Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 203 (2008) 3–24
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Merging Hierarchically-Structured Documents in Workflow Systems

Eric Badouel[1]

INRIA-*Rennes*
IRISA
Rennes, France

Maurice Tchoupé Tchendji[2]

IRISA
University of Rennes
Rennes, France

Abstract

We consider the manipulation of hierarchically-structured documents within a complex workflow system. Such a system may consist of several subsystems distributed over a computer network. These subsystems can concurrently update partial views of the document. At some points in time we need to reconcile the various local updates by merging the partial views into a coherent global document. For that purpose, we represent the potentially-infinite set of documents compatible with a given partial view as a coinductive data structure. This set is a regular set of trees that can be obtained as the image of the partial view of the document by the canonical morphism (anamorphism) associated with a coalgebra (some kind of tree automaton). Merging partial views then amounts to computing the intersection of the corresponding regular sets of trees which can be obtained using a synchronization operation on coalgebras.

Keywords: Context-Free Grammars, Coalgebras, Anamorphisms, Merging Structured Documents.

1 Introduction

The data perspective of a workflow management system puts emphasis on the flow of business documents between activities. A complex workflow system may consist of several subsystems distributed over a computer network. These subsystems can concurrently update partial views of the document. At some points in time we need to reconcile the various local updates by merging the partial views into a coherent global document. Merging of structured documents has already been considered

[1] Email: ebadouel@irisa.fr
[2] Email: mtchoupe@irisa.fr

doi:10.1016/j.entcs.2008.05.017

in previous studies on the structural merging of XML documents [23,8,9,21,4,24], on file synchronization [6], or on software merging [15,25]. However, the merge operation that we have in mind here is much simpler than these. One initiates a new activity (a case) in a workflow system by introducing a document (e.g., a form). During its lifetime within the system this document grows by incorporating new information until the processing of the case is complete and the document exits the system. Therefore, normally such a document has a finite lifetime and when updating such a document there is no need to erase information (In general, it would even be considered undesirable to do so). This situation greatly simplifies the merge operation in comparison to the existing strategies noted above, where conflicts during the merge operation essentially come from erasing or restructuring operations.

A hierarchical-structured document is intentionally represented as a tree decorated with attributes. Some of these attributes may describe the physical appearance of the document. The set of legal structures is given by a context-free grammar together with some constraints on attributes (that may be given, for instance, by the semantic rules of an attribute grammar). We forget about the attributes; these are related to semantic issues that can be treated independently of the purely structural aspects that we address in this article. Therefore, we do not consider any concrete syntax of the document, since these concrete syntaxes can also be given by specific attributes. Therefore, we forget about the terminal symbols of the grammars and end up with so-called abstract context-free grammars. These grammars may be identified with tree automata, thus characterizing legal documents as a regular set of (abstract syntax) trees. Under the mild assumption that any production is characterized by its left and right-hand side, abstract-syntax trees (trees labelled by productions) coincide with derivation trees (trees labelled by grammatical symbols).

We not only consider that documents flow in the system but also that some copies of a same document may simultaneously exist in different parts of the system and can asynchronously be manipulated by a set of independent actors. For instance, some actor, with a specific domain of expertise, can operate on the document through interfaces based on a domain-specific language corresponding to his particular role. There is no reason that these tools, that can have be developed independently, can be aware of the global grammar and can manipulate the document as a whole. Thus, each such actor operates on a distinct partial view of the whole structure of the document. In this paper we consider that a specific view (of the grammar) is given by a subset of grammatical symbols, namely those corresponding to syntactical categories that are meaningful in the corresponding domain of expertise. The partial view of the document interpreted as a derivation tree (or projection of the document according to the view) is the tree obtained by erasing invisible grammatical symbols while preserving its structure (a node in the projection is a successor of another node if and only if the same relation holds in the original document). Of course this notion of view is reminiscent of similar, and more involved, notions that were introduced in the context of database theory or more recently in the XML community [1,7]. In this context, views are often defined in terms of a set of operations on XML documents

[10] combining projection (by restriction to certain XML elements), selection (from the value of certain attributes) and swap (changing the order of appearance of certain elements). These XML views can be created using query languages like XSL [34] or XQuery [33] advocated by the World Wide Web Consortium. Since we have discarded attributes and, in particular, are not interested in the physical appearance of documents, the simple notion of view that we are considering fits perfectly our needs. We can relate the abstract view of documents used in this article to more concrete representations using bidirectional tree transformations [28,29,30].

We define a binary merge operator so that merging a finite set of partial views can be realized by an iterated application of that (commutative and associative) binary operator. Even if we assume that the set of all partial views should determine unambiguously the global document, there is no reason to believe that the binary merge operator produces a unique document; in general, it would even produce an infinite set of documents. It happens that the set of (legal) documents associated with a given partial view is a regular set of trees generated by a tree automaton starting from the partial view considered as the initial state. Such a tree automaton is presented as a coalgebra for a certain functor; the carrier of the (unique) fixed point of this functor is thus a coinductive data type whose elements can be interpreted as representations of a potentially-infinite sets of trees. We call the anamorphism associated with a tree automaton the canonical coalgebra morphism from the tree automaton, viewed as a coalgebra, to the terminal coalgebra. Then the set of trees recognized by a tree automaton from an initial state is encoded by the image of this initial state by the associated anamorphism. Merging partial views then amounts to computing the intersection of the corresponding regular sets of trees which can be obtained using a synchronization operation on coalgebras.

The structure of the document grows as it flows through the workflow system. Thus the tree representation of the document should contain open nodes, that are leaves from which the structure may evolve (the so-called buds of the tree). We deduce a natural order relation on documents where $t_1 \leq t_2$ if t_2 may be obtained from t_1 by replacing some buds of t_1 by trees, we then say that t_2 is an *update* of t_1. A document is complete if it contains no buds, it is a maximal element. Initially, we restrict our attention to complete documents and we address the problem of coherence of views: can we decide whether there exists some (global) document corresponding to a given set of partial views (as defined in Section 3) and in the affirmative, can we produce such a document? We provide a solution to that problem, in Section 4, based on an expansion algorithm which produces a representation of the (regular) set of documents associated with a given view. That representation is an arena, a coinductive data structure that we introduce in Section 2 to represent potentially-infinite sets of trees. We then adapt in Section 5 this expansion algorithm in the case of trees with open nodes so that it produces a representation of all possible updates of documents with the given partial view. Reconciliation of local updates may then be obtained by looking at a minimal element in the intersection of the expansions of the corresponding partial views. All algorithms are given in the functional language Haskell; a short appendix gives a set of Haskell notations that should be sufficient for understanding the code given in the article.

2 Trees and arenas

We introduce a coinductive data structure to represent potentially infinite sets of trees. Elements of that data type, defined as the fixed point of a functor, are called *arenas* by analogy with game theory: a tree is a member of an arena if it can be viewed as a strategy for the game defined by the arena. We show that we can decide whether a finitely-presented arena is empty. An arena presented by a coalgebra (a generator) together with an element of the carrier set of this coalgebra (a germ) is the image of the germ by the anamorphism associated with the generator (i.e., the unique coalgebra morphism from the generator to the terminal coalgebra). A coalgebra can be interpreted as a tree automaton and then the arena presented by a tree automaton and an initial state (the germ) is simply a representation of the regular set of trees generated by the tree automaton from the given initial state.

Definition 2.1 A **tree** over an alphabet $\mathbf{A}$ is a partial map $t : \mathbb{N}^* \to \mathbf{A}$ whose domain $Dom(t) \subseteq \mathbb{N}^*$ is a prefix closed set such that for all $u \in Dom(t)$ the set $\{i \in \mathbb{N} \mid u \cdot i \in Dom(t)\}$ is an interval $[1, \cdots, n] \cap \mathbb{N}$; integer n is the **arity** of node u. If $t_1, \cdots, t_n$ are trees and $a \in \mathbf{A}$ we let $t = a(t_1, \ldots, t_n)$ denote the tree t of domain $Dom(t) = \{\varepsilon\} \cup \{i \cdot u \mid 1 \leq i \leq n\,,\ u \in Dom(t_i)\}$ with $t(\varepsilon) = a$ and $t(i \cdot u) = t_i(u)$.

We choose Haskell, a lazy functional language, to present the various algorithms since the presented solution relies heavily on inductive data structures. Trees may be described in Haskell by the following data type definition

```
data Tree a = Node {top :: a, succ_ :: [Tree a]}
```

The polymorphic structure of an arena is then introduced as:

```
newtype Arena  a = Or  {unOr  :: [(a,Arenas a)]}
newtype Arenas a = And {unAnd :: [Arenas a]}
```

The first definition states the existence of an isomorphism

$$Arena\ a \;\cong\; [(a, Arenas\ a)]$$

given by the constructor *Or* and the selector *unOr*:

$$Or \quad :: [(a, Arenas\ a)] \to Arena\ a$$

$$unOr :: Arena\ a \to [(a, Arenas\ a)]$$

An arena `arena` represents the set of trees whose shape conforms to one of the patterns given in the corresponding list `unOr arena` (a disjunction). Each such pattern is a pair `(a,arenas)` made of an element `a` and a list of arenas (a conjunction). A tree conforms to that pattern if its root is labelled `a` and its subtrees conform to the respective patterns in the list `unAnd arenas`. The derived isomorphism

$$Arena\ a \cong [(a, [Arena\ a])]$$

shows that this data structure is the fixed point of functor `f a` where `f a x = [(a,[x])]` (using Haskell notations where `(a,x)` represents the cartesian product $a \times x$). The semantics of Haskell recursive datatypes, associated with regular functors, is given by their unique fixed-point in the category of pointed CPOs and strict continuous maps. Notice, however, that the functors for arenas has the form $\wp(F\ X)$ where the powerset monad, represented here by the list monad, represents a non-deterministic choice and $FX = A \times [X]$ is the functor whose initial algebra is the datatypes of trees, i.e. the inhabitants of the arenas. Thus, the corresponding coalgebras, which, as we see below, can be interpreted as special kind of tree automata, fit in the theory of systems as presented in [18] where monad $\wp$ and endofuctor F represent respectively, the non-deterministic branching and the transition type of these tree automata.

We graphically represent an arena as a tree with two kinds of nodes. An example of an arena is depiceted in Fig. 1 where the annotations (*) and (**) mean that the subtrees stemming from nodes with an identical annotations are the same. An "or" node represents an arena while an "and" node represents a list of arenas. An "or" node representing an arena `t = Or atss` has as many successors as there are elements `(a,ts)` in the list `atss`; the arc associated with this element is labelled `a` and the corresponding successor is an "and" node corresponding to the list `ts`. Thus, if the set `atss` is empty this node has no successor; and "or" with no successor represent an empty set of trees and we represent it by the symbol $\perp$. A list of arenas `ts` is associated with an "and" node whose outdegree is the length of the list. The i^{th} successor is the "or" node associated with the i^{th} element of the list. If this list is empty the node has no successor and is denoted $\top$. We say that a tree `t = Node`

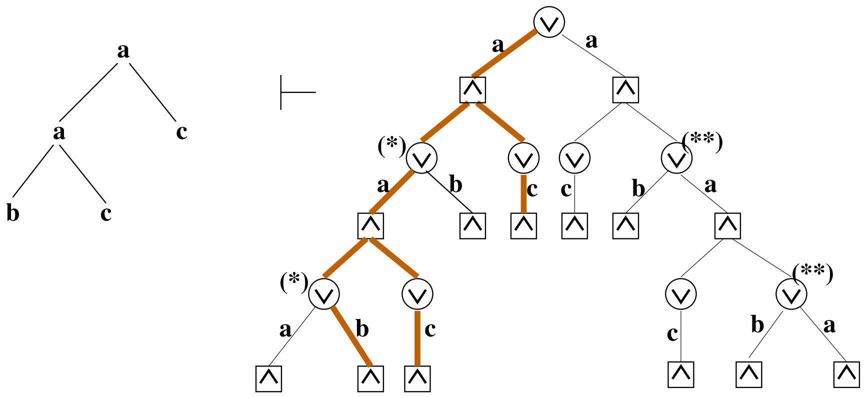

Fig. 1. An arena: a coinductive datatype representing a set of trees

`a ts` is a member of an arena if there exists, at the root, an arc labelled a leading to an "and" node whose outdegree coincides with the length of `ts` and, such that, recursively, each tree in the list `ts` is a member of the arena represented by the subtree rooted at the successor node of the "and" node with the same rank. This corresponds to the following function:

```
isMember :: (Eq a) => Tree a -> Arena a -> Bool
isMember (Node a ts) arena =
  or [and (zipWith isMember ts (unAnd arenas))|
                     (elet,arenas) <- unOr arena,
                      elet  == a,
                     (length ts)==(length (unAnd arenas))]
```

For instance, the arcs in bold face in Fig. 1 shows the membership of the tree

```
Node a[Node a [Node b [], Node c []],Node c []]
```

to the corresponding arena. We readily see in Fig. 2, the form of the trees in the arena of Fig. 1.

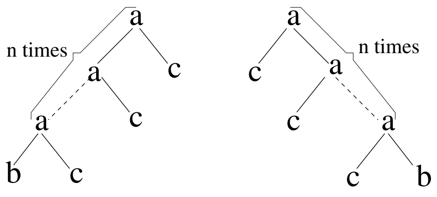

Fig. 2. Trees in the arena of figure 1

A (finite) tree that belongs to the set represented by an arena can be identified with the subtree of the arena induced by a set of nodes with the following property: it is a finite connected set of nodes containing the root and such that (i) every "or" node in this set admits exactly one successor of this node in this set, and (ii) every successor of an "and" node in this set also belongs to this set. We say that the tree is embedded in the arena and we identify the tree with its embedding. Similarly, we say that a tree is embedded in an arena at an "or" node if it is embedded in the arena represented by the tree rooted at this node. Intuitively, to embed a tree in an arena we start from the root of the arena and in each "or" node we choose an arc (whose label gives the label of the current node of the tree), and at each "and" node we visit in parallel all the successors of this node (each of them recursively provides the respective subtree of the embedded tree). We are only interested in finite trees, so this construction must terminate and, therefore, each such path should reach a node with no successor. These nodes cannot be "or" nodes because if such node has no successor there is no means to choose one of them at it is required by condition (i). The leaves of the embedded tree will therefore correspond to "and" nodes with no successor (the $\top$ nodes).

The interpretation of an "or" node (also called its extension) is the set of trees embedded from this node. The interpretation of an "and" node of outdegree n is the set of n-tuples of trees each of whose element is taken from the extension of the corresponding successor of this node. Therefore, if an "and" node contains a $\perp$ as a successor (i.e., an "or" with no successor) its extension will be empty and it will not contribute to the extensions of the nodes above it, and, in particular, to the root. Thus, we do not modify the extension of the arena by cutting off the arc leading to such a node. By doing so one can introduce a new node $\perp$. We, therefore, iterate this "cleaning" operation as long as there are no remaining $\perp$ node except maybe the root which is then the unique node representing an arena with an empty extension.

If the arena is finite one can enumerate its extension by the following function:

```
enumerate :: Arena -> [Tree a]
enumerate arena = [Node elet ts |
                       (elet,arenas) <- unOr arena,
                        ts <- dist (map enumerate (unAnd arenas))]


dist :: [[a]] -> [[a]]
dist [] = [[]]
dist (xs:xss) = [y:ys | y <- xs, ys <- dist xss]
```

and test the emptyness of this set

```
isEmpty :: Arena a -> Bool
isempty arena = and[or (map isEmpty (unAnd arenas)) |
                    (_,arenas) <- unOr arena]
```

A coalgebra for the functor `f a` is a map:

$$coalg : x \to [(a,\ [x])]$$

that can be interpreted as a (top down) tree automaton:

- a is the type of labels of the trees
- x is the type of states
- $q \to (A, [q_1, \cdots, q_n])$ is a transition of the automaton when the pair $(A, [q_1, \cdots, q_n])$ appears in the list $coalg\ \ q$.

To recognize a tree by a tree automaton starting from some initial state we proceed as follows:

- We associate the root of the tree with the initial state.
- If a node labelled A, associated with a state q, has n successors not yet associated with states, and if $q \to (A, \langle q_1, \cdots, q_n\rangle)$ is some transition of the tree automaton, then we associate each of its successor nodes with the respective states q_1 to q_n.
- The tree is recognized when we have associated each of its nodes with some state.

The anamorphism associated with the coalgebra is given as follows:

```
type Coalg a b = b -> [(a,[b])]
ana :: Coalg a b -> b -> Arena a
ana coalg gen = Or[(a,And (map (ana coalg) gens))| (a,gens)<-coalg gen]
```

An element in the carrier set of the coalgebra is a germ from which the anamorphism produces an arena. When a coalgebra is interpreted as a tree automaton `auto`, a germ is a state and the expression (`ana auto init`) is an arena whose extension is the set of trees recognized by the tree automaton from the initial state `init`. In the sequel we identify coalgebras and tree automata. For instance, the extension of the arena of Fig. 1 is the set of trees recognized by the tree automaton, with initial state q_0, and with the following transitions:

$$\begin{array}{llll} q_0 \rightarrow (a, [q_1, q_2]) & q_1 \rightarrow (a, [q_1, q_2]) & q_2 \rightarrow (c, [\,]) & q_3 \rightarrow (b, [\,]) \\ q_0 \rightarrow (a, [q_2, q_3]) & q_1 \rightarrow (b, [\,]) & & q_3 \rightarrow (a, [q_2, q_3]) \end{array}$$

One can decide whether the language of a tree automaton is empty if the set of states accessible from the initial state is finite by proceeding as follows. A *marking* of this set of states Q is a subset of states $m \subseteq Q$ such that for any transition $q \rightarrow (A, [q_1, \ldots, q_n])$, state q is marked as soon as each of q_1 to q_n is marked. The language of the tree automaton is non-empty if and only if the initial state belongs to the least marking m_{min}. More precisely, $m_{min} = \cup_{n \in \mathbb{N}} m_n$ where the sequence m_n is given by $m_0 = \emptyset$ and m_{n+1} contains those states q for which there exists a transition $q \rightarrow (A, [q_1, \ldots, q_n])$ such that every q_i in the right-hand side is in m_n. It immediately follows that $q \in m_n$ if and only if there exists a tree of depth less or equal to n recognized from state q. Of course we may, at each stage, mark only states that have not already be marked, which amounts to associating each state q with trees of minimal depth recognized from q.

We present a variant of the above classical marking algorithm in terms of the structure of arena. For that purpose we label each "or" node by the corresponding state of the tree automaton and we prune the resulting tree as follows. For each rooted path we consider the first occurrence of a node labelled by a state that already labels some node before it along this path (as the set of states accessible from the initial state is finite we can always find such a repetition at a depth bounded above by the cardinality of that set). Then we cut all arcs stemming from such a node which, thus, becomes $\perp$. Figure 3 gives the pruning of the arena of Fig. 1. In this figure we also have represented the cleaning operation (suppression of the internal $\perp$) applied after pruning.

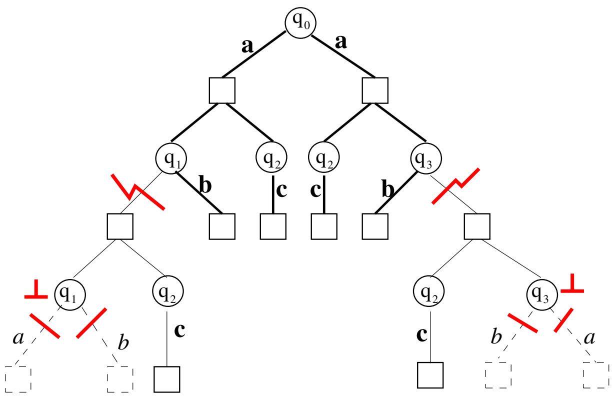

Fig. 3. Pruning of the arena of Fig. 1

After pruning we obtain a finite branching tree with no infinite branch, hence, a finite tree by Koenig lemma. Therefore, the functions `enumerate` and `isEmpty` previously introduced can be used. For instance, the extension of the arena of Fig. 3

is reduced to the following two trees:

$$Node\ a\ [Node\ b\ [\,],\ Node\ c\ [\,]]\ \text{and}\ Node\ a\ [Node\ c\ [\,],\ Node\ b\ [\,]]$$

Proposition 2.2 *An arena has an empty extension if and only if its pruning has an empty extension.*

Proof. On the one hand, it is clear that by pruning an arena one can only decrease its extension. We, therefore, have to check that the pruning of an arena with a non-empty extension has a non-empty extension. For that purpose we introduce a reduction operation that transforms any tree embedded in an arena into a tree embedded in its pruning. Let us, therefore, consider a (finite) tree embedded in an arena. We take the subtree rooted at a node marked $\perp$ by the pruning operation and we glue this tree in place of the subtree rooted at the node above along the same path that is labelled with the same state. As the arenas rooted from these respective nodes are identical, since they are generated from the same state, it follows that the new tree is still embedded in the original arena. By doing so the size of the tree has strictly decreased, therefore such transformations can only be applied a finite number of time leading to a tree embedded into the pruning of the arena. □

We present the following algorithm to decide the emptiness of the extension of a finitely-presented arena. It is an adaptation of the function `isEmpty` which amounts to applying this function to the pruning of the arena. For that purpose we add a parameter that accumulates the set of states encountered along the path so that we can prune at the first repetition of a state.

```
void :: (Eq b) => Coalg a b -> b -> Bool
void auto init = isVoid [] init where
 isVoid path state | elem state path = True
                   | otherwise =
                        and [or(map (isVoid (state:path)) states)
                           | (_,states) <- auto state]
```

In the same manner the following function provides an enumeration of the extension of the pruning of a finitely-generated arena. Thus, not only can we decide on emptiness, but when the extension is non-empty we can provide the "simplest" possible witnesses (taken from the extension of the pruning of the arena).

```
enum :: (Eq b) => Coalg a b -> b -> [Tree a]
enum auto init = enum_ [] init where
 enum_ path state | elem state path = []
                  | otherwise =
                    [Node elet trees |
                     (elet,states) <- auto state,
                     trees <- dist (map (enum_ (state:path)) states)]
```

3 Structured documents and their partial views

We consider only the structure of documents without paying attention to their contents or to any of their concrete attributes. A document is legal if it conforms to some abstract context-free grammar.

Definition 3.1 An abstract context-free grammar $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ consists of a finite set $\mathcal{S}$ of **grammatical symbols**, a particular grammatical symbol $A \in \mathcal{S}$ called **axiom**, and a finite set $\mathcal{P} \subseteq \mathcal{S} \times \mathcal{S}^*$ of **productions**. A production $P = \left(X_{P(0)}, X_{P(1)} \cdots X_{P(n)}\right)$ is noted $P : X_{P(0)} \to X_{P(1)} \cdots X_{P(n)}$ and $|P|$ gives the length of the right-hand side of P.

We represent a grammar with the following Haskell definition

```
data Gram prod symb = Gram{prods :: [prod],
                           symbols :: [symb],
                           lhs :: prod -> symb,
                           rhs :: prod -> [symb]}
```

where the selectors give, respectively, the list of production names, the list of grammatical symbols, the left-hand side and right-hand side of each production. To conform to Def. 3.1 we shall nevertheless assume that each production is characterized by its left and right-hand side. A grammar is a coalgebra:

```
gram2coalg :: (Eq symb) => Gram prod symb -> Coalg prod symb
gram2coalg gram symb = [(p,rhs gram p) | p <- prods gram,
                                              symb == lhs gram p]
```

The extension of the arena

```
ast gram symb = ana (gram2coalg gram) symb :: Arena prod
```

is the set of abstract syntax trees for the grammar associated with the given grammatical symbol, where

Definition 3.2 The set $AST(\mathbb{G}, X)$ of **abstract syntax tree** for grammar $\mathbb{G}$ and associated with grammatical symbol X is made of the trees of the form $P(t_1, \ldots, t_n)$ where P is a production of the grammar such that $X = X_{P(0)}$, $n = |P|$ and $t_i \in AST(\mathbb{G}, X_i)$ for all $1 \leq i \leq n$. Abstract syntax trees are therefore the terms for the multi-sorted signature whose sorts are the grammatical symbols and whose operators are the production where production $P : X_{P(0)} \to X_{P(1)} \cdots X_{P(n)}$ is viewed as an operator of arity $X_{P(1)} \times \cdots \times X_{P(n)} \to X_{P(0)}$.

For each grammatical symbol the corresponding set of abstract syntax trees is, thus, a regular set. A document is an abstract syntax tree associated with the axiom of the grammar. Since we have assumed that each production is characterized by its left and right-hand side, a document may equivalently be represented as a derivation tree, i.e., a tree whose node are labelled with grammatical symbols:

Definition 3.3 The set $Der(\mathbb{G}, X)$ of **derivation trees** for grammar $\mathbb{G}$ and associated with grammatical symbol X is made of the trees of the form $X(t_1, \ldots, t_n)$ for which there exists a production P such that $X = X_{P(0)}$, $n = |P|$ and $t_i \in Der(\mathbb{G}, X_i)$ for all $1 \leq i \leq n$.

A view $\mathcal{V} \subseteq \mathcal{S}$ is a subset of grammatical symbols which are meaningful for some part of the system. The projection associated with a view consists in erasing the invisible grammatical symbols while preserving the structure; it is a forest (a list of trees) reduced to a single tree in case the axiom is a visible symbol:

```
projection :: (symb -> Bool) -> Tree symb -> [Tree symb]
projection view der = if view (top der) then [Node (top der) sons]
                                        else sons
 where sons = concat (map (projection view)(succ_ der))
```

Example 3.4 Suppose that we have a grammar with $\{A, B, C\}$ as the set of grammatical symbols, where A is the axiom, and with the following productions:

$$\begin{array}{lll} P_1 : A \to C\ B & P_3 : B \to C\ A & P_5 : C \to A\ C \\ P_2 : A \to \varepsilon & P_4 : B \to B\ B & P_6 : C \to C\ C \\ & & P_7 : C \to \varepsilon \end{array}$$

Figure 4 shows a derivation tree for the grammar of Example 3.4 together with its projections on the subalphabets $\{A,\ B\}$ and $\{A,\ C\}$, respectively. By iteration

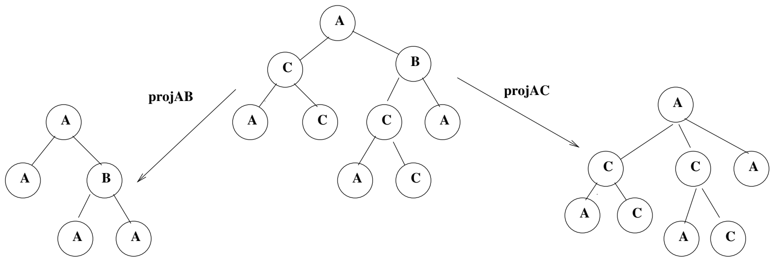

Fig. 4. A derivation tree and its projections on {A,B} and {A,C}

of production $p_6 : C \to C\ C$ we can produce (see Fig. 5) an infinite number of derivation trees having the same partial view.

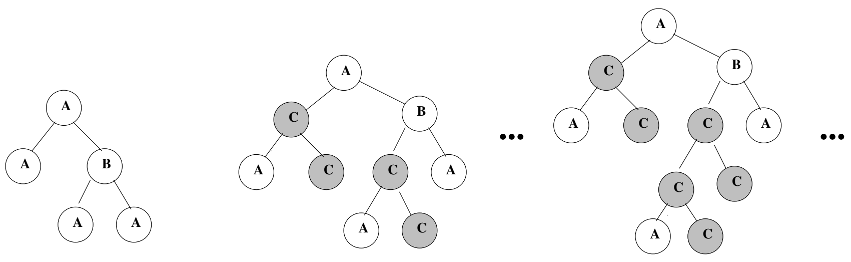

Fig. 5. An infinite set of documents with the same partial view

4 Coherence of views

The problem of coherence of views is to decide whether there exists some (global) document corresponding to a given set of partial views and, if so, to produce such a document. In this section we provide a solution to that problem based on an expansion algorithm which produces a representation of the (regular) set of documents associated with a given view. We are looking for a function

```
expansion :: (Eq symb) => Gram prod symb -> (symb -> Bool) -> symb
                                          -> [Tree symb] -> Arena prod
```

such that the value of the expression `expansion gram view symb forest` is an arena whose extension is the set of abstract syntax trees (associated with a given grammar and symbol) whose projection according to the view given in the argument list is the forest also given in the argument list. That function is defined as an anamorphism

```
expansion gram view axiom ts = ana gramview (axiom, ts)
 where gramview = gram2coalgview gram view
```

for a tree automaton (or coalgebra) `gramview = gram2coalgview gram view` associated with the grammar and the view. The states of this automaton are pairs $\langle X, ts\rangle$ consisting of a grammatical symbol X together with a list of trees. Trees to be recognized from this state are all abstract syntax trees whose root is labeled by a production with symbol X on the left-hand side and such that the corresponding partial view is either the list ts when X is an invisible symbol, or the list reduced to the tree $(Node\ X\ ts)$ if X is a visible symbol. If *forest* is the forest for which we want to compute the expansion then, either the axiom is visible and then that forest should be reduced to a unique tree $forest = [Node\ axiom\ ts_0]$ and we let $q_0 = \langle axiom, ts_0\rangle$ as initial state, or the axiom is an invisible symbol and we let $q_0 = \langle axiom, forest\rangle$ be the initial state. Suppose that a node associated with production $p : A_0 \to A_1 \cdots A_n$ is labeled by the pair $\langle symb, ts\rangle$. It should be the case that $symb = A_0$ and that we can find lists of trees $ts_1, \ldots, ts_n$ such that ts can be decomposed as $ts = ts'_1 +\!\!+ \cdots +\!\!+ ts'_n$ such that $ts'_i = ts_i$ if A_i is invisible and $ts'_i = [Node\ A_i\ ts_i]$ if A_i is visible. Transitions of this automaton are of the form

$$(A_0, ts) \to (p, [(A_1, ts_1), \cdots, (A_n, ts_n)]),$$

where $p : A_0 \to A_1 \cdots A_n$ is a production of the grammar, and symbol A_0 and sequences of trees ts and ts_i satisfy the above conditions. The function computing this tree automaton is given as follows:

```
gram2coalgview :: (Eq symb) => Gram prod symb -> (symb -> Bool)
                               -> Coalg prod (symb, [Tree symb])
gram2coalgview gram view (symb, ts) =
 [(p, zip (rhs gram p) tss) | p <- prods gram,
                              symb == lhs gram p,
                              tss <- match view (rhs gram p) ts]
```

The function *zip* associates each symbol in the right-hand side of the production with a list of trees in order to form the state to be attached to the corresponding argument in the transition of the automaton. The function `match view` takes as a first argument a list of grammatical symbols $A_1 \cdots A_n$, the second argument is a sequence of trees ts and it generates all lists $[ts_1, \ldots, ts_n]$ associated with decompositions $ts = ts'_1 ++ \cdots ++ ts'_n$ of ts where $ts'_i = ts_i$ if A_i is invisible and $ts'_i = [Node\ A_i\ ts_i]$ otherwise, in order to associate them with the symbols A_i.

```
match :: (Eq symb) => (symb -> Bool) -> [symb] -> [Tree symb]
                                     -> [[[Tree symb]]]
match view [] ts = if null ts then [[]] else []
match view (symb:symbs) ts = [(ts1:tss) |
                              (ts1,ts2) <- matchone view symb ts,
                              tss <- match view symbs ts2]
matchone :: (Eq symb) => (symb -> Bool) -> [symb] -> [Tree symb]
                          -> [([Tree symb],[Tree symb])]
matchone view symb ts = if view symb
                        then if (null ts || ((top (head ts)/=symb))
                             then [] else [(succ_ (head ts), tail ts)]
                        else split ts
split :: [a] -> [([a],[a])]
split [] = [([],[])]
split (x:xs) = [([],x:xs)] ++ [(x:xs1,xs2) | (xs1,xs2) <- split xs]
```

By unfolding the definition of function *expansion* we obtain the following reformulation:

```
expansion :: (Eq symb) => Gram prod symb -> (symb -> Bool) -> symb
                          -> symb -> [Tree symb] -> Arena prod
expansion gram view axiom ts = g (axiom,ts) where
 g (symb,ts) = Or [(p, And (map g (zip (rhs gram p)tss))) |
                              p <- prods gram,
                              symb== lhs gram p,
                              tss <- match view (rhs gram p) ts]
```

We illustrate the algorithm of expansion with the grammar of Example 3.4 and the partial view associated with $\{A, B\}$ given in Fig. 4. We represent forests by their encoding in the langage of Dyck on the two visible letters A and B using the opening and closing parenthesis '(' and ')' for the Dyck symbols for A, and similarly '[' and ']' for B. Each production of the grammar is then translated into a schema of transitions for the automaton:

$$\begin{array}{llll}
\langle A, w\rangle & \longrightarrow & (P_1, [\langle C, u\rangle, \langle B, v\rangle]) & \text{if } w = u[v] \\
\langle A, w\rangle & \longrightarrow & (P_2, [\,]) & \text{if } w = \varepsilon \\
\langle B, w\rangle & \longrightarrow & (P_3, [\langle C, u\rangle, \langle A, v\rangle]) & \text{if } w = u(v) \\
\langle B, w\rangle & \longrightarrow & (P_4, [\langle B, u\rangle, \langle B, v\rangle]) & \text{if } w = [u][v] \\
\langle C, w\rangle & \longrightarrow & (P_5, [\langle A, u\rangle, \langle C, v\rangle]) & \text{if } w = (u)v \\
\langle C, w\rangle & \longrightarrow & (P_6, [\langle C, u\rangle, \langle C, v\rangle]) & \text{if } w = uv \\
\langle C, w\rangle & \longrightarrow & (P_7, [\,]) & \text{if } w = \varepsilon
\end{array}$$

The first schema for instance states that an abstract syntax tree generated from state $\langle A, w\rangle$ may be obtained by using production P_1 with arguments respectively associated with states $\langle C, u\rangle$, and $\langle B, v\rangle$ with Dyck words u and v such that w can be decomposed on the form $w = u[v]$. It means that w should end by a ']'. By looking for the associated '[' one can unambiguously determine words u and v. In such a situation we say that the pattern associated with this schema of the transition is deterministic. This is the case for all schemas except for the one associated with production P_6 responsible for the infinite number of documents with the same partial views in Fig.5.

The Dyck encoding of the partial view associated with the derivation tree of Fig. 4 is $(()[()()])$. Since the axiom A is a visible symbol associated with Dyck letters '(' and ')', the initial state of the automaton is $q_0 = \langle A, ()[()()]\rangle$. By restriction to the states accessible from q_0 we obtain the following finite tree automaton:

$$\begin{array}{lllll}
q_0 & \longrightarrow & (P_1, [q_1, q_2]) & & \text{with } q_1 = \langle C, ()\rangle \text{ and } q_2 = \langle B, ()()\rangle \\
q_1 & \longrightarrow & (P_5, [q_3, q_4]) & & \text{with } q_3 = \langle A, \varepsilon\rangle \text{ and } q_4 = \langle C, \varepsilon\rangle \\
q_1 & \longrightarrow & (P_6, [q_4, q_1]) & \mid \quad (P_6, [q_1, q_4]) & \\
q_2 & \longrightarrow & (P_3, [q_1, q_3]) & & \\
q_3 & \longrightarrow & (P_2, [\,]) & & \\
q_4 & \longrightarrow & (P_6, [q_4, q_4]) & \mid \quad (P_7, [\,]) &
\end{array}$$

Fig. 6 gives the pruning (and the subsequent cleaning) of the arena generated from this automaton from its initial state. Certain parts, which are not explicitly indicated in this figure, are represented by colored triangles that should be replaced respectively by the subtree rooted at the node with the same color (they are the nodes labelled by states q_4 and q_1, respectively). There is exactly one tree in the extension of the resulting arena, namely the abstract syntax tree associated with derivation tree of Fig.4.

In order to solve the problem of coherence of views it remains to define a binary synchronization operation on coalgebras producing a new coalgebra from which a pair of partial views generates the intersection of the set of documents generated

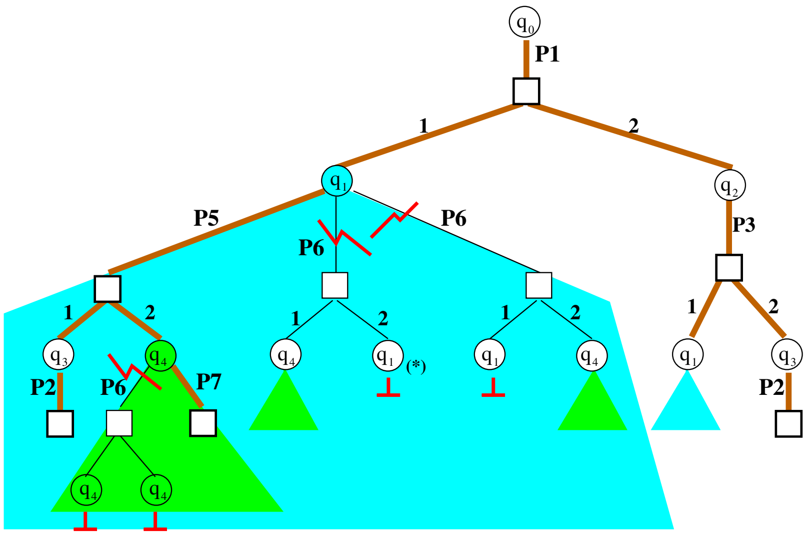

Fig. 6. Pruning (and cleaning) of the arena generated by the automaton and initial state associated with the partial view (()[()()])

by each coalgebra from the corresponding partial view. More precisely:

Proposition 4.1 *If* $(t \models coalg\ init)$ *means that tree t belongs to the extension of* $(ana\ coalg\ init)$*, then*

$$(t \models (coalg_1 \langle\$\rangle\, coalg_2)\,(init1,\ init2)) \Leftrightarrow (t \models (coalg_1\ init_1)\ \wedge\ t \models (coalg_2\ init_2))$$

The set of states of the compound automaton are pairs of states of the respective automata and its transitions are obtained by synchronization:

```
(<$>) :: (Eq a) => Coalg a b -> Coalg a c -> Coalg a (b,c)
(coalg1 <$> coalg2)(state1,state2) =
   Or [(a1, And (zip states1 states2)) |
                (a1,states1) <- coalg1 state1,
                (a2,states2) <- coalg2 state2,
                 a1 == a2,
                 length states1 == length states2]
```

The proof of Prop. 4.1 is straightforward and is omitted. Then we let

```
coherence :: (Eq prod,Eq symb) =>
             Gram prod symb -> (symb -> Bool) -> (symb -> Bool)
             -> symb -> [Tree symb] -> [Tree symb] -> Arena prod
coherence gram view1 view2 axiom ts1 ts2 =
 ana (gview1 <$> gview2)((axiom,ts1),(axiom,ts2))
 where gview1 = gram2coalgview gram view1
       gview2 = gram2coalgview gram view2
```

Given an abstract context-free grammar, two views, two projections associated with these views (partial views of documents) and the axiom of the grammar, the coherence function returns an arena representing the set of coherent common extensions of these two partial views.

5 The merge of structured documents

In this section we try to elaborate on a model of document manipulations in workflow systems using adaptations of the expansion algorithm and of the synchronization operator defined in the previous section. Adaptating the definitions is necessary because a document in a workflow system is bound to grow as it flows between activities. Some leaves of the document (viewed as a tree) are then marked as open to indicate that the tree may expand from these nodes; a marked leaf is said to be a *bud* of the tree.

Definition 5.1 A **document** associated with grammar $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ is an abstract syntax tree for the extended grammar $\mathbb{G}_\Omega = (\mathcal{S}, \mathcal{P} \cup \mathcal{S}_\Omega, A)$ obtained from $\mathbb{G}$ by adding a new production $X_\Omega : X \to \varepsilon$, with an empty right-hand side, for each grammatical symbol $X \in \mathcal{S}$. A node labelled X_Ω is an **open node** (or **bud**) of sort X. We let $t_1 \leq t_2$ if t_2 may be obtained from t_1 by replacing some buds of t_1 by trees of the corresponding sort, we then say that t_2 is an **update** of t_1.

A derivation tree is defined as before except that we add the information that certain leaves are buds. Such a derivation tree is viewed as a tree on the duplicate alphabet $\mathcal{S} \cup \overline{\mathcal{S}}$ where a node labelled by a symbol $\overline{X} \in \overline{\mathcal{S}}$ represents an open node of sort $X \in \mathcal{S}$ (therefore symbols in $\overline{\mathcal{S}}$ appear only at leaves). In that way we keep a bijective correspondance between documents (abstract syntax trees for $\mathbb{G}_\Omega$) and these (extended) derivation trees. The projection of a derivation tree on a subset $\mathcal{V} \subset \mathcal{S}$ of grammatical symbols is defined as before by considering that a symbol $\overline{X}$ is visible if and only X is visible. The coalgebra (tree automaton) that defines the expansion of a partial view (projection along a view) is defined as before except that we add the transitions $(X, [\,]) \to (X_\Omega, [\,])$ if X is an invisible symbol, and, $(X, [\mathit{Node}\ \overline{X}\ [\,]]) \to (X_\Omega, [\,])$ if X is a visible symbol. A transition of the form $q \to (X_\Omega, [\,])$ is called an *exit transition.* We enrich the automata by adding a predicate *exit* for specifying those states from which an exit transition exists. Therefore an exit transition $q \to (X_\Omega, [\,])$ in the above defined automaton is replaced by the information that *exit* q holds true. The initial state of the synchronization of two automata $\mathcal{A}$ and $\mathcal{A}'$ is the pair (q_0, q_0') made of the respective initial states of $\mathcal{A}$ and $\mathcal{A}'$. We let *exit* (q, q') if and only if *exit* q and *exit* q'. The transitions of this synchronized automaton are

$$(q, q') \longrightarrow (P, [(q_1, q_1'), \ldots, (q_n, q_n')])$$

with $q \to (P, [q_1, \ldots, q_n])$ in $\mathcal{A}$ and $q' \to (P, [q_1', \ldots, q_n'])$ in $\mathcal{A}'$ together with

$$(q, q') \longrightarrow (P, [q_1', \ldots, q_n'])$$

with *exit* q and $q' \to (P, [q_1', \ldots, q_n'])$ in $\mathcal{A}'$, and symmetrically,

$$(q, q') \longrightarrow (P, [q_1, \ldots, q_n])$$

with *exit* q' and $q \to (P, [q_1, \ldots, q_n])$ in $\mathcal{A}$. Let $\mathcal{V}_1$ and $\mathcal{V}_2$ be two views and u_1 and u_2 be two partial views on the corresponding set of symbols. Merging u_1 and u_2

consists in finding a document t satisfying the following:

$$(\exists t_1 \leq t \quad \pi_{\mathcal{V}_1}(t_1) = u_1) \ \wedge \ (\exists t_2 \leq t \quad \pi_{\mathcal{V}_1}(t_2) = u_2)\,,$$

where $\pi_{\mathcal{V}}(t) = u$ means that the tree u is the projection of document t according to view $\mathcal{V}$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are the automata associated with $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively, then a document satisfies the above condition if and only if it is an update of some document recognized by the synchronization of $\mathcal{A}_1$ and $\mathcal{A}_2$ from initial state (u_1, u_2). We can unambiguously reconstruct a (global) document as the merge of local views if the set of views is a *cover* in the sense of the following definition:

Definition 5.2 A set $\{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ of views $\mathcal{V}_i \subset \mathcal{S}$ is a **cover** of grammar $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ when the synchronization of the corresponding automata $\mathcal{V}_i$ recognizes at most one element whatever its initial state is.

As in [32] we choose to represent a workflow system as a statechart [13]. We could as well have chosen UML state diagrams (a standardization of statecharts) or some specific class of Petri nets. However, statecharts are a simple and powerful formalism which contains exactly the ingredients needed to illustrate our purpose (see Fig. 7). Actually a state can be decomposed hierarchically in two ways: an 'or' decomposition (depicted by box containment) and an 'and' decomposition (a box is split into subparts separated by dash lines). If a document is located in some 'or' state then it should be located in exactly one of its constituent substates; if it is located in an 'and' state, then a partial view of that document should be located into each constituent part of that state. A transition between two states

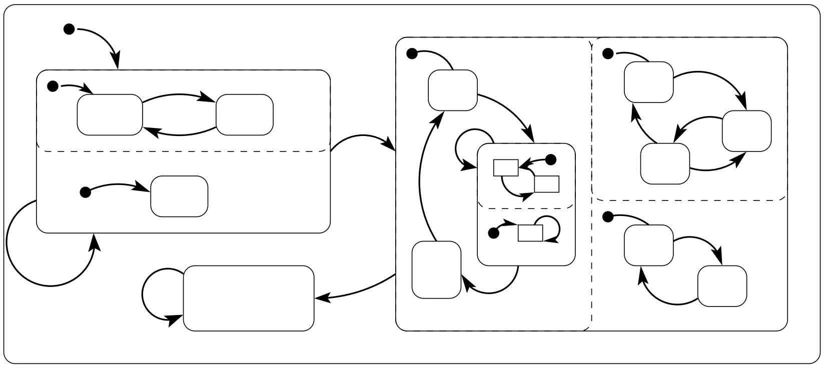

Fig. 7. A statechart

may be labelled by an event or by a guard expressing a condition that when satisfied provokes the corresponding state transition. In each state as well as in each 'and' part of a state, a default initial state is specified (using an arc originating from a dot) that we initially activate when entering its parent state. However, we may choose to enter directly at some other state (see [13] for examples). At each state the document may be updated. We associate each state with a grammatical structure to be defined below. Each constituent state of an 'or' block as well as their parent state, is associated with the same grammatical structure. If an 'and' block is associated with a grammatical structure $\mathcal{G}$, then any of its constituent parts is associated with a grammatical substructure $\mathcal{G}_i$ of $\mathcal{G}$ so that the set $\{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$ is a cover of $\mathcal{G}$.

Definition 5.3 A **grammatical structure** $\mathcal{G} = (\mathbb{G}, \mathbb{C})$ consists of two parts. A grammar $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ that gives the intentional representation of documents (as abstract syntax trees for the extended grammar $\mathbb{G}_\Omega$) and a **concrete representation** $\mathbb{C} = (\mathbb{S}, \mathbb{P})$ that consists of a set $\mathbb{S} \subseteq \mathcal{S}$ of **syntactical categories** and a set $\mathbb{P}$ of **patches**. The concrete view of a document is then given by its projection on the set of syntactical categories; however, a document should always be characterized by its concrete view, i.e., $\{\mathbb{S}\}$ is a cover of $\mathbb{G}$. A patch of sort $X \in \mathbb{S}$ is associated with a document of that sort. A grammatical structure $\mathcal{G}_1$ is a substructure of $\mathcal{G}$ if $\mathbb{S}_1 \subseteq \mathbb{S}$ and the projection of a document for $\mathbb{G}$ on $\mathbb{S}_1$ is the projection of a (necessarily unique) document for $\mathbb{G}_1$ on $\mathbb{S}_1$; a set of substructures $\{\mathcal{G}_1, \ldots, \mathcal{G}_n\}$ is a cover of $\mathcal{G}$ if $\{\mathbb{S}_1, \ldots, \mathbb{S}_n\}$ is a cover of $\mathbb{G}$.

A syntactical category corresponds to an element that can be identified and manipulated by the application; grammatical symbols in $\mathcal{S} \setminus \mathbb{S}$ are just artefacts that are used to describe the logical organization of documents (using the grammar) but have no semantic meaning. We assume that two different grammatical structures can share syntactic symbols only; the other grammatical symbols are purely local. When entering an 'and' state, the document is projected onto the various components and each such projection (partial view of the document) is directed to the corresponding initial state. Then each partial view flows into its subsystem where it may be updated. We can apply a patch at a given bud of the document with the same sort, and the result is the new document obtained by substituting the patch at the corresponding node. An update of the document is realized by a sequence of applications of patches. When exiting the 'and' state the various partial views are merged and the resulting document is forwarded to the target state.

6 Conclusion

In this paper we have shown an application of the use of coalgebras and of coinductive structures in the context of the business document manipulation within complex and distributed workflow systems. We have proposed for that purpose a simple extension of the model of statecharts. For the time being the problem is that the designer has to check the conditions that are enforced in Def. 5.3. These conditions are quite natural and will be satisfied in many pratical situations and not too difficult to check. However, it would be desirable to have an automated way to verify that a given system specification satisfies these properties, i.e., is well-formed. At the moment, such a tool does not exist. The main difficulty is to decide whether a given set of views is a cover of a given grammar. This problem, however, is very difficult since it generalizes the problem of ambiguity in context-free grammars. Indeed, a context-free grammar can be viewed as an abstract grammar whose grammatical symbols are all terminal and non-terminal symbols (up to the adjunction of extra productions $X \to \varepsilon$ for each terminal symbol X). Parsing a sequence of terminal symbols is equivalent to finding the expansion of that sequence when the view is given by the set T of terminal symbols. Hence, checking the unambiguity of the context-free grammar reduces to checking whether $\{T\}$ is a cover of the corresponding (abstract) grammar. Therefore, one has to look for some stronger condition to enforce on the corresponding vectors of views. Another way out is to try to cope with potential ambiguity of the system by replacing a document in the workflow by an arena (or an automaton generating it) representing all possible values of that document. But this would require us to lift all our constructions at the level of arenas (or of automata) in order to obtain some kind of symbolic workflow systems.

It will probably be simpler and more realistic to adopt a bottom-up approach: we suppose having a set of tools associated with specific activities in the workflow that we would like to coordinate. Each such tool is associated with a grammatical structure describing the family of manipulated documents together with a dynamic system representing all possible evolutions of these documents (using the tool). Thus, we have to fuse these grammatical structures into a larger one and, simultaneously, to synthesize some control so that any vector of partial views of a same global document can always be unambiguously merged into another global document as soon as each local modification is executed under the supervision of the controller.

As previously mentioned, the expansion algorithm presented in this article, can be seen as some kind of generalized functional parser where tokens used for parsing have an internal tree structure exploited by the algorithm. We would like to design a set of functional parser combinators, in the line of [11,17], in order to obtain a domain-specific language for writing the expansion algorithm by specifying, using these combinators, the abstract grammar and the view. Since expansion is an inverse to projection, this expansion algorithm is also related to the more general problem of *program inversion*. It could be interesting to verify whether our solution to the merge of partial views could be understood within the framework presented in [3,26,27].

Web services emphasize the use of *active documents* where queries are attached to some nodes in order to collect information from other documents [2]. It could be interesting to perform such queries on partial views during their merge operation as it would allow more information to flow between the corresponding activities. We intent to address this problem in the context of attribute grammars using the functional evaluations of attributes presented in [19,12,5]. Indeed, the value of an attribute of a visible grammatical symbol may depend on the value of attributes of invisible symbols (but visible for a different view) and thus, attribute computation may provide a means of coordination between the activities operating on different views of a document.

References

[1] Abiteboul, S., On Views and XML, in *Proceedings of the eighteenth ACM Symposium on Principles of Database Systems*, 1–9, 1999.

[2] Abiteboul, S., O., Benjelloun, and T. Milo, Positive Active XML, in *Proceedings of the twenty-third ACM Symposium on Principles of Database Systems*, 35–45, 2004.

[3] Abramov, Sergei M. and Robert Gluck. The universal resolving algorithm: inverse computation in a

functional language. Proceedings of Mathematics of Program Construction 2000. R. Backhouse and J. N. Oliveira, Eds. Springer-Verlag Lecture Notes in Computer Science vol. 1837, pp. 187–212. 2000.

[4] Asklund, U., Identifying conflicts during structural merge, in *Proceedings of the Nordic Workshop on Programming Environment Research*, Lund, Sweden, 231–242, 1994.

[5] Backhouse, Kevin S. A Functional Semantics of Attribute Grammars. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science, Springer-Verlag, 2002.

[6] Balasubramaniam, S., and Benjamin C. Pierce, What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking* (MobiCom'98), 98–108, 1998.

[7] Braganholo, V., Updating relational databases through xml views. Thesis proposal - in preparation, PPGC-UFRGS, Porto Alegre, 2002.

[8] Chawathe, Sudarshan, and Hector Garcia-Molina, Meaningful change detection in structured data, in *ACM SIGMOD International Conference on Management of Data*, ACM Press, 26–37, 1977.

[9] Chawathe, Sudarshan, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom, Change detection in hierarchical structured information, in *ACM SIGMOD International Conference on Management of Data*, ACM Press, 493–504, 1996.

[10] Chen, Y., T. Ling and M. Lee. Designing Valid XML Views. ER Conference, 2002.

[11] Fokker, J. Functional Parsers. In J. Jeuring and E. Meijer, editors, *First International School on Advanced Functional Programming*, volume 925 of Lecture Notes in Computer Science, 1-23, Springer-Verlag, 1995.

[12] Fokkinga,M., J. Jeuring, L. Meertens, and E. Meijer. A Translation from Attribute Grammars to Catamorphisms. *The Squiggolist*, 2(1):20-26, 1991.

[13] Harel, D., Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming* **8**, 231–274, 1987.

[14] Haskell, A Purely Functional Language. URL: http://www.haskell.org

[15] Horwitz, S., J. Prins, and T. Reps, Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems* **11**(3), 345–387, 1989.

[16] Hudak, P., J. Peterson, and J.H. Fasel. *A Gentle Introduction to Haskell 98.*, 1999.

[17] Hutton, G., and E. Meijer. Monadic Parsing in Haskell. J. Functional Programming 8(4), pp. 437-444, 1998.

[18] Hasuo, I., B. Jacobs, and A. Sokolova. Generic Trace Semantics via Coinduction. *Logical Methods in Computer Science* **3**(4), 2007.

[19] Johnsson, T. Attribute Grammars as a Functional Programming Paradigm. In G. Kahn, ed, *Proc. of 3rd Int. Conf. on Functional Programming and Computer Architecture*, FPCA'87, vol. 274 of Lecture Notes in Computer Science, 154-173, Springer-Verlag, 1987.

[20] Kiepuszewski, B., A.H.M. ter Hofstede, W.M.P. van der Aalst. Fundamentals of Control Flow in Workflows, *Acta Informatica* **39**(3), 143–202, 2007.

[21] R. la Fontaine, Merging XML files: a new approach providing intelligent merge of XML data sets, in *Proceedings of XML Europe*, Barcelona, Spain, 2002.

[22] Leymann, F., and D. Roller, *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1999.

[23] Lindholm, T., A three-way Merge for XML Documents, in ACM Symposium on Document Engineering, ACM Press, 1–10 ,2004.

[24] Manger, Gerald W., Generic Algorithm for Merging SGML/XML-Instances, in *Proceedings of XML Europe*, Berlin, Germany, 2003.

[25] Mens, T., A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* **28**(5), 449–462, 2002.

[26] Mu, Shin-Cheng . A Calculational Approach to Program Inversion. PhD Thesis, Oxford University Computing Laboratory, 2003.

[27] Mu, Shin-Cheng and Richard Bird. Theory and applications of inverting functions as folds. Science of Computer Programming (Special Issue for Mathematics of Program Construction, vol. 51, pp. 87–116, 2003.

[28] Mu, S.C., Z. Hu, and M. Takeichi. An injective language for reversible computation. In Seventh International Conference on Mathematics of Program Construction (MPC 2004), Stirling, Scotland, July 2004. Springer Verlag, LNCS.

[29] Mu, S.C., Z. Hu, and M. Takeishi, An algebraic approach to bidirectional updating, *Second Asian Symposium on Programming Languages and Systems*, 2–18, 2004.

[30] Mu, S.C., Z. Hu, and M. Takeishi, A programmable Editor for Developping Structured Documents based on Bidirectional Transformations, Higher-Order and Symbolic Computation **1**, 1–31, 2006.

[31] Peyton Jones, Simon (editor). *Haskell 98 Language and Libraries - The Revised Report*. Cambridge University Press, 2003.

[32] Wodtke, D., and G. Weikum, A formal foundation for distributed workflow execution based on statecharts, *in Database Theory – ICDT'97*, Lecture Notes in Computer Science, vol. 1186, 230–246, 1997.

[33] XQuery, URL: http://www.w3.org/TR/xquery

[34] XSL, URL: http://www.w3.org/TR/xsl/

Appendix: Some Haskell Notations

This annex is not an introduction to Haskell for which the interested reader is encourage to consult the technical document [31], and the introduction [16] both accessible on Haskell official website [14]. However the following should be sufficient for a good understanding of the Haskell code presented in our paper.

Some Haskell Combinators

length gives the length of a list: $length\ [1, 3, 5, 7]$ has value 4.

head, tail give, respectively, the first element of a non empty list and the residual list: $head\ [1, 2, 3]$ is 1 and $tail\ [1, 2, 3]$ is $[2, 3]$.

elem tests whether an element appears in a list: $elem\ 2\ [1, 3, 5, 7]$ is $False$.

concat, ++ for the concatenation of lists: $xs ++ ys$ is the concatenation of lists xs and ys; If xss is a list of lists, $concat\ xss$ is the concatenation of all lists in xss.

and, or the conjunction and disjunction of a list of booleans: $or\ [\,]$ is $False$ and $and\ [\,]$ is $True$ (respective neutral elements).

map applies a function to each element of a list: $map\ ::\ (a \to b) \to [a] \to [b]$, for instance $map\ (*2)\ [0, 1, 2, 3, 4]$ is $[0, 2, 4, 6, 8]$.

zip pair the elements of the same rank from two lists. If the two lists do not have the same length, the generation of the resulting list stops as soon as one of the input lists is exhausted: $zip\ ::\ [a] \to [b] \to [(a, b)]$, for instance $zip\ [1, 2, 3, 4]\ ['a', 'b', 'c']$ is $[(1, 'a'), (2, 'b'), (3, 'c')]$.

zipWith is similar to zip except that it further applies a binary function on each pair occurring in the output list. Thus $zipWith\ f\ xs\ ys$ can be written

$$map\ (\backslash(x, y) \to f\ x\ y)\ (zip\ xs\ ys)$$

for instance $and\ (zipWith\ (<=)\ xs\ (tail\ xs))$ tests whether is list is ordered.

Schema of list comprehension

That very intuitive notation is similar to the schema of set comprehension. On its simpler form we write

$$[f\ x \mid x \leftarrow xs]$$

to represent the list of elements $f(x)$ when variable x ranges the list given by the expression xs, called list generator. That expression is equivalent to *map f xs*. Now several extensions are possible:

with several generators We can use several generators. Each generator definition can then depends of the preceding variables. Thus, the order of generators is important. For instance
- $[(x,y)|x \leftarrow [1..3], y \leftarrow [x..3]]$ has value $[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]$,
- $[(x,y)|x \leftarrow [1,2], y \leftarrow ['a','b']]$ has value $[(1,'a'),(1,'b'),(2,'a'),(2,'b')]$ and
- $[(x,y)|y \leftarrow ['a','b'], x \leftarrow [1,2]]$ has value $[(1,'a'),(2,'a'),(1,'b'),(2,'b')]$.

with guards We can use guards to filter some elements. For instance, $[x\ 'mod'\ 3 \mid x \leftarrow [12,11,9,4,13,20,7], even\ x]$ has value $[0,1,2]$ (the remainders in the division by 3 of the even numbers of the input list).

with patterns One can also replace the variable by a pattern. We apply pattern matching to each element of the list and filter out the elements that fail the pattern match. In case of succesful pattern match, the variables of the pattern are instanciated accordingly. For instance, given a list of trees ts, the value of the expression $[a \mid Node\ a\ [\,] \leftarrow ts]$ is the list of labels of trees, taken from list ts, that are reduced to a leaf. This expression is equivalent to $[a \mid Node\ a\ ts' \leftarrow ts, ts' == [\,]]$.

To conclude, we present some examples that combine several of the above features:

concat : a way of writing combinator *concat* :

$$concat\ xss\ =\ [x \mid xs \leftarrow xss,\ x \leftarrow xs]$$

zipWith : another way to define *zipWith* in terms of *zip* :

$$zipWith\ f\ xs\ ys\ =\ [f\ x\ y \mid (x,y) \leftarrow zip\ xs\ ys]$$

and, thus, to write the function that tests whether a list is ordered:

$$sorted\ xs\ =\ and\ [x <= y \mid (x,y) \leftarrow zip\ xs\ (tail\ xs)]$$

positions: a function $positions\ ::\ (Eq\ a) \Rightarrow a \rightarrow [a] \rightarrow [Int]$ that returns the list of positions where a given element occurs in a given list (e.g. *positions* $0\ [1,0,0,1,0,1,1,0]$ has value $[2,3,5,8]$) :

$$positions\ x\ xs\ =\ [i \mid (x',i) \leftarrow zip\ xs\ [1..],\ x' == x]$$

sieve of Eratosthenes : function that generates the (infinite) list of prime numbers:

$$\begin{aligned} prime &= sieve\ [2..] \\ sieve\ (n:ns) &= n : (sieve\ [m \mid m \leftarrow ns, m\ 'mod'\ n\ \neq 0]) \end{aligned}$$