Document based modeling of Web services choreographies using Active XML

Loïc Hélouët and Albert Benveniste (surname.name@inria.fr)
INRIA-Rennes, Campus de Beaulieu, 35042 Rennes Cedex, France

***Abstract*—This paper proposes a document based framework for the modeling of web-based choreographies involving a tight combination of workflow and data management. Our starting point is Active XML proposed by S. Abiteboul — AXML documents are XML documents with embedded service calls. We enhance Active XML with a rich notion of interface and we propose an effective technique to decide if provided services and needs of callers (defined as interfaces) are compatible. We also explicitly take distribution into account and allow for the composition of distributed AXML systems.**

***Keywords*-Web services, composition, Active XML, choreographies.**

I. Introduction

E-business and supply chain management involve a combination of widely distributed workflow systems and data/information management. According to [1], these systems can be viewed from one of the following perspectives:

- Workflow based perspective, in which process is emphasized. Web services and their orchestrations are now considered an infrastructure of choice for managing business processes and workflow activities over the Web infrastructure [2]. BPEL [3] has become the industrial standard for specifying orchestrations. Besides BPEL, the Orc formalism has been proposed to specify orchestrations, by W. Cook and J. Misra at Austin [4].
- Information based perspective, which emphasizes the information dimension, viewing processes as operations that are triggered as a result of information changes. Information centric systems typically rely on database oriented technologies.

Today, technologies in use for these two aspects are mostly separated. The WIDE approach [5], [6] was a first attempt toward tightly combining them. This attempt was further developed in [1], where the three dimensions of process, information, and organization, are considered. In response to the same need, the notion of "business artifact" has been proposed at IBM as a framework tightly combining workflow and data management [7]. See [8] for a brief survey, and [9] for a use of business artifacts in the context of Active XML, i.e., closer to our present study.

This work was partially funded by the ANR national research program DOTS (ANR-06-SETI-003), DocFlow (ANR-06-MDCA-005) and the project CREATE ActivDoc.

The above attempts (with the exception of [9]) do not build on top of the reference for documents manipulation, namely XML [10]. Active XML (AXML) [11] was proposed by Serge Abiteboul et al. as a high-level specification language tailored for data-intensive, distributed, dynamic Web services. It mainly consists in XML documents with embedded service calls. It allows for lazy evaluation: services can return structured data that contain references to services that have to be called to continue the evaluation of the returned values if needed. AXML offers mechanisms to store and query structured data distributed over entities called *peers* but does not provide ways to define control flows. AXML was then extended with *guarded* service calls to capture control flows [12], [9].

However, distribution in Guarded AXML is only reflected through the localization of a service, and some guards apply to the whole system, without considering distribution of information. On the other hand, distribution and decentralization are central concepts in Service Oriented Architectures, in which software components know each other only through their interfaces. Interfaces must depict precisely the data that will be sent as an input to a service, and the expected valid returns. A component uses a service only if the service interface fulfills its needs.

In this paper we further build on top of Guarded AXML by, first, proposing a notion of *interface*, and, second, taking distribution into account explicitly. The resulting model is called *Distributed Active XML*. It offers a notion of composition that allows enriching in a modular way a set of peers owning services with more peers offering more services. Our extensions to the existing framework are illustrated through an existing case study, the Dell supply chain.

The paper is organized as follows. Section II recalls the needed background on AXML. The Dell supply chain example is discussed in section III; this is a challenging case, as it is a choreography that combines workflow and data management; the detailed modeling of the Dell supply chain example can be found in [13]. Section IV is the core of this paper, and presents our model of Distributed Active XML; a detailed and fully formal presentation can be found in the extended paper [13]. Issues of decidability and complexity are reported in section V; again, details are found in [13]. Finally, section VI concludes this work, and draws some perspectives on future work on AXML platforms.

II. Documents, patterns, and queries

In this section we give a sketch of the background material on documents that is used in Active XML. A more precise and formal definition can be found in [13]. An AXML description consists in (active) XML documents and services distributed over a set of agents called peers. Information owned by each peer is stored as XML documents. In this framework, documents are unranked and unordered *forests* F (i.e., sets of trees T) whose non terminal nodes are labeled with *tags* and whose leaves are labeled with *data* or *service calls*. Documents are hence used to specify records as in usual XML databases but also workflows in the form of "things to do" (todo's for short). Todo's appear in documents as service calls, represented with nodes labeled by $!f$ (where f is the name of a service). Examples of documents are found in Figure 2, top, and in Figure 4. A service call is marked with a "!" to indicate that the service is to be called, with a "?" to indicate that the service is being processed, or has no mark to indicate that the service call has been completed to its end.

Documents can be tested using *patterns*. A *pattern* is a tree with two types of edges: / to indicate the *child* relation, and // to indicate the *descendant* relation, where "descendant" means any finite iteration of "child". Internal nodes of patterns can be labeled with tags or with the special label $\star$ to refer to an arbitrary tag. Leaves can be labeled with tags, data, service calls, or *variables*. These patterns are exactly those defined in [14], and without variables, correspond to a fragment of Xpath frequently used in the literature [15]. An example of pattern is found on Figure 1, diagram on the left. This pattern checks for the presence of an order in a document. Patterns can be given *conditions* which are constraints relating their variables of the following form: $X \sim Y$, where $\sim \in \{=, \neq, \leq\}$.

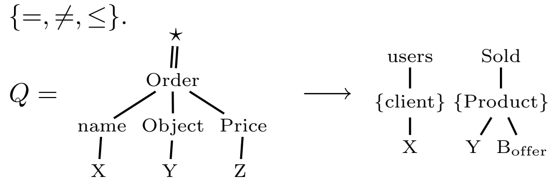

Figure 1. An example of nondeterministic query $Q = B \to H$; B is a pattern and H is a forest.

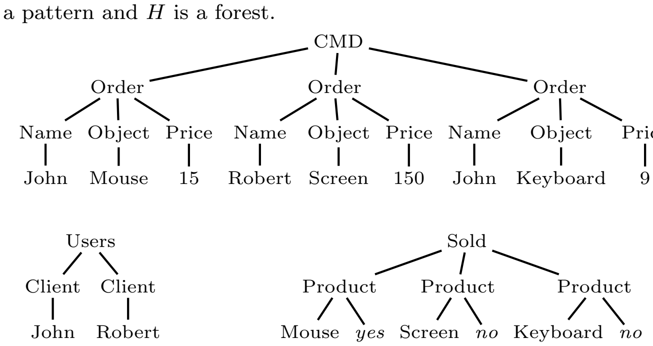

Figure 2. A sample document F (top) and a result of query $Q(F)$ (bottom), for Q as in Figure 1.

Testing a pattern B on a document F consists in searching for a *matching* of B into F, i.e., a map μ, from the nodes of B to the nodes of F, respecting child and descendant relations, mapping the root of B to a root of a tree in F, and preserving the labels, with $\star$ interpreted as any label. Matchings are not required to be injective. Pattern B of Figure 1 can be successfully tested on document F of Figure 2, top. For the considered document, there are three different maps sending pattern B onto the unique tree in F. Intuitively, each mapping corresponds to a recorded *Order* in the document. A document F *satisfies* a pattern B, written $F \models B$ iff there exists at least one matching from B to F.

Documents can be transformed by means of queries. A *query* is a pair $Q = B \to H$, where B is called the *body* and H the *head* of the query. In a query, the body is a pattern and the head is a forest, where each tree possesses a special node called the *constructor node*, shown with brackets. Figure 1 shows an example of query. A query operates on a documents F as follows: first, all matchings of pattern *body* into F are computed. These matchings define the valuations of variables in *body*. Then, each valuation is used to replace the variables in the subtrees of pattern *head* rooted at constructor nodes by their valuation, or by a nondeterministically chosen value if the variable does not appear in the *body* part. If the *head* pattern contains some constraints, then these valuations must be chosen consistently. The result of the query is then computed by replacing in the *head* the subtrees rooted at constructors by the subtrees computed from the mappings.

The result of applying query Q of Figure 1 to document F of Figure 2, top, is shown on of Figure 2, bottom. Note that variable B_{offer} is chosen nondeterministically in $\{yes, no\}$. Choosing nondeterministic values for variables was not originally proposed in [12] (though it could be emulated by some additional service), but seems an interesting way to allow for uncontrolled choices in modeled systems. In particular, non-determinism allows for the description of open systems (the environment can then be modeled as non-deterministic services) and also allows for underspecification of some parts of an AXML system.

At this point we did not model service calls, that were just introduced as distinguished labels. We will provide a semantics of calls and develop our model of Distributed Active XML (DAXML) in the following sections.

III. Motivation

In this section we motivate our approach with an interesting example, the Dell supply chain. This case study combines aspects of Web stored data management — the Dell Web portal — and complex distributed supplier chain involving logistics. Data are important and structured, both in the Dell catalog and in the processing of the supplier chain. The underlying workflow is rich and is a choreography [4] since there is no central orchestrator. Our study relies on the well documented description [16].

A. The Dell supply chain example

This example works as follows. After a customer places an order, either by phone or through the Internet on www.dell.com, Dell processes the order through financial evaluation (credit checking) and configuration evaluation (checking the feasibility of a specific technical configuration), which takes two to three days, after which it sends the order to one of its manufacturing plants in Austin, Texas. These plants can build, test, and package the product in about eight hours. The general rule for production is first in, first out, and Dell typically plans to ship all orders no later than five days after receipt. In most cases, Dell has significantly less time to respond to customers than it takes to transport components from its suppliers to its assembly plants. To compensate for long lead times and buffer against demand variability, Dell requires its suppliers to keep inventory on hand in the Austin revolvers (for "revolving" inventory). Revolvers are small warehouses located within a few miles of Dell's assembly plants. Each revolver is shared by several suppliers. Inventory in revolvers is owned and managed by suppliers and charged to Dell indirectly through component pricing. To help suppliers make good management decisions, Dell shares its forecasts with them once per month.

The overall architecture of the Dell supply chain is shown on Figure 3. Boxes denote peers (actors of the system), and multiple boxes indicate that there exist several instances of the considered peer. The customer and adjacent arrows are not within the scope of our model, so we will only provide the expectations of the customer in the form of an interface. As the "Dell supervisor" involves monitoring (a topic in itself) and a lot of algorithmic inventory management, we leave aside this part of the application. The Dell supply chain involves several services, which are summarized in Table I. Observe that the last three services need access to the revolver.

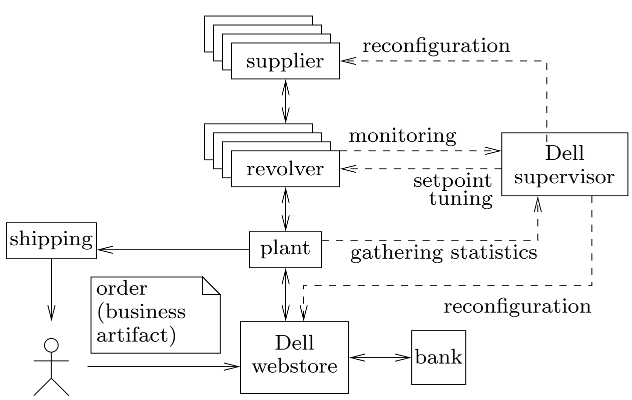

Figure 3. Architecture of the example.

B. Services, calls & distribution

Service specification uses the tree pattern matching mechanism defined in section II to define guards, and queries (in the spirit of Xquery [17]) to create new documents out of existing ones. Services decompose into a *call* — submitting a query — and a *return* — receiving a response and inserting it in the due place in the document. Both call and return are specified using *guarded queries* over documents. A call to a service usually proceeds as follows. When the service is owned by the peer calling it, the call consists in applying a query to the peer's document. The result of this query is defined to create a temporary working space (called the *workspace* of the service call) used to process the call (which can take several steps of computation if the workspace still contains todo's). A service call returns when one of its return queries is applicable to the workspace, i.e. when one of the guards of its return queries becomes true. The result of the considered query is then computed and inserted in the caller's document close to the node at which the service was requested.

When a peer calls a service owned by another peer, it places a new occurrence of service call in the distant peer's document, together with some parameters collected on its own document. When this distant service is completed, the distant peer sends back the result of his computation to the calling peer. Services calls and returns are executed in any order when their guards are true. The firing policy is local to each peer, which can be either eager (try to replace all service nodes tagged by some $!f$), lazy (perform service call only when needed), localized (focus first on the services of some tree/subtree), etc. So far, we hide all the details of this machinery, which will be made clear during the specification of (part of) the Dell example. A more formal presentation of our model is also provided in subsequent sections and in [13].

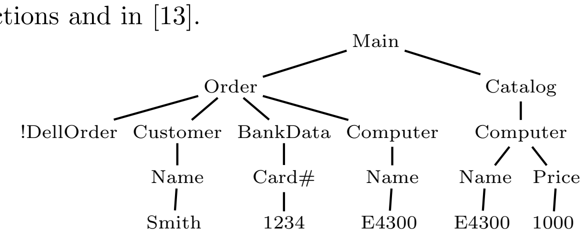

Figure 4. Initial document.

The Dell supply chain decomposes into several peers, namely the $Customer$, the $Webstore$, the $Plant$, the $Revolvers$, the $Suppliers$, the $Bank$, and the $Supervisor$. An example of document owned by peer Webstore is given in Figure 4. It contains one order that has been posted to the WebStore, represented by a subtree with root $Order$, that contains a call node with $!DellOrder$ tag. The parameters of the command appear as siblings of the call node. The document also contains a catalog, that lists types of computers together with their price.

We assume that peers are distinct computation units or actors in the system, and do not have direct access to the documents or internal machinery of other peers. Hence a peer sees another peer through the services it provides, i.e. via *interfaces* depicting the relations between inputs to a service during call, and the expected results that may be returned. More precisely, an interface consists of a

Service	*Description*
DellOrder	Returns a computer to a customer in conformance with the order form
GiveOrderId	Gives a new identifier to a computer order form
CheckCredit	Checks whether a customer has sufficient credit; performed by the bank
RejectOrder	Rejects an order if customer has insufficient credit on his bank account or order incorrect
ProcessOrder	Collects and assemble items, and delivers the assembled computer
GetItem	Gets a given item from the revolver
DecrementInventory	Removes one item from the revolver
RefuelInventory	Adds a certain number of items to the revolver

Table I: simplified description of the services in the Dell Supply Chain

pattern depicting the allowed parameters of a call, a list of patterns depicting the shape of returned documents, and a constraint over the variables of these patterns.

Let us illustrate interfaces with an example. In the dell supply chain, customers see service DellOrder provided by the Webstore only through an interface $I_{\text{DellOrder}}$. This interface is shown on Figure 6. It consists of two patterns and an assertion. *Call pattern* I^{C} specifies that the customer should submit a product reference, a catalog price, and his credit card information to the Webstore. In turn, the customer expects the delivery of the ordered product, as specified by the *return pattern* I^{R}. Together with an additional *assertion* $[X'_{\text{C}} = X_{\text{C}}]$, this interface specifies that, when calling DellOrder service, the customer should receive the computer she ordered.

Let us detail how the Webstore *implements* the DellOrder interface in Figure 7. The call guard for the corresponding DellOrder service is simply *true*, which means that a call to service *DellOrder* can be processed by the Webstore as soon as a node tagged by !*DellOrder* exists in the document. The call query Q^C of the DellOrder service copies all the needed parameters and creates instances of service calls (GiveOrderId, CheckCredit, RejectOrder, ProcessOrder) that have to be completed to process the order. Note that this also requires that the order has been correctly filled, and that the ordered reference appears in the catalog. When this is not the case, the order will not be processed.

The returned value depends on whether the order was accepted or rejected. Here, the implementation of the DellOrder service consists of a set of two guarded return queries, namely (G_1, Q_1^R) and (G_2, Q_2^R). Query Q_1^R returns a delivery certificate, but can only be executed when the order has been processed in due form, which is described by guard G_1. Query Q_2^R returns a rejected order when guard G_2 holds, that is when the order was tagged as *rejected* by service RejectOrder due to, either a negative answer from the bank, or an incorrect filling of the order form. Observe that the service specified in Figure 7 *is not* a correct implementation of the interface specified in Figure 6. The main reason is that the customer was too optimistic in not considering possible rejection of her payment by the bank. The interface of Figure 6 should in fact be corrected accordingly, i.e include a pattern for the rejection case — this is not done here to save space.

Further specifying the services GiveOrderId, CheckCredit, RejectOrder, and ProcessOrder, would complete the description of the DellOrder service by the Webstore. Among these, the first three are internal services held by the Webstore, whereas ProcessOrder is a service held by the Plant, a different peer. Hence, the latter should be specified as an interface when seen from the Webstore, with corresponding implementation by the Plant — this mimics what we did in figures 6 and 7.

The implementation of ProcessOrder is shown in part on Figure 5. The call query of this service creates as many instances of service GetItem as there are parts (we show only one); each instance is given a unique identifier X_{Ident}. Service GetItem is owned by the plant. The return guard G^R requires that each item has been properly obtained by the Plant (this process is not described in this shortened version of the case study). The return query of ProcessOrder is then simply $Q^R = \mathit{true} \rightarrow \mathit{processed}$, thus enabling the guard G_1^R of DellOrder, see Figure 7.

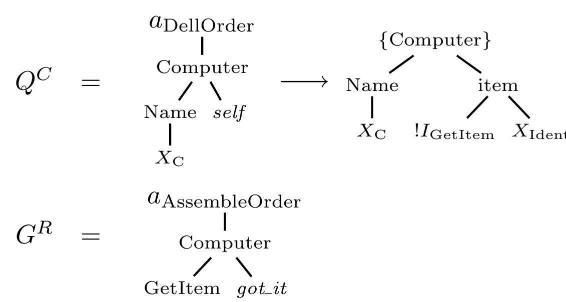

Figure 5. ProcessOrder as implemented by the plant.

At this point, the following reasoning can be performed on the DellOrder system. It can be easily shown that the service performs as specified, provided (of course) that the order is not rejected. Rejection can occur either because the order was not properly filled, or because the bank did not approve the transaction. Now, satisfying the guard G^R of ProcessOrder is in the hands of the peers *supplier*: they are responsible for maintaining the stock at appropriate level.

This is a separate issue that can be checked based on the following information: 1/ what is the query rate for each item by the plant, 2/ how frequently suppliers check stock level at the revolvers, and, 3/ what is the critical stock level, for each item, that will cause the supplier to refuel this stock. This is a more difficult property for checking, but it is now local to the peers Revolvers and

Suppliers, and does not need a system wide analysis.

C. Discussion

Based on the case study, several remarks can be stated:

a) Documents and workflow must be handled on an equal basis: the supplier chain involves a complex workflow that is a combination of several orchestrations: the Webstore orchestrates the progressive transformation of the order into a computer for delivery; then, each supplier maintains independently the stock level at the revolver. On the other hand, checking that the order form is properly filled and conforms information from the catalog is a data issue. However, data and workflow tightly interact through guards, which involve pattern matching on documents, and through service calls and returns, which modify documents.

b) Services are specified in a declarative way: this is done by specifying guarded *call* and *return queries* over XML documents. We use tree patterns and queries to state when a service can be invoked, what parameters must be given when calling it, when it is ready to complete, and what it can return. Hence, the control flow of a service execution is not explicitly given. *Interfaces* are defined in a similar way, and thus can be compared with the function that a service offers. We will show that this offers opportunities for efficient analysis, by replacing a distant service by an interface for it.

c) A service can involve recursively other services for its completion: the DellOrder service of Figure 7 is a typical example. Calling it adds the result of the call query to the documents owned by the Webstore as a temporary workspace. This additional document contains fresh instances of GiveOrderId, CheckCredit, etc., which must be evaluated. Computing an answer to a call may involve several recursive calls to other sub-services. With an ad hoc use of guards, a high-level service can then play the role of an orchestration of sub-services.

IV. Distributed Active XML

A. Services as Functions

Services are owned by peers. They are captured by the notion of "function", whereas external services as seen by another peer are modeled by "interfaces".

Definition 1: A function *is divided into* call *and a* return *part. The call part consists in a* guarded query (G^C, Q^C) *and the return part in a set of a* guarded queries $\{(G_i^R, Q_i^R)\}_{i\in 1..n}$. *Guarded queries are simply pairs consisting of a* guard *(a pattern) and a query. The query can be applied to a document only if the associated guard holds on the considered document.*

Let us consider a function DellOrder, with call query $(true, Q^C)$ and return queries $\{(G_1^R, Q_1^R)(G_2^R, Q_2^R)\}$, where $Q^C, G_1^R, G_2^R, Q_1^R, Q_2^R$ are the guards and queries of Figure 7. As the call guard of DellOrder is *true*, this service can be evaluated as soon as a document contains a !DellOrder node. Consider for example the initial document of Figure 4, call it F. The DellOrder service can be called by peer Webstore. The call query applies to the document owned by the WebStore.

Applying DellOrder to F with its description in Figure 7 proceeds as follows. First, the body of the call query Q^C is applied to F while node labeled *self* in the body is mapped to node labeled !DellOrder in F. Valuations for the variables result, namely X_{N}=Smith, and so on. The selected values are then assigned to the corresponding variables in the head of call query Q^C, and the resulting document is put in a *workspace* WS pointing to node ?*DellOrder* of document F; note the shift from mark ! to ? for *DellOrder*, indicating evaluation in progress. Initial document F is augmented with WS. Usually, workspaces are trees with root labeled by distinct tags of the form a_f for each service name f. This allows in particular specifying that a query applies to a workspace, or checking that a service is under evaluation on a peer. As a consequence of the *DellOrder* call, new service calls appear for evaluation by the different peers that own each new service, thus further modifying the status of the document.

If, as a result, return guard G_1^R holds in the workspace, then return query Q_1^R can be applied. The result $Q_1^R(WS)$, that is a forest containing the reference of the order, the actual reference of the assembled computer, and a node with tag *delivered* is appended as a sibling to the calling node, and the workspace WS is deleted. If, however, return guard G_2^R holds in the document, then return query Q_2^R can be applied to the workspace, which appends a forest containing the reference of the order, and a node with tag *rejected*, and deletes the workspace. In both cases, the calling node with tag ?*DellOrder* is relabeled with *Dellorder*, indicating that the call is completed.

The *move* from a document F_0 to another document F_1 as the result of a call to and a return from service f are written $F_0 \vdash_{\text{func}}^{f,\text{call}} F_1$ and $F_0 \vdash_{\text{func}}^{f,\text{ret}} F_1$, respectively, and we simply write $F_0 \vdash_{\text{func}}^{f} F_1$ to refer to any of the above two cases.

B. External services as interfaces

We aim at distributing active XML, and to this extend, distant services should be perceived only through the functionalities they provide. This notion is captured by *interfaces*. Figure 6 depicts an interface for the DellOrder service. This interface can be seen by the customer, who can thus say something about 1/ what parameters she must submit to the service call, and 2/ what are the possible returns of the service.

Definition 2: An interface *is a pair* $(I^{\mathrm{C}}, \{I_i^{\mathrm{R}}\}_{i\in 1..n})$ *consisting of a* call *and a set of* return patterns*, possibly enhanced with* constraints *over variables of* I^{C} *and* $I_i^{\mathrm{R}}, i \in 1..n$.

Interfaces can be seen as descriptions of distant services. When such services are not explicitly provided, it seems interesting to study the evolution of a system under any implementation. This is captured as follows. Document F_0 can *move* to document F_1 under interface

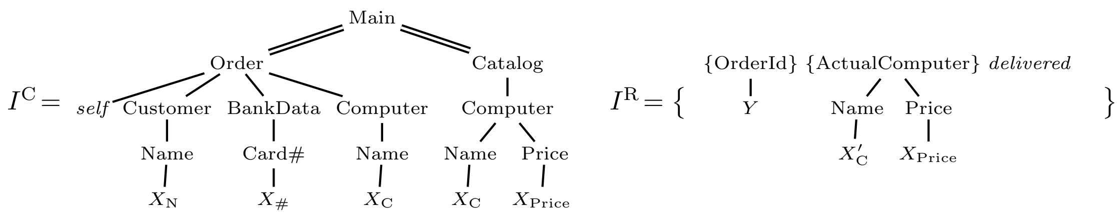

Figure 6. $I_{DellOrder}$, external view from customer as an interface, with its assertion $[X'_C = X_C]$.

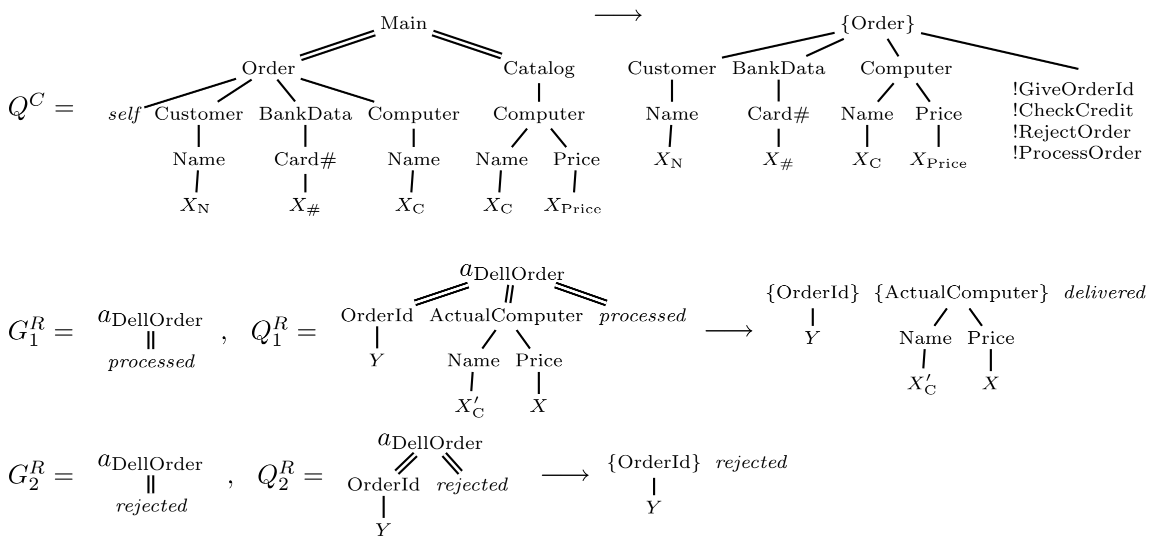

Figure 7. DellOrder, implementation by Webstore.

$I = (I^{C}, \{I_i^{R}\}_{i\in 1..n})$, written $F_0 \vdash^{I}_{\text{intf}} F_1$, if 1/ F_0 satisfies I^{C}, and 2/ the increment $F_1 \setminus F_0$ satisfies some $I_i^{R}, i \in 1..n$.

In the following, we will also use interfaces to depict the functional needs of some peer, and relate local needs to a distant implementation (an access to a local function provided by another peer). However, we need to ensure that a service provides a correct implementation before connecting an interface and a function. This is formalized under the notion of *implementation*:

Definition 3: A function f implements *an interface I, written $f \models I$ iff 1/ for every document F in which I^{C} holds, there is a matching from the body of the call query Q_f^C of f to F, and 2/ every document returned by f satisfies at least one return pattern I_i^{R}. Assertions of I must be satisfied as well, if any.*

Intuitively, condition 1/ means that when parameters of a call satisfy the call pattern of an interface, then they will be correct inputs to a service, and condition 2/ means that all returned values are compatible with the expected results specified in the interface. The function in Figure 7 is not an implementation of the interface in Figure 6, as the rejection case is not considered by the interface. Observe that interfaces involve no guard, and that relation $f \models I$ does not involve the call guard of f; the reason for this is that guards must be tested at the peer owning the service.

Relation $\models$ is formally defined in [13], where it is shown that it can be brought back to two instances of a *query containment* [18], [15], plus an instance of pattern matching problem. Implementation relation $\models$ is shown to be decidable and of complexity Co-Nexp-Time under some assumptions on the domains of variables and on the type of constraint used in the interface.

C. The Distributed Active XML model

An Active XML *system* $\mathcal{S}$, as described in [12] consists of a set $\mathcal{P}$ of *peers*, a set Φ_{int} of functions — the internal services — and a set Φ_{ext} of interfaces — the external services — both located on one of the peers. For each *initial document* F_0, $\mathcal{S}$ gives raise to a set of possible *runs*, which are the possible sequences $F_0 \vdash F_1 \vdash \ldots$ of documents obtained by performing moves under calls to internal and external services contained in F_0, and to the new service calls that result, inductively.

AXML systems do not consider the effect of distribution, and external services are just underspecified (via interfaces in our definition, and via DTDs in the original AXML model). In particular we have to describe the effect of a call to a distant service that implements an interface. We first enhance our previous notion of AXML system to take distribution and implementation into account, and then consider the semantics of DAXML.

Definition 4: A Distributed AXML system *(DAXML system)* $\mathcal{S} = (\mathcal{P}; \Phi_{\text{int}}, \Phi_{\text{ext}}; \mathcal{L}; \gamma)$ *is a tuple comprising:*

- *a set $\mathcal{P}$ of peers,*
- *two disjoint sets Φ_{int} and Φ_{ext} of functions and interfaces,*

- *a* location map $\mathcal{L} : \Phi_{\text{int}} \cup \Phi_{\text{ext}} \mapsto \mathcal{P}$*, and*
- *an* implementation map $\gamma : \Phi_{\text{ext}} \mapsto \Phi_{\text{int}}$ *which is a partial function such that* $\gamma(I) \Vert\!\models I$ *for each interface* $I \in \Phi_{\text{ext}}$*.*

It is required that $\mathcal{L}(\gamma(I)) \neq \mathcal{L}(I)$*, reflecting that service interfaces are used by distant peers other than the one owning the service.*

DAXML systems compose as follows: $\mathcal{S}_1 \parallel_\xi \mathcal{S}_2$ *is given by:* $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$*,* $\Phi_{\text{int}} = \Phi^1_{\text{int}} \cup \Phi^2_{\text{int}}$*,* $\mathcal{L} = \mathcal{L}^1 \cup \mathcal{L}^2$*,* $\Phi_{\text{ext}} = \Phi^1_{\text{ext}} \cup \Phi^2_{\text{ext}}$*, and* $\gamma = \gamma^1 \cup \gamma^2 \cup \xi$*, where the* pairing map

$$\xi : \left(\Phi^1_{\text{ext}} \cup \Phi^2_{\text{ext}}\right) \setminus \left(D(\gamma^1) \cup D(\gamma^2)\right) \mapsto \Phi^1_{\text{int}} \cup \Phi^2_{\text{int}}$$

assigns interfaces *of one system to* functions *of the other system* implementing *them. The parallel composition is only defined if* $\mathcal{L}^1$ *and* $\mathcal{L}^2$ *agree on the common part of their respective domains, and similarly for* γ^1 *and* γ^2*.*

Distributed documents: Documents are distributed over peers: a document is thus a forest F, and there exists a map ℓ such that $\ell(F) \in \mathcal{P}$ localizes each tree F of forest F on a peer. Distribution of documents is an important issue, as a peer should not have direct access to data or services owned by another peer. In the original AXML model, services are called and guards are evaluated over a global document, regardless of who owns the data. We propose an interpretation of distribution that allows access to remote data and services only via distant service calls. Distributed AXML documents located on a peer p contain references to services provided by p (nodes with labels of the form $!f$ with $f \in \mathcal{L}^{-1}(p) \cap \Phi_{\text{int}}$) and references to interfaces (nodes with labels of the form $!I$ with $I \in \mathcal{L}^{-1}(p) \cap \Phi_{\text{ext}}$). Each peer knows exactly which service on which distant peer implements its external functions.

The first easy cases are local service and interface moves. A local call to a service or interface owned by a peer p only modifies the part of the distributed document owned by p. Let us call this part of the distributed document F_p. Then, there is a local move $F \vdash F'$ from document F to document F' iff there is a move from F_p to some F'_p as defined in sections IV-A and IV-B, and the rest of the document located on other peers does not change (i.e. $F \setminus F_p = F' \setminus F'_p$). We only allow local interface moves for interfaces that are not implemented.

The remaining case is then distant service calls. An *external call* to an implemented interface $I = (I^{\text{C}}, \{I^{\text{R}}_i\}_{i \in 1..n})$ can be performed if a peer p owns a node labeled by $!I$. Calling I consists in replacing label $!I$ by $?I$ in F_p, and appending to the peer that implements I (say with a service g) a new tree with root a_I, that contains a node labeled $!g$ and the smallest set of subtrees of F_p where I^{C} holds. This can be considered as sending to a peer a request to execute g with some parameters, and creating a *distant workspace*. Note that this move is only allowed for implemented interfaces.

A return from a an external service call can be performed only when the distant service has been completed, that is the previously created distant workspace contains a node labeled g, with no ! or ? mark. The computed result is deleted from the callee's document, and appended as a sibling of the external calling node in the caller's document. The label of the calling node is then set to I.

We will not give more details on the semantics of DAXML, and refer to [13] for a complete operational description. In the rest of the paper, we will simply write $F \vdash F'$ and say that there is a *move* from F to F' when F' is a distributed document obtained after executing some operation (internal/external call or return) from F. We say that F is *deadlocked* if there exists no F' such that $F \vdash F'$. A *run* of a system $\mathcal{S}$ from a distributed document F_0 is an infinite sequence $\rho = F_0, \ldots, F_n, ..$ of instances over $\mathcal{S}$, such that for every i, either $F_i \vdash F_{i+1}$ or F_i is deadlocked and $F_i = F_{i+1}$. We denote by $\mathcal{R}uns(F_0, \mathcal{S})$ the runs of $\mathcal{S}$ starting at F_0.

Locality in DAXML: Let $\mathcal{S}$ be a DAXML system and $\mathcal{R} \subset \mathcal{P}$ be a subset of its set of peers. One can *restrict* $\mathcal{S}$ to $\mathcal{R}$, written $\mathcal{S}_{|\mathcal{R}}$, by simply restricting the set of functions and interfaces to those located on $\mathcal{R}$, and restricting the location map accordingly.

Similarly, for F a distributed document for $\mathcal{S}$, we can restrict F to its subset of trees located on $\mathcal{R}$, written $F_{|\mathcal{R}}$. Then, a run $\rho = F_0, F_1, \ldots$ of $\mathcal{S}$ can be *projected* over $\mathcal{R}$, written $\Pi_{\mathcal{R}}(\rho)$, by 1/ considering the sequence of restrictions $F_{0|\mathcal{R}}, F_{1|\mathcal{R}}, \ldots$, and, 2/ only keeping, in the so restricted run, changes that are local to $\mathcal{R}$. For $\mathcal{R}$ a singleton $\{q\}$, $\Pi_q(\rho)$ records what can be seen from peer q in run ρ. Local runs satisfy the following property for every system $\mathcal{S}$ and distributed document F_0 [13]:

$$\Pi_{\mathcal{R}}(\mathcal{R}uns(F_0, \mathcal{S})) \subseteq \mathcal{R}uns(F_{0|\mathcal{R}}, \mathcal{S}_{|\mathcal{R}})$$

which expresses that the local view, by a subset of peers, of any run of the overall system, conforms with the set of runs specified by replacing distant calls by their interfaces. This property is important, as it allows to check locally for safety properties such as the absence of a bad pattern in any run of an AXML system.

V. Issues of decidability and complexity

Issues of decidability and complexity impacted our design choices in the following aspects of our model: how interfaces should be defined, and what kind of property can be checked on a DAXML system. We begin with the notion of interface and study the implementation relation.

A. Complexity of the implementation relation

The first question is that of the complexity of pattern matching $F \models P$. Obviously, testing if $T \models P$ is an NP-complete problem as soon as variables are used (which allows encoding a SAT problem). We can reduce complexity of pattern matching, and then, of implementation, by stating assumptions regarding variables or the shape of patterns. The *child-only* assumption holds when a pattern does not use descendant edges. The *finite-domain* assumption holds when variables range over finite domains.

The following result is proved in [13]: let I be an interface and f be a function which variables range over finite domains, and such that additional constraints do not involve at the same time variables of I^{C} and $\{I_i^{\mathrm{R}}\}_{i\in 1..n}$ (that is we forbid expressions of the form $X < Y$, $(X = \alpha) \implies (Y = \beta)$, etc. when X is a variable of I^{C} and Y a variable of I_i^{R} for some $i \in 1..n$). Then, deciding whether $f \models I$ is a Co-NexpTime problem. However, without the finite domain assumption, or with generic constraints on variables $f \models I$, becomes an undecidable problem.

Observe that our implementation relation (definition 3) is not tractable with assertions such as the side conditions of functions and interfaces in the examples of figures 6 and 7. The reason is that considering them would immediately lead to undecidability of the implementation relation — this fact is explained in the next paragraph. However, it is still possible to leave the assertions aside when verifying implementation, and to handle them separately (for instance at runtime).

B. Reachability and Tree-LTL logic

It is shown in [13] that it is undecidable whether a pattern P is reachable by some run of a DAXML system $\mathcal{S}$ starting from a given initial document F_0. This undecidability result is not related to distribution; it holds already with a single peer and internal services only [12]. It can be avoided if negative patterns are forbidden in guards (they are considered in [13], but we did not allow them for simplicity) and if unbounded recursive calls between functions are forbidden too.

A consequence is that some restrictions on assertions are also needed to keep implementation decidable. Consider for instance the assertion $[X'_{\mathrm{C}} = X_{\mathrm{C}}]$ in figures 6 and 7. Ensuring it is a reachability problem as it amounts to requiring that X'_{C} can reach a prescribed value in some run. This is undecidable unless stringent assumptions are made, as we have seen. Hence, allowing any type of assertion in the implementation relation leads to undecidability.

More complex properties on the runs of DAXML systems can be formulated using formulas from the logic Tree-LTL [12], defined by the following grammar:

$$\phi ::= \mathit{Qpattern} \mid \phi \wedge \phi \mid \neg\phi \mid \phi\,\mathcal{U}\,\phi \mid \bigcirc\phi,$$

where *Qpattern* is a Qpattern, i.e., a pattern P in which some variables are declared *free* and universally quantified over the formula, and $\wedge, \neg, \mathcal{U}, \bigcirc$ have the usual meaning of LTL. It is shown in [13] that our new notions of interface and DAXML system keep complexity and decidability issues unchanged as compared to the original results of [12] concerning guarded AXML: distribution comes for free.

VI. Conclusion and perspectives

This paper has introduced an XML based distributed framework for services choreographies, thus supporting information and workflow management in a unified framework. We have defined a rich notion of service interface for this framework. Distant calls via interfaces ensure interoperability between agents of a system, but also encapsulation of data and services in peers. Interfaces are useful to analyse partially specified designs where some implementations are only known via their interfaces. We see this as a promising research direction.

Our model relies on a querying mechanism that uses tree patterns (a fragment of Xpath). However, AXML can easily be adapted to other query mechanisms (possibly with different decidability issues). Finally, let us mention that an experimental AXML platform has been developed by the Active XML community (`www.activexml.net`). This implementation relies on standards for web services and is then open to all systems implementing these standards.

Acknowledgment: We are indebted to Serge Abiteboul, Victor Vianu, and Luc Segoufin, for fruitful discussions on this work.

References

[1] J. Wang and A. Kumar, "A framework for document-driven workflow systems," in *Business Process Management*, 2005, pp. 285–301.

[2] W. van der Aalst and K. van Hee, *Workflow Management: Models, Methods, and Systems.* MIT Press, 2002.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, I. Thatte D. (Editor), Trickovic, and S. Weerawarana, "Business Process Execution Language for Web Services. [BPEL4WS.] version 1.1."

[4] J. Misra and W. Cook, "Computation orchestration," *Software and Systems Modeling*, vol. 6, no. 1, pp. 83–110, March 2007.

[5] S. Ceri, P. Grefen, and G. Sanchez, "Wide: A distributed architecture for workflow management."

[6] D. Chan, J. Vonk, G. Sanchez, P. Grefen, and P. Apers, "A specification language for the wide workflow model," in *symposium on Applied Computing (SAC)*, 1998, pp. 197–199.

[7] A. Nigam and N. Caswell, "Business artifacts: An approach to operational specification," *IBM Systems Journal*, vol. 42, no. 3, pp. 428–445, 2003.

[8] R. Hull, "Artifact-centric business process models: Brief survey of research results and challenges," in *On the Move to Meaningful Internet Systems (OTM) Conferences*, ser. Lecture Notes in Computer Science, vol. 5332. Springer, 2008, pp. 1152–1163.

[9] S. Abiteboul, L. Segoufin, and V. Vianu, "Modeling and verifying active XML artifacts," *IEEE Data Eng. Bull.*, vol. 32, no. 3, pp. 10–15, 2009.

[10] "The Extensible Markup Language (XML) 1.0 (2nd Edition)," `http://www.w3.org/TR/REC-xml`.

[11] S. Abiteboul, O. Benjelloun, and T. Milo, "The active XML project: an overview," *VLDB J.*, vol. 17, no. 5, pp. 1019–1040, 2008.

[12] S. Abiteboul, L. Segoufin, and V. Vianu, "Static analysis of active XML systems," in *Symposium on Principles of Database Systems (PODS)*, 2008, pp. 221–230.

[13] L. Hélouët and A. Benveniste, "Distributed active XML and service interfaces," INRIA, RR no 7082, 2009.

[14] C. David, "Complexity of data tree patterns over XML documents," in *Mathematical Foundations of Computer Science (MFCS)*, ser. Lecture Notes in Computer Science, vol. 5162. Springer, 2008, pp. 278–289.

[15] G. Miklau and D. Suciu, "Containment and equivalence for a fragment of xpath," *J. ACM*, vol. 51, no. 1, pp. 2–45, 2004.

[16] R. Kapuscinski, R. Zhang, P. Carbonneau, R. Moore, and B. Reeves, "Inventory decisions in Dell's supply chain," *Interfaces*, vol. 34, no. 3, pp. 191–205, 2004.

[17] W. W. W. Consortium, "Xquery 1.0: An XML query language," W3C, Tech. Rep., Jan. 2007, D.Chamberlin, A. Berglund, S. Boag, et. al., Editors.

[18] T. Schwentick, "Xpath query containment," *SIGMOD Record*, vol. 33, no. 1, pp. 101–109, 2004.