The AXML Artifact Model

Serge Abiteboul, Pierre Bourhis, Alban Galland, Bogdan Marinoiu

The AXML Artifact Model[†]

(Invited Paper)

Serge Abiteboul*, Pierre Bourhis, Alban Galland, Bogdan Marinoiu
INRIA Saclay & LRI, University Paris Sud
Orsay, France
Email: firstname.lastname@inria.fr
* *Also at LSV, Ecole Normale Supŕieure, Cachan*

***Abstract*—Towards a data-centric workflow approach, we introduce an *artifact model* to capture data and workflow management activities in distributed settings. The model is built on Active XML, i.e., XML trees including Web service calls. We argue that the model captures the essential features of business artifacts as described informally in [1] or discussed in [2]. To illustrate, we briefly consider the *monitoring* of distributed systems and the *verification* of temporal properties for them.**

***Keywords*-Active XML, database, workflow, artifact, documents**

I. Introduction

The evolution of shared data is at the center of most human activities. The novel notion of *business artifact* [1] has been proposed to specify such evolution. The main idea is to capture both the flow of control (workflow) of the application but also data evolution (data cycle). See [2] for a brief survey on the topic and research directions. In the spirit of this approach, we assume as a context a network of autonomous systems evolving and interacting by exchanging data. Building on Active XML (AXML for short) [3], we propose a new artifact model. The thesis is that AXML is a proper formal foundation for such an artifact model.

Workflow and database systems are two essential software components that often have difficulties interoperating. Data-centric workflow systems are meant to integrate both the control of workflows and the data of databases. They allow the management of data evolution by tasks with complex sequencing constraints as encountered for instance in scientific workflow systems, information manufacturing systems, e-government, e-business or healthcare global systems. We follow here a data-centric workflow approach where both data and tasks, and also the "actors" (humans, processes, systems) are captured by objects called *artifacts*. Artifacts present the following facets that, in our opinion, have to be captured by an artifact model:

State An artifact is an object with a universal identity (e.g., URI). Its state is self-describing (e.g., XML data) so that it may be easily transmitted or archived. It has a host that is a peer or another artifact.

Evolution An artifact is created, evolves in time (possibly space), hibernates, is reactivated or dies according to a logic that is specified declaratively. Its evolution may be constrained to obey some laws, e.g. some workflow.

Interactions An artifact interacts with the rest of the world via function calls (e.g., Web services) both as a server and a client. An artifact provides for communications, storage and processing for the artifacts it hosts.

History As in scientific workflows, an artifact has a history with time and provenance information that may be recorded and queried.

These requirements have been in part motivated by [4].

We claim that the AXML model [3] provides a proper foundation for an artifact model. An AXML document consists of an XML document with embedded function calls. The calls may be activated from inside (the artifact as a client) and then receive answers in push or pull mode. The calls may also be activated from outside (the artifact as a server). Rules are used to specify the logic of functions, declaratively [5].

In this paper, we introduce the AXML artifact model and illustrate its use with recent works on distributed system monitoring and verification. The paper is organized as follows. In Section II, we present the model. We briefly discuss monitoring in Section III and verification in Section IV. The last section is a conclusion. To conclude this section, we present a motivating example and mention related works.

Example: To illustrate, we use as a running example, a simplified view of the Dell manufacturing system [6]. See Figure 1. When a new Web order arrives (1), a new *webOrder* artifact is created. Then the new artifact creates a subartifact that is sent to a credit service (2). Once credit has been approved, the subartifact returns to the *webOrder* but now its state contains all the credit data (and notably the fact that the credit request was successful). A plant is then selected and the artifact moves to that plant (3). It initiates a new subartifact for gathering parts, that is sent to a warehouse and another local artifact for communications

[†] This work has been partially funded by the Advanced European Research Council grant Webdam and the ANR grant Docflow.

```
< plant artID="plant02" >
...
< webOrder artID="wo3" >
      < client >
        < name > Sue Leroux < /name >
        < address > ... < /address >
      < /client >
      < order > ... < /order >
      < order > ... < /order >
   < creditApproval artID="wo3-ca" >
     ...
   < /creditApproval >
   < fun funID="?warehouseOrder"/ >
   < fun funID="?comm"/ >
< /webOrder >
...
< /plant >
```

Figure 2. An AXML artifact

with the customer (4). Once the product has been built, the artifact is sent to a delivery service (5). Finally, once the Web order has been completed, the artifact moves to an archive where it is stored as a text-based XML serialization that includes all the information it has gathered during its life cycle (6). (Subartifacts may also be archived separately.)

The state of an artifact is an AXML document. See Figure 2 where the tree is represented using an XML syntax (a text-based serialization of the tree). The figure shows part of a *webOrder* artifact immediately after it enters the plant. The *creditApproval* element denotes a subartifact (the one that has been processed by the bank). The functions *?warehouseOrder* and $?comm$ will be activated next in order to create the *warehouseOrder* and *communication* subartifacts that will then work concurrently (and somewhat autonomously). Observe that we have simplified here the exact AXML syntax. For instance, the artifact IDs are more complex. (They include the URI of the peer where the artifact has been created and an identifier within this peer.)

Related work: Although the notion of artifact has been recently articulated by [1], similar ideas of data centric workflows have been around, e.g., in AXML [3], in the Vortex system [7] or scientific workflows [8]. The models that are considered are often restricted, e.g., [9], [4]. For instance, a single artifact is usually considered, vs. a system of artifacts in the present paper. Also, these models are often based on the relational model so have difficulties with the collections of artifacts or nested tasks/artifacts. Formal models for data-centric workflows have been considered in [10] (that focuses on verification) and [11] (that discusses the synthesis of artifacts).

II. The model

In this section, we present the *AXML artifact model*. We assume the existence of infinite alphabets: $\mathcal{P}$ for peer names; $\mathcal{A}$ for artifact identifiers; $\mathcal{C}$ for class names; $\mathcal{S}$ for stage names; $\mathcal{L}$ for labels; $\mathcal{D}$ for constants (say strings).

Artifact tree: The identifier $\alpha \in \mathcal{A}$ of an artifact is a URI that it acquires at creation time. The artifact identified by α is an *object* that has an AXML document for state. In the following, we use α to denote the identifier as well as the object itself. We next modify the notion of active documents of [3] to meet the needs of an artifact model.

An (AXML) artifact tree is a tree with *nodes* in $\mathcal{N}$ defined as follows. A node in $\mathcal{N}$ is a pair (α, i) where α is the ID of the artifact this node belongs to and $i \in \mathcal{D}$ a unique ID within this artifact. This guarantees that two distinct nodes in the system will not have the same identifier. The nodes in an artifact α are called α-nodes. Some nodes are labeled by element names in $\mathcal{L}$ (element nodes), some by constants in $\mathcal{D}$ (content nodes) and finally some by "function names" (function call nodes) in $\mathcal{F}$, that is defined next. A function name in $\mathcal{F}$ is an expression $f@\alpha$ where α is in $\mathcal{A} \cup \perp$ and f is in $\mathcal{D}$. For $f@\alpha$ with α in $\mathcal{A}$, we say that we call f at artifact α. For $f@\perp$, we say that we call the external service f (in practice, f is the URL of a Web service).

An artifact tree regroups several artifacts (on the same peer):

Definition: An *(AXML) artifact tree* is a finite, labeled, unordered, unranked tree with nodes in $\mathcal{N}$ and labels in $\mathcal{F} \cup \mathcal{L} \cup \mathcal{D}$ where nodes labeled by $\mathcal{D}$ are only leaves[1] and verifying:

> (+) If an α-node in the tree has a β-node as child, then this β-node is a maximum for all β-nodes in the tree and none of its descendants is an α-node.

If a β-node is a child of some α-node, we say that β is a *subartifact* of α, and this transitively.

Figure 2 illustrates these notions. The figure shows a possible state of a *Plant* artifact tree. Observe the $wo3$ *webOrder* artifact. Its ID is $wo3$. It contains some element nodes (*name*), some content node ("Sue Leroux"), some function nodes (a call to a $?warehouse$ function), and a subartifact ($wo3$-ca).

Artifact schema: We next define the notion of schema by enriching the schemas of Guarded AXML (GAXML for short) [5] to meet the needs of an artifact model. A schema imposes constraints on a system of artifacts or its evolution: (i) typing constraints (as in DTD or XSD); (ii) service signatures (as in WSDL); and (iii) dynamic constraints (as in BPEL). We next introduce our notion of schema based on different components that are detailed further.

Definition: An (*artifact*) *schema* $S = (\Pi, C, \tau, \omega, \theta, \iota)$ consists of:

- A finite set $\Pi = p_1, ..., p_K$ of peer names.

[1] In [5], function nodes are leaves. Here we will see that the function nodes have children that will in particular contain their "call guard" and "argument query". See further.

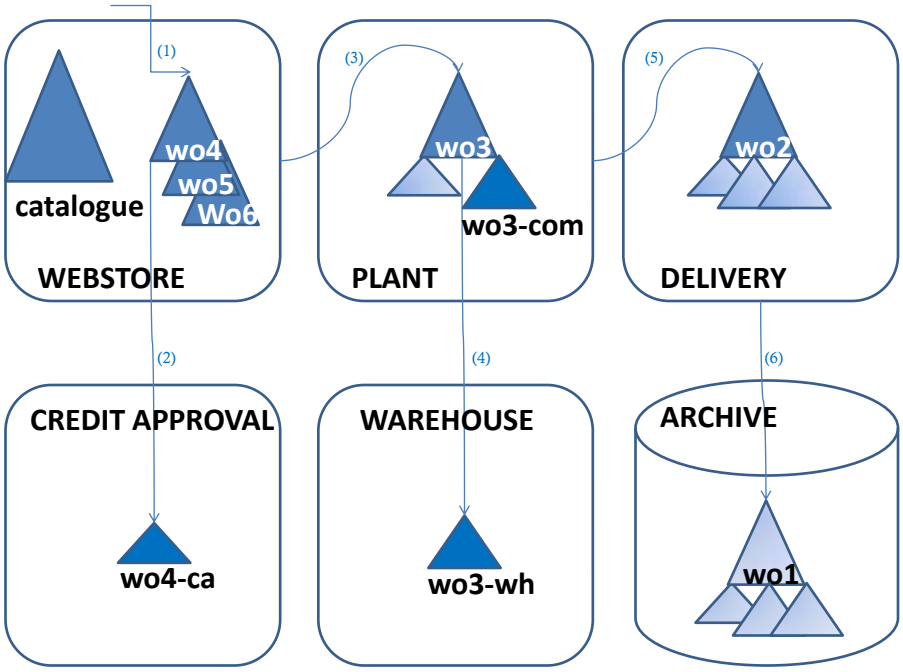


Figure 1. Artifacts in the Dell application

- A finite set $C = \{c_1, ..., c_N\}$ of (artifact) class names and, for each class c_i, $\tau(c_i)$ specifies a typing constraint on the state of c_i-artifacts.
- For each artifact class, the function θ specifies the semantics of internal functions. The function ι specifies the interfaces of the internal/external functions.
- The workflow function ω specifies constraints on artifact evolution (beyond those already specified by θ).

Peers: To simplify, we assume here that the set Π of peers is fixed. In general, it may vary when, during a run, we discover new peers. From an implementation viewpoint, a peer provides storage, communications and computing resources for the artifacts it hosts. As we will see, artifacts will be allowed to exchange data with other peers or to move to another peer. AXML data (e.g., in function arguments and results) are sent as strings and reconstructed at the receiving peer with new node IDs. On the other hand, when an artifact moves, its nodes preserve their IDs.

Classes: To simplify, we assume here that the set C of classes is fixed. In general, it may vary when we discover new classes of artifacts. To simplify again, we assume that the class of an artifact is an immutable property. For each class, the function τ specifies the type of the state of artifacts in this class. For each class, the artifacts in that class go through different *stages*. We use a variant of DTDs for unordered trees [5]. Let c_i be a class of artifacts. Then the *class typing* τ_i is a set $\{(s_{i1}, \delta_{i1}), ..., (s_{iM}, \delta_{iM})\}$ where each s_{ij} is one of the possible *stages* of class c_i, and each δ_{ij} is an unordered DTD that characterizes this particular stage. To discriminate between stages, we impose that for two distinct j, k, there is no data that satisfies both δ_{ij} and δ_{ik}.

Interface: The function ι defines the *interface* of artifacts. For each class c, $\iota(c)$ specifies the list of functions, artifacts in that class are willing to support, as well as the types of the input and output of these functions.

Note that the interfaces we consider here focus on signatures and are therefore rather simple. One could consider more complex interfaces, and in particular some that would include states. For instance, suppose we view *creditApproval* as an external service. Such a service may have in its interface the constraint that it will always refuse credit to a customer that has already been denied credit twice (unless some special procedure is performed). Such more complex interfaces may be captured as well by artifacts.

The function ι similarly defines the list of external functions and their input/output types ($\iota(\perp)$).

Function semantics: Artifacts evolve in time according to some "logic" specified declaratively. We assume here each artifact carries with it, its *logic*. (This is a departure from AXML where functions are not defined within a document but in their hosting peers. We will examine this issue further.) We first consider "internal functions", i.e., functions supported by the artifacts, then "external functions", i.e.,

those supported by external services. As in GAXML, there are 4 components in a function call:

Function call activation is controlled by a *call guard*.

Call argument is computed by an *argument query*.

The return of the result is controlled by a *return guard*.

The result of the call is computed by a *result query*.

Observe that the guards impose a control over runs of the system. They are specified using *Boolean combinations of tree-pattern queries* over the documents (BTPQ for short). More precisely, they are Boolean BTPQs, i.e., BTPQs with no free variables. The argument and return queries are also BTPQs but possibly with free variables that define their results. These are all queries that are evaluated *relative* to the function call node, respectively function return node.

An issue is where are these four components specified. Let f be a function defined in class c, i.e., each c-artifact α supports calls $f@\alpha$. Then $\theta(c, f)$ specifies the return guard and return query of function f in class c, i.e. for all such call $f@\alpha$. (In practice, the definitions of these two components are specified in the definition of the particular class of the receiver.)

On the other hand, the call guard and the argument query are chosen by the caller[2]. They are given as children of function call nodes. Note that one can therefore specify different call guards and argument queries for different calls of the same function, even in the same artifact. Note also that call guards allow in particular sequencing function calls. For instance, given two calls to functions f_1, f_2 in the same document, to force f_1 to be evaluated first, one may require in the call guard of f_2 that the result of f_1 has already been received.

To illustrate, consider the running example. One can specify that when a product has been ordered, a bill is issued to the customer. The guard may be that this product is available. In GAXML syntax, the query to compute the argument of the call may be as in Figure 3.

A last aspect of function in the AXML artifact model is that like AXML, AXML artifacts support both pull and push services. A pull service call is just a standard remote method call. A push service call is a subscription. It is typically answered by "yes". Then the server sends a stream of data possibly eventually closed by an *end-of-stream*.

Now consider external functions. From a peer viewpoint, there is no difference between calling an internal or an external function. Thus, a call to an external function also has a call guard and an argument query. On the other hand, the external service is outside the artifact realm. Thus the return guard is simply a constraint on the data that is returned and there is no return query. So an external function simply returns non-deterministically some data obeying typing constraints. Observe that one can use these external queries to model arithmetic functions (to ignore their semantics and avoid reasoning about arithmetic) or user inputs.

[2]In GAXML, all calls to a function f use the same call guard and argument query .

Workflow: For each class c_i, a *workflow* $\omega(c_i)$ imposes constraints on how an artifact in this class may evolve. This may be specified with triples $(event, precond, postcond)$. The key events that can be considered correspond to the activation of a function call at some peer, the reception of the call at some other peer, the sending of the result, and the reception of that result. Many high level events such as artifact creation, move or archiving may be seen as instances of function calls. The conditions $precond$ and $postcond$ are (BTPQ) formulas over the states of the artifacts (including their times and locations).

One could prefer a more standard workflow approach in the style BPEL. The workflow is then given by specifying admissible transitions between the stages. Suppose that an event e occurs that moves an artifact in class c_i from state σ to σ'. Then there must exist $(e, p, p') \in \omega(c_i)$ with $\sigma \models p$ and $\sigma' \models p'$. If this is not the case, a workflow constraint violation is detected.

In both approaches, the system should specify what to do when such a violation occurs, e.g., transaction abortion or roll back.

Artifact system and runs: An *artifact system* for a schema $(\Pi, C, \tau, \omega, \theta, \iota)$ is a function $\mathcal{I}$ that (i) maps each $p \in \Pi$ to an artifact tree $\mathcal{I}(p)$; and (ii) for $p \neq p'$, no artifact ID occurs in both $\mathcal{I}(p)$ and $\mathcal{I}(p')$.

An artifact system defines the set $art(\mathcal{I})$ of artifacts occurring in $\mathcal{I}$. Note that (ii) is some form of "non-ubiquity" condition for artifacts. Each is entirely in one host. On the other hand, an artifact may create a subartifact and "send" it out in the world, e.g., the *warehouseOrder* subartifact of a *webOrder*. The *localization* of an artifact α in $art(\mathcal{I})$, denoted $\lambda(\alpha)$, is the peer that it is included in.

A *run* of a system is a sequence $\mathcal{I}_j$ of systems, where each transition corresponds to some elementary operation, e.g. a function call. An interesting issue is the time of the transition. Time is always defined with respect to a peer, e.g., a *time* is a pair (p, t) where p is a peer ID and $t \in Q$. The order between these times, denoted $\prec$, is only partial. Consider for instance the activation of a function call. As already mentioned, this involves some synchronicity between the caller and callee. The transition happens at two simultaneous times, one for the caller p_1 and one for the callee p_2 and $(p_1, t_1) \equiv (p_2, t_2)$.

From a formal point of view, these exchanges and moves occur instantaneously. For instance, for a message from some peer p to some peer p', the message is sent and received simultaneously. Of course, this synchronicity does not correspond to the fundamentally asynchronous nature of the distributed world we model. To overcome this shortcoming, one can introduce a new peer, namely *Net*, to represent the communication network. So, for instance, to move from p_1 to p_2, an artifact will move (instantaneously)

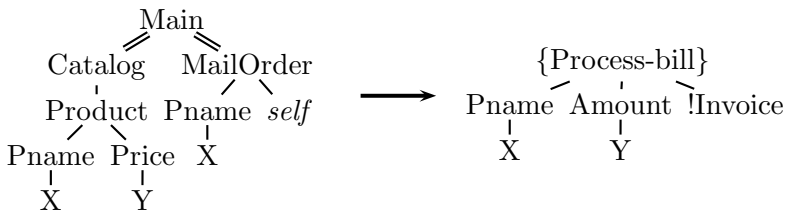


Figure 3. A GAXML query

from p_1 to *Net*, then later on move (instantaneously) to p_2. So, we will see two distinct times, $(p_1, t_1) \equiv (\mathit{Net}, t_1') \prec (\mathit{Net}, t_2') \equiv (p_2, t_2)$. Note that the inequality captures the (desired) constraint that a message arrives *after* it is sent.

We conclude this section with remarks on important aspects of the artifact model:

Tasks and services, states and stages, activities All these notions (in our opinion, sometimes considered in confusing ways) are formally captured in the artifact model. This will be best seen with our running example. A *Plant* artifact receives streams of *webOrder*s, a collection of tasks. To each of these tasks, corresponds an artifact for us. Different subtasks for a *webOrder*, e.g., *creditApproval*, *warehouseOrder*, are represented by subartifacts. This is where the natural nesting in the model comes in handy. In general, a task may require the collaboration of several artifacts. A *Plant* artifact provides a service (that corresponds physically to building the product). At some point in time, a *Plant* artifact may be processing a large number of *webOrder*s. The state of each *webOrder* is an (Active) XML document. Some of them may be in a *waitingForParts* stage, others in *inConstruction*, *waitingForTests*, etc.

The notion of *activity* often arising in functional decompositions of business processes, may be seen as a view over a system of artifacts. It may focus on several artifacts, e.g., the warehouse activity would group the artifacts involved in the warehousing task for a *webOrder*. It may also isolate a particular period of the process, e.g., the different stages in a product construction.

Time and provenance The system handles *time* and queries talk about it, e.g., which *webOrder*s took more than one week of processing and which *webOrder* were delivered before being paid? The system also handles *provenance* and queries talk about it, e.g., which *webStore* created a particular *webOrder* and which plant built the products for it?

Life and death An artifact may be created in a number of ways, e.g., from scratch, by cloning an existing artifact, or by activating some (passive) data. An artifact may hibernate temporarily or terminate. Typically some *archiving* may be performed at the end of the artifact life cycle or at particular milestones.

Asynchronous communications The system provides reliable communications, where an artifact can send a message to any other artifact just by knowing its ID. An issue is the localization of the destination artifact. This is straightforward to implement for artifacts that are anchored at a particular peer. For artifacts that move, the system is responsible for forwarding them messages. This is a standard issue in system with mobile objects, e.g., cell phones.

Where are the artifact rules? The fact that an artifact (in an object-oriented style) carries its own logic is a departure from AXML where the rules that define local functions are installed at the hosting peer. Observe that this does not imply that the artifact has to have some deep knowledge of the organization of its host. For instance, an artifact entering the *webStore* does not have to know where the catalog is located; it only has to know the name of the function to obtain information from the catalog. A more serious issue is that this may be viewed as some security violation since the peer is going to run the rules of the artifact inside its own environment. We believe that this is the standard cost (as in Java applets) we have to pay for the flexibility of the approach. A peer can always deactivate an artifact it is hosting or control it very tightly (e.g., in a sandbox).

III. Monitoring

One of the problems we studied is the monitoring of distributed systems based on the notion of artifacts. In this section, we briefly mention this work to illustrate the use of artifacts. A fundamental issue in that setting is that we do not see all that is going on inside the artifact. We may not even see all the communications. Hence we have to handle incomplete information in particular for query processing.

We have developed the P2PMonitor monitoring system [12]. Both the monitored and the monitoring systems are P2P. We have used the Dell manufacturing system [6] in a

demo [13] to validate these ideas with *webOrder*s modeled as AXML artifacts.

Functionalities: In general, the surveillance of artifact in our system is based on the following functionalities. The *alerters* are in charge of the detection of basic events in the systems we monitor (e.g., sending/reception of a function call, sending/reception of result). The *stream processors* process streams of alerts and evaluate on-line queries. *Repositories* store surveillance data and maintain access structures such as indexes or materialized views. And, finally, *XML processors* process this surveillance data and evaluate off-line spatio-temporal queries.

We next briefly discuss interesting issues encountered when developing this system.

The alerters: AXML peers can provide self-monitoring functionalities for the artifacts they host. In these systems, monitoring may be run simultaneously with the application logic (if there is no risk to interfere, e.g., with the application performance). However, most of the systems we want to monitor are not AXML. For such a non-AXML system, we have limited access to the data and control of the system. We can observe (some of) its interactions with the rest of the world. This may be achieved e.g., by installing a "spy" on its Web server that observes the service calls and responses of the Web server. For database systems, an alerter may rely on the triggering system of the database or on the surveillance of update calls to the database. For instance, in P2PMonitor, we have developed a Webserver alerter and an eXist-database alerter.

Stream processing: Stream processing is needed for gathering and organizing the desired traces of runs and for on-line queries. The techniques we use for stream processing are described in [14]. The central component is the *axlog widget* defined by tree-pattern queries over active documents (in the AXML style) that include some input streams of updates. A widget generates a stream of updates for each query, the updates that are needed to maintain the view corresponding to the query. To support widgets we use a novel algorithm [14], [15] based on datalog technology (Differential or MagicSet) and efficient XML automata filtering (YFilter).

Localization of the artifacts to monitor: This is an important aspect in monitoring. Different styles of localization may be considered:

- Static Monitoring: We know in advance the peers to monitor and their location as, e.g., in the Dell Supply Chain demonstration [13].
- Self-monitoring: An AXML artifact does itself the monitoring so localization is not an issue.
- Monitoring by contamination: When an artifact moves to a different location (say a new peer), its monitor sends to the monitor of the new location all the indications needed for the monitoring of the artifact to continue. (It may have to "install" a monitor at the new location if none exists.)
- Primary-site-driven monitoring: In the style of the MobileIP protocol [16], a primary site of the artifact is responsible for always knowing the location of the artifact (e.g., the artifact has to inform it) and installing monitoring on the host of the artifact (when possible).

IV. Verification

In [5], the analysis of the behavior of GAXML systems, and in particular the verification of temporal properties of their runs, are considered. For instance, one may want to verify whether some static property (e.g., all ordered products are available) and some dynamic property (e.g. an order is never delivered before payment is received) always hold. The temporal properties are specified in Tree-LTL, a data-tree-based temporal logic that allows expressing a rich class of temporal properties. Two examples of tree-LTL formulas are given in Figure 4.

The analysis establishes the boundary of decidability of satisfaction of Tree-LTL properties by GAXML systems. In particular, decidability is obtained by disallowing recursion in GAXML systems, which leads to a bound on the number of total function calls in runs.

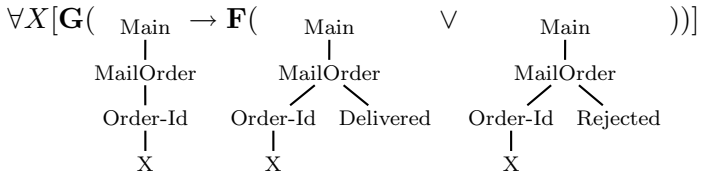
Although recursion-free GAXML is a rather restricted language for specifying artifacts, it already captures some essential aspects of the control of applications in presence of data.

V. Issues and on-going work

There has been a number of works around AXML. See [3]. We mention the ones that are most relevant to the discussion and discuss issues.

1) *Model:* The present paper is an attempt to specify a model of artifacts based on AXML. This is on-going work.
2) *Monitoring, tracing, querying history:* We mentioned works in that direction in Section III.
3) *Verification:* We mentioned works in that direction in Section IV. The problem is also studied in [17], [18]. An interesting direction is to enrich (beyond GAXML) the class of system that may be verified.
4) *Time:* an issue is the absence of global clock. So reasoning about time is more complicated.
5) *Interface:* in general, we don't know the internal logic of some artifacts or that of some tasks performed on another peer. We need to be able to describe abstractly such behaviors. There is some on-going work on interfaces in the context of AXML [19].
6) *Access control:* we are working on access control mechanisms in such a distributed environment, where partners want to keep control over their own data.

Every mail order is eventually completed (delivered or rejected):



Every product for which a correct amount has been paid is eventually delivered (note that the variable Z is existentially quantified):

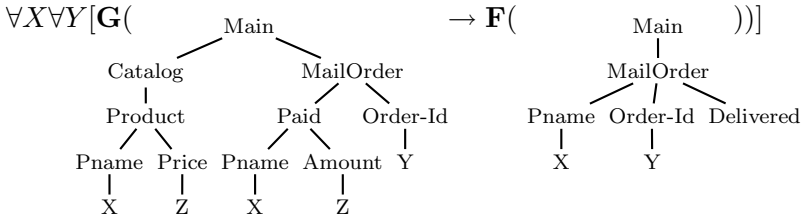


Figure 4. Tree-LTL formula

REFERENCES

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

[2] R. Hull, "Artifact-centric business process models: Brief survey of research results and challenges," in *OTM*, 2008.

[3] S. Abiteboul, O. Benjelloun, and T. Milo, "The Active XML project: an overview," *VLDB J.*, 2008.

[4] K. Bhattacharya, R. Hull, and J. Su, "A data-centric design methodology for business processes," in *Handbook of Research on Business Process Management*, J. Cardoso and W. van der Aalst, Eds. N.A., 2009.

[5] S. Abiteboul, L. Segoufin, and V. Vianu, "Static analysis of Active XML systems," in *PODS*, 2008, pp. 221–230.

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

[7] G. Dong, R. Hull, B. Kumar, J. Su, and G. Zhou, "A framework for optimizing distributed workflow executions," in *DBPL 1999*, 2000.

[8] S. Davidson and J. Freire, "Provenance and scientific workflows: Challenges and opportunities," in *ACM SIGMOD Int. Conf. on Management of Data*, 2008.

[9] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su, "Towards formal analysis of artifactcentric business process models," in *Int. Conf. on Business Process Management*, 2007.

[10] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu, "Automatic verification of data-centric business processes," in *ICDT*, 2009.

[11] C. Fritz, R. Hull, and J. Su, "Automatic construction of simple artifact-based business processes," in *ICDT*, 2009.

[12] S. Abiteboul and B. Marinoiu, "Distributed Monitoring of Peer to Peer Systems," in *Workshop On Web Information And Data Management*, 2007, pp. 41–48.

[13] S. Abiteboul, B. Marinoiu, and P. Bourhis, "Distributed Monitoring of Peer to Peer Systems (demo)," in *ICDE*, 2008.

[14] S. Abiteboul, P. Bourhis, and B. Marinoiu, "Efficient maintenance techniques for views over active documents," in *EDBT*, 2009.

[15] ——, "Satisfiability and Relevance for Queries over Active Documents," in *PODS 2009*, 2009.

[16] I. E. T. Force, "Rfc 3344, IP mobility support for IPv4," http://tools.ietf.org/html/rfc3344, 2002.

[17] S. Abiteboul, O. Benjelloun, and T. Milo, "Positive Active XML," in *ACM PODS*, 2004, pp. 35–45.

[18] B. Genest, A. Muscholl, O. Serre, and M. Zeitoun, "Tree pattern rewriting systems," in *ATVA, LNCS 5311*, 2008.

[19] A. Benveniste and L. Helouet, "Interface for Active XML," private communication.