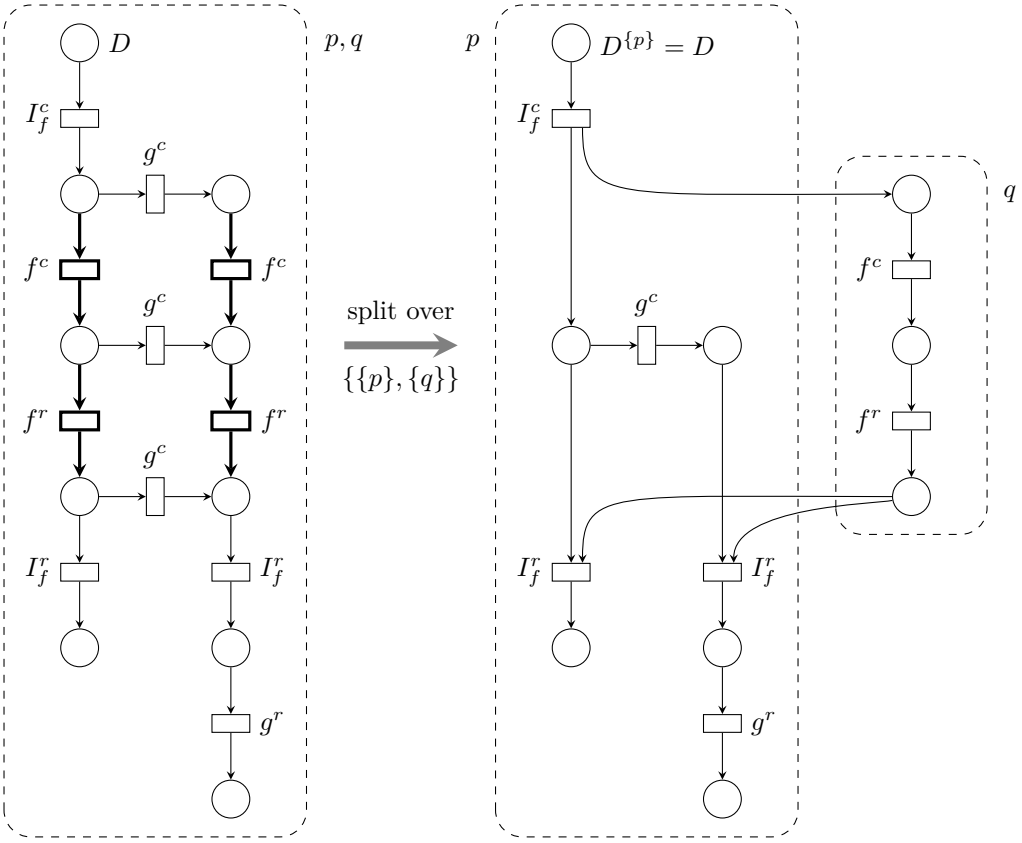


Figure 6: Split of the net $N_{S,D}$ over $\{\{p\},\{q\}\}$. We assume two distinct peers p and q in schema S, and services f, g, I such that $f \not\models I$ and $\nu(f) = q$, $\nu(I) = \nu(g) = p$. For simplicity, the initial instance D contains documents located exclusively on p, but data is created on q when I is called. Such a call induces a call to service f on q (thick transitions), but also allows to call g on p. Service g may only return after I has completed. In this simple example, we also assume that merging the workspace of I and the net projected on p introduces no new arc in the flow relations.

Definition 5.2 (split)**.** Given a schema S with set of peers $\mathcal{P}$, an instance $D \in \mathcal{D}_S$ and a partition of $\mathcal{P}$ into $\mathcal{P}_1 \uplus \mathcal{P}_2$, we denote by $N_{S,D}^{\mathcal{P}_1,\mathcal{P}_2}$ the *split* of $N_{S,D}$ over $\{\mathcal{P}_1, \mathcal{P}_2\}$. This labeled net is constructed as follows.

(*i*) For all $I \in \mathrm{dom}(\gamma)$ and $f = \gamma(I)$ such that $\nu(I) \in \mathcal{P}_i$ and $\nu(f) \notin \mathcal{P}_i$ (for some $i \in \{1, 2\}$), let Θ_I be the sets of transitions from the net $N_{S,D}$ which are labeled with I_f^c. Let δ be the function which associates each transition t in some Θ_I with the set of transitions corresponding to return of the interface call initiated by t. Consequently, every transition in $\delta(t)$ is

labeled with I^r_f if t is labeled with I^c_f. The sets Θ_I are used to characterize the transitions which create workspaces in peers not in $\mathcal{P}_i$, with $\mathcal{P}_i \supset \nu(I)$, for $i \in \{1,2\}$.

(*ii*) For each $i \in \{1,2\}$, start by projecting $N_{S,D}$ on the peers $\mathcal{P}_i$, to obtain $N^{\mathcal{P}_i}_{S,D}$ as described in Definition 5.1.

(*iii*) For each $I \in \mathrm{dom}(\gamma)$, for each $t \in \Theta_I$, create a new net $N(t)$ isomorphic to the net $N_{S,D_f(F)}$, where F are the parameters of the call to I in transition t. Consider $N_1 = N^{\mathcal{P}_1}_{S,D} \oplus N^{\mathcal{P}_2}_{S,D} \oplus \left(\bigoplus_{I\in\mathrm{dom}(\gamma)} \bigoplus_{t\in\Theta_I} N(t)\right)$, this merges the nets corresponding to the implemented interface calls.

(*iv*) For every $I \in \mathrm{dom}(\gamma)$, for every transition t in Θ_I, add to N_1 the arc (t, D), where D is the initial place of $N(t)$. Also add the arcs (D', t'), with $t' \in \delta(t)$, for each D' which is a place of $N(t)$ in which I may return with the results expected by t'.

(*v*) Since we may have added transitions corresponding to implemented interface calls in the nets $N(t)$, steps (*i*) to (*iv*) should be iterated on the nets $N(t)$ until there are no more. Note that with the no-recursion hypothesis, the process stops and the net remains finite. The resulting net is $N^{\mathcal{P}_1,\mathcal{P}_2}_{S,D}$.

We easily extend the split operation to more than two sets of peers, and denote it $N^{\mathcal{P}_1,\ldots,\mathcal{P}_k}_{S,D} = N^{(\mathcal{P}_i)}_{S,D}$ for a family of pairwise-disjoint sets of peers $\mathcal{P}_i$ $(1 \le i \le k)$, with $\biguplus_{i=1}^k \mathcal{P}_i = \mathcal{P}$. The corresponding LPN is $P^{(\mathcal{P}_i)}_{S,D} = (N^{(\mathcal{P}_i)}_{S,D}, m'_0)$, with $m'_0(D^{\mathcal{P}_i}) = 1$ for each $1 \le i \le k$ and $m'_0(D') = 0$ for all other places D'.

Remark 5.1. Splitting a net destroys the *eval* link between the active nodes making implemented interface calls, and their respective workspaces, as written in Remark 2.1. However, this link still exists in the split net, since each interface call is associated with a fresh subnet $N(t)$ (see item (*iii*) of Definition 5.2). Thus, it can be restored without ambiguity when generating an instance from a marking of a split LPN (this is a minor change in Definition 3.4).

The following theorem states that splitting preserves the global behavior.

Theorem 5.1. *Given a schema S with set of peers $\mathcal{P}$ and an instance D over S, consider a family of sets of peers $\mathcal{P}_k$, $k \in K$, for some finite set K such that $\biguplus_{k\in K} \mathcal{P}_k = \mathcal{P}$. The LPNs $P_{S,D}$ and $P^{(\mathcal{P}_k)}_{S,D}$ are equivalent.*

Proof. We follow the same scheme as in the proof of Theorem 3.2. Since we can assume there are no running interface calls in D (it is supposed to be an initial instance), the *eval* function is preserved when projecting, therefore $D(m'_0) = \biguplus_{k\in K} D^{\mathcal{P}_k} = D$ (see Remark 2.1) and $D(m_0) \equiv D(m'_0)$. We also use Remark 5.1 to generate correct instances from markings. □

This theorem leads to a more general result, useful in the proof of our main result given in Theorem 5.4, which provides more efficient means to check compatibility.

Corollary 5.2. *For any schema S containing a function $f \in \mathcal{F}_{env} \subseteq \mathcal{F}_{int}$ whose call query has body $\mathbf{B}^c$, for any forest $F \in [\mathbf{B}^c]$, instance $D \in \mathcal{D}_S$, and partition of the peers of S into $\mathcal{P}_1 \uplus \mathcal{P}_2$, the LPNs $P^{\mathcal{F}_{env}}_{S,D}(F)$ and $P^{\mathcal{F}_{env}}_{S,D}(F)^{\mathcal{P}_1,\mathcal{P}_2}$ are equivalent. Similarly, $P^{\mathcal{F}_{env}}_{S,D}[F]$ and $P^{\mathcal{F}_{env}}_{S,D}[F]^{\mathcal{P}_1,\mathcal{P}_2}$ are equivalent.*

Proof. Simply note that the underlying net of $P^{\mathcal{F}_{env}}_{S,D}(F)$ is the merging of several nets of the form N_{S,D_i}, for a finite family of instances (D_i) over S, to which one place and a finite number of transitions are added (see Figure 5).

By a simple extension of Theorem 5.1, each LPN (N_{S,D_i}, m_i) is equivalent with $(N^{\mathcal{P}_1,\mathcal{P}_2}_{S,D_i}, m'_i)$, if m_i is a marking such that all places but D_i are empty and D_i contains an arbitrary number of tokens (this is useful because this place may have received an arbitrary number of tokens, due to the extra place). The marking m'_i is the result of the projection on $\mathcal{P}_1$ and $\mathcal{P}_2$, i.e., $m'_i(D_i^{\mathcal{P}_j}) = m_i(D_i)$ for $j \in \{1,2\}$ and $m'_i(D) = 0$ otherwise. By Proposition 3.3, the merging of these nets preserves equivalence. Finally, the addition of the extra places and transitions (input and output of the merged nets) also preserves the property, since these places do not contain trees which satisfy any guard and these transitions do not require any guard to hold, hence the result.

The same technique applies to $P^{\mathcal{F}_{env}}_{S,D}[F]$. □

5.2 Distributivity and Semi-Decision Algorithm for Compatibility

We now show that compatibility "distributes" over schemas: if a set of schemas are pairwise-compatible, then any composition of several of them is compatible with any other composition (Theorem 5.4). Recall that composition $\|_\xi$ of schemas w.r.t. a pairing map ξ is defined in Section 2.2.

This result has two practical consequences. One is a faster way to check compatibility between large schemas, another is to preserve confidentiality, since a schema does not need to know what resources another schema uses to answer its requests.

All the results in this section are proved for strong compatibility, but work nevertheless with weak compatibility.

The following lemma is a first step towards proving Theorem 5.4. In its proof, notation $\mathcal{F}^1_{env} \sqcup_\xi \mathcal{F}^2_{env}$ refers to the set of functions that can be triggered by an environmental call in schema $S^1 \|_\xi S^2$, it is a subset of $\mathcal{F}^1_{env} \cup \mathcal{F}^2_{env}$ and a superset of $\mathcal{F}^1_{env} \cup \mathcal{F}^2_{env} \setminus \xi(\mathrm{dom}(\xi))$: some functions of a set $\mathcal{F}^i_{env}$ may not remain in $\mathcal{F}^1_{env} \sqcup_\xi \mathcal{F}^2_{env}$, because they implement interfaces of S^{3-i} and are not provided to the environment anymore.

Lemma 5.3. *Consider three instances D^1, D^2, D^3 over three schemas S^1, S^2, S^3. If $S^3 \ {}_{D^3}\overset{\xi_{13}}{\bowtie}_{D^1} \ S^1 \ {}_{D^1}\overset{\xi_{12}}{\bowtie}_{D^2} \ S^2 \ {}_{D^2}\overset{\xi_{23}}{\bowtie}_{D^3} \ S^3$, then $S^3 \ {}_{D^3}\overset{\xi_{123}}{\bowtie}_{D^1 \uplus D^2} \ (S^1 \|_{\xi_{12}} S^2)$ for any pairing maps $\xi_{12}, \xi_{13}, \xi_{23}$ such that $\mathrm{dom}(\xi_{13}) \cap \mathrm{dom}(\xi_{23}) = \emptyset$, with $\xi_{123} = \xi_{13} \cup \xi_{23}$.*

Proof. Obviously, for any $I \in (\mathcal{F}^1_{ext} \cup \mathcal{F}^2_{ext}) \cap \mathrm{dom}(\xi_{123})$, $f \in \mathcal{F}^3_{env} \subseteq \mathcal{F}^3_{int}$ such that $f \not\models I$, then $(S^1 \|_{\xi_{12}} S^2) \ {}_I\triangleleft^f_{D^3} \ S^3$. Indeed, suppose $I \in \mathcal{F}^1_{ext}$ (and thus $I \in \mathrm{dom}(\xi_{13})$), denote by $\mathbf{P}^c$ its call pattern. Since $S^1 \ {}_I\triangleleft^f_{D^3} \ S^3$, we already know that the different LPNs $P^{\mathcal{F}^3_{env}}_{S^3,D^3}(F)$, for $F \in [\mathbf{P}^c]$, have the property required in Definition 4.3. The case $I \in \mathcal{F}^2_{ext}$ is identical.

For the other direction, assume $I \in \mathcal{F}^3_{ext} \cap \mathrm{dom}(\xi_{123})$. Because $\mathrm{dom}(\xi_{13}) \cap \mathrm{dom}(\xi_{23}) = \emptyset$, either $I \in \mathrm{dom}(\xi_{13})$ or $I \in \mathrm{dom}(\xi_{23})$, assume the first one (the other case is similar) and consider $f = \xi_{13}(I) \in \mathcal{F}^1_{env}$. By compatibility hypothesis between S^1 and S^3, $f \not\models I$. We want to show that for any $F \in [\mathbf{P}^c]$,

for any reachable marking m in the LPN $P^{\mathcal{F}^1_{env} \sqcup_\xi \mathcal{F}^2_{env}}_{S^1 \|_{\xi_{12}} S^2, D^1 \uplus D^2}(F)$, there exists a marking $m' \in \mathcal{RM}(m)$ such that $m'(D) = 1$ for some $D \in \mathcal{D}^r_f(F)$.

Fix some $F \in [\mathbf{P}^c]$, let m be a reachable marking of $P^{\mathcal{F}^1_{env} \sqcup_\xi \mathcal{F}^2_{env}}_{S^1 \|_{\xi_{12}} S^2, D^1 \uplus D^2}(F)$. Let us split this LPN to obtain $P = P^{\mathcal{F}^1_{env} \sqcup_\xi \mathcal{F}^2_{env}}_{S^1 \|_{\xi_{12}} S^2, D^1 \uplus D^2}(F)^{\mathcal{P}^1, \mathcal{P}^2}$, where $\mathcal{P}^i$ is the set of peers of schema S^i. By Corollary 5.2, $P \equiv P^{\mathcal{F}^1_{env} \sqcup_\xi \mathcal{F}^2_{env}}_{S^1 \|_{\xi_{12}} S^2, D^1 \uplus D^2}(F)$ and hence there exists a reachable marking m_1 in P such that $D(m_1) \equiv D(m)$. Denote by $\mathcal{D}^i$ the sets of places of P located on peers $\mathcal{P}^i$, and consider $m_2 = m_{1|\mathcal{D}^1}$. By construction, m_2 is a reachable marking in $P^{\mathcal{F}^1_{env}}_{S^1, D^1}(F)$, so because $S^3 \ {}_{D^3}\overset{\xi_{13}}{\bowtie}_{D^1} S^1$, there exists a marking m'_2 reachable from m_2 such that $m'_2(D) = 1$ for some $D \in \mathcal{D}^r_f(F)$. Let σ_2 be the occurrence sequence such that $m_2 \, [\sigma_2\rangle \, m'_2$.

We show that there exists an occurrence sequence σ_1 and a marking m'_1 of P such that $m_1 \, [\sigma_1\rangle \, m'_1$, with $m'(D) = 1$ for some $D \in \mathcal{D}^r_f(F)$. By definition of the split, any transition of σ_2 labeled with something different from J^r, with $J \in \mathcal{F}^1_{ext} \cap \mathrm{dom}(\xi_{12})$, can still occur in P. If there is a transition t in σ_2 labeled with such an event J^r, then a transition t' labeled with J^r_g, with $g = \xi(J) \in \mathcal{F}^2_{env}$ and $g \not\models J$, can be enabled in P only after S^2 has finished the computation of service g. By the compatibility hypothesis between S^1 and S^2, this is guaranteed to occur at some time since the computation of g requires a finite number of steps and we assume the LPN behaves fairly. Thus, σ_1 is obtained from σ_2 by adding n_t transitions $(t^t_1, t^t_2, \ldots, t^t_{n_t})$ just before the transitions t labeled with J^r, corresponding to the steps before J is ready to return, and by replacing t by t'. Moreover, the markings m^i_2 and m^i_1 obtained after every transition t_i of σ_2 are such that $D(m^i_1)^{\mathcal{P}^1} \equiv D(m^i_2)$, because these transitions have the same effects on the documents contained on the peers from $\mathcal{P}^1$, while the extra transitions only affect the peers from $\mathcal{P}^2$. So, there exists some $D \in \mathcal{D}^r_f(F)$ such that $m'(D) = 1$.

Finally, because $P \equiv P^{\mathcal{F}^1_{env} \sqcup_\xi \mathcal{F}^2_{env}}_{S^1 \|_{\xi_{12}} S^2, D^1 \uplus D^2}(F)$ and $m_1 \, [\sigma_1\rangle \, m'_1$ in P, there exist a marking m' and an occurrence sequence σ which verify that $m \, [\sigma\rangle \, m'$ in $P^{\mathcal{F}^1_{env} \sqcup_\xi \mathcal{F}^2_{env}}_{S^1 \|_{\xi_{12}} S^2, D^1 \uplus D^2}(F)$, with $D(m') \equiv D(m'_1)$, so that marking m' allows I to return, which concludes the proof. □

Theorem 5.4. *Given a set of instances D^i over pairwise-compatible schemas S^i, $i \in K$, for any finite and disjoint sets $K_1, K_2 \subseteq K$ and any pairing maps defined over disjoint domains it holds that*

$$(\|_{i \in K_1} S^i) \ {}_{(\uplus_{i \in K_1} D^i)}\bowtie_{(\uplus_{j \in K_2} D^j)} \ (\|_{j \in K_2} S^j) \ .$$

Proof. For clarity sake, we omit to mention the pairing maps in the statement and in the proof, they can be deduced from the schemas taken into account.

Let $K_1 = \{i_1, i_2, \ldots, i_k\}$ and $K_2 = \{j_1, j_2, \ldots, j_l\}$ be a partition of K. By Lemma 5.3, $S^{i_1} \ {}_{D^{i_1}}\bowtie_{D^{j_1} \uplus D^{j_2}} (S^{j_1} \,\|\, S^{j_2})$, but also $S^{j_3} \ {}_{D^{j_3}}\bowtie_{D^{j_1} \uplus D^{j_2}} (S^{j_1} \,\|\, S^{j_2})$ since all schemas are pairwise compatible. Thus we can apply again Lemma 5.3 on S^{i_1}, $(S^{j_1} \| S^{j_2})$, and S^{j_3}, to show that $S^{i_1} \ {}_{D^{i_1}}\bowtie_{D^{j_1} \uplus D^{j_2} \uplus D^{j_3}} ((S^{j_1} \| S^{j_2}) \| S^{j_3})$. By iterating this process on the right side we get $S^{i_1} \ {}_{D^{i_1}}\bowtie_{(\uplus_{j \in K_2} D^j)} (\|_{j \in K_2} S^j)$, and then iterating on the left gives the expected result. □

Remark 5.2. Note that the converse implication of Theorem 5.4 is not always true, i.e., $(\|_{i \in K_1} S^i) \ {}_{(\uplus_{i \in K_1} D^i)}\bowtie_{(\uplus_{j \in K_2} D^j)} (\|_{j \in K_2} S^j)$ does not imply that

$S^i \ {}_{D^i}\!\bowtie_{D^j} S^j$ for all $i \in K_1$ and $j \in K_2$. Indeed, the return of an interface I of S^i implemented by a function of S^j may be enabled by the composition between S^j and some other schemas S^k, $k \in K \setminus \{i, j\}$. The composition with the schemas S^k restricts the possible behaviors of S^j, and this may remove some behaviors which prevented S^j from returning a result to S^i.

This last theorem provides an algorithm (Algorithm 1) to check compatibility between two schemas S^1 and S^2. It first "splits" both schemas by isolating in $\mathcal{P}_1^1 \subseteq \mathcal{P}^1$ and in $\mathcal{P}_1^2 \subseteq \mathcal{P}^2$ the peers p_j^i which are involved in the communication between S^1 and S^2: for $i \in \{1, 2\}$, let $\mathcal{P}_1^i = (p_j^i)_{j \in K^i}$ be the set of peers of S^i which either own the interfaces in $\mathcal{F}_{ext}^i \cap \mathrm{dom}(\xi)$ (interfaces implemented in a peer of S^{3-i}), or own the functions in $\mathcal{F}_{env}^i \cap \xi(\mathrm{dom}(\xi))$ (functions implementing interfaces located in a peer of S^{3-i}).

Define the schemas S_j^i as the restrictions of S^i to peer p_j^i, for $i \in \{1, 2\}$ and $j \in K^i$. Then, compatibility is checked between all these S_j^i by using the algorithm described in Theorem 4.2 (called here `Compatibility_check`), and finally by recursively checking compatibility between each S_j^i and the remaining schema S_0^i (the restriction of S^i to $\mathcal{P}_0^i = \mathcal{P}^i \setminus \mathcal{P}_1^i$). The algorithm returns `true` when all the checks are positive, thus proving global compatibility (Theorem 5.4). It returns `false` otherwise, which does not necessarily mean that the schemas are not compatible (Remark 5.2).

In this algorithm, the function `Project` is used to project a schema, a pairing map, or an instance, to a set of peers, by keeping only the services and the documents they own. Each map $\xi_{j,j'}^i$ is defined over the interfaces of S^i which either belong to peers of S_j^i and are implemented in $S_{j'}^i$, or the converse.

The complexity of this algorithm is hard to compute, since the number of calls to `Compatibility_check` depend on the schemas themselves. However in general, although many calls to `Compatibility_check` are made by this algorithm, it is much faster than a single check on the full schemas. Indeed, the size of the constructed LPNs and the complexity of `Compatibility_check` is highly exponential in the number of functions in $\mathcal{F}_{env}^i$ and of their parameters, which our improved algorithm reduces as much as possible, since it only involves two peers at a time.

Note that the results shown in this section also hold for weak compatibility.

6 Conclusion and perspectives

We have proposed a Petri net semantics for a fragment of DAXML. We used this representation to define two new notions of compatibility between DAXML schemas, in order to guarantee correct orchestration of services, and we proved their decidability. Furthermore, we proposed a more efficient semi-algorithm to decide compatibility.

This work can be extended in several directions. First of all, our compatibility notions can be refined to cases where some environment calls needed to ensure progress of a service are guaranteed to occur by a contract. This notion of compatibility would be a good compromise between strong and weak compatibility.

Another possible development is to allow some recursion. Although recursion leads to undecidability [3] in general, it is likely that if the allowed recursion

Algorithm 1: `Compatibility_semi_check`

Input: two schemas S^1, S^2, instances $D^1 \in \mathcal{D}_{S^1}$, $D^2 \in \mathcal{D}_{S^2}$, a pairing map ξ.
Denote $S^i = (\mathcal{P}^i, \mathcal{F}^i_{int}, \mathcal{F}^i_{ext}, \mathcal{F}^i_{env}, \nu, \gamma)$, for $i \in \{1, 2\}$.

```
// Initialization
for i ← 1 to 2 do
    /* Select the peers of S^i which own interfaces implemented
       in a peer of S^{3-i}                                        */
    P^i_1 ← F^i_ext ∩ dom(ξ)
    /* Add the ones which own functions implementing
       interfaces in a peer of S^{3-i}                             */
    P^i_1 ← P^i_1 ∪ (F^i_env ∩ ξ(dom(ξ)))
    /* Store the remaining peers in P^i_0                          */
    P^i_0 ← P^i \ P^i_1
end
// Project schemas according to each peer p^i_j ∈ P^i_1, and to P^i_0
for i ← 1 to 2 do
    foreach p^i_j ∈ P^i_1 do
        S^i_j ← Project(S^i, {p^i_j})
        D^i_j ← Project(D^i, {p^i_j})
    end
    S^i_0 ← Project(S^i, P^i_0)
    D^i_0 ← Project(D^i, P^i_0)
end
// Check compatibility between single-peer schemas S^i_j
foreach p^i_j, p^{i'}_{j'} ∈ P^1_1 ∪ P^2_1 do
    if i = i' then
        ξ' ← ξ^i_{j,j'}
    else
        ξ' ← Project (ξ, {p^i_j, p^{i'}_{j'}})
    end
    if not Compatibility_check (S^i_j, D^i_j, S^{i'}_{j'}, D^{i'}_{j'}, ξ') then
        return false
    end
end
// Recursively check compatibility with remaining schema S^i_0
for i ← 1 to 2 do
    foreach S^i_j do
        ξ' ← ξ^i_{j,0}
        if not Compatibility_semi_check (S^i_j, D^i_j, S^i_0, D^i_0, ξ') then
            return false
        end
    end
end
return true
```

does not create an unbounded number of non-evaluated service references, then decidability of compatibility still holds.

Finally, although we prove (semi-)decidability, our algorithms rely on several reachability checks on complex Petri nets. In order to obtain practical results, it is necessary to improve complexity by reducing drastically the size of the LPNs. An interesting question is to study how to obtain the minimal number of places and transitions to represent the DAXML behaviors, which would at the same time be the best representation of the inner concurrency of the model. Another challenge is to avoid enumerating variable values, which would require to adapt the DAXML model to use value "types". The key is to find a good trade-off between the compact representation of workflows (as it is done with XML nets for example, using high-level Petri nets), and the existence of verification techniques similar to the ones we used here in the low-level Petri nets context.

References

[1] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: A data-centric perspective on web services. In *Journées Bases de Données Avancées (BDA 2002)*, 2002.

[2] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of Active XML systems. In *Symposium on Principles of Database Systems (PODS 2008)*, pages 221–230, 2008.

[3] A. Benveniste and L. Hélouët. Distributed Active XML and service interfaces. Technical Report 7082, INRIA, 2009.

[4] H. Carstensen. Decidability questions for fairness in Petri nets. In *Symposium on Theoretical Aspects of Computer Science (STACS 1987)*, volume 247 of *Lecture Notes in Computer Science*, pages 396–407, 1987.

[5] H. Che, Y. Li, A. Oberweis, and W. Stucky. Web service composition based on XML nets. In *Hawaii International Conference on System Sciences (HICSS-42)*, pages 1–10, 2009.

[6] D. de Frutos Escrig and C. Johnen. Decidability of home space property. Technical Report 503, LRI, 1989.

[7] J. Esparza and M. Nielsen. Decidability issues for Petri nets. *Journal of Information Processing and Cybernetics (EIK)*, 30(3):143–160, 1994.

[8] R. Howell, L. Rosier, and H.-C. Yen. A taxonomy of fairness and temporal logic problems for Petri nets. *Theoretical Computer Science*, 82(2):341–372, 1991.

[9] L. Hélouët and A. Benveniste. Document based modeling of web services choreographies using Active XML. In *IEEE International Conference on Web Services (ICWS 2010)*, pages 291–298, 2010.

[10] P. Jančar. Decidability of weak fairness in Petri nets. In *Symposium on Theoretical Aspects of Computer Science (STACS 1987)*, volume 349 of *Lecture Notes in Computer Science*, pages 446–457, 1989.

[11] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *International Conference on Concurrency Theory (CONCUR 2006)*, pages 477–491, 2006.

[12] K. Lenz and A. Oberweis. Inter-organizational business process management with XML nets. In *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 243–263. 2003.

[13] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[14] OASIS. Web services business process execution language. Technical report, OASIS, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[15] D. Park. Concurrency and automata on infinite sequences. In *GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, 1981.

[16] W3C. Web services description language (WSDL) 1.1. Technical report, W3C, 2001. http://www.w3.org/TR/wsdl.

A Expansions

Additional definitions for trees are needed. We denote by $F[n/x]$ the forest F in which node n is relabeled with x. Within a given forest, we denote by $T(n)$ the unique tree containing n. We also write $F \boxplus_n F'$ to denote the forest obtained by attaching a forest F' (or a tree) as a sibling of node n in forest F, and $F \boxminus T'$ to denote the forest obtained by removing a subtree T' from a forest F, etc.

We define an *expansion* of a pattern as a possible "simple" tree matching the tree pattern and the conditions. For this purpose, we use *valuations*, i.e., functions $v : \mathcal{X} \to \mathcal{V}$ which associate variables with data values. Intuitively, these associations are given by the different matchings with the body of the query. Then, an expansion consists of the nodes of the tree pattern plus extra nodes to extend the descendant edges, and a replacement of the variables by a forest of data values, according to some valuations (see Fig. 1). We first define formally the case where the constructor node of a tree pattern is located at the root for a single valuation.

Definition A.1. Let $v : \mathcal{X} \to \mathcal{V}$ be a valuation, let $G = (M, H, c, \mu_M, \mu_H)$ be a tree pattern of root c, and denote by $\{e_1, \ldots, e_k\} = \mu_H^{-1}(\mathit{descendant})$ the k descendant edges of G. A *rooted-expansion* of G with respect to v is a tree $T = (N, E, \mu)$ such that T is empty (i.e., $N = \emptyset$), or

- $N = M \uplus \biguplus_{i=1}^{k} M_i$, where for all $1 \leq i \leq k$, M_i is a finite but possibly empty set of l_i new nodes $M_i = \{n_i^1, \ldots, n_i^{l_i}\}$, with $l_i \geq 0$;
- $E = \mu_H^{-1}(\mathit{child}) \uplus \biguplus_{i=1}^{k} H_i$, where for all $1 \leq i \leq k$, H_i contains the new edges linking the nodes from M_i: denote by $(m_i^1, m_i^2) = e_i$ the descendant edge to replace, then $H_i = \{(n_i^j, n_i^{j+1}) \mid 1 \leq j < l_i\} \cup \{(m_i^1, n_i^1), (n_i^{l_i}, m_i^2)\}$ (or $H_i = \{e_i\}$ if $M_i = \emptyset$);
- $\mu(x) = \begin{cases} \mu_M(n) & \text{if } n \in M \text{ and } \mu_M(n) \in \mathcal{T} \cup \mathcal{V} \cup \overline{\mathcal{F}}, \\ v(\mu_M(n)) & \text{if } n \in M \text{ and } \mu_M(n) \in \mathcal{X}, \\ a & \text{if } n \in M \text{ is an internal node, } \mu_M(n) = \star, \text{ and } a \in \mathcal{T}, \\ \sigma & \text{if } n \in M \text{ is a leaf, } \mu_M(n) = \star, \text{ and } \sigma \in \mathcal{T} \cup \mathcal{V} \cup \overline{\mathcal{F}}, \\ a & \text{if } n \in M_i,\ 1 \leq i \leq k, \text{ and } a \in \mathcal{T}. \end{cases}$

Because of last three cases for μ (stars and descendant edges), there can be several (sometimes even infinitely-many) rooted-expansions of a single tree pattern with respect to a given valuation v. Denote by $[G]_v$ the set of possible rooted-expansions of a tree pattern G with respect to v. We now define an expansion in the general case, with respect to a set of valuations. The construction is illustrated in Fig. 1.

Definition A.2. Let V be a set of valuations, let $G = (M, H, c, \mu_M, \mu_H)$ be a tree pattern, and G_c be the subtree pattern rooted at c. An *expansion* of G with respect to V is a tree $T = (G' \boxplus_c F_c) \boxminus G_c$, where

- G' is a tree obtained from the tree pattern G by adding new nodes for the descendant edges in $G \boxminus G_c$, and by giving labels to nodes labeled with $\star$ in $G \boxminus G_c$, as detailed in the previous definition;

- $F_c = \bigcup_{v \in V} \{T_c^v\}$, with $T_c^v \in [G_c]_v$ for all $v \in V$ (F_c is an empty forest if $V = \emptyset$).

We extend the notation $[G]_V$ to denote the set of possible expansions of a general tree pattern with respect to a set of valuations V.

Definition A.3 (expansion)**.** Given a pattern $\mathbf{G} = (\{G_1, \ldots, G_k\}, cond)$, an *expansion* of $\mathbf{G}$ is a forest $F = \{T_1, \ldots, T'_k\}$, $k' \leq k$, such that for some set V of valuations consistent with *cond*, for all $1 \leq i \leq k$, there exists $T_j \in [G_i]_V$ for some $1 \leq j \leq k'$. The set of all expansions (varying V and the elements of $[G_i]_V$) of a given pattern $\mathbf{G}$ is written $[\mathbf{G}]$.

B Semantics of DAXML

In this section, $D = (F, eval, \ell)$ and $D' = (F', eval', \ell')$ are two instances over the same schema $S = (\mathcal{P}, \mathcal{F}_{int}, \mathcal{F}_{ext}, \nu, \gamma)$. Moreover, the symbol $f = \left(\mathbf{G}^c, Q^c, \{(\mathbf{G}_i^r, Q_i^r)\}_{i \in \{1,\ldots,K_f\}}\right) \in \mathcal{F}_{int}$ denotes a function, while the symbol $I = \left(\mathbf{P}^c, \{\mathbf{P}_i^r\}_{i \in \{1,\ldots,K_I\}}\right) \in \mathcal{F}_{ext}$ is used for an interface. Also, the operations on trees defined in the first paragraph of Appendix A are used.

Function calls and returns.

$D \xrightarrow{f^c} D'$ *(function call)*

If there is a node n in the forest F labeled with $!f$, such that $\ell(T(n)) = \nu(f)$ and the calling guard of f holds, i.e., $F \models \mathbf{G}^c$, then the internal function f can be called and D is modified into D' by adding a new workspace. Formally,

- $F' = F[n/?f] \cup W_n$, where the tree W_n is the workspace of n, i.e., a root node labeled with a_f whose children are the different trees of the call query $Q^c(F, n)$;
- $eval'(m) = eval(m)$ for all active nodes m in F, and $eval'(n) = W_n$;
- $\ell'(T) = \ell(T)$ for all trees T in the forest F, and $\ell'(W_n) = \ell(T(n)) = \nu(f)$.

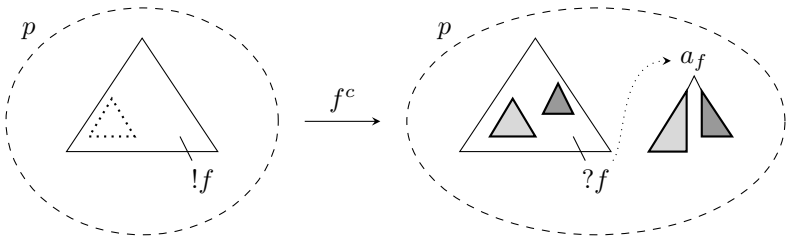


Figure 7: Illustration of a call to an internal function f. The dotted triangle represents the part of the forest witnessing that the call guard holds in D. The thick triangles inside the document are the occurrences of the body of the call query, their respective heads (the "parameters" of the call) form a new tree, the workspace of f.

$D \xrightarrow{f^r} D'$ *(function return)*

When there is a node n in F labeled with $?f$, if for some $i \in \{1, \ldots, K_f\}$ it holds that $eval(n) \models \mathbf{G}_i^r$, and in addition if $eval(n)$ contains no node labeled with $?\alpha$ for $\alpha \in \mathcal{F}_{int} \cup \mathcal{F}_{ext}$, then f may return. In this case, D' is obtained from D by attaching as a sibling of n the return query of f, and by removing the workspace of n:

- $F' = (F[n/f] \uplus_n Q_i^r) \setminus \{eval(n)\}$;
- $eval' : (\mathrm{dom}(eval) \setminus \{n\}) \to F'$ is such that $eval'(m) = eval(m)$ for all active nodes $m \neq n$;
- $\ell' : F' \to \mathcal{P}$ is defined by $\ell'(T) = \ell(T)$ for all trees $T \in F'$.

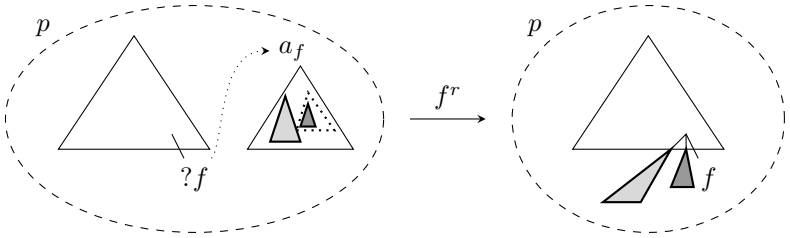


Figure 8: Illustration of a return from an internal function f. The dotted triangle witnesses that the return guard holds in the function workspace. The thick triangles inside the workspace are the occurrences of the body of the return query, their respective heads (the "results" of the call) are appended to the original document as siblings of f, and the workspace is deleted.

Implemented interface calls and returns.

$D \xrightarrow{I_f^c} D'$ *(Interface call)*

Given a node n of F, let T_n denote the smallest subtree containing n, its siblings, and all their descendants (if n is a root node, then $T_n = T(n)$, otherwise T_n is the tree rooted at the parent of n). If n is labeled with $!I$, where $I \in \mathrm{dom}(\gamma)$ is an implemented interface such that $\ell(T(n)) = \nu(I)$, and if $T_n \models \mathbf{P}^c$, then the interface I may be called. Assume $\gamma(I) = f$. The new schema D' consists of the addition of a tree containing the parameters for a future call to f, on the peer on which f is localized:

- $F' = F[n/?I] \cup W_n$, where the tree W_n is the workspace of n; it consists of a root node labeled with a_I, whose children are a single node labeled with $!f$ (for the function call), and the set of copies of all the smallest trees $T_{I,i}^c$, $1 \leq i \leq k$, such that $T_{I,i}^c \models \mathbf{P}^c$, which represent the different parameters of the call;
- $eval'(m) = eval(m)$ for all active nodes $m \neq n$ in F, and $eval'(n) = W_n$;
- $\ell'(T) = \ell(T)$ for all trees T in the forest F, and $\ell'(W_n) = \nu(f) \neq \nu(I)$.

$D \xrightarrow{I_f^r} D'$ *(Interface return)*

Assume there exists a node n in F labeled with $?I$ for some implemented interface $I \in \mathrm{dom}(\gamma)$, and denote $f = \gamma(I)$. If $eval(n)$ contains a node labeled with f (the computation of f is finished), no nodes labeled with

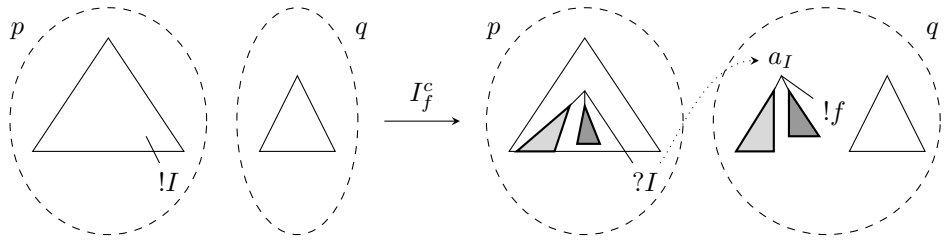


Figure 9: Illustration of a call to an interface I, implemented by a function f on another peer. The thick triangles inside the document on p are the parameters $T^c_{I,i}$, they consist of the minimal trees (among the siblings of $?I$) matching the call pattern of I. They are copied as such on peer q in the workspace of I, along with a node allowing to call f.

$?\alpha$ for $\alpha \in \mathcal{F}_{int} \cup \mathcal{F}_{ext}$ (no other computation is active), then the interface can return. The instance D appends the result to its forest after minor corrections:

- $F' = (F[n/I] \boxplus_n F'_n) \setminus \{eval(n)\}$, where F_n is the forest obtained by removing W_n defined above from $eval(n)$ (so that only the results returned by f remain in F_n, the parameters $T^c_{I,i}$ are discarded), and F'_n is a relabeling of the nodes m of F_n labeled with $!\alpha$:
  - if $\alpha \in \mathcal{F}_{int}$ and $\alpha = \gamma(J)$ for some $J \in \mathcal{F}_{ext}$ such that $\nu(J) = \nu(I)$ (i.e., α implements an interface J localized on the peer which made the call to I), then m is relabeled with $!J$;
  - if $\alpha \in \mathcal{F}_{ext}$ and there exists $g \in \mathcal{F}_{int}$, $\nu(g) = \nu(I)$, such that $g = \gamma(\alpha)$ (i.e., α is an interface implemented by a function g localized on the peer which made the call to I), then m is relabeled with $!g$;
- $eval' : (\mathrm{dom}(eval) \setminus \{n\}) \to F'$ is such that $eval'(T) = eval(T)$ for all active nodes $m \neq n$;
- $\ell' : F' \to \mathcal{P}$ is defined by $\ell'(T) = \ell(T)$ for all trees $T \in F'$.

Note that the relabeling of nodes is used here for convenience, since a renaming can be simulated by extra tags and functions in the existing formalism. For example, instead of returning a call to interface I to peer p and relabeling it to f, one could return a special tag ε_I, so that a function f_I with guard ε_I can be called by p, and this function would eventually return a query head containing $!f$, with $f \Vdash I$.

Non-implemented interface calls and returns.

$D \xrightarrow{I^c} D'$ *(Interface call)*

Recall that for any node n, T_n denotes the smallest subtree containing n, its siblings, and all their descendants. If n is labeled with $!I$, where $I \in \mathcal{F}_{ext} \setminus \mathrm{dom}(\gamma)$ is a non-implemented interface such that $\ell(T(n)) = \nu(I)$, and if $T_n \models \mathbf{P}^c$, then the interface I may be called. In such a case, D' is simply defined by $D' = (F[n/?I], eval, \ell)$.

$D \xrightarrow{\overline{I^r}} D'$ *(Reception of interface return)*

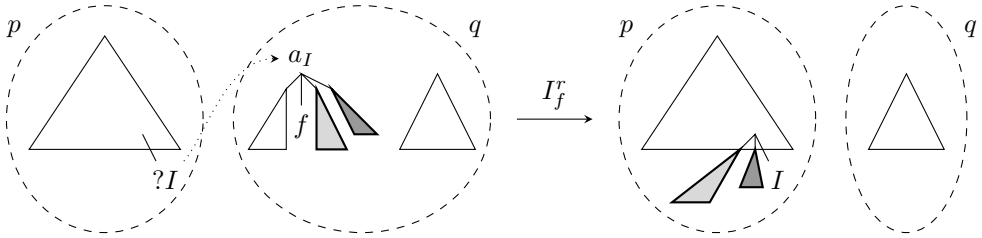


Figure 10: Illustration of a return from an interface I implemented by a function f. The thick triangles in the workspace of I are the results, they consist of the children of a_I which are not the parameters from the call. When f is completed, they are copied (up to a relabeling of the services) as siblings of the node I which initiated the call, while the workspace is deleted.

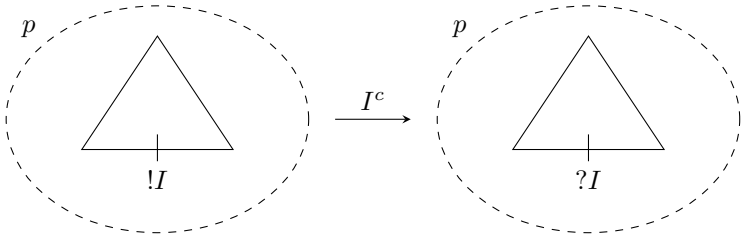


Figure 11: Illustration of a call to a non-implemented interface I. The label of the node is simply changed to indicate a running computation elsewhere.

A non-implemented interface call may return at any time, the peer which made the call "receives" the different possible values of the return pattern from an external (unknown) peer. Formally, if there is a node n in F labeled with $?I$, where $I \in \mathcal{F}_{ext} \setminus \mathrm{dom}(\gamma)$, then D leads to $D' = (F[n/I] \boxplus_n F^r, eval, \ell)$, where F^r is an expansion of one of the return patterns of I (see Definition A.3), i.e., there is $i \in \{1, \ldots, K_I\}$ such that $F^r \in [\mathbf{P}_i^r]$.

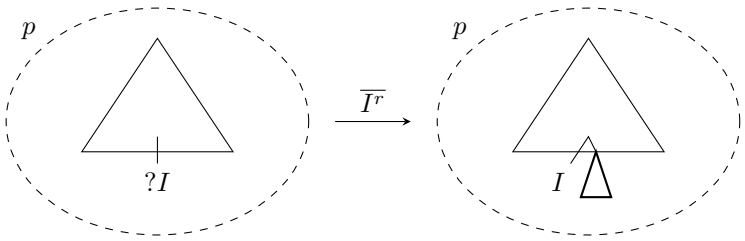


Figure 12: Illustration of a return from a non-implemented interface I. The thick triangle is an expansion of one of the return patterns of I, it represents the expected result.

To summarize, for a given a schema S, the set of events $\mathcal{E}$ consists of $\mathcal{E} = \{f^c, f^r \mid f \in \mathcal{F}_{int}\} \cup \{I_f^c, I_f^r \mid I \in \mathrm{dom}(\gamma), \gamma(I) = f\} \cup \{I^c, \overline{I^r} \mid I \in \mathcal{F}_{ext} \backslash \mathrm{dom}(\gamma)\}$ and is finite.

C Environment calls and returns.

$D \xrightarrow{\overline{E_f^c}} D'$ *(Reception of environment call)*

At any time, a peer may receive an external call from the environment, this call being associated with an internal function f. As for implemented interface calls, the new schema D' consists of the addition of a tree containing the parameters for the future call to f:

- $F' = F \cup W$, where the tree W is the workspace of the environment call; it consists of a root node labeled with a_{E_f}, whose children are a single node labeled with $!f$ (for the function call), and an arbitrary element of $\left[\mathbf{B}^c\right]$, which represents the different parameters of the call as f admits them ($\mathbf{B}^c$ denotes the body of the call query Q^c of f);
- $eval' = eval$ (there are no new active nodes);
- $\ell'(T) = \ell(T)$ for all trees T in the forest F, and $\ell'(W) = \nu(f)$.

$D \xrightarrow{E_f^r} D'$ *(Return to environment)*

Assume that D contains a tree W whose root node is labeled with a_{E_f}, which is the image of no node by $eval$. If W contains a node labeled with f and no nodes labeled with $?\alpha$ for $\alpha \in \mathcal{F}_{int} \cup \mathcal{F}_{ext}$ (no other computation is active), then the environment call may return. The instance D just removes the workspace W, as the computation is complete: $D' = \left(F \setminus W, eval, \ell_{|F \setminus W}\right)$.

Remark C.1. In the above, we implicitly extended the definition of instances. Indeed, in an instance, the workspace of an environment call associated with a service f is not related to any active node by the function $eval$. In fact, the newly created workspace W would be linked to a node from another instance, defined over another schema S', which made a call to a non-implemented interface $I \in \mathcal{F}'_{ext}$, and such that $f \Vdash I$. Composing both schemas would allow to add the appropriate link in $eval$, while merging I^c and $\overline{E_f^c}$ into I_f^c, and $\overline{I^r}$ and E_f^r into I_f^r.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399