Towards Continuous Workflow Enactment Systems

Panayiotis Neophytou, Panos K. Chrysanthis, and Alexandros Labrinidis

University of Pittsburgh, Pittsburgh, PA 15260, USA
{panickos,panos,labrinid}@cs.pitt.edu

Abstract. Traditional workflow enactment systems and workflow design processes view the workflow as a one-time interaction with the various data sources, executing a series of steps once, whenever the workflow results are requested. The fundamental underlying assumption has been that data sources are passive and all interactions are structured along the request/reply (query) model. Hence, traditional Workflow Management Systems cannot effectively support business or scientific monitoring applications that require the processing of data streams. In this paper, we propose a paradigm shift from the traditional step-wise workflow execution model to a continuous execution model, in order to handle data streams published and delivered asynchronously from multiple sources.

Keywords: workflow, continuous workflows, patterns, data streams.

1 Introduction

Many Enterprises use workflows to automate their operations and integrate their information systems and human resources. Workflows have also been used to facilitate outsourcing or collaboration beyond the boundaries of a single enterprise, for example, in establishing Virtual Enterprises [1]. Recently, workflows have been used in the context of scientific exploration and discovery to automate repetitive, complex and distributed scientific computations that often require the collaboration of multiple scientists [4,7,9].

A common class of applications in both business and scientific domains is monitoring applications that involve the processing of continuous streams of data (updates) [3]. Examples include financial analysis applications that monitor streams of stock data to support decision making in brokering firms and environmental analysis applications that collect and analyze sensor data to support discovery of air and water pollution.

Most recent workflow enactment/management systems orchestrate the interactions among activities within a workflow along the lines of web services [18]. Several business process modeling languages have been designed to capture the logic of a composite web service, including WSCI [19], BPML [2], BPEL4WS (with the latest update WS-BPEL 2.0 [8]), BPSS [15] and XPDL [20]. However, these interactions are usually one-shot interactions between the sender and the

E. Bertino and J.B.D. Joshi (Eds.): CollaborateCom 2008, LNICST 10, pp. 162–178, 2009.

receiver of the request and it is not clear whether or not these existing workflow management systems and languages are suited for monitoring applications.

Our goal in this paper is to examine the capability of current workflow models and workflow management systems to support business and scientific monitoring applications. We will base our examination on the Workflow Pattern framework which was developed in [17]. This framework proposed a set of 20 common workflow patterns and a set of 6 communication patterns in [13]. This framework was used to evaluate the capabilities of the languages mentioned above in [22] and [10], showing that these languages could not support nearly half of the 20 workflow patterns, and also 2 of the communication patterns. These 2 communication patterns are Publish/Subscribe and Broadcast which, interestingly, are essential for enabling monitoring applications. YAWL [16], which is a more recent workflow definition language, makes the effort to support all the workflow patterns, but there is no reference in its definition for the support of the two communication patterns which are prevailing in monitoring applications.

The two missing communication patterns from existing workflow models are a direct result of the fundamental assumption that data sources in workflows are passive (e.g., stored in databases or data files) whereas data consumers (users, tasks) are both active and passive. These two missing communication patterns assume that some data sources are active, supporting continuous data streams.

In order to address the lack of support for continuous data streams in existing workflow models, in this paper, we consider a paradigm shift towards the idea of "continuous" workflows (analogous to the recent data processing shift from Database Management Systems to Data Stream Management Systems). The main difference between traditional and continuous workflows is that the latter is continuously (i.e., always) active and continuously reacting on internal streams of events and external streams of updates from multiple sources at the same time at any part of the workflow network.

The key contributions of this paper are:

1. Identify the limitations of the current workflow model in terms of supporting streams of data events, internally and externally.
2. Propose a new Continuous Workflow model and introduce two key primitives to support it, namely, queues and window operators.
3. Illustrate how the 20 existing workflow patterns could be implemented in order to support data streams.
4. Identify 4 new Continuous Workflow patterns.
5. Illustrate the expressive power of our continuous workflow model, in terms of simplicity and flexibility, by comparing the implementation of a monitoring application in our model and using a Timed Petri net implementation [6].

The rest of the paper is organized as follows: Section 2 sets the stage by providing background to existing workflow systems. We also discuss the Workflow Pattern framework that was used to evaluate the abilities of these existing systems. In Section 3, we study the need for continuous workflow constructs and then propose the new workflow model for Continuous workflows in Section 4. We finally conclude and give our directions for future work in Section 5.

2 Existing Workflow Model

A *workflow* (also referred to as workflow process) is defined as the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules. A *workflow management system* (WfMS) is one that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants (human or machine) and, where required, invoke the use of IT tools (databases, job schedulers etc.) and applications. A *workflow process* can be defined by set of sub-processes which form part of the overall process. Multiple levels of sub processes may be combined to form a workflow hierarchy.

A *workflow activity* is a description of a piece of work that forms one logical step within a process. An activity may be a manual activity, which does not support computer automation, or an automated activity. A workflow activity requires human and/or machine resources(s) to support process execution; where human resource is required an activity is allocated to a workflow participant. A workflow activity is specified in terms of name, preconditions, actions, rules of exception handling, completion and temporal constraints. Every workflow specification formalism is built around a set of control flow relationships and concepts, such as those defined in [21]. Examples include simple one-to-one precedence constraints to denote *sequential* execution, or *OR* and *AND* relationships to denote *parallel* execution. These are subdivided into *OR-split* and *AND-split* to specify branching and, *OR-join* and *AND-join* to specify convergence to initiate the next activity in the workflow.

A comprehensive study [17] enumerates the various control patterns required by workflow applications. A pattern "is the abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts" [12]. The 20 patterns studied in [17] include more complex control structures, than the ones described by WfMC [21], such as XOR-split, Differed Choice, Multiple Instances etc. These help to define the workflow model in more detail and down to specific imperative workflow requirements. The study also elaborates on which of these patterns could be realized in workflow management systems and languages, available at the time of the study. Some of the patterns mentioned cannot be realized by these systems because their design did not take them into consideration. They then proposed a new workflow language [16] that is able to implement these patterns.

Workflow events are distinguished into internal and external events. External events, are relevant input workflow data, pushed into the workflow as a response to a request, from applications, users, databases and other entities external to the workflow. Internal events are workflow control data, as defined in [21], but limited to internally exchanged data between activities. This does not include the engine state data store in the WfMS database. Usually internal events mark the completion of an activity and signal the execution of the next one.

A *workflow request* is the initiating event of a workflow. Once it is received by the WfMS it creates a new instance of the workflow. The request includes

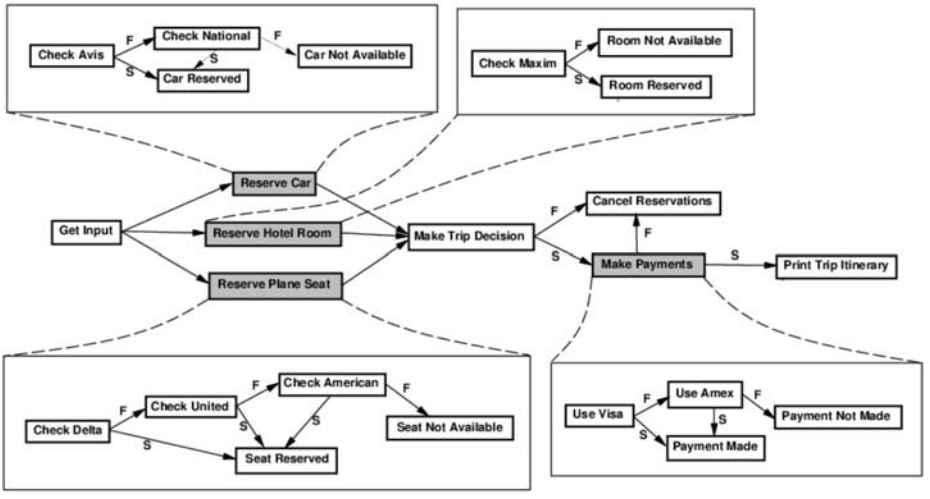

Fig. 1. Continuous Workflow enabling architecture

relevant data and constraints defined by the requester. This is the first piece of information being fed to a workflow instance.

Most workflow languages model workflows either as State charts or Petri nets. Figure 1 represents the state chart of a vacation trip booking workflow from [5], where AND-splits (and AND-joins) are implicit when more than one arrow originates from (or is coincident to) a node. For example, *Get Input* represents an AND-split and *Make Trip Decission* represents an AND-join. OR-splits and OR-joins are depicted with arrows annotated with selection conditions. For example, *Make Payments* represents an OR-split with condition **S** (Success) and **F** (Faillure). There is also the case where conditions are not mutually exclusive and more than one branches is activated. The workflow patterns observed in this workflow, are defined as WP1-WP5 in [17]. One can also discern some activities being defined as sub-processes.

Regarding the execution of activities, according to the transition definition in [21], any two activities in the same sequence cannot run in parallel. The first one will give the thread of execution to the next one. That means that if we have two activities A and B where A comes before B in a sequence, then B cannot start running (even on partial results from A) unless A is completely terminated (Figure 2).

An attempt was made, by using Time Petri nets, to apply temporal constraints on events in [6]. The effort covers some cases of workflow patterns for monitoring supply chains and reacting on events such as "Out of stock" and "Order arrived". The Petri net approach is difficult to implement and although is able to capture operations on multiple events, it cannot do it for an arbitrary number of events, known only at runtime. Also events are consumed whenever an activity is activated, and they have to be replaced if there is a need to reprocess them. These are all considerations that the designer has to make before hand.

We will take their example and show an easier way to implement the supply line patterns described, in Section 4.

In the next section we will examine if the existing workflow model is suitable to support monitoring applications.

3 From the Existing to the Continuous Workflow Model

In this section we examine the ability of existing workflow models to support monitoring applications. This analysis is based on the communication patterns described in [22] and how those can be used inside a workflow using the internal workflow patterns described in [17].

3.1 Communication Patterns

Communication patterns are divided into two categories: Pull and Push. In the pull model the data consumer gets at most one reply per request. Three patterns from [17] follow this model. (1) Request/Reply, where a sender makes a request to the receiver and waits for a reply before continuing execution; (2) One-Way, where the sender makes a request to the receiver and waits for an acknowledgment reply before continuing execution; and (3) synchronous polling, where a sender makes a request to a receiver and continues processing. It then periodically checks to see if a reply was sent by the receiver. When it detects a reply it stops polling.

In the push model the data consumer receives multiple data items per request. Two patterns interest us which follow this model. Publish/Subscribe is a form of asynchronous communication where a request is sent by a process and the receivers are determined by a previous declaration of interest. The declaration of interest could also express constraints on the kind of replies each receiver is interested in. Lastly, Broadcast, is a form of asynchronous communication in which a request is sent to all participants, the receivers, of the network. Each participant determines whether the request is of interest by examining the content.

From the aforementioned patterns current workflow management systems and languages provide support for just the pull model communication patterns. In the best of our knowledge no system provides support for either of the two push model communication patterns.

3.2 Ability of Existing Workflows to Support Push Input

As we have mentioned in the introduction, monitoring applications monitor continuous streams of data. The only way to receive updates as soon as they happen is by using a Push mechanism such as Publish/Subscribe. In existing WfMSs the only point in a workflow that is able to handle push data is at the initial activity, where the request to instantiate a workflow comes in. This way, each event belonging to a stream will be individually handled by an instance of the workflow.

The workflow is able to notify humans or machine resources in the case that a specific event needs further handling.

An alternative processing model would be to use pipelined execution of the workflow. Since a high volume of events is expected from the data stream, a pipeline model could be used to save resources. In pipelined workflow enactment, activity instances are being shared by multiple workflow instances. Each activity is thought of as a pipeline stage. Buffering takes place between steps to independently handle individual events. The control flow is handled in the same way as in the case of multiple instances of the workflow.

There two problems with the pipeline approach. First, only one stream can be supported. Second, no multiple events can be handled together since each event runs on a separate instance or pipeline stage, thus the requirement that a monitoring application needs to run on a history of events in real time, is not met.

We will now examine how existing workflow models could support monitoring of multiple streams, and see where they fall back. Consider Figure 2. In this example, activity C is required to get continuous updates from a streaming data source. Since there are currently no constructs to allow for activity C to receive events directly from a data stream and act upon them immediately, let's assume that there is a buffer between activity C and the data stream. Now activity C is polling the buffer to get the new updates. In most systems, activity D cannot run on intermediate results of C since C has to terminate and then activate D. This is the case of synchronous polling, as described in Section 3.1. To alleviate this problem one can use loops (Figure 2.b.) We set activity C to query the buffer once and return the results to D. D then might split the results to G and back to C (for it to query the buffer once again). Now since G is an AND-join, and branch E-F was activated only once (not a loop), then G will only run once, consume the event that came from E-F, and then block. It will not be able to process all the results coming out of the loop. This problem could be solved by having G feed back the same event to its input, every time it executes, so that it will process it together with the results from the loop. This implementation, although it does not provide real-time monitoring of a data stream, it shows how a workflow system can monitor a buffer of a data stream, and internally produce a stream of events using a loop. There exist though workflow definition languages like BPEL, that do not allow loops that have an output on every iteration, like the one described above. The most common loops allowed have one input and one output.

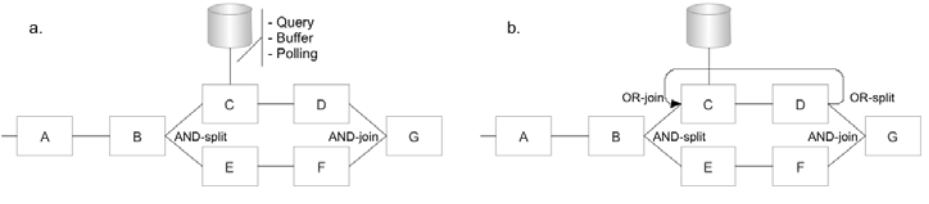

Fig. 2. Abstract Workflow example

If the designer adds another similar loop to the E-F branch then two internal asynchronous streams will be created, being joined at activity G. The results on random pairs of events would probably not make any sense. Also the two streams could have a big volume difference in number of events per unit of time, thus one of them would either drop events or should have means of buffering them. Introducing queues to the inputs of the joins should workout this problem. Moreover the results would probably make more sense if there was a way to synchronize the two streams in terms of temporal and value based functions on windows of these data, similar to the ones found in continuous queries [11].

We saw that polling is one way to monitor a stream, but this approach does not allow for real time reaction to the incoming stream, and in fact, even this is only allowed in systems where arbitrary loops are allowed. This makes us come to the conclusion that parallel execution of sequential activities is required to process streams of events, much like in the pipelined execution, because consecutive activities need to be continuously active processing the events. The difference here is that buffering of multiple events in the stream is required to be able to satisfy the requirement of monitoring applications to run on subsets of the history of the stream.

Another operation that monitoring applications need to be able to make on workflows processing data streams, is event invalidation. For example, if you have a stream from an airline which publishes fares, and a new fare update comes in that invalidates a previous fare, then the earlier update should be invalidated downstream, in order to avoid processing a fare that is invalid. Invalidation or otherwise known as cancelation is supported in YAWL [16] but it is not possible to selectivly invalidate events in the workflow, since they consider each workflow instance independent for each event and they do not support data streams.

4 Continuous Workflow Model

During our analysis in the previous section we made some observations on the functional and expressive ability of existing workflow languages and models. In light of those observations we now present our definition of "Continuous Workflows" and then elaborate on how this new workflow model can be applied to existing workflow patterns. We also identify 4 new patterns which we consider essential for new classes of applications, that require interaction with data streams either internally or externally.

Definition 1. *A "Continuous Workflow", is a workflow that supports enactment on multiple streams of data, by pipelining the flow and processing into various parts of the workflow. Continuous workflows can potentially run for an unlimited amount of time, constantly monitoring streams. To achieve that, the proposed Continuous Workflow Model introduces:*

1. *Concurrent execution of sequential activities, in a pipelined way.*
2. *Queues on the inputs of activities to buffer data in between activities.*

3. *Windows and window functions on the queues to allow the definition of synchronization semantics between multiple data streams. Windows are also used on multiple invocations of a single execution pipeline whose results are buffered in a queue. This means that an event can be considered as part of multiple pipeline invocations.*
4. *Interactions between pipeline steps. That is, the ability to notify a downstream or upstream activity of an update and cancel its execution.*

In order to introduce new synchronization semantics into Continuous Workflow we define the notion of event waves, as follows:

Definition 2. *A wave of internal events is created at a split node (or when initiating multiple instances) and it is synchronized at a join node (or when merging multiple instances). A split creates a wave of events that are disseminated in multiple branches that run in parallel. When these branches merge at a synchronizing join then all of the events in the wave must be joined and processed together. Even if an activity produces multiple events as a result of one invocation (or multiple invocations as part of a loop), then these are marked and considered part of the same wave. An event within a wave may create sub waves, creating a hierarchy of waves that need to be synchronized. These can be taken care by the system with appropriate instrumentation of the waves.*

4.1 Windows

A *window* is generally considered as a mechanism for adjusting flexible bounds on an unbounded stream in order to fetch a finite, yet ever-changing set of events, which may be regarded as a temporary bundle of events. We are introducing the notion of windows on the queues of events which are attached to the activity inputs. The windows are calculated by a window operator running on the queue. Windows are defined in terms of an upper bound, lower bound, extend and mode of adjustment as time advances. The upper and lower bounds are the timestamps of the events at the beginning and the end of the window. The extend is the size of the window. This can be defined in two measurement units: (a) Logical units, which are time based, and define the maximum time interval between the upper and lower bound timestamps. (b) Physical units, which are count based, and define the number of events between the upper and lower bounds. The mode of adjustment, also known as *window step*, defines the period for updating the window. If a step is not defined, then the window is evaluated every time a new event comes into the queue. This makes the window operator more accurate in terms of reacting to events on time, but in cases of high event rates this could seriously diminish the overall system performance. A flag called "delete_used_events" is also defined to denote if events that were used in the window that triggered the firing of an activity should be deleted from the queue. The signal to delete used events from queues comes as part of the post-conditions of an activity.

Another feature introduced for windows, is that every queue has two outputs: (1) the current window, as it is calculated according to the window constraints,

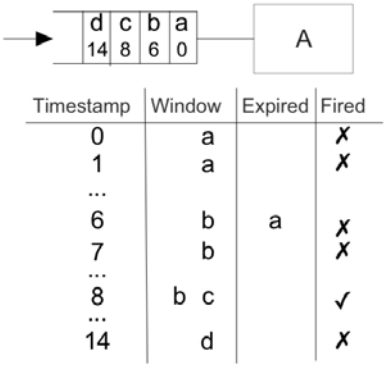

Fig. 3. Window operator example

and (2) the events that are being expired with every recalculation of a window. For example, if an order is waiting in a queue and for some reason was delayed, the window might expire it and it will be transferred to a different queue to be handled as a delayed order.

To better understand the window operator we describe an example with the help of Figure 3. Letters represent events and numbers represent timestamps (in minutes). The window attributes are: Size=5 minutes, Step=1 minute, Delete used events=true. Firing of the activity depends on the contents of the window and the preconditions of the activity which could be dependent on the contents. Assume the preconditions include $if(window.length >= 2)$ *then activate*. The window is calculated for every step (of 1 minute). If there is a change to the window operator's results then the preconditions of the activity are evaluated to determine whether the activity should be fired. If the activity is fired, then the events pushed are deleted from the queue. Notice in the example in Figure 3, that *a* was not used in firing activity A and that at timestamp 6, *a* does not fall inside the window, thus it was expired and returned to the expired output of the queue. However at timestamp 8, where the window pushed includes events b and c, the activity is fired and once it is completed these events are both deleted from the queue.

4.2 Workflow Patterns in Continuous Workflows

In this section, we consider the 20 workflow patterns presented in [17] whose implementation changes with the introduction of continuous workflow enactment, but their semantics remain the same. That means that continuous workflows are backward compatible with the existing workflows. The reader can easily verify that the examples in [17] are still valid for the patterns described here. All of the patterns except WP11 (Implicit termination) can be implemented using continuous workflows. We also introduce 4 new patterns that are unique to continuous workflows.

Basic Control Flow Patterns. These patterns capture the elementary aspects of process control. These patterns closely match the definitions of elementary control flow concepts covered in [21]. **WP 1,2** and **4** (Sequence, Parallel split and XOR-split) do not require any modifications to fit our continuous workflow model since they can be scheduled to execute without any synchronization dependencies on consecutive events.

In **WP3**, multiple parallel branches converge into a single thread of control (AND-join), thus synchronizing multiple threads. The join will be activated once all the branches have completed.

In continuous workflows, the assumption that a branch cannot be completed again before the execution of the join is relaxed, since the workflow reacts on streaming events. Events produced by multiple executions of a branch can be buffered in the join's queues. Figure 4.a shows that two data events were produced by activity A while activity B is still processing and may eventually drop the item. Given the AND-join semantics, activity C blocks until both queues have a result. Activities belonging to a branch that has already finished processing can be scheduled to execute on the next wave of events.

In **WP5**, two or more alternative branches come together without synchronization (XOR-join). The assumption in this pattern is that only one branch is activated. That means that each wave of events has only one event propagated through the only activated branch. The joining activity runs once on each wave coming into the queue.

Advanced Branching and Synchronization Patterns. This subsection includes more advanced patterns for branching and synchronization. Again the assumption for this set of patterns is that the activating stream is the same for each branch thus we use notion of waves. **WP6** (Multi-choice) and **WP8** (Multi-Merge) from this set are the same in the continuous setting, since no synchronization among events of the same wave is needed.

In **WP7**, multiple branches converge into a single thread (Synchronized Join). The join activity has to wait for all of the activated branches to finish and then execute. Some branches may not activate in which case a *null* event is propagated to the join. Figure 4.b shows a case where we have a branch (A) that has finished processing 2 events, branch (B) which is still processing the first event and branch (C)

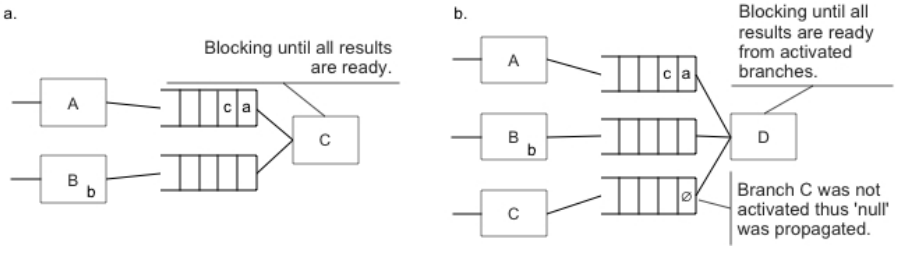

Fig. 4. (a) Pattern 3: AND-join and (b) Pattern 7: Synchronized-Merge

which was not activated and the *null* event was propagated. Activity D will only execute if all activated branches have finished.

In **WP9** (Discriminator) multiple branches merge and execution is initiated on the first result to arrive. Multiple branches are activated per wave. To avoid mixing of events from multiple waves, a cancellation signal is triggered for the specific wave after the join is finished executing. We present more details on cancellation in the WP19 pattern.

Structural Patterns. This set includes patterns for arbitrary cycles and workflow termination. Cycles are categorized in structured cycles which in programming languages resemble WHILE loops, and interleaved cycles which resemble GOTO loops, where loops can be interleaved. Termination can be explicit (a set of conditions are met) or implicit (the workflow terminates when nothing is left to process).

WP10 (Arbitrary Cycles). In the continuous workflow model arbitrary cycles can be implemented, both structured (Figure 5.a) and interleaved cycles (Figure 5.b). Arbitrary cycles make the understanding of semantics difficult since more complicated situations can arise, thus the workflow designer must be able to understand the risks and possible confusion that may arise from such a design especially in the continuous workflow model.

WP11 (Implicit termination). This patterns captures the behavior of traditional workflow systems when nodes can be terminated when activity ceases. This pattern is not directly supported by the continuous workflow model since streams of data are infinite and the only way to terminate the execution is only by defining a set of termination parameters. However, we could essentially implement similar functionality by relying on some sort of punctuations from the data sources [14].

Patterns Involving Multiple Instances. The patterns in this subsection involve cases where multiple threads of execution share the same definition. The number of instances could be known a priori or at runtime. We refer to a bundle as the set of instances required for a data event. If the results of the instances

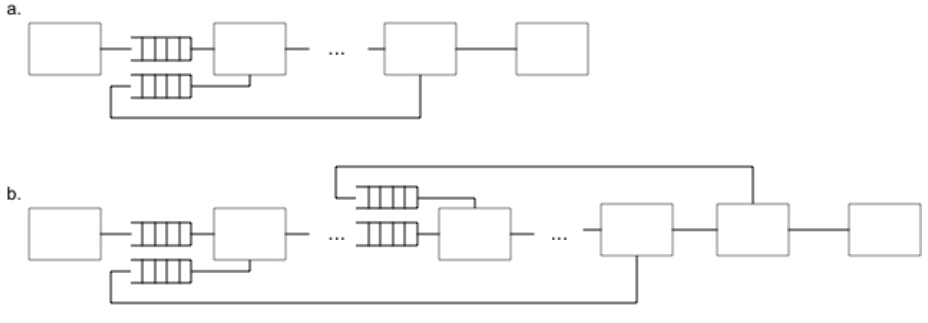

Fig. 5. Cycle patterns: (a) structured cycle and (b) interleaved cycle

of a bundle are needed for the execution of the rest of the workflow, then these results are gathered in a single queue at the activity joining the instances. The activity then runs on a window with the size of the number of instances. In the continuous model there are two cases to consider. The first one is running a number of bundles that were caused by the same number of events in parallel. The second case is running the bundles serially. When running serially, each event executes on the same bundle where the previous event was executed. The bundle will expand or shrink accordingly. Note the each bundle must finish execution before the next event can run on the same bundle.

WP12 (MI without Synchronization): This pattern refers to executing the instances in each bundle in parallel; their results are not required to be synchronized. Careful consideration must be taken if the results of an instance trigger the execution of the rest of the workflow. In the case of parallel bundles, results from multiple instances that correspond to one event can interleave with results that correspond to another event.

In **WP13-WP15** (MI with synchronization), instances within a bundle are synchronized once they end their execution. In WP13 the number of instances for each event/bundle is known at design time, in WP14 the number is known at runtime and in WP15 the number of instances is dynamic. In these cases the window size of the join node can be set to the number of instances, if the instances bundle is per-stream. If the instances bundle is per event then before moving on to the next step in the workflow each bundle has to synchronize internally and then forward the results. For WP15 to work, the process that creates the bundles has to also update the size of the windows at the corresponding queues inside the bundles.

State-Based Patterns. There are three state based patterns.

In **WP16** (Deferred choice), several branches have been activated (by AND-split or OR-split) but only one should execute. The decision is delayed until the occurrence of some event, where the one that finally executes sends cancel signals to the rest. In continuous workflows, since events are queued and the execution happens once some preconditions on the queues are met, the cancellation signal can be acted by removing the event from the activity's queue.

In **WP17** (Interleaved parallel routing), a set of activities is executed once, in an arbitrary order decided at runtime but no two activities can run at the same time for the same event. To achieve this, a list of mutual exclusion semaphores, (one for each wave of events) keeps the activities from running concurrently. The semaphore shows the wave id, a flag showing if some branch is acting on this event and a number for the active branches remaining to act on this wave. Once the number reaches 0 then the semaphore is removed from the list.

In **WP18** (Milestone), an activity can only run if a certain milestone is reached and has not expired. A milestone is a point in the process where a given activity A has finished processing an event, and a subsequent activity B has not yet started. It is important to keep track of which events have reached

a certain milestone and if that milestone has expired. A similar approach to the one in WP17 is taken, where a list is used to keep track of the waves of events.

Cancellation Patterns. There are two cancellation patterns. **WP19** and **WP20** refer to the canceling of an activity and the withdrawal of the whole workflow respectively. Activity cancellation in Continuous workflows works much like in [16], but instead of flushing everything in the queues, events are canceled only if they belong to the same wave as the event that triggered the cancellation. To cancel an activity on a specific event, you have to either remove it from the queue of the activity, or if the activity is running on that event, notify the scheduler to terminate the execution of the activity. Withdrawal of a workflow instance can only happen once the termination conditions are met, or if the user has requested to terminate the execution of the continuous workflow.

Continuous Workflow Patterns. In addition to the somewhat radical changes to the previously mentioned patterns, we now describe 4 patterns that are required in the context of Continuous Workflows.

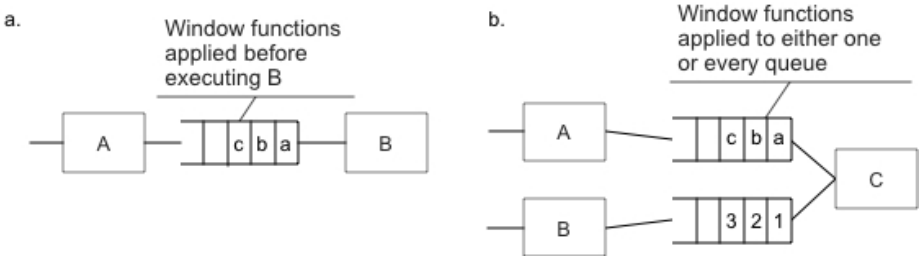

Fig. 6. (a) CWP1: Sequential aggreate and (b) CWP2: Stream-join

CWP1 Sequential Aggregate: A point in a workflow where two activities are to be run sequentially on a stream of events, one after the other. The later activity may need to run on the result of multiple invocations of the previous activity. The event results of the first activity are buffered in the second activity's queue. A function and/or window operations can be performed on a set of the resulting events as those are stored in the queue. The events in the queue can be involved in multiple invocations of the second activity until they are expired by the function. **Example**: Activity analyze_last_hour will analyze a one hour buffer of results produced by receive_temperature. The window function can also define the interval between invocation of the analysis part, like every 30 minutes.

CWP2 Stream-join: This pattern covers the case where each branch of the join activity is activated by a different stream of events. In this pattern the notion of event waves is not considered since the two streams are not synchronized. Again in this case the workflow can define functions on the individual queues. **Example**: In a travel agency application, activities receive_fares and receive_hotel_prices are joined into one stream by adding the prices.

CWP3 Stream-synch: A point in the workflow where two or more different event streams meet to get synchronized. The result is waves of events that are synchronized. This pattern is used to feed these waves to branches that require waves of events (see WP1-WP18). In CWP3, usually the slowest stream gives the pace and the other streams get sampled on some window of their events. **Example**: In a travel agency application, activities receive_fares and receive_hotel_prices are synchronized according to some window definitions, and split into pairs where the $hotel.price + fare.price < 300$. They are then handled individually but are considered part of the same wave.

CWP4 Workflow data view: This pattern refers to the ability to extract any kind of data being exchanged inside the continuous workflow and streamlining them into a separate event stream that can be used as an input to another workflow. An example usage of this pattern is to monitor the execution of the workflow and to debug it. Usually the views are not known at design time thus incorporating them into the workflow is not feasible. The view can be expressed as a set of predicates that can be evaluated on any arbitrary set of the data inside the workflow network. **Example**: Somewhere in the workflow an activity produces a result that is above expected values. The designer can add a view that will give her the message/event with the outlier value as soon as it is produced (i.e. $value > 100$). The message is annotated with meta-data regarding the activity it was last processed by.

4.3 Applicability of Continuous Workflow Patterns

We have evaluated the expressiveness of our continuous workflow model over a set of patterns which applies to the supply chain monitoring applications introduced in [6]. With the introduction of queues and window operators, designing those patterns is made much easier and it is more flexible.

In Figure 7, you can see two versions of the same pattern implemented using a Petri net approach and a Continuous Workflow approach. The pattern concerns the case where multiple occurrences of one event within a certain time period cause another event to occur. In the example shown if two out-of-stock events occur within a time period of T2 then a notification to the Supply Chain manager will be initiated. In Figure 7.a, transitions $t2$ and $t3$ wait for time interval $T2$ before consuming an event from either $e'1$ or $e''1$. This is used for expiring events that occurred time T2 ago. A notification by $t1$ is only fired if two events are allowed to coexist in $e'1$ and $e''1$.

In continuous workflows (Figure 7.b) this can be implemented by simple having a queue for Out-of-stock events and a window operator on that queue, which constructs windows of size $T2$. No step is defined thus the window is calculated for every new Out-of-stock event, and a notification is fired only if the window has two events in it (according to the precondition). Events are not deleted once used but they are eventually expired and handled by another activity, thus keeping the semantics of the two implementations the same. You can see that our implementation is much simpler. Moreover, if the designer wants to change the semantics and requires 3 out-of-stock events to happen before notifying the

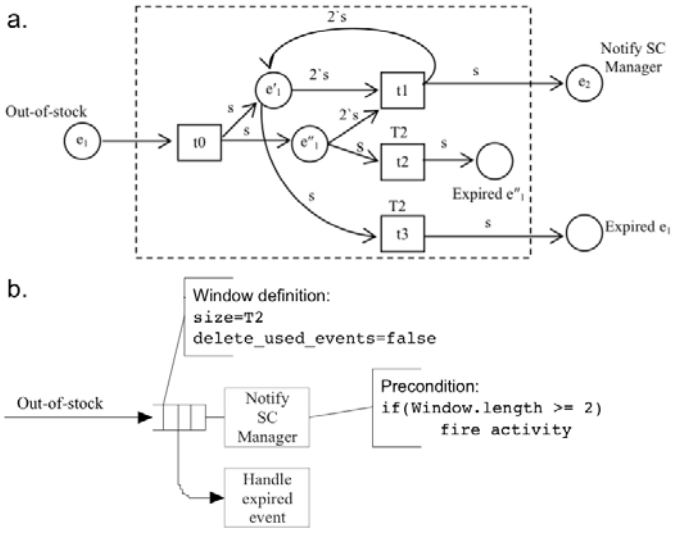

Fig. 7. (a) Petri net of "repeat cause-one effect" (b) Continuous workflow of "repeat cause-one effect"

supply chain manager, then, in the Petri net case she would have to add another transition like $t3$ and another like $t2$ and change the numbers on the arcs going to $t1$, from 2 to 3. In the continuous workflow case she would only have to change the precondition to $window.length >= 3$.

5 Conclusions and Future Work

In this paper we analyzed current workflow management systems' ability to enable the development of monitoring applications. A class of applications where it is possible for organizations to share resources and publish events, in order to enable real-time reaction management. We have shown where the existing WfMS's are lacking in functionality and proposed a new workflow enactment system which is capable of fulfilling the requirements of these applications. The new enactment model constitutes a paradigm shift from the traditional reactive model to one that is proactive towards multiple streams of events behaving in an almost random fashion.

Next on our agenda is to develop an architecture capable of enacting this new model, in an efficient and application centric way, and achieve the requested Quality of Results, in terms of response time, throughput and data quality. Three major challenges in this task: (1) Finding proper scheduling policies, that are able to handle different workflows and workloads, (2) Enable backwards compatibility with existing workflow definitions, in order to make the transition to the new model as smooth as possible, (3) Try to achieve maximum resource utilization in distributed environments, by integrating multiple workflow enactment systems and multiple resource providers (such as Grid computing platforms),

thus enabling further inter-organizational collaborations, and (4) Provide a user interface to enable the participant of a collaboration, to collectively build and maintain collections of continuous workflows.

Acknowledgement

This research was supported in part by NIH-NIAID grant NO1-AI50018 and NSF grant IIS-0534531.

References

1. Berfield, A., Chrysanthis, P.K., Tsamardinos, I., Pollack, M.E., Banerjee, S.: A scheme for integrating e-services in establishing virtual enterprises. In: RIDE, pp. 134–142 (2002)
2. BPMI. Process modeling language (bpml) (2002), `www.bpmi.org` (accessed, november 2002)
3. Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring streams: A new class of data management applications. In: VLDB (2002)
4. Churches, D., Gombás, G., Harrison, A., Maassen, J., Robinson, C., Shields, M.S., Taylor, I.J., Wang, I.: Programming scientific and distributed workflow with triana services. Concurrency and Computation: Practice and Experience 18(10), 1021–1037 (2006)
5. Ramamritham, K., Chrysanthis, P.K.: Advances in concurrency control and transaction processing. IEEE Computer Society Press, Los Alamitos (1997)
6. Liu, R., Kumar, A., van der Aalst, W.M.P.: A formal modeling approach for supply chain event management. Decision Support Systems 43(3), 761–778 (2007)
7. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the kepler system. Concurrency and Computation: Practice and Experience 18(10), 1039–1065 (2006)
8. OASIS. Web services businedd process execution language, `http://docs.oasis-open.org/wsbpel/2.0/os/wsbpel-v2.0-os.html`
9. Oinn, T.M., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, R.M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics 20(17), 3045–3054 (2004)
10. Aalst Marlon Dumas, W.M.P., Arthur, H.M., Hofstede Petia, W.: Pattern based analysis of bpml (and wsci)
11. Patroumpas, K., Sellis, T.K.: Window specification over data streams. In: Grust, T., Höpfner, H., Illarramendi, A., Jablonski, S., Mesiti, M., Müller, S., Patranjan, P.-L., Sattler, K.-U., Spiliopoulou, M., Wijsen, J. (eds.) EDBT 2006. LNCS, vol. 4254, pp. 445–464. Springer, Heidelberg (2006)
12. Riehle, D., Züllighoven, H.: Understanding and using patterns in software development. TAPOS 2(1), 3–13 (1996)
13. Ruh, W.A., Maginnis, F.X., Brown, W.J.: Enterprise application integration: A wiley tech brief (2001)
14. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. Knowl. Data Eng. 15(3), 555–568 (2003)

15. UN/CEFACT and OASIS. ebxml business process specification schema, www.ebxml.org/specs/ebbpss.pdf
16. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: yet another workflow language. Inf. Syst. 30(4), 245–275 (2005)
17. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
18. W3C. Web services glossary, http://www.w3.org/tr/ws-gloss/
19. W3C. Service choreography interface (wsci) 1.0 (2002), www.w3.org/tr/wsci
20. WfMC. Workflow process definition interface - xml process definition language, http://www.wfmc.org/
21. WfMC. Workflow management coalition: Terminology & glossary (wfmc- tc-1011) (1999)
22. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Analysis of web services composition languages: The case of bpel4ws. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 200–215. Springer, Heidelberg (2003)