Querying and Splicing of XML Workflows

Vassilis Christophides* Richard Hull† Akhil Kumar†

*Institute of Computer Science, FORTH,
Vassilika Vouton, P.O.Box 1385, GR 711 10,
Heraklion, Greece
christop@ics.forth.gr

†Bell Laboratories, Lucent Technologies
600 Mountain Avenue, Murray Hill, NJ, USA
{hull,akhil}@research.bell-labs.com

Abstract. In both industry and the research community it is now common to represent workflow schemas and enactments using XML. As a matter of fact, more and more enterprise application integration platforms (e.g., Excelon, Bea, iPlanet, etc.) are using XML to represent workflows within or across enterprise boundaries. In this paper we explore the ability of modern XML query languages (specifically, the W3C XML Algebra underlying the forthcoming XQuery) to query and manipulate workflow schemas and enactments represented as XML data.
The paper focuses on a simple, yet expressive, model called Workflow Query Model (WQM) offering four primary constructs: `sequence`, `choice`, `parallel`, and `loop`. Then three classes of queries are considered against WQM workflows: simple (e.g., to check the status of enactments), traversal (e.g., to check the relationship between tasks, or check the expected running time of a schema), and schema construction (e.g., to create new schemas from a library of workflow components). This querying functionality is quite useful for specifying, enacting and supervising e-services in various e-commerce application contexts and it can be easily specified using the W3C XML Query Algebra.

1 Introduction

During recent years, workflow interoperation has received considerable attention. Numerous research projects and prototypes have been proposed while basic interoperability between various vendor WFMSs has been a subject of standardization efforts by the Object Management Group (see Workflow Management Facility [OMG98]), and the Workflow Management Coalition (see the Xf-XML binding of the WfMC Interface 4 [WMC99]). Recently, XML has become widely accepted as the standard for exchanging not only business data but also information about the enterprise process (e.g., e-services) operating on these data. More and more enterprise application integration platforms (e.g., Excelon, BEA systems, iPlanet, Vitria BusinessWare, icXpertFLOW[1] etc.) and research prototypes [WSFL01,vdAK01,LO01,SGW00,MK00,TAKJ00] are using XML to represent workflow schemas and enactments, within or across enterprise boundaries.

[1] See www.exceloncorp.com, www.bea.com, www.iplanet.com, www.vitria.com and www.icomxpress.com respectively.

The use of appropriate XML query languages to access information, about both XML workflow schemas and their enactments, appears as the natural solution for specifying, enacting and supervising e-services within or across organizations. In this paper we show that modern XML query languages (in particular, the W3C XML Algebra [FSW01,FFM+00] underlying the forthcoming XQuery) are expressive enough to issue a variety of useful queries for a simple, but still expressive, workflow model. Furthermore, we propose the incorporation of a number of user-defined functions allowing us to easily navigate through the XML trees representing workflow schemas and enactments.

In this paper we focus on three classes of queries:

Simple: These check on the status of a workflow enactment, i.e., an execution that is in process.

Traversal: These check properties that are more global to workflow schemas and enactments, including issues such as the relationship between tasks (parallel, sequential, exclusive), and the expected (min, max) running time of a schema.

Schema Construction: These queries can be used to construct workflow schemas, using components from a library of "templates" and "base templates"

These queries provide a basis for improving the design and efficiency of workflows, both at compile-time and run-time. For example, potential bottle-necks and race conditions on data usage might be found at compile-time, and potential delays might be detected and averted at run-time.

Given the plethora of models in existing WorkFlow Management Systems (WFMS) we rely in this paper on a pragmatic workflow model allowing us to illustrate concretely how XML query languages can used. This model, called *Workflow Query Model* (*WQM*), involves flowchart constructs with parallelism. More specifically, we focus on four key workflow constructs, namely *sequence*, *choice*, *parallel*, and *loop*. We assume that in a workflow schema these constructs are *properly nested* as per the notion of a structured workflow [KHB00]. Intuitively, this means that there are no go-to's that point into a loop, or that point out of a group of parallel activities. Properly nested workflow schemas can simulate all structured and some unstructured workflows as shown in [KHB00]. As a matter of fact, WQM captures the common features of several commercially available WFMSs, (e.g., Fujitsu i-flow, IBM MQSeries, SAP Business Workflow, FileNet Workflow, Ultimus Workflow[2], or even UML activity diagrams). However, it does not directly address the issues that arise when querying flowchart models based on Petri nets [vdA98] or state charts (e.g., Statemate MAGNUM[3]).

In a nutshell, WQM schemas have a natural tree-based representation, making them ideal for representation and manipulation in XML. With regards to workflow enactments, we follow the same approach for representing them in XML by annotating schema nodes with appropriate status information (e.g., running,

[2] See www.i-flow.com, www.software.ibm.com/ts/mqseries/workflow, www.sap.com, www.filenet.com, www.ultimus1.com, respectively.

[3] See www.ilogix.com.

finished). It should be stressed that workflow querying has received surprisingly little attention so far beyond analysis of workflow logs [KAD98,GT97]. In addition, workflow querying using modern query languages for XML data (e.g., XPath[CD99], Quilt [DC00], XML Query Algebra [FSW01,FFM+00]) is not even addressed in previous research work [WSFL01,LO01,SGW00,MK00,TAKJ00].

This paper is organized as follows. Section 2 provides background and preliminary discussion of our WQM model. Then Section 3 presents how simple status queries can be expressed against enactments created according to the WQM model. Section 4 turns to more complex kinds of traversal queries against both schemas and enactments. Later Section 5 focuses on a novel kind of queries for on demand schema construction. Finally, Section 6 concludes the paper.

2 Background and Preliminaries

This section provides background and preliminary definitions for the rest of the paper. In particular, we present (i) the workflow model we are using to abstract commercial WFMSs, (ii) the variant of XRL [KZ98,vdAK01] we are using to represent in XML workflow schemas and enactments created according to our model, and (iii) how the XML query processor we envision can be integrated into the standard WfMC architecture.

2.1 A Workflow Query Model (WQM)

Figure 1 illustrates a representative workflow schema created using the WQM model which will be used as a running example in the rest of the paper. Task nodes are shown pictorially as rectangles, and the other kinds of nodes are shown using ovals that are labeled with the node type. More precisely, nine kinds of nodes are foreseen to model processes: `start`, `end`, `task`, `split-choice` (with arbitrary out-degree > 1), `join-choice` (with arbitrary in-degree > 1), `split-parallel` (with arbitrary out-degree > 1), `join-parallel` (with arbitrary in-degree > 1), `start-while_do`, and `end_while_do`.

A `split-choice` permits branching in the workflow, where exactly one of the out-edges is taken. The choice may be either deterministic or non-deterministic. In the former case, a specific task out of several alternatives may be chosen after checking a condition. For instance, if two managers can sign an expense approval, the one whose workload is less may be chosen explicitly. On the other hand, a non-deterministic choice may be made as follows. The expense claim could be "offered" to both the managers, and once one of them has accepted the task, it may be withdrawn from the other manager. The parallelism construct is self-explanatory. The `while_do` permits looping over a set of tasks, or in general, over a group of nodes, multiple times. In a WQM workflow schema the `split-choice` and `join-choice` must be matched, as must the `split-parallel` and `join-parallel` nodes, and also `start-while_do` and `end-while_do`. This restriction is primarily to simplify the structure of the workflow schemas that are studied, and thus the queries that need to be expressed.

The WQM workflow model also supports specifications indicating which data objects are manipulated by which tasks. In particular, included in a workflow

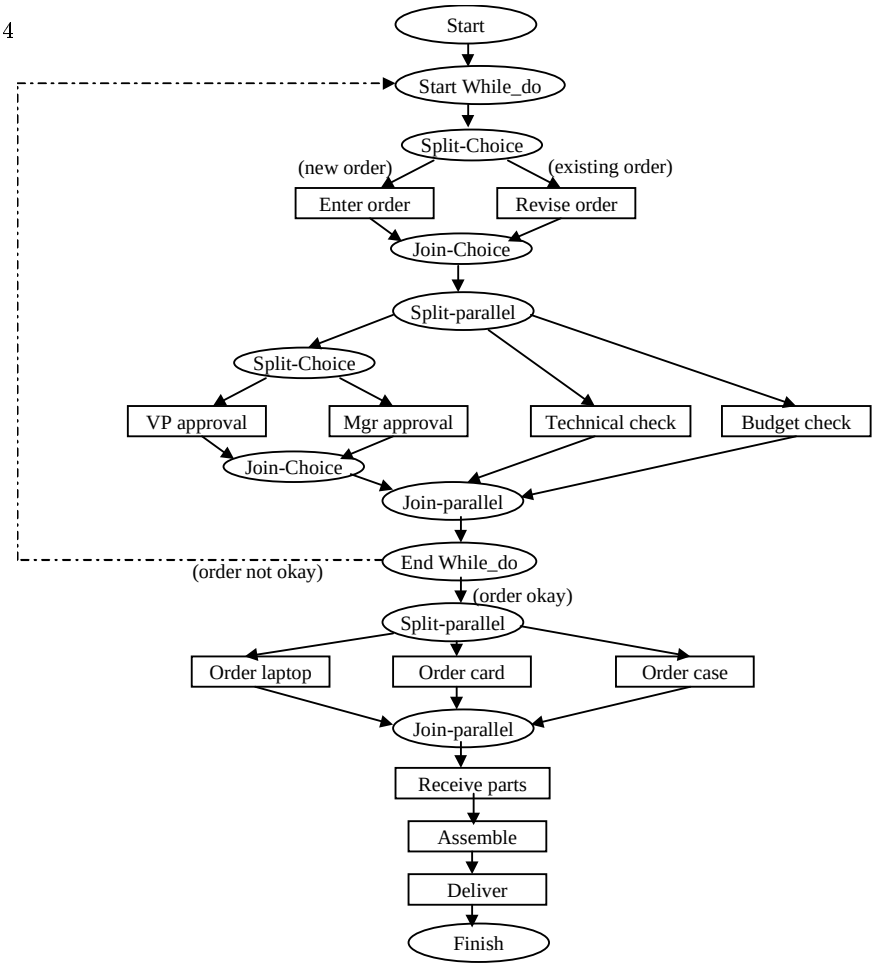

Fig. 1. An example workflow schema using our model

schema is the list of a finite number of named data objects that can be created, read, and/or written by the tasks of the workflow. These data objects can essentially be viewed as global variables that an enactment of the schema can access. For simplicity, we view the data objects as XML documents. Associated with each task is a listing of three disjoint categories of the named data objects, those that the task might create, those it might read, and those it might read and write. Note that exception handling (e.g., out of stock situations, delays, order changes, etc.) is not treated in this paper. For a more theoretical discussion related to structured workflow models, the reader is referred to [KHB00].

2.2 XML representation of a schema and an enactment state

This subsection describes how workflow schemas and enactments can be specified in the WQM model using XML. For this purpose, we are using the type system and syntax of the XML Query Algebra [FSW01,FFM+00] which captures the core semantics of XML Schema [TBMM00,MM00].

```
let buy_pc1 : WQMEnactment =
  Route [
     Data_list["order_form","engg_spec"],
     Sequence[
       while_do [@count ["0"], @condition ["new-order()? or review(not_okay)?"],
         Sequence [
           Choice[@condition ["new-order()?"]
             Task [ @name ["enter-order"], @status[ "finished" ],
                    @d_create [ "order_form","engg_spec" ]]
             Task [ @name ["revise-order"], @status ["not_ready" ],
                    @d_read [ "engg_spec" ], @d_update [ "order_form"]]
                 ]  (* end Choice *)
             Parallel[
              Choice[@condition ["manager(available)?", @branch ["1"],
                 Task [ @name ["manager-approval"], @status[ "running" ],
                        @d_read [ "engg_spec"],
                        @d_update [ "order_form"]]
                 Task [ @name ["vp-approval"],
                        @d_read [ "engg_spec" ],
                        @d_update [ "order_form"]]
                    ]  (* end Choice *)
               Task  [ @name ["technical-approval"], @status ["ready" ],
                       @d_update [ "order_form","engg_spec"]]
               Task  [ @name ["check-budget"], @status ["finished" ],
                       @d_read["engg_spec"],@d_update [ "order_form"]]
                   ]    (* end Parallel *)
                 ]  (* end Sequence *)
               ]  (* end while_do *)
           Parallel[
            Task  [ @name ["order-pc"], @status ["not_ready" ],
                    @d_read["engg_spec"],@d_update [ "order_form"]]
            Task  [ @name ["order-card"], @status ["not_ready" ],
                    @d_read["engg_spec"],@d_update [ "order_form"]]
            Task  [ @name ["order-case"], @status ["not_ready" ],
                    @d_read["engg_spec"],@d_update [ "order_form"]]
                 ]    (* end Parallel *)
           Task  [ @name ["receive-parts"], @status ["not_ready" ],
                   @d_update [ "order_form"]]
           Task  [ @name ["assemble-system"], @status ["not_ready" ],
                  @d_read["engg_spec"],@d_update [ "order_form"]]
           Task  [ @name ["deliver"], @status ["not_ready" ],
                  @d_update [ "order_form"]]
               ] (* end  sequence *)
         ] (* end Route *)
```

Fig. 2. The workflow of Figure 1 in syntax of XML Query Algebra

First, observe that workflow schemas in the WQM model can be naturally represented as XML trees. Each schema has a root (i.e., a starting element), and may contain several routing elements such as `sequence, parallel, choice, task and slot`, which may be properly nested. Due to space limitations, we are giving in Figure 2 only an enactment example of the workflow schema of Figure 1 defined in WQM.

The WQM XML representation of an enactment essentially captures the information in the coresponding schema with minor variations. Speaking in broad terms, this is achieved by annotating nodes of the tree that encode the workflow schema. In Figure 2 a WQM enactment called `buy_pc1` is presented with a `Route` element which consists of a `Data_list` and `Sequence` subelements. The `Data_list` subelement consists of data object names that are used in this workflow instance. The sequence in turn consists of a `While_do`, a parallel and three task subelements. In general subelements are nested inside their parent elements as comma separated lists enclosed in square brackets. The attributes of an ele-

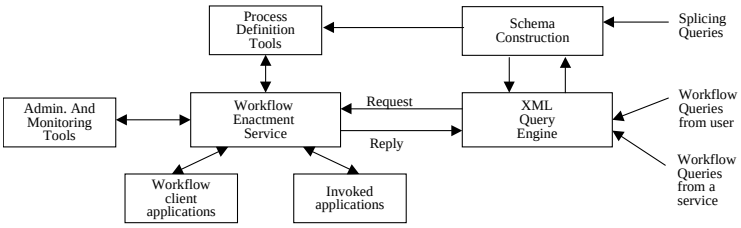

Fig. 3. Architecture for querying of workflows

ment are also enclosed inside the list associated with it and are preceded by the `@` symbol. The value of an attribute is given in square brackets immediately after its name. Thus, in Figure 2, the `While_do` element has two attributes, `count` and `condition`. The value of the `count` attribute is 0 and the condition attribute has a string value that defines the condition to be checked before executing the loop. Moreover, the `While_do` has a `Sequence` subelement.

The tasks have several attributes, including the `status` of the task, which takes one of the following values: `not_ready`, `ready`, `running`, `finished`. A `not_ready` task is one which is not eligible to start immediately because of precedence constraints. A task is in `ready` state when it becomes eligible to start; however, it has not been assigned to a specific worker. Upon assignment to a worker, it moves into the `running` state and a completed task is in the `finished` state. In this paper, we assume that all tasks initially have the status `not_ready`, although other approaches can be taken. It is conceivable that subtree nodes of a workflow enactment, like `choice` and `parallel` nodes, may also be assigned status values. In our example, we can see that the enactment is currently running the `manager-approval` task. Moreover, the `check-budget` tasks in the first `parallel` node has already finished, and the `technical-approval` task is ready to start. Also, note that the attribute `branch` is associated with a `choice` node and its value gives the `index number` of the branch that was chosen. However, if this attribute is missing for a `choice` node, it means that the enactment has not yet reached this node. Similarly, a `While_do` node has a `count` attribute that stores the number of times the loop has been executed.

2.3 Architecture

This subsection briefly discusses how an XML query engine could fit into an overall workflow management system architecture. To this end, we can see in Figure 3 a slightly modified version of the Workflow Management Coalition (WfMC) architecture. At the heart is an enactment service (also called workflow engine) that runs workflow enactments or instances. Other components that interface with this service are process definition tools, client applications and applications that are invoked by the enactment service. The service also interacts with administration and monitoring tools. In Figure 3 we have singled out the XML query processor as one of these tools. The processor will receive requests from an end user applications regarding a workflow schema or the status of an enactment. It will in turn ask the enactment service to send the corresponding instance(s) to it. Then, the query processor will run the query against the enactment(s)

and return an answer to the user. The query processor may also be invoked by another workflow enactment service.

3 Simple Queries

This short section considers straightforward queries that are simple to express *against XML specifications of enactments*. These queries are "local" in two ways: in essence (i) they focus on individual nodes of an XML tree, and (ii) they focus on a snapshot of an enactment. This contrasts with the queries of the following sections, which consider relationships between different parts of an XML tree, and the expected behavior of an enactment over time.

We focus here on some representative queries:

(S.1) what is the status of a task?
(S.2) list all tasks with a given status?
(S.3) what tasks are currently working with a data object?
(S.4) what is the number of times a `While_do` loop has been executed in an enactment?

In addition to giving information about a single enactment, these queries can be used against the set of all currently active enactments, to give a picture of where the "hot spots" are.

All of the queries in this section can be expressed using XPath [CD99]. For example, to answer queries of type S.1, i.e., find the status of task 'shipping', the XPath query would be:

```
/Route//Task[@name="shipping"]/@status
```

The answer to this query would give information about the status of this task as to whether it is `not_ready`, `ready`, `finished`, etc.

A query of type S.3 can also be expressed in a straightforward way in XPath. For instance, to find the tasks that are currently `running` with a data object, say `invoice`, the query would be written as:

```
/Route//Task[@status="running" and (@d_read = "invoice" or
                @d_update="invoice" or @d_create="invoice)]
```

Queries S.2 and S.4 can also be expressed is a similar way using the XPath constructs to get information about the status of the workflow.

4 Traversal Queries

This section considers richer queries, *against both schemas and enactments*, that involve graph traversals. Some of the queries involve traversals of the XML tree representing a schema or enactment, e.g., to determine the relationship between two tasks. Other queries involve traversals of the (annotated) flowchart that embodies the workflow schema (enactment). For both kinds of traversals, we rely heavily on the ability of the XML Query Algebra to express structural recursion, and also use a built-in function for `parent`[4].

The following queries are considered here:

[4] A built-in function for `ancestor` would provide even more succinctness.

(T.1) what is the relationship of two tasks (parallel, sequential, disjoint)?
(T.2) what is the set of tasks that *may be* touched by a currently active enactment?
(T.3) what is the set of tasks that *must* be touched by a currently active enactment?
(T.4) what is the set of tasks that lie inside a `While_do`?
(T.5) among the tasks inside a `While_do`, what tasks will definitely be executed in each iteration through it?
(T.6) can a given data object be updated further?
(T.7) what is the expected (average, maximum, minimum) estimated running time for a schema/enactment (or a part of it)?

Each of these queries has a variety of uses. Query T.1 can be used to check whether two tasks might compete for updating a data object, or whether two related tasks might occur in `Parallel` (even though that should be prevented). The remaining queries help with analyzing future resource usage of an enactment (or group of enactments), in terms of processing or personnel required, the data objects used, and the timing of when things will complete and when resources will be needed. Queries T.2, through T.6 focus more on *throughput*, while query T.7 focuses on *response-time*. Queries T.2 through T.6 can give information about bottlenecks and "hot spots" where further resources should be allocated over the long term. There is subtle difference between T.2 and T.3 in that T.2 returns a list of all tasks that may be done, while T.3 gives all tasks that will definitely be done. Thus, T.3 returns a subset of T.2, and does not include the tasks inside a Choice element. As a matter of fact, our example queries are approximate in the sense that they do not examine the actual conditions residing at `Choice` or `While_do` nodes. In general, until a branch has been selected at a choice node, it is not possible to predict which path the enactment may take. Thus, all the branches of such a choice element of an enactment are treated as "may be's". Queries T4 and T5 are variants of T2 and T.3, respectively, in that they pertain to tasks that lie inside `While_do` loops. Query T.7 is focused on the short-term and can be used to trigger exceptions if timing thresholds will be exceeded, help re-schedule tasks to alleviate short-term bottlenecks, and to modify future time commitments made about when existing or new enactments might be completed.

As we shall see shortly, to answer these workflow queries we need a structural recursion capability as well as user-defined functions. Both these features are lacking in XPath but they are supported by the XML Query Algebra. Then, the five types of queries can be grouped into three categories and implemented in the XML Algebra.

4.1 Relationship queries

A query of type T.1 to check the relationship between two tasks can be written in XML Query Algebra, using a user-defined function called *leastCommonAnc*. Depending upon whether the least common ancestor of two tasks is a `Sequence`, `Choice` or `Parallel`, the relationship between these two tasks is accordingly one of sequential, choice or parallel, respectively. We first write an ancestor function `anc` to check if one task is an ancestor of another and then use it in the *leastCommonAnc* function.

```
fun anc(t1:AnyTree;t2:AnyTree):Boolean =
   if (t1 == t2) then true
    else
   let p = parent(t2) do
     if p = () then false
      else anc(t1;p)
```

The above function takes two arguments, t1 and t2, and returns a boolean answer (true, if t1 is an ancestor of t2). Note that the `anc` function uses the parent function proposed in XML Query Algebra. The ancestor function can in turn be used to write a `leastCommonAnc` function which determines the least common ancestor of two tasks, or in general, of any two nodes in the XML tree.

```
fun leastCommonAnc(t1:AnyTree;t2:AnyTree):AnyTree =
   if anc(t1;t2) then
     t1
   else
     ( if anc(t2;t1) then t2
       else leastCommonAnc(parent(t1);parent(t2)))
```

This algorithm also takes two arguments and returns a tree rooted at the least common ancestor node. It works by checking if one of the two nodes is an ancestor of the other; if so, that is the answer. On the other hand, it considers the parents of each of the nodes and checks again for the ancestor relationship between them. Since both the nodes belong to the same tree, a least common ancestor is eventually found. Finally, the next function finds the relationship between two distinct tasks. It calls the `leastCommonAnc` function and prints out the results, i.e., the type of the node which is a least common ancestor. Note that two distinct tasks cannot have a least common ancestor of type `Route` or `While_do`, since those node types have a single child.

```
fun PrintRelation(t1:Task;t2:Task):String{0,*} =
   match  leastCommonAnc(t1;t2)
      case v:Parallel do "Parallel"
      case v:Choice do "Choice"
      case v:Sequence do "Sequence"
      else()
```

4.2 Status queries

The next three types of queries relate to the status of a running enactment with regards to unfinished tasks and pattern of data object usage. One useful query of type T.2 is to list all the tasks that *may still be* performed on an enactment. This query can be performed by function `maybe-tasks`.

The full text of this function is omitted for lack of space; however, a brief description follows. `maybe-tasks` takes a starting or root node of an enactment as input and walks recursively, depth-first, through the tree, listing the "unfinished" tasks that it encounters. This is accomplished by recursively calling the child

nodes of a given node (using the nodes function) until all nodes are exhausted. In the case of a choice node, the branch attribute is checked first. In a running enactment, the value of this attribute is set to the index of the path that is taken after a choice is made. However, an empty value in this variable means that this node has not been reached yet. Hence, all the branches should be listed since any one of them *may be* selected. On the other hand, if a value for the branch attribute is known, then only that branch is pursued. This function may be modified slightly to produce a list of nodes that *must be* traversed to handle a query of type T.3. For this query, the case of a choice node would simply be ignored in the above function. For brevity, we omit the listing of the `mustbe-tasks` function. Moreover, since queries T.4 and T.5 are similar in flavor to T.2 and T.3, respectively, they are left as an exercise for the reader.

Next, we give an example of a query of type T.6 to check if a data object, say d1, will be updated by a subsequent task in the enactment after, say, task t2. The following function `more-updates` takes two input parameters, a tree name corresponding to a running enactment, and a data object name (as a string).

```
fun more-updates(t1:AnyTree; d1:String):Boolean =
  let list1 = maybe-tasks(t1) do
    not(empty(
         for t in list1 do
            match t
               case t':Task do
                 where contains(t'/@d_update/data(),d1) do t'
               else ()
        ))
```

The above function calls the `maybe-tasks` function to make a list of tasks that may still follow further in a running enactment. Then for each task in the list, it checks if the task writes to the data object of interest. If a match is found then a true value is returned, else the answer is false.

4.3 Time computation queries

Lastly, as an example of a type T.7 query, consider the computation of the expected flow time for an enactment. The following `flow_time_left` function, also expressed in the XML Query Algebra, computes the expected time for the completion of an enactment, starting from a given enactment state[5].

```
fun flow_time_left(t1:AnyTree):Float =
     match  t1
      case p:Parallel do max(for c1 in nodes(p) do flow_time_left(c1))
      case c:Choice do dot_product(c/@probs/data(); c)
      case s:Sequence do sum(for c1 in nodes(s) do flow_time_left(c1))
      case t:Task do if t/@status/data() != "finished" do t/@time/data() else do 0
      case w:While_do do w/@repeat_factor/data() * flow_time_left(nodes(w))
      else ()
```

[5] Variations of this function for minimum and maximum expected time can easily be constructed.

We assume that an attribute `time` giving the expected time has been assigned for each task. For a `Parallel` step, function `flow_time_left` includes the maximum expected flow time for its children elements; for a `choice` element, the weighted average is computed using a function `dot_product`. This function (not specified here) takes two lists as argument, computes the product of corresponding pairs, and takes the sum of those products. Here we apply `dot_product` on a list of probabilities for the children of the `choice` (stored in choice node attribute `@prob`) and the `flow_time_left` values for the children of the choice. With a `While_do` node, we compute the `flow_time_left` for its child, and then multiply by a `repeat_factor`, which gives the expected number of times the `While_do` iterates. If none of the tasks have `finished` status, then this function will give the total expected flow time for a schema from start to finish.

5 Schema Construction

This section illustrates a novel application of query processing to workflow schemas. In particular, we show how the representation of workflow schemas in XML along with emerging XML query languages can be used to help support the automated construction of workflow schemas. The central idea of the approach is based on a form of hierarchical planning [EHN94], in which workflow schema templates of differing granularity are selected from a repository and then expanded by filling in the slots of those templates appropriately, using additional templates (some of which have no slots), also from the repository. This approach can be used for the on-demand, automated production of specialized workflow schemas, as illustrated in this section. It can also be generalized for use in connection with e-services composition, that is, combining e-services that are provided over the internet and/or the telephony and wireless networks.

To summarize, this section describes how to specify queries of the form:

(C.1) Build a workflow schema from a template and a set of other templates that have been selected for the slots of that template

The overall approach to building workflow schemas was introduced in [CHKS01]. Here we focus almost exclusively on how the XML Query Algebra is used to support the approach.

5.1 Overview of approach and example application

We view workflow schema construction as a form of *workflow mediation*, because of its analogy with database mediation [Wie92]. However, in our context mediation focuses primarily on enterprise processes, rather than on enterprise data. A workflow mediator can be used to help insulate one organization from the intricacies of interacting with multiple other organizations that together provide some coherent family of e-services. For example, a representative (workflow) mediator might be used by an organization such as Lucent to substantially automate the selection and purchase of PCs (including outsourced assembly and shipping).

Workflow mediators include three main modules, for Planning, Execution, and Data Transformation. The *Planning module* builds workflow schemas based

```
WQMTemplate [
  @name [ "PC_purchase1" ],
  Route [
    Data_list [ "pc_model", "assembler_address", "pc_invoice", ... ],
    Sequence [
      Parallel_sync [
        Slot [ filler_name [ "buy_pc_template" ] ],
        Slot [ filler_name [ "buy_modem_template" ] ],
      ],          (* end Parallel_sync *)
      Slot [ filler_name [ "assembly_template" ] ],
    ]             (* end Sequence *)
  ]               (* end Route *)
]                 (* end Template *)
```

(a) `PC_purchase1`, example template with slots for purchasing and assembling a PC

```
WQMTemplate [
  @name [ "buy_from_Micron1" ],
  @params [ "model", "ship_to", "invoice" ]
  Route [
    Sequence [
      Task [ @name [ "send_order_to_Micron" ],
             @address [ "buy_Micron1.exe" ],
             @d_read [ "model", "ship_to" ],
             @d_update [ "invoice" ]
      ],          (* end Task *)
      ...         (* more Tasks *)
    ]             (* end Sequence *)
  ]               (* end Route *)
]                 (* end Template *)
```

(b) `buy_Micron1`, an example base template for purchasing a PC from Micron

Fig. 4. Example templates from a repository `mytemplatedb`

on goals to be achieved (e.g., investigate possible PCs to be purchased; execute the purchase and assembly of selected PCs). The outputs of the Planning process are workflow schemas expressed in XML (and in our current context, in WQM); these are executed by the *Execution module*. The *Data Transformation module* is essentially an XML query processor, which is used by the other two modules.

5.2 Workflow templates

This and the next subsection together provide an illustration of how the Planning module creates workflow schemas, using the technique of *schema splicing*. This subsection illustrates the pieces of workflow schema that are used, and the next one illustrates how they are spliced together.

In WQM, workflow schemas can be completely specified, or might be *templates* which provide the high-level specification of a workflow but include `Slot` elements where selected other templates can be inserted. Templates without slot elements are called *base templates*. Figure 4(a) shows `PC_purchase1`, an extremely simplified template that might be used for purchasing PCs. Figure 4(b) shows `buy_from_Micron1`, a simplified base template that might be used to fill the `buy_pc_template` slot of `PC_purchase1`. In `PC_purchase1`, a sequence of two activities will occur. The first activity involves the parallel execution of two inserted tasks (which will involve ordering items from a PC vendor and a modem vendor, respectively, and having them shipped to an appropriate location). The

second activity consists in executing an inserted task, which asks an assembler to assemble the PC and modem, and ship to another location.

The base template `buy_from_Micron1` may be used to purchase a PC from vendor Micron. The schema for this template includes a task which is to send the `pc_model` and `ship_to` address to Micron, and receive an `invoice` in return. The attribute `params` permit parameter passing in and out of the template; in general these refer to the XML data that a workflow enactment will manipulate. The attribute `address` of the task gives the executable file containing the program required to perform the task. This program might invoke a wrapper or other functionalities that are resident in the mediator, or provided by external systems. In practice, this schema should include actions to be taken if Micron doesn't respond in a timely fashion, or if the PC model is not available, and actions to ensure that the receiving party eventually receives the PC in good condition.

5.3 Schema splicing

In general, "schema splicing" refers to the creation of new workflow schemas from existing workflow schemas and templates. Figure 5(a) illustrates a mapping (with type `Mapping`, not specified here) that provides the correspondence between the parameters appearing in `PC_purchase1` and either scalar values (e.g., `"Millennia MAX XP"`) or selected elements (e.g., `buy_from_ Micron1`) of a template repository. We assume that the Planning module has selected `PC_purchase1` and constructed `mymapping` assigning the parameters of `PC_purchase1`.

Figure 5(b) shows an XML Algebra function that can be used to replace slots of a template with other templates, choosing the other templates according to a mapping such as `mymapping`. Figure 5(c) shows a query that will fill in the template `PC_purchase1` using `mymapping`. We now describe the operation of this function and query in some detail[6].

The function `fill_slots` of Figure 5(b) takes in a `WQMtemplate` and "instructions" for how to fill the `Slots`, and produces a WQM workflow schema that has no `Slots`, i.e., a fully grounded WQM workflow schema. As with some of our previous queries, `fill_slots` involves structural recursion at the outer layer. Speaking intuitively, the function `fill_slots` performs a depth-first traversal of input `x` (of type `WQMTemplate`), and produces a new, somewhat modified "copy" of `x`. For each element of type `Slot` (i.e., in the first case of the `case` expression) it chooses an appropriate base template from `templatedb` and places it in the output XML object. In the opposite, (i.e., in the `else` part of the `case` expression), the function copies the high-level structure into the output XML object, and recursively calls `fill_slots` on the inner parts of that element.

We now give more detail on function `fill_slots`. Given a `Slot` element `x'`, we first find the element `m` in `mapping` that matches with `x'` (i.e., such that `m/filler_name = x'/filler_name`). For that element `m` we next find the matching template `t` in `templatedb` (i.e., such that `t/@name = m/value`). Finally, we

[6] We assume for now that each slot of the input template will be filled with base templates. That is, we do not have to recursively fill in the slots of templates used to fill other slots.

```
let mymapping : Mapping =
  map [ filler_name [ "pc_model" ],
        value_name [ "Millennia MAX XP" ] ],
  map [ filler_name [ "buy_pc_template" ],
        value_name [ "buy_from_Micron1" ],
        param_assign [ "pc_model", "assembler_address", "pc_invoice" ] ],
  ...
```

(a) Part of example mapping used for schema splicing

```
fun fill_slots(x:WQMTemplate;
               mapping:Mapping;
               templatedb:TemplateDB ) : WQMTemplate =
  match x
    case x':Slot do
      for m in mapping do
        where m/filler_name = x'/filler_name do
          for t in templatedb do
            where t/@name = m/value_name do
              fill_params(t/route/*; m/param_assign)
    else ~(name(x))[for v in nodes(x) do
                      fill_slots(v; mapping; templatedb)]
```

(b) Recursive function used to splice templates together

```
query
  for x in mytemplatedb do
    where x/@name/data() = "PC_purchase1" do
      fill_slots(x/*; mymapping; mytemplatedb)
```

(c) Query used to construct a particular schema

Fig. 5. Data, function, and query for constructing a workflow schema

generate from template `t` the content needed for the output XML object. This content is basically the routing part of `t`, namely `t/route/*`. However, we apply a substitution function `fill_params` (not defined here), that replaces all occurrences of the parameters of `t` listed in the `@params` attribute by the parameter values indicated in the `fill_params` element of map `m`. For example, this would replace parameters `"model"`, `"ship_to"`, and `"invoice"` in `buy_from_Micron1` by the parameter values `"pc_model"`, `"assembler_address"`, and `"pc_invoice"` in the first slot of `PC_purchase1`.

The `else` clause of `fill_slots` involves some aspects of XML Algebra that we haven't seen before. Intuitively, as function `fill_slots` performs the traversal of input `x`, when it comes to a non-`Slot` node n then it produces a node in the output XML object that has the same name as n (this is achieved by the construct `~(name(x))`). The contents of the replacement of node n are specified within the square brackets. To produce this content, `fill_slots` is called on each of the children of the original n.

The query of Figure 5 invokes `fill_slots` on the template from `mytemplatedb` whose name is `PC_purchase1`, i.e., the template of Figure 4(a). The output of this query will be a (ground) workflow schema for buying a PC from Micron. The mapping `mymapping` should be used when enacting this schema, so that the input data objects for the enactment (e.g., `pc_model`) will be initialized appropriately.

In the above example just a single layer of hierarchical planning was used, i.e., slots in the top-level template (`PC_purchase1`) were filled with base templates. In general, multiple layers can be used, such that slots of the top-level template are in turn filled with successive lower-level templates, and finally with base templates at the lowest level of the hierarchy. In terms of the XML Query Algebra function `fill_slots`, this recursion is achieved by having a call to `fill_slots` embedded within the function `fill_params`.

6 Conclusions

We believe that using XML to represent and query workflow schemas and enactments opens new perspectives in workflow management within or across organizations. In particular, it permits improving the design and efficiency of workflows, both at compile-time and run-time. For example, potential bottle-necks and race conditions on data usage might be found at compile-time using analysis queries, and potential delays might be detected and averted at run-time by using status queries. This functionality is required for specifying, enacting and supervising e-services in various e-commerce application contexts.

To this end, we have introduced a simple yet expressive workflow model, called WQM, that uses flowchart-based constructs that are "properly nested". Properly nested schemas are particularly convenient for querying with XML query languages. Indeed, we have demonstrated the power of the XML Query Algebra (offering structural recursion and user-defined functions for traversing workflow schemas) to express a broad range of queries against properly nested workflow schemas and enactments, and to dynamically construct workflow schemas. It should be finally stressed that WQM abstracts several commercially available WFMSs, based on procedural or associative task-based models. There is ongoing work on WQM in order to enable a generic wrapping in XML of heterogeneous workflow models. It remains open how our techniques can be applied in the context of flowchart-based workflow models that are not properly nested, or those based on Petri nets or state charts.

Acknowledgements

The authors thank Jérôme Siméon and Geliang Tong for help in specifying many of the queries in this paper in the latest version of the W3C XML Query Algebra. The third author is currently on leave from the University of Colorado, Boulder.

References

[CD99] J. Clark and S. DeRose. XML Path Language (XPath). Technical report, World Wide Web Consortium, 1999. W3C Recommendation 16 November 1999.

[CHKS01] V. Christophides, R. Hull, A. Kumar, and J. Siméon. Workflow mediation using VorteXML. *IEEE Data Engineering Bulletin*, 24(1), March 2001.

[DC00] D. Florescu D. Chamberlin, J. Robie. Quilt: An xml query language for heterogeneous data sources. In *WebDB'2000*, pages 53–62, Dallas, US., May 2000.

[EHN94] K. Erol, J. Hendler, and D.S. Nau. Semantics for hierarchical task-network planning. Technical Report CS-TR-3239, Computer Science Department, University of Maryland, 1994.

[FFM+00] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The XML query algebra. W3C Working Draft 07 June 2001. Available at `http://www.w3.org/TR/query-algebra/`.

[FSW01] M. Fernandez, J. Siméon, and P. Wadler. A semi-monad for semi-structured data. In *Proc. of Intl. Conf. on Database Theory*, 2001.

[GT97] A. Geppert and D. Tombros. Logging and post-mortem analysis of workflow executions based on event histories. In *Proc. 3rd Intl. Workshop on Rules in Database Systems*, Skoevde, Sweden, June 1997.

[KHB00] B. Kiepuszewski, A. ter Hofstede and C. Bussler On Structured Workflow Modelling In *Proc. CAISE '00*, Stockholm, Sweden, 2000.

[KAD98] P. Koksal, S. Arpinar, and A. Dogac. Workflow history management. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(1), 1998.

[KZ98] A. Kumar and L. Zhao. XRL: An extensible routing language for electronic applications. In *Intl. Conf. on Telecommunications and Electronic Commerce*, 1998.

[LO01] K. Lenz and A. Oberweis. Modeling Interorganizational Workflows with XML Nets In *Proc. of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, January 2001. Available at `http://dlib.computer.org/conferen/hicss/0981/pdf/09817052.pdf`.

[MM00] M. Maloney and A. Malhotra. XML schema part 2: Datatypes. W3C Recommendation, October 2000. Available at `http://www.w3.org/TR/xmlschema-2/`.

[MK00] , M. zur Mühlen and F. Klein. AFRICA: Workflow interoperability based on XML-messages In *Proc. of CAiSE*00 Workshop on Infrastructures for Dynamic Business-to-Business Service Outsourcing (IDSO'00)*, Stockholm, June 2000.

[OMG98] Object Management Group. Workflow management facility, joint submission bom/98-06-07, revised, July 1998. Available at `ftp://ftp.omg.org/pub/docs/bom/-98-06-07.pdf`.

[SGW00] G. Shegalov, M. Gillmann, and G. Weikum. Xml-enabled workflow management for e-services across heterogeneous platforms. In *1st Workshop on Technologies for E-Services (TES)*, Cairo, Egypt, September 2000.

[TBMM00] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. W3C Recommendation, October 2000. Available at `http://www.w3.org/TR/xmlschema-1/`.

[TAKJ00] A. Tripathi and T. Ahmed and V. Kakani and S. Jaman., Implementing Distributed Workflow Systems from XML Specifications, Available at `http://www.cs.umn.edu/Ajanta/papers/asa-ma.ps`.

[vdA98] W. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[vdAK01] W. van der Aalst and A. Kumar. XML based schema definition for support of inter-organizational workflow. Technical Report in review, CU Boulder, 2001.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[WMC99] Workflow Management Coalition. Workflow standard - interoperability Wf-XML binding document number wfmc-tc-1023, April 1999.

[WSFL01] Web Services Flow Language (WSFL), IBM Corporation. Available at `http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.