A Restful Workflow Implementation on Top of Distributed XQuery

Nicola Onose
UCSD

Rania Khalaf
IBM Research

Kristoffer Rose
IBM Research

Jérôme Siméon
IBM Research

ABSTRACT

Workflow languages are the norm when it comes to representing and implementing business processes. With the emergence of Web-enabled workflow languages, such as BPEL, there is an increasing need to support XML processing along with those languages. In this paper we extend the REST-based workflow language Bite with XQuery processing capabilities. We show how the resulting language can be implemented on top of a stand-alone XQuery processor by compiling its core constructs into DXQ, a distributed extension of XQuery. From an XQuery perspective, this approach demonstrates the expressiveness of the DXQ framework. From a workflow perspective, it opens interesting opportunities for light-weight implementations of Web workflows, cross-activity optimization, and experimentation with distributed workflows.

1. INTRODUCTION

Workflows are used extensively to model and implement business processes that involve both computational and human activities. Traditional applications of workflow technology include supply chain management, scientific analysis, procurement, etc. In the context of Web applications, Workflow activities typically involve XML data manipulation and rely on Web services infrastructure as a messaging layer. BPEL [24], which relies on WSDL [9], is probably the most commonly used in the context of large enterprise-wide applications and central to a number of commercial products [30]. However, the growing interest in Web 2.0 technology has led to the development of new approaches, notably based on REST-ful Web services [12]. The main motivation for this trend is to alleviate the perceived complexity in developing and deploying Web-based applications that involve Web services access, data processing, and human interaction. Bite [10] is a recent proposal for a light-weight, REST-based alternative to BPEL, capturing BPEL's core functionality. In this paper, we present an extension of Bite with XQuery processing activities, and show that the resulting language can be implemented directly by using Distributed XQuery [11], with minor extensions. Integration between database and workflow technology has been investigated in a number of contexts [6, 7, 15, 5, 2, 28]. Our approach is original, as it leverages the concurrent and distributed features of the DXQ extension for XQuery [11] and it is the first one to implement workflows using an XML query platform. We believe it to be interesting to the XQuery community because it steps into a new application area and it demonstrates the power and flexibility of the language.

Salsa Workflow. Bite is able to express simple to complex processes in which activities involve computation or Web interactions specified through HTTP requests. As an example, consider the sample workflow in Figures 1 and 2, which implements (part of) a simple application allowing users to look for and enroll in classes of Latin dances in the New York area. The graph representation is quite standard, and similar to that used for BPEL or other workflow languages.

In our extension of Bite, most activities are either HTTP interactions or XQuery expressions (complemented by loops, assignments etc.). The first part of the flow, depicted in Figure 1, contains just the first exchange of HTTP messages: the user opens a Web browser and types the URL of the application, issuing an HTTP request. The server receives the message (the receiveGET activity), does some data processing (in the xquery) activity, and sends back a response (built in the replyGET) containing an HTML form that can be used to select dance types.

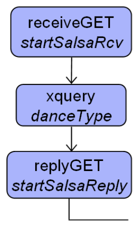

Figure 1: First part of the flow: HTTP request/response

The second part of the flow, depicted in Figure 2, consists of a *structured* activity: a while loop that contains other activities inside it. The loop is activated once the activities in Figure 1 are completed and continues as long as the value of the $enrolled variable stays false.

An iteration inside the while activity starts by enabling a receivePOST that waits for the dance types sent by the user. Once the form data are received, the flow engine starts in parallel two xquery activities that call remote services (GoogleBase and Craigslist) to retrieve information regarding relevant dance classes in the New York area. The information is aggregated in a subsequent xquery activity that joins the control flow. Its output is then used in a replyPOST

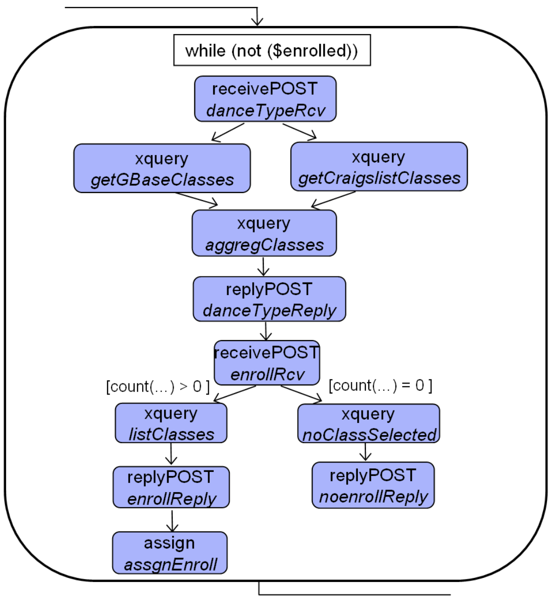

Figure 2: Second part of the flow: *while* loop

that sends to the user a form for class enrollment and enables a new receivePOST. The latter retrieves the answer, mapped into an XML message. If the count of chosen classes is zero, (the right branch in the figure) the server sends a reply saying that no class was selected and, since all other activities either have completed or are disabled, another loop iteration is started. If there are any chosen classes (the left branch in the figure), the engine constructs and sends a reply displaying them and calls an assign activity that sets the $enrolled variable to true, leading to the termination of the loop.

Besides support for sequential and parallel activities, as well as `while` loops which were illustrated here, Bite supports a number of other control flow expressions (e.g., the ability to *pick* between several choices). We will cover the rest of the language in Sections 2 and 3. As we will see, all of Bite's control flow capabilities can be implemented using a combination of imperative features and parallelism that are available as part of the Distributed XQuery language [11].

Challenges and Approach. Creating the infrastructure to support such applications raises a number of challenges. First, the data sources are RSS feeds, so we must provide XML processing capabilities. Not surprisingly, we use XQuery for that purpose. The choice of XQuery 1.0 ensures that the programmer can rely on a well-established language, and provides more than enough expressiveness for potentially data intensive applications. The design of Bite facilitates this integration by providing a specific script activity that allows to inline processing in a language supported by the runtime engine. Still, a number of specific semantic issues must be addressed, notably the relationship between Bite and XQuery variables, as well as scoping rules.

The second challenge has a less clear approach, and it concerns the choice of an implementation strategy. In many cases, workflow engines delegate the task of data processing to a DBMS, which leverages the powerful infrastructure it provides, but often requires heavy system integration and makes deployment complex. In addition, this often makes optimization of the data processing across multiple activities difficult. In the XML area, the availability of libraries for XPath or XQuery processing somewhat facilitates integration and deployment, however it leaves open most of the optimization issues. A variety of more integrated approaches have been used, such as implementing the workflow semantics using database triggers [5], or the nested relational calculus [14], or building full-fledged data processing within the workflow engine itself [15]. Our solution relies on a compilation and runtime infrastructure that is itself XQuery–based. More concretely, we exploit XQuery's expressiveness and recently developed imperative [8] and distributed features [11] to execute Bite's workflow language. Interestingly, since Bite's syntax is XML based, the compiler is also written as an XQuery 1.0 program that transforms Bite specifications into a DXQ program.

Contributions. The paper makes the following contributions.

- We present an extension to the Bite workflow language with XQuery activities.
- We describe the compile-time and runtime architectures and show how the workflow language can be compiled into a Distributed XQuery (DXQ) program.
- The approach has been implemented on top of the Galax XQuery processor, and tested on workflows that are small, but cover typical usage scenarios, such as the one serving as an illustration in this paper.

We would like to point out that the proposed approach yields promising additional benefits. First, both data manipulation and workflow control/data flow semantics are handled seamlessly. Second, the compiler can serve to provide an executable formal operational semantics. It also opens interesting avenues for future work, such as optimization by analyzing the queries and control structures and distributed flow execution, or making use of the distributed execution capabilities of DXQ to deploy distributed workflows.

Organization. The rest of the paper is organized as follows. Section 2 describes the XQuery extension of Bite. Section 3 explains how to compile Bite workflows into DXQ. Section 4 describes the implementation and the architecture of our system. We discuss related work in Section 5 and conclude in Section 6.

2. BITE WITH XQUERY

This section presents the Bite language, focusing on its control and data flow features, and describes our proposed XQuery extension. A more detailed presentation of Bite can be found in [17].

The basic Bite process model consists of an acyclic graph containing activities connected by conditional links. The graph allows a restricted type of nesting: only one type of activities (while) can be nested and links are not allowed to cross the boundaries of a nested activity. The execution semantics of links and activities is the same as the BPEL flow activity with suppressJoinFailure set to *yes*. Once the activity that is the source of a link completes, the condition is evaluated, and the status of the link is set to the boolean value obtained, or to true, if no condition was specified. An activity which is in the default (initial) state and has incoming control links, must wait that all activities it depends on terminate or are disabled, and then evaluates a predicate over the values of the links, called the *join condition*. If the result is *true*, then it enters the enabled state, meaning it can be scheduled to run, provided it receives all its inputs (which can also be external messages). If the result is *false*, then it becomes *disabled*, and its outgoing links are set to false. Error handling is provided by special error links to error handling activities. The semantics also includes Dead-Path Elimination (DPE) [19]. DPE is a technique of propagating the disablement of an activity via its outgoing links so that downstream activities do not hang waiting for it.

A Bite flow exposes itself as an HTTP-accessible service, responding to GET, PUT, POST, DELETE requests targeted at the different URLs of the flow's entry points. It composes interactions with other services over HTTP and with people over email. Unlike BPEL, no typing for the flow services is required (i.e. no WSDL, Schema, etc) and service endpoints (URL, email address, etc) are encoded directly into the calling activities.

Bite itself has only a small set of basic activities; however, this set is extensible by design, enabling users and developers to define new extension activities. The basic activities consist of (1) communication primitives for receiving and replying to HTTP requests (receiveGET/PUT, replyGET/PUT) and making HTTP requests to external services (GET, POST, PUT, DELETE), (2) utility activities for waiting, calling local code, or terminating the flow, (3) control helpers such as structured loops (while) and external choice, which enables reacting to an exclusive choice from a set of possible external inputs (pick). The state of the flow is stored into variables, described in detail later in this section. Bite adapts the pick construct of BPEL, by turning it into a flat activity whose output variable contains which choice was taken (using an index of the choice's name if given), and the received message data. The process may use the variable like any other, especially in transition conditions to go down a different branch based on the selected choice. while activities resemble loops in classical programming, however they must take into account parallelism. Thus, the loop control must wait for all activities nested inside the while to be completed or disabled before it reevaluates the condition. Explicit side effects are modeled as assign activities that copy data resulting from the evaluation of an expression to a variable or a location inside a variable.

A Bite flow can be executed on several different runtimes catering to different platforms: a servlet container, an HTTP server embedded in the Java platform, the IBM Project Zero [16] where Bite is provided as "the Assemble Flow language" and others.

Bite extension activities created by the user/developer community can be plugged into a Bite runtime. The activity implementations may be defined in any language that is supported, for extension activities, by the chosen runtime (in Project Zero, either Java or Groovy). We exploit that built-in extensibility by allowing users to use XQuery extension activities. An example of an XQuery extension activity is shown below. The input element has the body of the query that reads a variable named salsaClassRcv_Output, containing the output of the activity salsaClassRcv, and constructs a sequence of RequestSummary elements. The control element defines a control link, which here states that the xquery activity cannot run until the activity named *salsaClassRcv* has completed. The input and control XML elements are part of the Bite language and are available for every activity type.

```
<xquery name="transform">
  <input>
    <![CDATA[ 'for $x in $salsaClassRcv_Output return
     <RequestSummary>
        Dance {$x/dance_type/text()},
        Location {$x/location/text()}
     </RequestSummary>' ]]>
  </input>
  <control source="startSalsaRcv"/>
</xquery>
```

Data in Bite is stored in variables that are visible to all activities and expressions in the flow. These global variables can either be declared explicitly, at the beginning of the flow specification, or implicitly, as the target of an assign. In addition, the convention is that the output of an activity is contained within an implicitly defined variable, as salsaClassRcv_Output above. The standard semantics of Bite is more permissive, allowing variables to be directly used in any expression, as in Javascript. However, in order to be able to use XQuery, which has stricter scoping rules, we allow only the types of flow variables described above.

The use of shared variables makes data flow implicit, but explicit data dependencies can be specified using some syntactic sugar.

Our XQuery extension has also support for XQuery prolog statements, which can be inserted inside the initialization element of a reserved Bite variable. All declarations from that prolog are visible to the XQuery expressions inside the flow. However the XQuery variables are not visible at the flow level and should be distinguished from Bite variables.

3. COMPILATION INTO DXQ

3.1 DXQ Overview

DXQ is composed of several layers of extensions over XQuery 1.0, namely: XQueryP [8] which extends XQuery 1.0 with procedural features and immediate update application, DXQ [11] which extends XQueryP with closures and distributed computation. We briefly review the main features of DXQ, notably those relevant to our workflow compiler. At the language level, DXQ distinguishes the interface of a module from its implementation. Module interfaces include declarations of functions and variables that are exported. A client query may remotely call functions in a module exported by a DXQ server, to which it can refer by using an import interface statement. DXQ also adds a let server implement expression, which dynamically asserts that a DXQ server at a particular URI implements an imported interface. The expression language is further enriched with two remote evaluation expressions, from *Server* return { *Expr* } and at *Server* do { *Expr* }, which redirect evaluation of an expression to a DXQ server synchronously or asynchronously. In the context of implementing Bite, we merely need to observe that DXQ asynchronous remote calls result in parallel evaluation. The idea is that each remote call creates a thread and we can create local threads by declaring a DXQ server Self bound to the current module, as illustrated on the following code snippet[1].

```
import interface namespace Workflow = "URI"
  at "interface-file";
declare server Self implements Workflow;
...
at Self do { E }  (: creates a thread and evaluates E :)
...
```

In order to provide control between those threads, we add two new DXQ library functions for thread synchronization: dxq:wait, to wait idly for a thread to complete, and dxq:signal, to signal thread completion.

Architecturally, DXQ is used by deploying one or more XQuery that can communicate through HTTP. DXQ's native support for HTTP makes it fairly natural to implement the REST-based approach of Bite. In Section 4 we describe specific tweaks needed in the run-time architecture of DXQ to precisely handle Bite's HTTP interface. Finally, we also added one function for explicit HTTP calls, glx:http-request, which is used to implement http requests from within Bite activities. It takes as input the HTTP method, the URL and the request content, which must be XML.

[1]The top-level server declaration is a straightforward extension from [11] that is available in the implementation.

3.2 Compilation Approach

We adopt the XQuery datamodel for any data manipulated by the flows and use XQuery 1.0 as our expression language.

Shared resources and environment. The state visible to expressions and activities in the flow is the state residing in the variables of the flow, by definition of the flow language itself. Our mapping maintains the semantics and makes the state accessible via XQuery variables. We are also compliant with Bite with respect to which assignments to global variables are persistent. Assignments can be explicitly made persistent either by specifying them in an assign activity or in the initialization part of a flow variable. In addition, there are implicit assignments to an Output variable, as a result of running the activity to which that variable corresponds (as in the semantics of Bite). At implementation level, the entire state of each process, including process variables, state (disabled, enabled, default, completed), condition variables is stored in the global environment, and updated inside critical sections.

Activities. The main principle of compilation is to create one DXQ function for each activity, based on the intuition that a function can be called asynchronously to reflect the triggering of an activity. Each function is started in a new thread using the approach explained earlier and it is passed as argument a process id, designating the *logical instance* to which it belongs.

Messaging. An important aspect of this approach is to isolate the interpretation of incoming messages into a dispatch function that forwards the content of the HTTP requests to the appropriate process and activity.

Control flow. As XQuery is also our chosen expression language, transition, join and loop conditions are mere XQuery expressions. They are boolean predicates testing variable values (stored in the global environment) or data (retrieved through queries). In the current implementation, the join condition is the conjunction of the (boolean) values of all incoming links, but it can easily be extended to allow for generic predicates. The conjunction was chosen because it facilitates the implementation of an aggregation scenario, as the one used several times in Section 1.

Calling queries. The choice of XQuery as expression language provides a seamless integration of data management into workflows. XQuery queries can be used in any expression, either in xquery activities or in other constructs that contain expressions (assign, variable initialization, conditions used in control flow). A query can reference global variables and external XML instances, as in a regular XQuery module.

3.3 Flow Compilation

In the following, we show on some examples how each essential Bite flow construct is compiled into DXQ.

branching control flow A typical branching example is given in the top part of Figure 2: the termination of *danceTypeRcv*, a receivePOST activity, enables two xquery activities: *getGBaseClasses* and *getCraigslistClasses*. The generated code contains two asynchronous calls that spawn the threads corresponding to the two new activities:

```
...
at Self do {Self:getGBaseClasses($pid)};
at Self do {Self:getCraigslistClasses($pid)};
...
```

joining control flow paths Consider, in Figure 2, the *getGBaseClasses* and *getCraigslistClasses* activities that together enable *aggregClasses*. The functions implementing *getGBaseClasses* and *getCraigslistClasses* end by calling a helper function that decrements a counter (initially set to 2), and, upon reaching zero, launches *aggregClasses*. Here are the corresponding portions of the body of that helper function:

```
... (: enter critical section :)
let $new_cnt := $aggreg_cnt - 1 return
  replace value of node
  $env/entry[@pid=$p]/aggreg_cnt with $new_cnt;
...
if ($new_cnt eq 0) then
  at Self do {Self:aggregClasses($pid)
...
```

dead path elimination If all incoming control links are set to false, an activity is *disabled* and its output control links set to false. Following the *dead path elimination* rule, the workflow engine has to disable recursively other activities whose incoming links become false, according to the join condition. (Remember that in the current implementation the join condition is a conjunction.) The implementation is straightforward: generate one special *DPE-function* for each activity and call the DPE-functions of the activities that depend on it. Here is the most significant part of the of the body of aggregClasses_dpe, the DPE-function generated for *aggregClasses* from Figure 2:

```
if ($status_getGBaseClasses eq ``disabled'' and
    $status_getCraigslistClasses eq ``disabled'')
  then {
    replace value of node
      $z:env/entry[@pid = $z:pid]/
       activity[name = ``aggregClasses'']/status
    with ``disabled'';
    local:danceTypeReply_dpe($z:pid) }
  else ()
```

transition conditions The flow language allows for refining the control links by adding predicates called transition conditions. When the activity that is the target of the link is a receive or pick activity, if the transition condition is false and its state is set to *disabled* (i.e. the dispatch will not forward messages to it), otherwise, if all incoming links have true values, the state is set to *enabled*. If it is not a receive or a pick activity, then, if the condition is true, it is started asynchronously, otherwise, only its DPE-function is called.

For instance, in Figure 2, the *listClasses* and *noClassSelected* activities both depend on the second receivePOST *enrollRcv*. But they are in fact mutually exclusive, because the transitions are annotated with the conditions checking that the number of chosen classes is strictly positive and that it is zero, respectively. We show below the fragment of code generated to implement the first of the two transition conditions:

```
if (count($enrollRcv_Output/class_choice) > 0)
 then at Self do {Self:listClasses($z:pid)}
 else local:listClasses_dpe($z:pid)
```

receive/reply While activity threads run asynchronously, the external HTTP interface is synchronous: for a given HTTP request, the web server must return an HTTP reply. Therefore the thread of a receive has to wait for the reply to complete and produce its result. In practice, we use synchronization primitives (condition variables and mutexes) to block the receive, make the reply function store its Output at a special environment location and signal the waiting thread. The latter can then retrieve the result and return it to the client.

For instance, for the activities in Figure 1,[2] the blocking receive thread runs the code:

[2]One may notice that this example could be optimized into only one function that does all the work of the three activities. We plan to exploit these kinds of rewritings in the future.

```
at Self do {Self:danceType($z:pid)};
dxq:wait($cv1, $mx1);
$env/entry[...]/startSalsaReply_Output/node() }};
```

and the reply thread wakes the receive thread after it finishes its computation:

```
...
replace value of node
$env/entry[...]/startSalsaReply_Output with ...
...
dxq:signal($cv1)}};
```

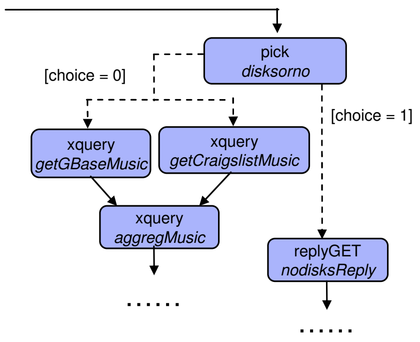

Figure 3: Sample pick activity

pick Our implementation of pick supports only choice based on incoming messages, hence it is conceptually similar to grouping several receives (which we call choices) together, such that when one of them is started, all others are disabled. In DXQ this is fairly easy, as it is enough to generate one function that processes all requests for a pick activity because the URL to which the message was sent to uniquely identifies a particular choice. All these steps can be implemented based on conditional statements.

We will examplify by looking at the pick in Figure 3 that allows choosing either a flow path for buying music or to skip that step, continuing the flow described in Section 1. For the link that is activated on the second choice (choice = 1) and triggers a replyGET (corresponding to the case in which the user does not want to buy any disks), the code will be:

```
if ($disksorno_Choice = 1)
  then at Self do {Self:nodisksReply($z:pid)}
  else local:nodisksReply_dpe($z:pid)
```

The code corresponding to the first choice is similar, but it will have two asynchronous calls, for the two xquery activities.

while Each time an activity inside the while completes, or is disabled, it decrements a counter, initialized at the beginning of each iteration. When the counter reaches zero, it reads the values of the variables from the context and it tests the loop condition. If it is true, then it re-initializes the state of the activities in the while. For the loop in Figure 2, this translates to a while loop within the function of the while activity:

```
while (not $enrolled) return {
  ... (: enable activities within the loop
        and init. counter :)
  dxq:wait($cv_loop, $mx_loop);
  ... (: re-read vars. from the environment :)
}
```

The compiler also creates a helper function that decrements the counter, similar to the one for joining flow paths, which, upon reaching a zero counter, will signal the main while activity: `dxq:signal($cv_loop)` .

Thus, we were able to implement most of the Bite language, just by taking DXQ and adding a few function calls. We have not implemented data dependencies (the input element in Bite), but it does not seem to pose any big challenges. There are other missing features such as error links and timeouts, for which DXQ has no support yet. Also, since we are focusing on XML processing, we limited the expression language to XQuery and we do not support any other extensions, while Bite allows for arbitrary extensions.

4. IMPLEMENTATION

Figure 4 gives an overview of the architecture of our workflow compiler and of the runtime engine.

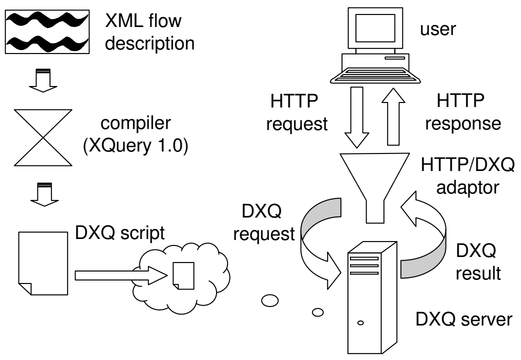

Figure 4: Compilation and Runtime Architecture

Our compiler implementation consists in 915 lines of standard XQuery 1.0, that we tested with two distinct open source XQuery processors: Galax [13] and Saxon [26]. The compiler takes as input a description of the flow, in XML format, and produces a DXQ module that implements it. It starts by generating the part of the prolog common to all applications, containing the declarations of the global environment and of the associated mutex. The next step is to generate a special *dispatch* function (henceforth called *dispatcher*) that accepts requests and forwards them to the appropriate instance or creates a new instance. Multiple workflow instances are simulated by using threads that carry logical process ids and by partitioning the global environment based on those ids. For a new instance, the dispatcher needs to create environment locations for variables used in synchronization, as well as for global variables of that process, shared by all its threads. The rest of the compiler is dedicated to implementing the compilation principle "one function per activity", where each flow constructs is implemented as described in Section 3.

The target platform of our compiler is the DXQ runtime. After compiling a workflow specification, the resulting DXQ module and interface are run as any DXQ server. In order to provide an external interface compatible to the protocol used by Bite (pure REST), we add an adaptor which acts as a proxy between the HTTP clients and the DXQ runtime. The adaptor is also used to translate non-XML input, such as the one coming from an HTML form, into an XML encoding. When a request is received, the adaptor extracts the URL and out of the URL (standard Bite convention) the process id, if specified. The id, together with the content and the HTTP method of the request are sent as arguments in a call to the dispatcher of the

DXQ server identified by that URL. If the process id is mentioned in the URL, then the dispatcher forwards the request to an enabled activity of that process. Otherwise, a new instance, with a new process id, is created and a new entry in the global environment is allocated.

5. RELATED WORK

A number of projects have studied the interaction of Database technology and Workflows. VorteX [15], GridDB [21], DFL [14] propose models specifically adapted for data-centric workflows. The BPQ project [4] focuses on querying business processes expressed in BPEL. The emergence of Web-enabled workflow languages has resulted in a renewed interest in the integration of data manipulation features with workflow technology. In [28], Vrhovnik et al extend a BPEL engine with SQL capabilities. Most existing commercial BPEL implementations support data queries using SQL [25, 23], XPath [25], or XQuery [7, 30]. Our work follows a similar idea, but focuses on a more light-weight workflow language designed for Web 2.0 applications. In addition, we rely on an original implementation approach that enables a tighter integration between the data processing and workflow components (through compilation of workflows into a database language) instead of relying on a more complex system integration. Several other recent work focus on Web data manipulation and data flow such as Yahoo Pipes [31], DAMIA [3], XProc [29] or AXML [1]. Each of those efforts include support for XML as the data format and are optimized towards data manipulation, but do not have full workflow capabilities.

From an implementation point of view, several techniques have been proposed to support workflow processing directly on top of a database system, e.g., using triggers [5], or mapping into an Object-Oriented Database [2]. A system [20] supporting the FDL language, a precursor of BPEL and therefore having a closely related semantics, stores a representation of navigational state in a database. Navigation is then performed by executing appropriate SQL calls on top of the database system. Srivastava et al. [27] study optimization of relational queries over web services, considering precedence constraints, which can encode only a small part of BPEL's semantics. Even though we implement the workflow semantics using a "query language", our use of DXQ's support for concurrency relates our approach to using languages for concurrent and distributed programming e.g., the pi-calculus [22] or Orc [18], to implement workflows.

6. CONCLUSION

In this paper we have described the integration between XQuery and the REST-based workflow language Bite, and shown how the resulting language can be implemented on top of DXQ, a distributed extension of XQuery. The proposed compiler supports all the control flow features of Bite, and is well suited for the deployment and execution of light-weight Web applications that require data processing and human interactions. We believe this work opens an interesting new avenue for the XQuery language. It also provides some opportunities for future work on query optimization in the context of workflow processes, as well as on distributed workflows by exploiting the features existing in DXQ.

7. REFERENCES

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

[2] A. Ailamaki, Y. E. Ioannidis, and M. Livny. Scientific workflow management by database management. In *SSDBM 1998*, pages 190–199, 1998.

[3] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh. Damia: a data mashup fabric for intranet applications. In *VLDB*, pages 1370–1373, 2007.

[4] C. Beeri, A. Eyal, T. Milo, and A. Pilberg. Monitoring business processes with queries. In *VLDB*, pages 603–614, 2007.

[5] C. Blecken. Media360 workflow-implementing a workflow engine inside a database. In *VLDB*, page 692, 2000.

[6] What to look for when BPEL 2.0 becomes available for public review. http://www.jpasley.com/2006/07/what-to-look-for-when-bpel-20-becomes.html.

[7] BEA WebLogic integration 8.1 documentation. Using the BPEL export tool. http://edocs.bea.com/wli/docs81/bpel/export.html.

[8] D. Chamberlain, M. Carey, D. Florescu, D. Kossmann, and J. Robie. XQueryP: Programming with XQuery. In *XIME-P*, 2006.

[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1, W3C note, 2001.

[10] F. Curbera, M. J. Duftler, R. Khalaf, and D. Lovell. Bite: Workflow composition for the web. In *ICSOC*, pages 94–106, 2007.

[11] M. F. Fernández, T. Jim, K. Morton, N. Onose, and J. Siméon. DXQ: A distributed XQuery scripting language. In *XIME-P*, 2007.

[12] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, UC Irvine, 2000.

[13] Galax: An implementation of XQuery. http://www.galaxquery.org.

[14] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. DFL: A dataflow language based on petri nets and nested relational calculus. *Inf. Syst.*, 33(3):261–284, 2008.

[15] R. Hull, F. Llirbat, E. Siman, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modification and dynamic browsing. In *WACC '99*, pages 69–78, 1999.

[16] IBM. Project Zero, 2007. http://www.projectzero.org/.

[17] R. Khalaf, N. Mukhi, and S. Weerawarana. Service-oriented composition in BPEL4WS. In *WWW2003*, May 2003.

[18] D. Kitchin, W. R. Cook, and J. Misra. A language for task orchestration and its semantic properties. In *CONCUR*, pages 477–491, 2006.

[19] F. Leymann and A. Altenhuber. Managing business processes as information resources. *IBM Systems Journal*, 33(2), 1994.

[20] F. Leymann and D. Roller. *Production Workflow*, chapter 10. Prentice Hall, 2000.

[21] D. T. Liu and M. J. Franklin. The design of GridDB: A data-centric overlay for the scientific grid. In *VLDB*, pages 600–611, 2004.

[22] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, January 2007.

[23] Windows workflow foundation. http://msdn2.microsoft.com/en-us/netframework/aa663328.aspx.

[24] OASIS. *Web Services Business Process Execution Language Version 2.0*, 11 April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[25] Oracle BPEL process manager. http://www.oracle.com/technology/products/ias/bpel/index.html.

[26] Saxon: The XSLT and XQuery processor. http://saxon.sourceforge.net.

[27] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB '06*, pages 355–366, 2006.

[28] M. Vrhovnik, H. Schwarz, O. Suhre, B. Mitschang, V. Markl, A. Maier, and T. Kraft. An approach to optimize data processing in business processes. In *VLDB*, pages 615–626, 2007.

[29] N. Walsh, A. Milowski, and H. S. Thompson. XProc: An XML pipeline language, W3C working draft, 2008.

[30] Websphere process server. http://www-306.ibm.com/software/integration/wps/features/.

[31] Pipes: Rewire the web. http://pipes.yahoo.com.