# CONFLuEnCE: Implementation and Application Design

Panayiotis Neophytou, Panos K. Chrysanthis, Alexandros Labrinidis

*Advanced Data Management Technologies Laboratory*
*Department of Computer Science*
*University of Pittsburgh*
{panickos, panos, labrinid}@cs.pitt.edu

***Abstract*—Data streams have become pervasive and data production rates are increasing exponentially, driven by advances in technology, for example the proliferation of sensors, smart phones, and their applications. This fact effectuates an unprecedented opportunity to build real-time monitoring and analytics applications, which when used collaboratively and interactively, will provide insights to every aspect of our environment, both in the business and scientific domains.**

**In our previous work, we have identified the need for workflow management systems which are capable of orchestrating the processing of multiple heterogeneous data streams, while enabling their users to interact collaboratively with the workflows in real time. In this paper, we describe CONFLuEnCE (CONtinuous workFLow ExeCution Engine), which is an implementation of our continuous workflow model. CONFLuEnCE is built on top of Kepler, an existing workflow management system, by fusing stream semantics and stream processing methods as another computational domain. Furthermore, we explicate our experiences in designing and implementing real-life business and scientific continuous workflow monitoring applications, which attest to the ease of use and applicability of our system.**

***Index Terms*—workflow, continuous workflows, data streams, monitoring applications**

## I. Introduction

In the last decade we have witnessed the proliferation of data streams and the exponential increase in data production rates. This is mainly the result of the increase in bandwidth speeds and the availability of wireless broadband, and more importantly, due to the spread of heterogeneous sensors (thermal sensing, telescopes, GPS enabled sensors etc.) as well as the increasing population and world-wide distribution of smart phone devices and their applications. Smart phones enable mobile users to (over-)share their observations, ideas, opinions etc., all marked with contextual meta-data, practically rendering them as human-sensors. Moreover, these smart mobile devices provide an almost pervasive connectivity to their users, which creates an ideal collaborative environment, since it allows more frequent involvement.

This fact effectuates an unprecedented opportunity to build real-time monitoring and analytics applications, which when used collaboratively and interactively, will provide insights to every aspect of our environment, both in the business and the scientific domains. Example applications from the business domain include real-time supply chain management, on-line marketing strategy decision making through on-line analysis of social network feeds etc. In the scientific domain, examples are laboratory information management systems (LIMSs) including remotely established underwater labs to monitor sea life and conditions in real-time [1]. Other examples of applications of scientific workflows and their requirements are described in [2]. We have also been working on a collaboration platform built using continuous workflows [3] to monitor astronomers' interactions with an on-line annotations management system, as well as coordinate their reactions to transient events (e.g., supernovae, asteroids passing through etc.), to help them identify object types, find out trending objects and/or events. By exchanging opinions and feedback through our system the astronomers are collaboratively defining the morphology of the sky. All of these applications enable the collaboration between different users with each one feeding back to the workflow their own decisions, conclusions, and actions.

Most recent workflow enactment/management systems orchestrate the interactions among activities within a workflow using web services [4]. Several business process modeling languages have been designed to capture the logic of a composite web service, in the form of a workflow, including WSCI, BPML, BPSS, XPDL and WS-BPEL 2.0. However, these interactions are usually one-shot interactions between the sender and the receiver of the request; it is clear that the existing workflow management systems and languages are not suited for reactive applications.

As an alternative approach for collaborative monitoring over data streams one might consider using Continuous Queries (CQs) and data stream management systems (DSMSs) [5], [6]. The main drawbacks of CQs are: (1) have a static configuration and (2) are unable to facilitate user interaction. This makes CQs alone unsuitable as a complete solution for enabling reactive and collaborative applications.

In order to address the lack of support for continuous data streams in existing workflow models, we proposed a shift towards the idea of "continuous" workflows. The main difference between traditional and continuous workflows is that the latter are continuously (i.e., always) active and continuously reacting on internal streams of events and external

COLLABORATECOM 2011, October 15-18, Orlando, United States

DOI 10.4108/collaboratecom.2011.65

streams of updates from multiple sources, concurrently and at any part of the workflow network. We have shown in [3] how the workflow and communications patterns are cast into continuous workflows (CWfs) and also proposed four new CWfs patterns that complement the existing "traditional" workflow patterns [7], [8]. These extensions effectively enable the integration of a DSMS as a data source besides the traditional ones such as database and file systems.

We have implemented our proposed CWf model as a prototype system, called CONFLuEnCE, which is short for *CONtinuous workFLow ExeCution Engine*, and was built on top of Kepler [9]. In this paper, we describe the realization of our CWf model into a prototype system and show how CONFLuEnCE can enable reactive and collaborative applications. To show that CONFLuEnCE can facilitate both business and scientific collaborative applications we describe the design and implementation of a Supply Chain Management application, and *Astroshelf*, a collaborative exploration of the sky.

**Contributions:** In summary, our contributions are as follows:

1) We present the design of our continuous workflow model.
2) We show how our model has been implemented into a fully functional CONtinuous workFLow ExeCution Engine, namely CONFLuEnCE.
3) We present our experiences in designing and implementing two real-life business and scientific continuous workflow monitoring applications, which attest to the ease of use and applicability of our system.

**Roadmap:** Following this introduction, in Section II, we describe the continuous workflow model along with all the requirements of a continuous workflow system. In Section III we describe how all the necessary components of the model were implemented on top of the Kepler workflow management system. Then in Section IV we describe two representative applications from both the business and scientific domains which make use of CONFLuEnCE to enable real time data-stream monitoring and collaboration between users. Finally we compare our system to existing approaches in Section V and conclude in Section VI.

## II. Continuous Workflow Model

A *Continuous Workflow*, is a workflow that supports enactment on multiple streams of data, by parallelizing the flow of data and its processing into various parts of the workflow. Continuous workflows can potentially run for an unlimited amount of time, constantly monitoring and operating on data streams. Our proposed Continuous Workflow model supports these characteristics by:

- Active queues on the inputs of activities which support windows and window functions to allow the definition of synchronization semantics among multiple data streams.
- Concurrent execution of sequential activities, in a pipelined way.
- The ability to support push communication, i.e., receiving *push* updates from data stream sources.

In the following subsections we will elaborate on the basic primitive components of our continuous workflow model, namely *waves*, *windows*, and *push communication*.

### A. Waves

A wave is a set of internal events associated with an external event and as such these internal events can be synchronized at different points of the workflow. A wave is initiated when an external event $e_i$ enters the system and is associated with a wave-tag which is $e_i$'s timestamp $t_i$. When the external event $e_i$ or any internal event in its wave is processed by a task, any new internal events produced by this task become part of the wave as well. Specifically, if the task processing the event with wave-tag $t_i$ creates $n$ events then these resulting events will have wave-tags $t_i.1, t_i.2, ..., t_i.n$. The wave-tag of the last event of the wave is marked as such. This is useful when a task downstream needs to synchronize all of the events belonging to a single wave. Moreover, a sub-wave may be formed when an event which is part of a wave is processed by a task. In this case a wave hierarchy is formed where an extra serial number is attached to the wave-tag. For example, if $t_i.3$ is involved in a task then the resulting $m$ events will have wave-tags $t_i.3.1, t_i.3.2, ..., t_i.3.m$.

For example, consider a supply chain management application: When a customer submits an order with multiple products, that order is split by a task into individual data items for each product. Those data items belong to the same wave. Then the items are dispatched to the various warehouses that carry those items (usually more than one warehouse). Once the items are individually shipped then the confirmation events for each of those items are synchronized downstream all together to form the final notification to the customer to inform her that the order was shipped.

In effect, waves capture the lineage of events. Even though some workflow management systems keep track of the lineage of the processed data to be used for playback and trace-back, our model, in addition, allows the usage of this information by the application designer to enable the synchronization of these events.

### B. Windows

A *window* is generally considered as a mechanism for setting flexible bounds on an unbounded stream of data events in order to fetch a finite, yet ever-changing set of events, which may be regarded as a logical bundle of events. We have introduced the notion of windows on the queues of events in workflows which are attached to the activity inputs. The windows are calculated by a *window operator* running on the queue. The window operator will try to produce a window whenever it is asked by the attached workflow activity. When events expire they are pushed to an *expired items* queue which are optionally handled by another workflow activity. Five parameters are required to define the window semantics for that operator: *size*, *step*, *window_formation_timeout*, *group-by*

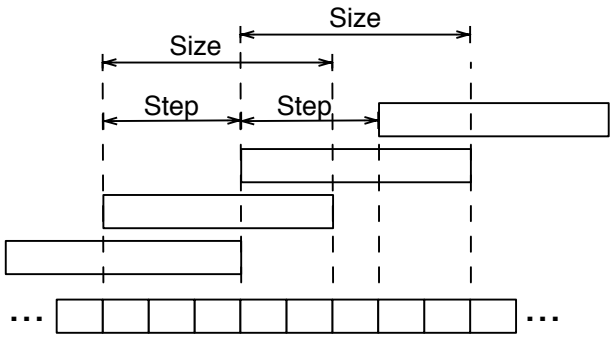


Fig. 1: Size and Step for event-based window semantics. The bottom series of boxes represent the stream of events. In this example the size is 5 events and the step is 3 events. The windows produced are represented by the boxes stacked above the queue.

functionality, and *delete_used_events*. A description of these parameters follows:

*1) Size and Step:* In general, 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. When a window is initiated its lower bound is defined and its upper bound is computed using the size. The mode of adjustment, also known as the *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.

The size and step of a window definition can be defined in four ways:

(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.
(c) *Wave-based*: where the upper and lower bounds of a window are defined by the first and last events of a wave currently being processed.
(d) *Semantics-based*: where a general predicate over the data stream can be used to define the window start/end points.

These ways can be intermixed between size and step to form disparate semantic meanings. A depiction of how the size and step parameters define the windows can be seen in Figure 1.

*2) Window formation timeout:* In the case of time-based windows, in order to produce a window, an event belonging to the next window has to appear to close the current window. In the case of sparse data streams, this could take a while and the window operator would block without producing any windows, even if the logical bounds of the current window in production have passed. Part of the window specification is the setting of a timeout to close a time-based window after $T \times x$ amount of time, where $x$ is the size of the window, and $T$ is a factor defined by the workflow designer. This means that if an event which closes the window does not arrive before the timeout, the window is automatically closed at the timeout, producing whatever events are currently within the window. Any event arriving after the timeout, which is also after the window's upper bound, with a timestamp before the upper bound, will be discarded and not considered as part of any subsequent windows. Note that only window definitions involving time require a timeout (i.e., (a) and possibly (d)).

*3) Group-by:* In many cases the streams of events contain closely related elements, where the application needs to process them in groups (e.g., calculating the average number of tweets per unit of time containing the same hashtags). This requires the window operator to support multiplexing of the stream based on some grouping attribute(s).

In our workflow model the data types could be simple (e.g., integer, string, float, etc.) or complex (records which allow hierarchy, matrices or arrays). In the case of a simple data type events would be grouped based on value equality, similarly to traditional query processing. When the data type is complex then the grouping is defined specifically for a particular element of that complex type. For hierarchical records we use an XPath-like notation[1] where you may define the grouping attribute by means of a path query (e.g., `/entities/hashtags`). For matrices and arrays we need to specify the index of the element we want to group-by.

In addition to the simple path queries, the path language for group-by's also supports functions. For example, if the `/entities/hashtags` query returns an array, but we only want to test equality based on the set of the elements in the array, i.e., ignoring the order, we could define the predicate as `as-set(/entities/hashtags)`. Furthermore, it supports multiple predicates which are evaluated in the order of their definition. For example, if we first want to group-by the user id in the tweet and then by hashtags we would define a predicate as `/userid, /entities/hashtags`.

The window operator then keeps a separate queue for each group of events, and applies the window semantics on each and every queue. Each time a new window is produced, the window operator's time is set to that window. The operator makes sure that the window produced is the earliest available window among all queues.

*4) Deleting used events:* Events on a queue could either be consumed or only used by an activity. A flag, called "delete_used_events" is used to denote consumption. That is, to denote if events that were used in the window that triggered an activity should be deleted from the queue upon their usage. The signal to delete used events from queues comes as part of the post-conditions of an activity.

**Window operator example**: To better understand the window operator we describe an example with the help of Figure 2. The goal is to trigger activity A when two events occur within 5 minutes of each other.

[1]www.w3.org/TR/xpath

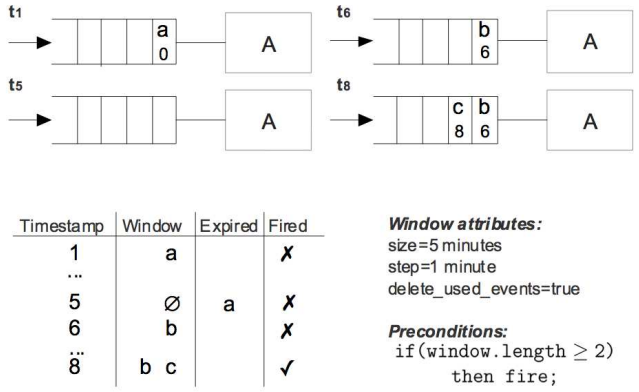


Fig. 2: The window is calculated at every step (of 1 minute). Letters represent events and numbers represent timestamps (in minutes). The window starts at timestamp 0, at which time event $a$ arrives and is enqueued. Between timestamps 1 and 5, on every minute the queue is evaluated and no action is taken, as the precondition is not satisfied, since only $a$ is part of the window. At timestamp 5, event $a$ is expired because the current window's upper bound is more than its timestamp plus the size of the window. Meaning that $a$ cannot be part of any subsequent windows. At timestamp 6, event $b$ is enqueued. The precondition is evaluated and no action is taken, since only $b$ is part of the window. At timestamp 8, event $c$ is enqueued. This time the precondition is satisfied since both $b$ and $c$ are part of the window, and that makes it of size 2 events. The activity is then triggered and once its execution is completed events $b$ and $c$ are deleted from the queue, since the "delete_used_events" flag is set.

### C. Push communication

In the push communication model, the data consumer receives multiple data items asynchronously. We are interested in two communication patterns which follow this model [10]:

*a*) *Broadcast*, the form of asynchronous communication in which a data producer sends the data items over a broadcast medium (i.e., channel), and the consumers "tune" into the channel to receive the available data. Each consumer determines whether a data item is of interest or not.

*b*) *Publish/Subscribe*, the form of asynchronous communication in which the consumers (subscribers) register their interest at a producer (publisher). Once data becomes available, the producer sends the data to the individual subscribers based on their expressed interest.

The push model has not been supported by any workflow system, until recently when, parallel to our work, another workflow management system started to support it [11]. The lack of support of these patterns so far has been a direct result of an underlying assumption that data sources in workflows are passive (e.g., data is stored in databases or data files) and data consumers (users, tasks), are the only active entities that can request and synchronously retrieve the data. These two missing communication patterns require that the data sources involved are active as well.

There are two basic ways of supporting push communications in continuous workflows. First, since a CWf is a long running process, during the initialization phase it could open indefinite connections with the data sources, from where the workflow can receive updates in real-time. A second way would be for the workflow to keep an open port waiting for connections from outside parties that want to push data to the workflow. This would mean that the end point of the workflow is well known to outsiders and fairly constant. For example a data source may be a DSMS or a third party data mediator to which a CWf can register and either open a connection (first method) or open a port (second method) to receive push data.

## III. Extending Kepler

In order to implement our continuous workflow model, we have to implement the three basic primitives presented in the previous section in addition to all the primitives provided by a traditional Workflow Management System. Instead of building a new system from scratch we evaluated a number of open sourced workflow systems such as Taverna [12] and Kepler [9]. We chose Kepler as the base for CONFLuEnCE because of our common aim to support scientific workflows and because of its extensibility. Kepler is a free open-source scientific workflow system, which was built on top of PtolemyII, a software system for modeling, simulating, and designing concurrent, real-time systems. The suitability of Kepler, for implementing our CWf model, comes from the underlying PtolemyII system: "the use of well defined models of computation that govern the interaction between components"[2]. Also, Kepler's code is inherently extensible, proven by the fact that is being actively developed by nearly 20 different scientific projects. It provides a large library of basic actors (i.e., components representing various tasks) as well as specialized actors, for easy reusability and composition of new applications. The library includes actors for database interfacing, data filtering, etc. and it is easily extensible to include domain-specific actors for actions such as automatically annotating astronomical objects.

Furthermore, programming workflows in Kepler is made easy for domain experts without any knowledge of programming structures. Kepler provides an intuitive high-level visual language for building workflows, where the designer can drag and drop components and connect inputs with outputs quite easily. Configuring parameters is easily done using dialog boxes and it also gives useful displays for debugging the workflows. Finally, Kepler was implemented in Java that simplifies our implementation of CONFLuEnCE. All in all Kepler was an ideal platform for us to build our model.

### A. Kepler's actor-oriented modeling

A workflow in Kepler is viewed as a composition of independent components called *actors*. Actors have parameters configuring and customizing their behavior which can be set statically during the workflow design as well as dynamically during runtime. Communication between them happens

[2]http://ptolemy.eecs.berkeley.edu/objectives.htm

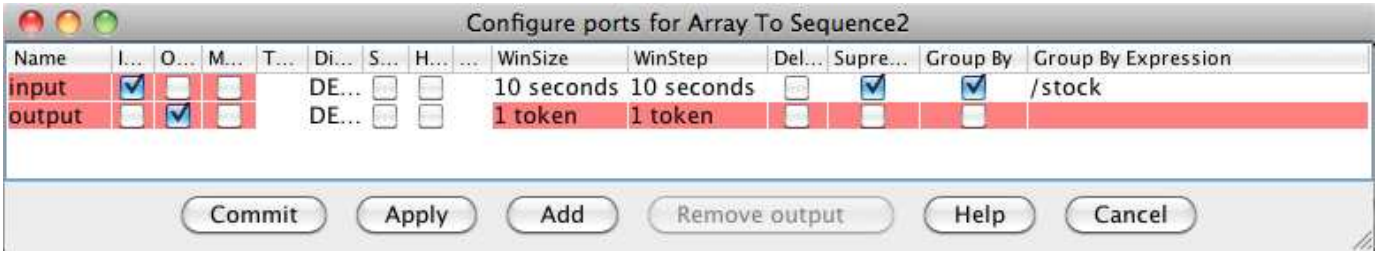


Fig. 3: Modified Kepler form for configuring actor ports. On this form the workflow designer can define in freeform text the size and step of the window associated with specific ports. Shaded cells denote non-editable parameters.

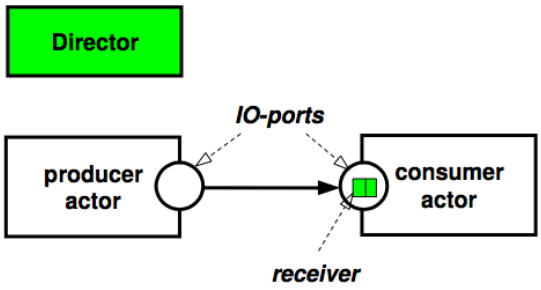


Fig. 4: The semantics of component interaction is determined by a director, which controls execution and supplies the objects (called receivers) that implement communication

through interfaces called *ports*. These are distinguished into input ports and output ports and the connection between them is called a *channel*. As part of the communication, between the two ports, a data item (referred to as token in Kepler) is propagated from the output port to the input port. The receiving point of a channel has a *receiver* object, which controls the communication between the actors. The receiver object is not provided by the actor but by the workflow's controlling entity, called the *director*. The director defines the execution and communication models of the workflow. As such, whether the communication is synchronous or asynchronous (buffered) is determined by the designer of the director, not by the designer of the actor. Figure 4 [9] shows how all these components are organized as part of a workflow.

The execution and communication model of the workflow is governed by the model of computation defined by a *director* entity. That is, given the same actor configuration, different execution semantics can be specified through the choice of a particular *director*. Kepler provides five main directors, each exposing a different model of computation.

1) The *Synchronous Data Flow* (SDF) director is designed for sequential and simple workflows with the number of tokens produced by the actors known a-priori, thus the scheduling order of the actors is defined before the execution starts.
2) The *Dynamic Data Flow* (DDF) director, like SDF, executes the workflow in a single execution thread. However, unlike SDF it does not use static scheduling, but does so at runtime, since the number of tokens produced by each actor is unknown.
3) The *Process Network* (PN) director is designed for managing workflows that require parallel processing. This director wraps each actor in its own execution thread and the workflow is driven by data availability.
4) The *Continuous Time* (CN) director introduces the notion of time for modeling workflows able to predict how a system evolves over time.
5) The *Discrete Event* (DE) director, which also works with timestamps, measures average wait times and occurrence rates. All the events (data and timestamp pairs) emitted from actors are placed in a global workflow timeline.

A detailed description of these directors can be found in [13].

Throughout the workflow execution, the director goes through a set of phases, summarized as follows (described in more detail in [9]):

1) *Pre-initialize*: Calls the pre-initialize method of all the actors just before starting the workflow execution. This phase is reached only once per workflow execution.
2) *Type-checking*: to make sure that all the data types of tokens between the sending and receiving ends of the actors are compatible.
3) *Initialize*: calls the initialize method of each actor, every time the workflow is run[3]. Initialization tokens may be transmitted from one actor to another, web-service pings may take place or other actions which need to be taken before the workflow starts running.
4) *Iteration:* A workflow run may consist of multiple iterations, where in each iteration the director calls the methods: *pre-fire* (actor tests its firing preconditions), *fire* (actor performs its main function, usually by consuming tokens from the input ports and producing results in its output ports) and *post-fire* (where the actor evaluates the postconditions and decides if it should be fired again in the next iteration) of each actor.

Workflows may be reused as part of a larger workflow (parent). They are called sub-workflows. The parent workflow views a sub-workflow as a self contained actor and manages it just like any other actor. Sub-workflows may use different

[3]Note that a workflow run is different than a workflow execution, since before a run a re-initialization of the workflow with possibly different parameter values, data input etc. takes place. An execution consists of many different runs. In the CWf model though an execution contains just one run since the workflow is long running and real-time.

director (i.e., employee different model of computation) than the parent workflow, thus forming a *hierarchical heterogeneity*.

### B. Continuous Workflow Director

The first requirement of our CWf model is the concurrent execution of sequential activities which is governed as mentioned earlier by the director entity. Concurrent execution is natively enabled by the PN director included with Kepler, since every actor is executed in parallel in its own thread. However, PN does not support any notion of time needed by the window operator. The notion of time is supported by the CN and DE directors both of which, however, do not support parallel processing. Since none of the existing directors could be used to support the CWf requirements, we decided to implement a new Continuous Workflow director by incorporating time-based techniques used by DE.

To add the notion of timed events to the PN model of computation we encapsulate each data token within an event object. The event carries its timestamp (either the creation time of the data or the time it entered the system), and its wave-tag. Since all current actors were implemented without being timestamp aware, they cannot output the timestamp of the events to the next receiving actor. To solve this problem the CWf director associates a time-keeper object to each actor at initialization time. The time-keeper keeps track of the timestamp information of the latest event processed by the actor. When the actor sends the result on a channel, the receiver objects at the receiving ends of that channel (which are receivers defined by the CWf director) will ask the time-keeper associated with the producing actor to provide a timestamp for that event. This way we also solve the problem of backwards compatibility, to support all the legacy actors available in Kepler's library. Any CWf-specific actors can be implemented with timestamp and wave awareness.

The new Process Network Continuous Workflow (PNCWf) director is defined as a class which extends the PN director, and implements the Interface for Continuous Workflow Directors (ICWF). The ICWF requires the implementation of some continuous workflow specific methods, e.g., for getting the time-keeper objects of actors, for initializing the receivers capable of accommodating the timestamped events communications etc.

Even though the PN director model of computation fits the continuous workflows model, by allowing actors to run concurrently, it is oblivious to any application or user Quality of Service (QoS) requirements since it relies on the scheduler of the underlying operating system. For this reason, we are currently developing a new CWf director that adapts the scheduling techniques proposed for scheduling continuous queries [6].

### C. Windowed Receiver

The second requirement is to add queues on the inputs of actors to buffer data. Although this is already implemented in certain models of computation in Kepler, window semantics on these queues do not exist in any model of computation. We have implemented a new type of receiver which is associated with the directors that implement the window specifications. This new type of receiver defines windows by size and step. The unit of measurement of these two parameters can be of type token, time, or wave. These parameters can be set in freeform text in the modified form which is provided by Kepler for configuring actor ports (Figure 3). Additionally the designer may define the windowed receiver as a "Group-by". As we mentioned in Section II-B3, depending on the data type of the tokens, the user may set a predicate in the form of a path query or scalar index to define the grouping element of the token. Since the group-by function may create a lot of groups, if the window definition is time-based the user may also choose to suppress the empty windows by toggling a checkbox.

### D. Push communication

We have implemented the push communication patterns described in Section II-C in three ways:

1) *Web-sockets*: An input actor initializes, within itself, a web socket server listening to a specific port. The application built on top of the specific CWf has knowledge of the port number and whenever it needs to push data to the CWf it connects to the specific port and sends the data. The use of web sockets enabled us to build applications that run on the client's browser.
2) *Direct TCP connection*: An input actor initializes a connection with a specific data stream source. This could be a DSMS, or a generic service providing streaming data (e.g., Twitter streams). The socket connection object runs within the containing actor's thread and blocks whenever there are no data to receive. When new data arrive they are broadcasted to the input ports of the actors connected to that input actor's output port.
3) *Using a mediator*: In order to support more generic data feeds such as RSS streams, we used a mediator platform called PubSubHubub[4]. This service will aggregate data updates from multiple RSS feeds and push them directly to a URL deployed by the workflow server. Once that URL is called, it will in turn forward the updates to the CWf using a predefined TCP port, much like what happens in the above case with the web-sockets server within the CWf. This allows us to integrate our workflows with a larger set of data sources.

Since we are dealing with continuous workflows, most of the time the results are also manifested as data streams. Thus the implementation also provides output actors to support the same kind of push communications as those used for the input actors. For instance, a client may connect to a known port and get results pushed to her. Another way would be using the mediator. Alternatively the output could be pushed using email, SMS, or other asynchronous types of communication as well as stored in a database that can be queried later.

[4]http://code.google.com/p/pubsubhubbub/

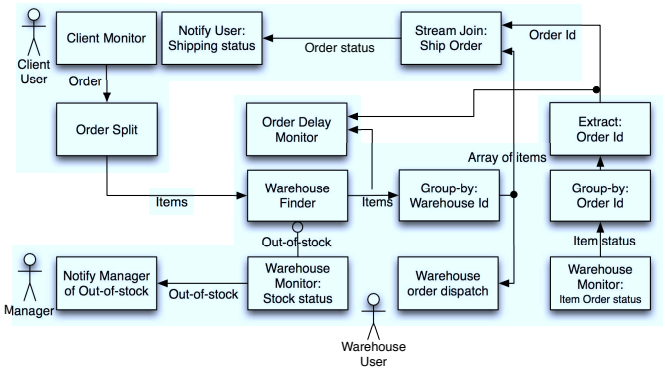


Fig. 5: High level design of a continuous workflow for Supply Chain Management reactive application.


(a) Customer's Ordering Panel (b) Company Manager's Panel

Fig. 6: Supply Chain Management application: The customer's panel where the users submit their orders. Company manager's panel updated in real-time with statistics on the number of orders submitted per second and the number of shipments made per second.

## IV. Continuous Workflow Applications

Having CONFLuEnCE as a working prototype we started building applications to show its applicability both in the business as well as the scientific domains. In this section we present two of these applications, from which the first one was demonstrated during SIGMOD 2011 [14]. The second one has been developed as part of the Astroshelf[5] project, for enabling collaboration and data analysis within the astronomy community.

### A. Supply chain management

Supply chain management applications are generally built to manage the workload of a business which handles product orders from customers, fulfillment of these orders from the warehouses and seamless bookkeeping of all of the transactions that take place. Ideally the system should provide analytics on the state of the supply chain. In this scenario the objective was to design a continuous workflow capable of serving as the integration layer between databases in the warehouses, the web server providing the user interface and ordering system, and other administrator interfaces. Additionally, it should provide real-time analytics and event notifications to help the managers alleviate problems as quickly as possible. The high-level design of the workflow is depicted in Figure 5. The users of this system are split into four categories: (1) Clients, (2) Warehouse manager, (3) Company Manager and (4) Administrator. Roles 1-3 interact with the workflow through a web interface (through a mobile device or a laptop) and role 4 interacts with the workflow directly through the Kepler interface.

A client submits orders with multiple items, using the web interface depicted in Figure 6a, and receives a notification once her order has been shipped. A warehouse manager notifies the system when an item is out of stock and also receives order requests from the system and fulfills them. Note that an order may contain objects that are available in different warehouses. The workflow takes care of routing the order requests to the appropriate warehouse manager. The company manager receives notifications when things go wrong more than once and in more than one way, e.g., when an item is reported out of stock more than once in some specified period, or when multiple orders have been delayed or canceled. The company manager also has a real time view (Figure 6b) of the current volume of orders and shipments to customers, updated every second. The statistics are computed using window semantics in certain parts of the workflow. The windows have a size of 1 second and a step of 1 second. The administrator's role is to change parameters, such as window sizes, or tune up settings in the scheduler to make the execution fit the application's requirements.

Figure 7 shows the precise specification of the continuous workflow supporting the Supply Chain Management application. The director can be seen at the top-center of the workflow definition. The workflow is also marked with the three sections, each one responsible for supporting roles 1-3. As part of this application the only actors that we had to implement ourselves were the push communication enabling actors. The rest of the workflow uses "of-the-shelf" actors provided by Kepler. Even for processing window operator results, such as counting the size of the window every second, we used the already available "Array Length" actor. The windowed receiver is designed to produce windows as array tokens, thus making the use of the "Array Length" actor, plug-and-play.

Furthermore, to test the capacity of the system and its ability to handle high loads we implemented the roles of customers and warehouse managers as automated processes that automatically interacted with the workflow. In the case of the customers the automated process would randomly pick some products from a list and submit them as an order. The warehouse manager process would wake up in intervals and

[5]http://db.cs.pitt.edu/group/projects/astroshelf

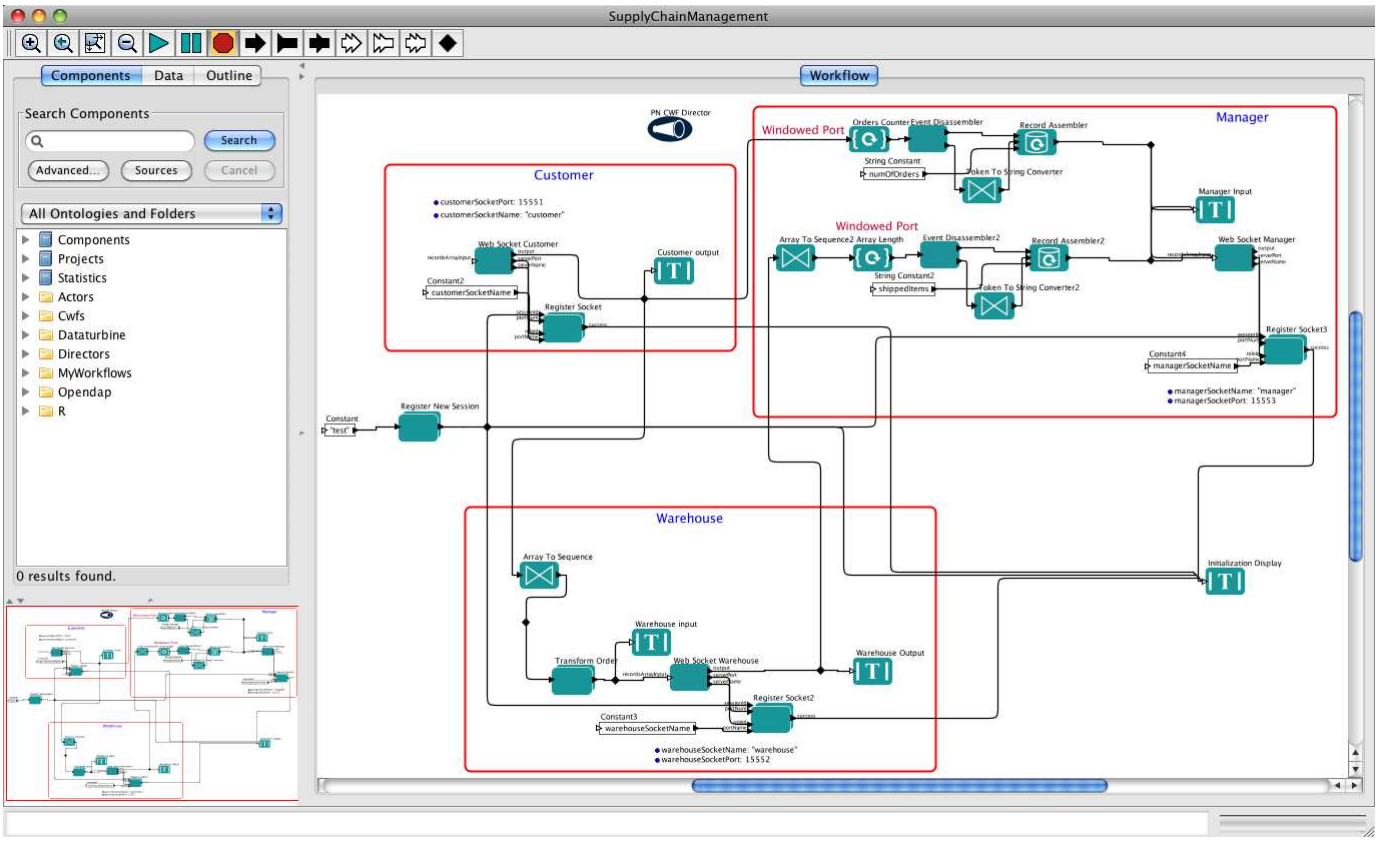


Fig. 7: Implementation a continuous workflow for Supply Chain Management reactive application.

service orders from its work list. As an initial stress test we spawned 20 customers and 3 warehouse manager processes to see the robustness of our system. The system was running without any problems, until we stopped it after three hours. As part of our future work we plan to further test the system by measuring more metrics (e.g., response time) and scalability in number of users that can be supported.

### *B. Astroshelf's collaboration backend*

In the context of the NSF project funding the research and development of CONFLuEnCE, we have been working with a group of astrophysicists to develop a complete platform, called *AstroShelf* which will enable them to collaboratively annotate sky objects and phenomena, as well as visualize parts of the sky using different algorithms. This includes a user interface (dashboard) with the ability to display sky images, an annotations management system and a monitoring module for real-time processing of annotations and sky update events. The monitoring module is realized within CONFLuEnCE. A high-level design of the system is shown in Figure 8.

Specifically, we have designed a continuous workflow to run on CONFLuEnCE as part of the monitoring module. The goal of this workflow is to monitor the activity of inserting, updating or deleting annotations as well as integrating the detection of transient events from various sky surveys that are of interest to the users, all in real-time. After processing these events the workflow will ask for feedback from the users. By interacting with the workflow, the users may refine the annotations, iterating over them until they reach a consensus. Manipulating annotations can be done using the SkyView, the Galaxy Classifier or the Supernovae Classifier.

The system interactions and flow of events are as follows (numbered as in Figure 8):

1) Using the Astroshelf's user interface the astronomers can define and name areas in the sky that are of interest to them. Additionally they may define the type of events they are interested in (e.g., new annotation, new supernova, galaxy classification, etc.) This expression of interest is pushed to CONFLuEnCE and it is registered into an R-Tree spacial index which resides inside the actor "Tag Interest". We used an R-Tree for its ability to index multi-dimensional information and quickly match spatial queries with the areas defined by the users.
2) Using the SkyView the users can annotate objects, group of objects or arbitrary points in the sky with any information they deem important to share. Using either the Galaxy or Supernovae classifiers the users can classify types of galaxies or supernovae, respectively. These classifications are recorded as annotations as well. All of these annotations are inserted into the Annotations Engine through a specialized API.
3) Every time a new annotation is inserted into the Annotations Engine, this event is detected by the Event Reporting

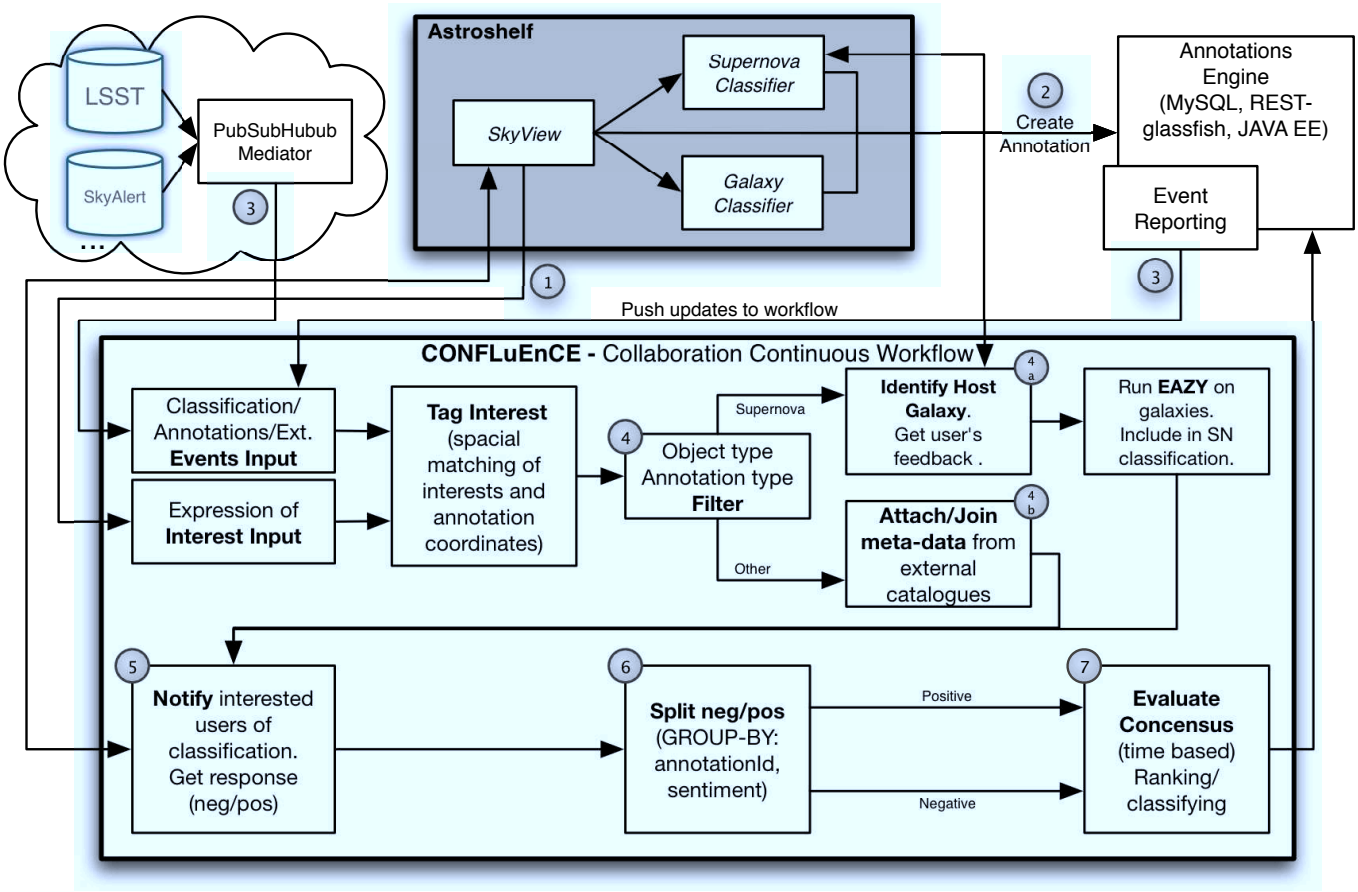


Fig. 8: High-level design of a continuous workflow for the Astroshelf collaboration platform.

module and directly forwards it to the continuous workflow on CONFLuEnCE. Other types of events pushed to the monitoring workflow are transient events detected by various sky surveys (e.g., LSST[6]), which are also available through an aggregation service, called SkyAlert.

4) Once the aforementioned events enter the system they are tagged by the "Tag Interest" actor with the user ids of those who previously expressed interest in the area and type of the object attached to the annotation (interaction 1). Then they are filtered depending on the event type, and follow different paths in the workflow.
   a. Supernovae events need to be handled differently than other events. Firstly, the supernova object is matched with its host galaxy and this matching is verified by the user through the browser interface. Then the object is run through the EAZY algorithm to calculate the redshift probability distribution.
   b. All other events are joined with data available from various external catalogs. This information will help the users when they provide feedback about an annotation.
5) Once all the necessary data have been attached to the data objects, then the users tagged on those objects are notified directly on their browser (as shown in the figure) or through email, SMS, twitter etc.
6) The notified users then use the Astroshelf interface to express their opinion on the annotated objects or classifications. The opinions are tagged as positive or negative and split accordingly. The "Split neg/pos" step groups the opinions according to the object id and the sentiment of the opinion. The window size of the group by is time based to measure the density of each opinion with respect to temporal bounds.
7) The final step of the workflow is to evaluate the overall consensus on the various opinions (positive or negative). It will then create another annotation on the object that captures the consensus. This new annotation goes back into the workflow and the cycle continues.

As it can be seen from the steps described above, the process of annotating sky objects is a loop which runs until the users collaboratively converge to a significant opinion.

## V. Related Work

A comprehensive study is presented in [8] which enumerates the various control patterns required by workflow applications. A pattern "is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts" [15]. The 20 patterns studied in [8] include more complex control structures, than the ones described by WfMC [16], such as

[6]http://www.lsst.org

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. A newer study [17] also evaluates how each pattern can be implemented in each of Kepler [9], Triana [18] and Taverna [12]. What we are mostly interested in are the patterns which are basic to enabling data stream monitoring applications. Specifically the *Push* communication patterns. Moreover Taverna and Triana only support one Model of Computation, and are also less effective in supporting as many patterns as Kepler.

Parallel to our work, another survey/position paper which deals with the basic characteristics and requirements of scientific workflow management systems makes the distinction between *stateless* and *stateful* actors [19]. Stateless actors are oblivious to their previous invocations in the same run, as opposed to stateful actors, which are required in models of computation that allow loops in the workflow definition, and/or support pipeline parallel execution, which are aware of the previous runs and may even involve data items from previous invocations (as we are doing by employing windowed queues on the actor inputs). We have also identified these requirements in our continuous workflow model paper [3].

Nova [11] is a system built by Yahoo! which supports stateful incremental processing. It deals with data in large batches using disk-based processing, and does batch incremental processing for Yahoo's data processing use-cases, which deal with continually arriving data. Even though it is similar to CONFLuEnCE, in the sense that it is continually processing data pushed from various Yahoo! data sources, on Pig/Hadoop workflows with a goal of low latency, it lacks support of window semantics, extensibility in scheduling policies and it is constrained by the limited number of workflow patterns supported by Pig/Hadoop. It is also short of a High Level Visual workflow programming language which makes systems like this more accessible to other domain experts (e.g., physicists, astronomers). A similar approach to cloud based data stream processing using Pig/Hadoop workflows is followed by HStreaming[7]. It has the same limitations in terms of workflow flexibility and high level visual workflow language, but this one supports a subset of the window semantics that CONFLuEnCE supports.

## VI. Conclusions

In this paper we presented the fundamental primitives of a continuous workflow model, which laid the basis for us to build CONFLuEnCE, our CONtinuous workFLow ExeCution Engine. Towards this we used Kepler, which is a complete workflow management system; in particular, we implemented our continuous workflow model as a new module in Kepler. Our module includes a director implementing a new model of computation with its own communications model (windowed receivers), a set of source actors designed to handle input from various types of data stream providers, and user interface modifications for defining window semantics. Finally, we described the implementation of two real-life applications using our model, one from the business domain and the other from the scientific domain. These applications enable real time collaboration between different users within the application, attesting to the ease of use and applicability of our system.

**Acknowledgments**: This research was supported in part by NSF grants IIS-0534531 and OIA-1028162. We thank the Astroshelf team: Liz Marai, Timothy Luciani, Rebecca Hachey, Roxana Gheorghiu and our astronomy collaborators: Arthur Kosowsky, Jeffrey Newman, Michael Wood-Vasley, Brian Cherinca, and Anja Weyant.

[7]http://hstreaming.com/

## References

[1] J. Delaney and R. Barga, "A 2020 vision for ocean science," in *The Fourth Paradigm: Data-Intensive Scientific Discovery*, T. Hey, S. Tansley, and K. M. Tolle, Eds. Microsoft Research, 2009, pp. 27–38.
[2] T. Hey, S. Tansley, and K. M. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.
[3] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis, "Towards continuous workflow enactment systems," in *CollaborateCom*, 2008, pp. 162–178.
[4] W3C, "Web services glossary - http://www.w3.org/tr/ws-gloss/."
[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: A new class of data management applications," in *VLDB*, 2002.
[6] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs, "Algorithms and metrics for processing multiple heterogeneous continuous queries," *ACM Trans. Database Syst.*, vol. 33, no. 1, 2008.
[7] W. van der Aalst, A. Barros, A. ter Hofstede, and B. Kiepuszewski, "Advanced workflow patterns," in *CoopIS*, 2000, pp. 18–29.
[8] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.
[9] B. Ludäscher *et al.*, "Scientific workflow management and the kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.
[10] W. A. Ruh, F. X. Maginnis, and W. J. Brown, "Enterprise application integration: A wiley tech brief," 2001.
[11] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang, "Nova: continuous pig/hadoop workflows," in *Proceedings of SIGMOD*, 2011, pp. 1081–1090.
[12] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.
[13] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.
[14] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis, "Confluence: Continuous workflow execution engine," in *Proceedings of SIGMOD*, 2011, pp. 1311–1314.
[15] D. Riehle and H. Züllighoven, "Understanding and using patterns in software development," *TAPOS*, vol. 2, no. 1, pp. 3–13, 1996.
[16] WfMC, "Workflow management coalition: Terminology & glossary (wfmc- tc-1011)," 1999.
[17] S. Migliorini, M. Gambini, M. L. Rosa, and A. ter Hofstede, "Pattern-based evaluation of scientific workflow management systems," February 2011.
[18] D. Churches, G. Gombás, A. Harrison, J. Maassen, C. Robinson, M. S. Shields, I. J. Taylor, and I. Wang, "Programming scientific and distributed workflow with triana services," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1021–1037, 2006.
[19] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers, "Scientific workflows: Business as usual?" in *Business Process Management*, 2009.