Contents

List of Figures

Ph.D. Thesis

Chathura Herath
Thesis advisor: Beth Plale

August 8, 2011

Abstract

Scientific workflows provide a widely accepted programming model that allows scientists to model their scientific experiments focusing on computational models, simulations and analysis, facilitating an integral part of the e-sciences paradigm. Scientific workflows are designed around static data, yet there are many scientific domains that process continuous event streams. These are not well captured by the current scientific workflow programming model. A hybrid programming model will enable data mining of high volumes of event streams, and facilitate setting up gateways for much needed features such as triggered computing, alert systems and real time analysis. One of the contributions of this thesis is a programming abstraction that preserves the simplicity and user friendliness of scientific workflows while allowing event streams to be first class citizens in scientific workflows. It is necessary to overcome the static nature of the conventional workflow inputs before event streams can be introduced. This can be accomplished by introducing streaming semantics that allow a pipelined execution model over the scientific workflow service components. The composition, execution and monitoring framework proposed in this thesis is called Streamflow.

Implementation of the stream semantics presented in this thesis requires consolidation of different technologies such as existing scientific workflow technologies and stream processing technologies. Because of the varying intensity of computer and data resource requirements, the Streamflow framework needs to use different workflow and resource management components based on the class of the computation. The framework identifies three classes of computation to which stream semantics are compiled, requiring three new management frameworks. The three classes are light-weight operator-based computations, medium-size computations, and long-running computations. The thesis argues that this classification is necessary to have a stable stream processing system. Thus it is necessary to place the different stream semantics in the right resources to be managed using right runtime framework. Further, this classification allows matching high rates of raw data streams with limited resources available for heavy computations. This classification facilitates better utilization of resources, allowing pruning and mining of streams, so their rates can be kept at manageable levels. Stream processing systems, such as complex event processing, provide languages and operators optimized for high throughput event streams. Long running compute intensive computations, on the other hand, require supercomputing resources. The BPEL workflow systems provides the quality of service requirements necessary to manage such supercomputing applications. Medium-sized computations take several minutes to compute. These computations are beyond the computation capability of stream operators, yet have quick turnaround

time. Such computations are compiled using XBaya workflow engine which is lightweight yet highly flexible, with low turnaround time. This thesis presents a programming abstraction that allows the different sections of the computation to be compiled and deployed to these different runtimes, depending on the characteristics of those sections; thus they can be managed with higher quality of services and better sustainability.

Defining streaming semantics and providing a runtime that is capable of executing such semantics may not necessarily mean the system could be used out of the box for any stream processing application. Resources are limited, and the rates at which each stream processing component would be able to operate, given the resources that are available, would determine the sustainable schedulability of a given event processing application. Another contribution of this thesis is dynamic analysis of a given Streamflow to find its schedulability given its resources and the event rates.

1 Introduction

The computational sciences have contributed considerably towards scientific research and development. As science moves towards the fourth paradigm identified in the book "Fourth Paradigm", this is unlikely to change[59]. The insight and perspective that scientific computing provides to the scientific community, although vast, is impeded by the complexities associated with mapping a specific domain science application into compute infrastructure, both hardware and software. The underlying compute infrastructures that facilitate scientific computing can be both expensive and complicated. The learning curve for domain scientist to use such systems is relatively steep.

Grid computing [41] , scientific gateways [27] [83] , workflow infrastructures [112][92][34] and others [25] promise to mitigate the challenges posed by the management of scientific computing infrastructures and to ease the burden on domain scientists. One popular programming abstraction is scientific workflows. Many geosciences and life sciences [93] have, over the years, adopted scientific workflows as a suitable abstraction for capturing scientific experiments [113]. Scientific workflows have contributed towards scholarly activities including publications. The importance of such workflow systems is further expanded in [68].

Because of the maturity of e-science, usage and sharing of research applications, data, and distributed computing resources among end user communities has accelerated. These collaborations are multi-disciplinary in nature and are building gateways to empower seamless access to high-end integrated infrastructure like TeraGrid and cloud resources. Figure 1 depicts a generic computer science architecture which is often referred to as a science gateway. Most science gateways use scientific workflow systems as an abstraction to access the underlying computer resources and to manage the scientific experiments below the workflow layer of Figure 1 .

Experience with geoscience workflows suggests that scientists are more likely to reuse a workflow than create a new one. This is partly influenced by the domain

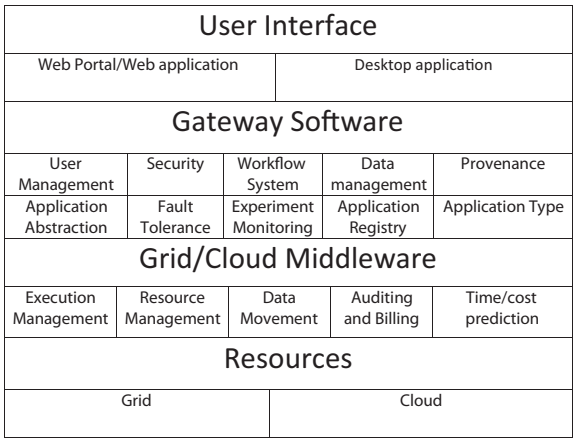

Figure 1: Layered diagram for science gateways consisting of the compute and data resources at the bottom layeR, grid and cloud middleware sitting as an abstraction on top of it. The scientific gatewaysaccess layer makes use of grid and cloud middleware to provide a domain specific compute infrastructure.

knowledge required to compose a semantically accurate workflow that solves a real research problem. In addition, the reusability of a scientific workflow is oftentimes shaped by the repetitious nature of the problem, and of the input datasets. Scientific workflow systems in many science gateways [120] such as LEAD (Linked Environment for Atmospheric Discovery) [32], ACES (Asia-Pacific Cooperation for Earthquake Simulation) [45] and Astroportal [106], deal primarily with the sensor data produced by scientific sensors; workflows are rerun with different data sets produced by different sensors.

Solutions to data streams such as sensor sources that interact with grid services [40] are sometimes loosely referred to as grid data streaming. The survey of Zang et al [126] classifies and identifies characteristics of such data streams. We identify the following data stream characteristics hich fit into the stream processing framework proposed.

- Tend to produce data regularly over an interval of time unlike static datasets or static finite file sets
- Transient data that normally passes through a system once
- Data event for different processes may arrive at different rates

- Large volumes of continuous data, potentially infinite
- May require different levels of processing requiring and have varying resource requirements
- Data events may be large files so stream processing system may process metadata or references to big data files in place of moving the big files
- Data can be structured or unstructured

It is important to identify the distinction between the data stream rate and the event stream rate in the context of a stream processing application. The event stream rate captures the rate of occurrence of discrete events at some point of time. An event could be of any size. For example an event could be few a GBs of weather forecast data, a few MBs of NEXRAD [12] radar file, or FIX protocol [105] events of a few KBs each in a stock market event setting. Event streams rate is expressed as events per second. Data stream rate, on the other hand, refers to the amount of data that flows through the system and is expressed as bits per second (bps). If the event stream rate is steady and the data size per event is the same for all events, the data event rate is the product of event size and event rate. For event streams that have non uniform interarrival rates and events that are non-uniform in size the data event rates are calculated based on time window averages.

There are many science gateway applications where the data streams and stream processing seem to find suitable applications. Some of the science gateways that deal with data from instruments includes LEAD (Linked Environment for Atmospheric Discovery) [32], which uses Doppler radar [76] data for detecting event storms, ACES (Asia-Pacific Cooperation for Earthquake Simulation), [45] that uses seismic sensors to detect movements in the tectonic plates, Astroportal [106] a project that takes in telescope imaging, and environmental monitoring using DataTurbine [42] to manage sensor data from coral reefs.

Given the increasing volume of instruments generating data in real time, well grounded integration of event streams with the workflow paradigm is important.

The operators or processors that evaluate streams continously monitor current and past events in the stream. Workflow activities, on the other hand, are mostly discrete request response based invocations. Workflow execution shows clear evidence of control flow dominance, though it has data dependencies. The control structure in workflows is, in most cases, precompiled and preserved during execution. The stream processors follow a clear dataflow execution approach where the data flows are triggers with the availability of the data. Further, the continuous operation of stream operators makes it hard to sort out a clear control structure [17].

The scientific computing experiments appreciate the determinism in the computational model whenever possible. For example consider a scientific computation that has four components A, B, C and D, as shown in Figure 2. The component D depends on outputs of component B and C. Components B and C depend on the output of component A; assume A has a single input. The computational model will be different when this simple graph is processed in a pure control-flow environment versus an event processing environment. In a control-flow environment the inputs for component A need to be available before the computation can start. Once the graph starts executing components B and C will produce exactly one output sets OB1 and OC1 which will become inputs to components D. This makes the computation deterministic after the input to component A is specified the inputs to component D does not depend on the compute time of any of the components not the time at which the input arrived at A.

In an event processing environment components will start executing immediately when their local data inputs are available. Also in an event processing environment as and when the multiple events arrive at these components they will continue to execute and produce output events. In such an environment there can be certain amount of non determinism for various reasons. Multiple events may arrive at component A which will trigger outputs of A to execute components B and C. If the components B and C have different computation duration, their outputs may be sparsely distributed in time. Because of the output events from

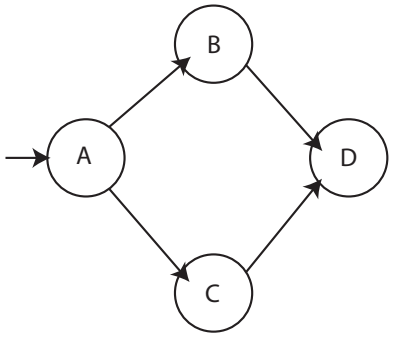

Figure 2: Simple task dependency graph to illustrate the control-flow and data-flow distinction. Control-flow has an execution instance per event that doesn't interact with executions of other instances. Data-flow may have events that execute activities as data becomes available.

components B and C may arrive at different times at component D, it is necessary to define when cmponent D should be invoked. Assume that component D is triggered at every distinct input set. If the component B produced two events, OB1 and OB2 when component C produced one event OC1, component D will be invoked with inputs (OB1, OC1) and (OB2, OC1). This might be different from the original expectation of this computational graph. This will mean that component D needs to be aware that the events produced by different input events may arrive at its input at different times and needs to provide for this non-determinism. The non-determinism can also be due to external input event stream inter-arrival time delays.

There are ways to mitigate this by using queues or correlations but the purpose of this example was to illustrate the contrast in the computational models of the two paradigms, control-flow and event processing systems sometimes referred to as data-flow. This example demonstrates that preservation of the control-flow is a useful feature because it removes the non-determinism, thus simplifying the processing components and making it simple to follow the computational trace to understand the application. Such deterministic control structures make it easier for scientist to setup their experiments.

Scientific workflows are a good abstraction for scientific experiments, but they are not designed to work with stream operators. This thesis explores the possi-

bilities of integrating high volumes of sensor data event streams into a scientific workflow abstraction extended with support for events to form a coherent programming model [102]. This approach was chosen because the workflow abstraction is well known. The breadth of the acceptance is arguably due to its ability to veil the complexity while giving perspective of the scientific experiment that allows the scientists who use these systems to focus on their science problem rather than the complexities of the compute infrastructure.

Scientific workflows are parallel by nature and support for them is often built as distributed systems. Models exist [53] to analyze the parallelism and the possible performance improvements that could be reached given the structure of a particular workflow. Sensor events are discrete by nature and may or may not trigger computations. When sensor events do trigger computations, the computational time may very well exceed the inter-arrival time of the sensor events. This leads to a new dimension of parallelism sometimes referred to as pipeline parallelism where the computations that are triggered by different events can be interleaved and run in parallel. The potential problems with allocating resources will be discussed in detail in chapter 5; but despite these issues, the possibilities of this new parallelism are worth exploring.

There have been different approaches to scientific sensor event stream processing. One possible approach is to build a decoupled stream processing system [102] where event streams are processed by an event processing system with trigger mechanisms used to launch experiments [98]. There is also a pure workflow approach [14] where the repetitive nature of the event streams is captured in pure workflow abstraction. These approaches and other related work will be compared in detail in the section to follow. The proposed approach lies in between these two alternatives by introducing workflow semantics that are equipped to handle data streams and stream inputs; stream components are considered to be first class components.

The consumption of scientific data in scientific workflow systems provides useful insights on potential challenges facing event streaming frameworks. The sensor

event streams have a relatively high event rate which normally drives the rate of the rest of the system. In a typical event driven scientific workflow system, most of the events that are generated by input sensors may not contain interesting data, but occasionally some of them carry events of significance that require further investigation. Such investigations may lead to further experiments that would provide further insight. Most experiments of this nature are scientific workflows. When considering the percentage of events that trigger such compute intensive workflows relative to the total amount of events generated, it is a significantly low percentage. Figure 3 depicts the change in event rate, information rate, and resource requirements. The data rates are high at the input end of an event processing system, where information content may actually be quite low. As more computation is done, information content increases requiring significant computing power to process such information. In order to sustain such a system, it is necessary to manage and reduce event rates using filtering and mining techniques, to a level that can still be serviced using the available resources when they trigger scientific workflows and other compute intensive tasks. Figure 4 shows data and compute needs of a typical compute intensive workflow. Data production may yield higher data rates due to the production of more data during the workflow, yet the event rate normally stays close the input event rate. There are certain workflow patterns [107] and applications that may yield higher event rates, but the premise of compute intensive tasks yielding similar event rates continue to be the case in many applications.

Scientific workflows can consume significant amounts of data. As the experiment evolves, they tend to produce less amounts of data yet contain greater insight. This can be viewed as a system that has high event rates at the beginning and the data rates are shed as the experiments progress. Workflow programming semantics[48] of conventional workflows are primarily focused on static, single run dataflow and control flow executions based on Directed Acyclic Graph (DAGs). Coupling programming semantics of static workflows with the event processing system tend to provide an effective way of reducing data rates. Introducing stream

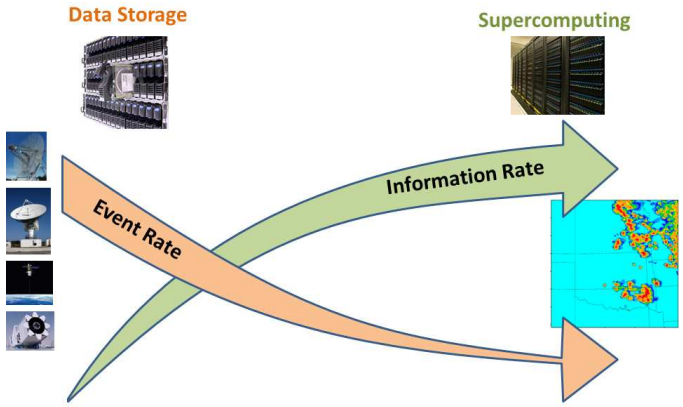

Figure 3: In a typical scientific event processing gateway event rate and information rate vary in an inverse relation. At the begining of the gateway the system has to process and mine lot of events and as important event are mined event rates will reduce and information rate would increase.

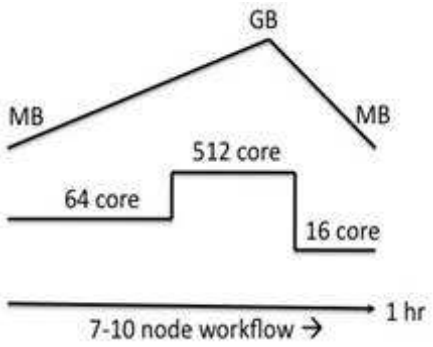

Figure 4: Data production and compute power requirement in a LEAD workflow.

aware workflow semantics will allow the scientists to compose scientific workflows that can directly interface with the data stream yet preserve the abstraction that is provided by the scientific workflow which would give an overview of the experiment so the scientist can still keep track of the big picture.

1.1 Contributions

The main contribution of this thesis is threefold; the first is a programming abstraction that allows for seamless integration of data event streams with the scientific workflow programming model. To achieve this, we propose new streaming scientific workflow semantics, and we present a framework to compile these workflow semantics to a scientific workflow and stream processing environment. This programming abstraction is referred to as Streamflow. We evaluate this new framework by measuring compilation and deployment overhead of Streamflows in comparison to conventional scientific workflows. The evaluation measures the completeness of the stream semantics in comparison to data streaming semantics.

Secondly we propose a formal model that attempts to analytically solve the schedulability of a given Streamflow and a set of input event rates. Given the difficulty of finding an analytical solution, we evaluate runtime health monitoring of a given Streamflow to monitor bottlenecks. In this analysis, we propose a workflow infrastructure and a stream processing system that caters to the processing needs of a given Streamflow and explain how design decision were made based on this analysis. We evaluate this by measuring the pipeline performance of data events in the Streamflow and latencies introduced by the framework.

Thirdly, we propose a Streaming Map Reduce framework based on Apache Hadoop that is capable of interacting with data streams instead of the static data. In this framework, we define slight variations to the programming interface and introduce a windowing technique and buffer while allowing garbage collection thus utilizing the resources optimally. We evaluate this by measuring the comparison of the stream implementation with the conventional Hadoop Map Reduce framework. Performance improvements may be made for specific applications that may

capitalize on windowing based caching techniques.

2 Related Work

Related work appears in four main areas of stream processing programming approaches. The first is workflow approaches and workflow execution environments. The second is database continuous queries and related approaches. The third area is the sensor networks and overlay networks designed for stream processing and the dataflow approach. The final area is on the related work in streaming approaches to Map reduce.

2.1 Workflow approach to data streaming

The conventional workflow management systems are good abstractions to capture control structures and control flows [123]. This is often cited as a strength of the workflow management systems because of the "what you see is what you get" nature of the computational trace that a given workflow may exhibit when it is scheduled for execution [82]. The data streams are repetitive in nature and require iterative triggering of activities for each element in the data stream. Most of the proposed approaches that have tried to address stream processing requirements within a workflow management system stream identify the lack of out-of-the-box control structures that can meet the repetitive processing requirements demanded by the data streams. The most recent proposal for defining structures in conventional workflows was presented in [14] and that thesis introduces flow control semantics that work within the workflow system and then compile into a workflow execution environment by defining buffers and queues to handle and manipulate streams in the runtime. The stream integration to workflow is either a push or pull mechanism [13] where the push mechanism resemblances the Event Condition Action (ECA) rule model and the pull mechanism shows striking similarities to the lazy evaluation of streams model that is discussed later. From the point of workflow execution, an idea of "continuous workflow" is introduced in [91] where workflow enactment semantics are laid out to further the understanding of how the workflow systems underlying execution engines may facilitate the data

streams. This work refers to window and queue based implementation mechanisms to manage the repetitive nature of the data streams. This windowing approach in workflow management system implementation allows the streams to be part of the control structure. Changing the pure workflow abstraction to incorporate the data streams somewhat complicates the programming model. The workflow programming languages like WS-BPEL [4] are not equipped to facilitate data structures such as queuing or windowing. The Dryad framework [61] along with DryadLINK [125] programming language allows pipelining of compute tasks in much more general way than the constraints imposed by Map Reduce. Dryad provides means to configure these compute tasks in a computational graph, allowing a dataflow pipeline.

2.2 Database approach to streaming

Stream processing and workflow programming form two different paradigms of computing. Most early approaches kept the two systems decoupled and developed each separately; in many cases a stream processing system triggers the workflow system. The stream processing framework developed by NCSA in [78] is a stream processing engine that can trigger a scientific workflow toolkit, CyberIntegrator, [11] thus keeping the stream processing separate from workflow systems. Therefore, no significant are required modifications to the control flow system to which conventional workflows adhere.

Early work by Pale [103] proposed a model and system for stream processing that utilized a declarative query language and event-action construct for query rules[101]. Calder [79] extended this to query processing over streams in a grid setting using a web service that abstracted a set of sensors into a single source[51]. The stream processing language of Calder was then extended to call out to arbitrary functions and this was applied in the LEAD project to continuously mine [100] an incoming radar stream for severe storm signatures [77][116]. The work in this thesis addresses the limitations inherent in the model used in [77] where continuous query [10] processing was poorly integrated with the graph oriented workflow

processing. Continuous query based stream processing system that allows a flow networks composition mechanism similar to that of the workflow management systems is explored to Aurora [22] [1]. The operators available for composition are primarily stream operators such as join, split, aggregate, etc. A similar system for data mining in data mining workflows, exploring Event-Condition-Action ECA rules instead of continuous queries is presented in [38].A grid middleware that is built upon the Globus toolkit named GATES [21] allows distributed streams to be implemented using pipelining and presented dynamic and static resource allocation using the Grid resource allocation. The work presented in this thesis distinguish with this work by presenting its programming model and iterative approach to event processing and ability to pick the Model of Computation depending on the runtime behavior of the event processing application.

2.3 Process networks and dataflow

Filter networks have been used for processing data streams, and [43] provides a message brokering approach to handle the stream processing by setting up broker network paths and routing the message streams through these paths. Stream Based Functions as a computational model proposed in [67], is a combination of the dataflow model [73] and the process network model [74]. Narada broker is an event brokering system that is utilized to manage real-time data management [95] [44] for grid services, and these grid services may be used in this context [43]; they can be orchestrated as a dataflow system using the event brokering system. This provides a possible stream processing mechanism that is built upon the event brokering system Dataflow systems based on the dataflow architecture are a popular programming model particularly in DSP systems because of the data flow nature of the signal processing discipline. This ability to have loose control flow over sequence of event makes it a candidate for data stream processing. The Kepler workflow system [81] provides a dataflow director which is an enactment mechanism, allowing workflows such as DAG to be integrated with data streams; the pipelined execution that is [3] triggered by the availability of the data in the stream

makes it a dataflow execution model. The Ptolemy framework [18] designed as a simulation framework supports the Dynamic Data Flow computational model, among others, that would enable the extension of the data flow principles used in Digital system processing and used for data streaming. The Triana [23] workflow system and proposed supporting data stream processing based on system named Styx [15] for interfacing data streams with grid services. European workflow management system ICENI workflow management system [83] also proposed a dataflow based computation model that may allow event streams [84]. IBM Spade and System S [47] provides a comprehensive stream processing framework that includes declarative query based event processing. The work presented in this thesis is distinguished from System S and kepler/Ptolemy by its ability to actively preserve controlflows that suits well with the scintific computing environments as well as presenting the ability to pick the Model of computation based on the runtime requirements.

2.4 Map Reduce Streaming

There have been applications of lightweight xml processing through streaming data into Map Reduce in [127]. There are other efforts such as Apache Hadoop PIG [94] which introduces notions of streaming into Map Reduce programming. Apache Hadoop introduced a script based programming model that can be used similar to Unix piping. The Twister iterative Map Reduce [33] programming model allows map tasks to be called iteratively with an access to a data cache that would fit very soundly for certain iterative scientific applications that require iteration before converging. The semantics of streaming in the Twister system follows the idea of the feedback loop in control circuits where the semantics of PIG streaming in Hadoop is more focused on piping a file into Map Reduce as it is.

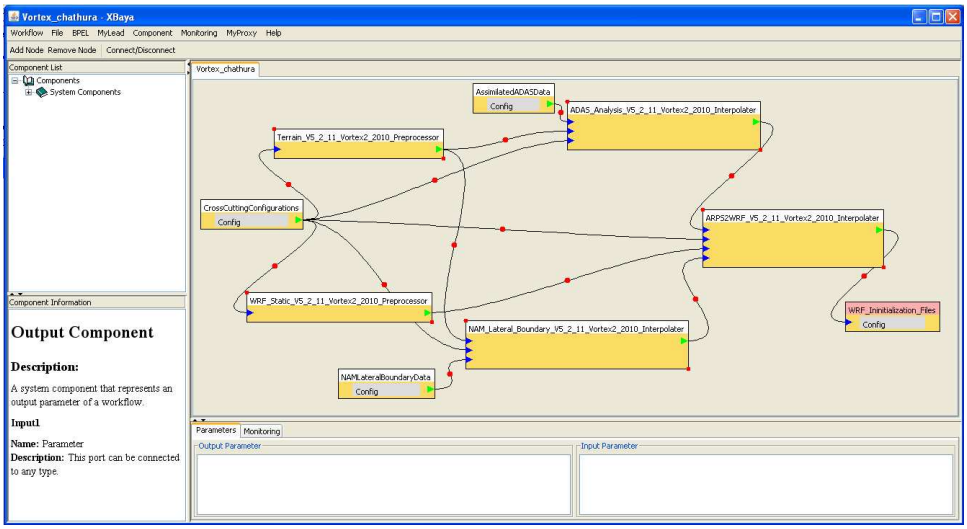

Figure 5: XBaya Workbench is a scientific workflow woorkbench that allows service discovery, workflow composition, workflow execution and monitoring environment.

3 Background

This chapter describes the author's contribution to the XBaya workbench shown in Figure 6 which drew upon the Indiana University Computer Science department thesis work of S. Shirasuna[110]. XBaya is relevant because the Streamflow model presented in this thesis is implemented using it. XBaya enables composition, execution and monitoring of scientific workflows. Three aspects built into XBaya serve as prior contributions of this thesis and will be described in this chapter. First,XBaya is a high level workflow description language, so it allows the abstract description to be compiled into different workflow runtimes like BPEL[65], PEGUSUS [28] and Taverna [114]. Second, is a lightweight and flexible workflow enactor/ workflow engine that was designed to handle short running jobs with high throughput execution. Third a widget framework that allows inference and implementation of on-the-fly service widget components that would allow flexible workflow composition.

3.1 High level Workflow abstraction

Most workflow systems like Taverna, Triana and Kepler provide a visual workflow composition tool (sometimes referred to as a workflow workbench) which enables workflow modeling while providing facilities for workflow enactment or execution. In most workflow systems, workflow enactment is easily distinguishable from workflow composition and the monitoring framework; for example the Kepler workflow system uses the Ptolemy framework and the, Pegasus workflow system [26] uses Condor DAGMan [46]. Workflow enactment systems normally support a workflow language that provides workflow execution semantics for a given workflow execution system, and these languages try to capture semantics that are common in the science domains in which the workflow system is designed to operate. There have been attempts to standardize web service based workflow systems by defining a standard, WS-BPEL, but scientific workflow languages in general have semantics that are more far-reaching than what is defined in WS-BPEL specification; for example, the Pegasus workflow system uses DAG, which is used for heavy parametric studies in seismology use the SCUFL workflow language, and Kepler actor-based workflow description use MoML [72].

Most workflow systems share common workflow semantics so they utilize a high level workflow description language that is independent of a given workflow runtime. At the time of execution, XBaya compiles its high level workflow description to the target runtime making it an interoperable workflow management system. This ability to compile workflows to different workflow execution engines is used to implement the Streamflow framework. The main advantage of a system with high level workflow description, which can be compiled into different target execution engines, is the ability to select the most suitable target workflow execution system, depending on the nature of the work and capitalize on its strengths.

The workflow composition interface of XBaya provides an easy to use drag-and-drop graphical user interface that allows scientific workflow users to compose workflow using activities such as static inputs, services, conditional branch activities, and so on. Once the workflow is composed the user can execute the workflow

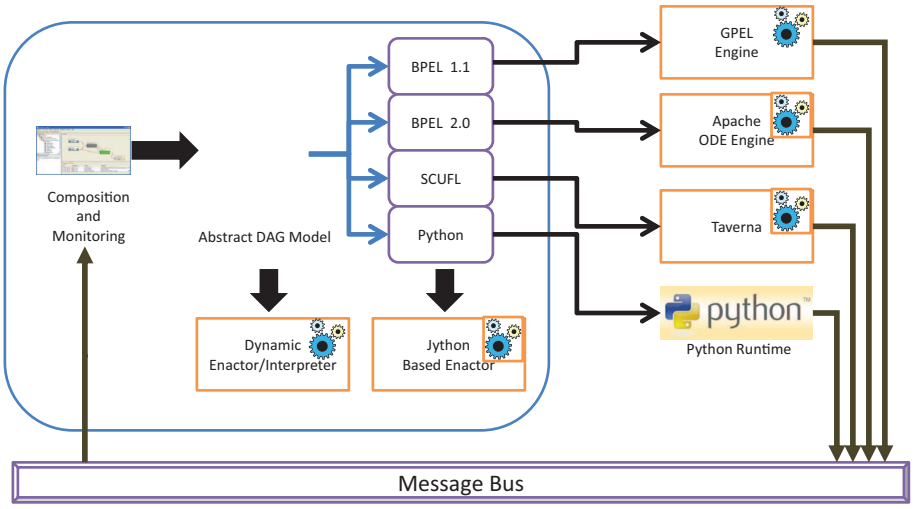

Figure 6: XBaya High level workflow description DAG model with the ability to get compiled into different workflow execution environments

and for that execution XBaya provides a few options. XBaya provides a Jython based enactment engine which compiles the workflow into a Jython script and executes it. Furthermore XBaya provides the option of deploying the workflow into a WS-BPEL based workflow engine such as Apache ODE. The emphasis in this thesis is a dynamic enactment engine which is designed specifically to cater for dynamic interactive workflows. Graphical representation of the workflow in XBaya and the actual enactment of the workflow are decoupled so the workflow monitoring is achieved through asynchronous messaging using a WS-Eventing [16] based notification broker [60]. As the workflow enactment engine (whether WS-BPEL or something else) executes the workflow, it publishes notification to an event broker; the XBaya monitoring system subscribes to this event broker and thus gets notified about the progress of the workflow.

3.2 XBaya workflow engine

Out of the different workflow script compilers and workflow execution runtimes shown in Figure 6, two are used extensively in this thesis, the BPEL Apache ODE

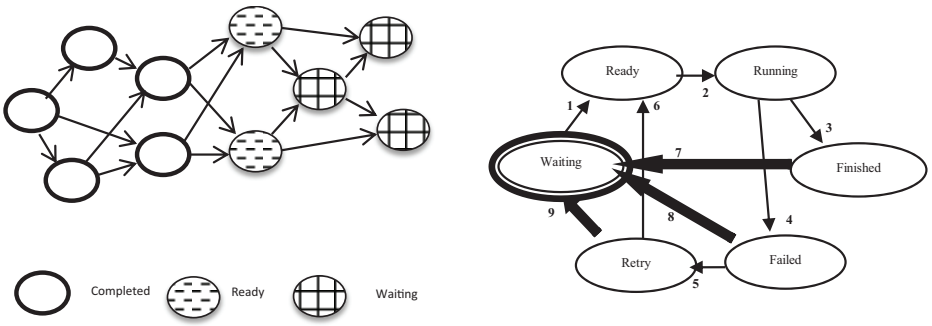

Figure 7: XBaya workflow running instance states where different nodes are in different execution states

runtime [54] and the Xwf XBaya Engine that is used for dynamic workflow execution, debugging and steering. Xwf is the workflow scripting format that XBaya uses for its runtime based on directed acyclic graph scripting. The distinction between the XBaya Engine and the ramainder of the workflow runtime of XBaya is that the XBaya Engine executes the workflow against a Xwf DAG and during the execution of a particular instance of a workflow, execution can be paused and the DAG may be dynamically changed. Those changes will be incorporated once execution is resumed.

The individual nodes representing workflow activities have one to one correspondence to the graphically composable components available in the workflow composer. A particular workflow can be identified as a collection of such activities that have control and/or data dependencies to one another. When considering the point of view of the XBaya workflow engine an individual activity can be in six different states, as represented by the state machine in Figure 7.

A given workflow activity will have certain data and/or control dependencies and the activity will remain in a "waiting" state until its dependencies are satisfied, both data and control dependencies. Then the activity moves into a Ready(1) state and waits for the interpreter to schedule the activity (2) with successful completion of the activity invoking a Finished state (3). Such finished activities may satisfy the control or data dependencies of other activities in the workflow, thereby allowing the workflow to progress. When the XBaya workflow engine is execut-

ing a workflow activity there is a possibility that the activity may not complete execution successfully. Subsets of such failures are transient and retrying the execution of activity may lead to success. Retrying is only effective when an activity is idempotent, because the retrying process is unable to recognize application side effects. Such activities move into the Retry state(5) which implements the retry policies such as the time delay before retry and the number of times it should be retried and it would become ready to be executed(6). In the state diagram the thick arrows (7, 8, and 9) represent the workflow being interrupted by the user and the user performing user interactions. Before the user can intervene with the workflow, workflow execution must be paused. For example, a user interaction like a rerun or a smart rerun could trigger the 7, 8, 9 paths. It is important to note that already running activities will not be interrupted by the workflow pause and to avoid race conditions the workflow would pause once the currently executing activities have finished or failed. The individual Finished activities in the workflow model is required to save the output port values so that in an event of a smart rerun those values will be used as already computed data dependency values. This can be viewed as check pointing the results of the workflow activity after its successful completion. Following is a listing of different functionalities designed by the author in XBaya workflow Engine.

- Derivations during workflow Execution
  - Fault handling.
  - Dynamic change workflow inputs, workflow rerun.
  - Dynamic change in point of execution, workflow smart rerun.
- On the fly workflow composition
  - Dynamic addition of activities to the workflow.
  - Dynamic remove or replace of activity to the workflow.
- Reactive workflow evolution

- Reactive rescheduling of resource adaptively
- Reactive addition of activities that is a possible next step in workflow evolution

3.3 Workflow service creator for dynamic composition

In addition to an interoperable abstraction and a high throughput enactment engine, XBaya also provides a just-in-time service definition feature which includes the capability for the workflow user to create a web service on-the-fly. During the workflow composition or when changing the workflow, there are situations where the output types of one activity slightly mismatch the input types of another service. The user has prior semantic knowledge of the compatibility of the inputs and outputs of the two services. XBaya enforces type safety based on WSDL schema based type checking so it prevents the workflow from composing tasks having incompatible types or even what is referred to as duck types, yet have different schema definitions. In such cases XBaya would allow the user to create a new activity on the fly while the workflow is active, thus allowing scientist to make the maximum use of the workflow run.

The usability of this option goes well beyond the type incompatibility problem, even though that was the motivation that led to this feature. When the user creates this new activity the user will be presented with a java skeleton capturing the type incompatibility. For example if Service A has an output type of type T1, Service B has an output type of T2 and Service Chas an input type of T3,then the assumption is that the user knows an algorithm to make an object of type T3 using the objects of type T1 and T2. In such a case the java skeleton presented to the users is shown in Figure 8.

An interesting aspect is that if the types T1 and T2 are complex types, then XBaya can generate java types using xml-beans [7] and allow the user to save the jar file containing the type classes. Once the user fills in the implementation of the above class it can be made a java widget and can be included in the execution as a temporary workflow activity, or if the user wants this activity to be preserved for

```
1  public class ServiceName{
2
3    public T3 operationName(T1 t1, T2 t2){
4      //Fill in the implementation
5      return null;
6
7    }
8
9  }
```

Figure 8: Template generated for new activity. User will implement the method signature and will be able to deploy this java based activity as a widget component to be included during execution.

later use then XBaya will allow the user to export the activity as a web service. The web service that is generated is an Apache Axis2 [96] service, and for this feature to be available the user should provide an endpoint reference of an Axis2 service with a special service called ServiceCreator already deployed in it. This approach is taken to make sure that this would not lead to any security vulnerabilities.

4 Programming abstraction for data event streaming in scientific workflows

The research proposed here builds upon earlier work with XBaya and workflow systems and proposes a model for extending a workflow system with support for data streams. This solution is not only a natural evolution of scientific workflow, but it also allows the use of existing programming models of scientific workflow composition without major changes and alienations of users. A typical workflow system provides a means to compose, orchestrate and monitor workflows, and in this framework we use a WS-BPEL [4] based generic scientific workflow system developed for the LEAD workflow system [25]. We described XBaya in section 3 [110]. The WS-BPEL based workflow execution engine is Apache ODE [5]. The interaction between these components and how they fit in a larger workflow system is described in [97].

As described in section 3, a workflow composition tool allows a scientist to compose a workflow from existing software and service components without having to be familiar with workflow languages. Further, workflow composition tools allow for optimizations that can occur when compiling a workflow graph into an executable workflow language [29]. The proposed programming model is based on the programming abstractions XBaya offers. XBaya workflow composer provides a high level workflow description language called Abstract DAG model that is independent of conventional workflow execution languages. This independent representation decouples workflow composition and workflow execution and allows the internal representation to be transformed to multiple execution languages. XBaya currently supports BPEL 1.1, BPEL 2.0, SCUFL [49], and Python scripts. The different workflow enactment environments have their merits and demerits, and depending on the domain science the optimal workflow enactment environment should be chosen to capitalize on the merits. For the purpose of this paper we focus only on the BPEL execution environment. The work presented in this paper builds upon the functionalities and the programming model presented in scientific

workflow systems and provides a programming abstraction for processing data streams.

4.1 Requirements

Our goal is to extend the existing support for workflow execution such as that accomplished through XBaya in a way that satisfies the following requirements:

- Preserve the workflow programming model. Users are familiar with DAG execution. Stream processing is an event driven paradigm that when put side by side in an interface to the user during workflow composition, it tends to confuse the scientists [56][115]. In this research we strive to integrate continuous events processing into the familiar DAG model while preserving the familiar DAG control flow. Other research directions have proposed a pure workflow approach which has made the programming model somewhat complicated or completely disjointed the workflow and stream processing [14].

- Make the changes transparent to the workflow execution engine so the approach works with an out-of-the-box engine. Not making changes to the workflow execution model means using existing functionalities provided by workflows as they are and using an external system that reduces the burden on the workflow system in terms of available functionality. This external system we call a Complex Event Processing system.

- Keep the simple, simple. If users do not need support for streams, the system should act and feel the same as it always has.

- Define workflow patterns for use as new workflow semantics that provide a computational model where Complex Event Processing is a first class entity in the workflow.

The initial focus is to identify the best approach to integrate stream processing into workflows without disturbing the existing workflow paradigm. This leads to

the formalization of several patterns that are discussed in the formal model in section 5.

The integration of the Streams into the workflow system is addressed as a programming model in workflow composition. The graphical workflow composition has been very successful as a programming model and has had enormous success among the community as the means for representing scientific experiments. In extending this programming model to encompass stream processing, special consideration is given to not alter the existing workflow programming model that is comfortable for most scientists.

4.2 Programming model - Streamflow

A top down approach is taken in defining the programming model. The main focus is to identify the best approach to integrate stream processing into workflows without disturbing the existing workflow paradigm. This leads to the formalization of several patterns.

When dealing with live observational data feeds during workflow composition, a researcher continues to use a workflow composer under our new model. If the experiment involves a data stream of a particular data type, the scientist would focus on setting up the workflow as if this workflow is going to process a single event of the data stream. XBaya builds a mechanism to remove the static workflow inputs and connect a data stream of the same type to that workflow node. At the time of execution of the workflow it is necessary to understand that this workflow has different execution semantics because of the stream nature of the inputs; the XBaya workflow system manages the execution in a way that is complexity transparent to the user.

This new node type is configured through setting a query, an EndPoint Reference (EPR) and a time range. The query represents a subsequence of events within a temporally ordered data stream that are of interest to the researcher. Since a workflow executes once from beginning to end in a linear fashion, yet the data stream is continuous, a time range defines the period of time for which the data

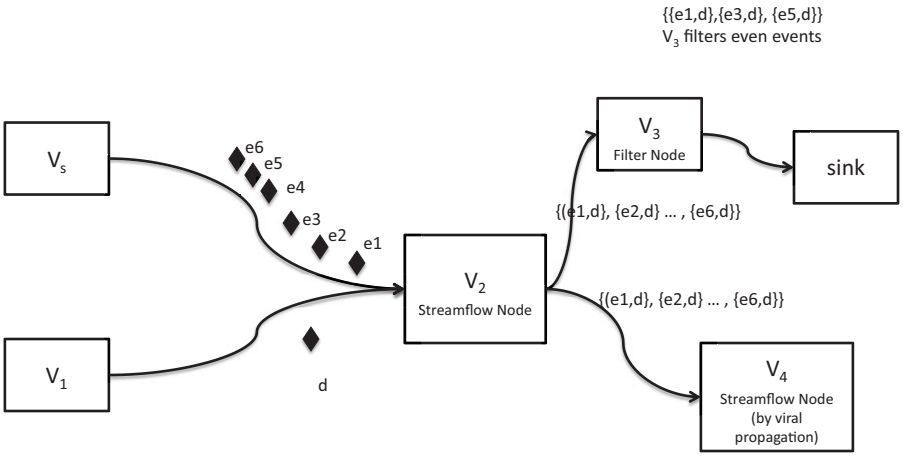

Figure 9: Example Streamflow that shows how streams of different cardinalities can be composed together

stream should be monitored. Finally the EPR indicates to the event processing service where to send data events matching the given query.

The objective of defining a programming model is to define programming constructs that enable users to compose workflows that have streaming in them. In doing so we first define the Streaming entities that are introduced in this thesis. The different streaming workflow components available for Streamflow programming model are listed in the following listing and the Figure 9 shows a composition that involves some of these components.

- Streamflow node - This is the most common node in Streamflow graph and represents a typical dataflow node that supports pipelining of event streams.
- Filter node - Filter node represents a filtering mechanism based on a predicate provided. It acts on a input event stream and based on the predicate of on individual event, it filters out certain events.
- Event generator - Event generator represents an external event source or a service that produces a stream of events.

- Stream sink node - Stream sink node allows streams to be published back to the CEP Engine
- Aggregator Node - This is a stream node that allows events in a stream to be integrated or bundled together to generate another stream.
- Merge Node - Merge node merges events into a single event stream.
- Join Node- This is a stream node that allows two event streams to be joined.
- CEP node - This is a generic node that allow Complex Event Processing query to be run against an event stream.

To enable the user to incorporate real time, data streams into workflows, XBaya introduces a general workflow abstraction type we call streaming node type. The significance of this type that is all the nodes that belong to this general category show one of the Streamflow patterns listed below and shown in Figure 41. When these Streamflow patterns are used during a workflow composition, we refer to the resulting control-flow and data-flow structure as a Streamflow.

4.2.1 Streamflow node

The Streamflow nodes are for all practical purposes web service nodes that might be found in a conventional scientific workflow. A node becomes a Streamflow node because of the context in which it appears in the Streamflow. A web service node that appears downstream from a stream generator node, Streamflow node, and other streaming nodes is a Streamflow node. The Streamflow node V2 in Figure 9, takes a data stream of e1, e2, and e6 as input and produces a data stream as output, subject to the following rules:

- The Streamflow node executes a web service component over each event in the data stream.
- Simplifying assumption: a Streamflow node has only 1 stream input and may have one or more static inputs, this does not affect the model's gener-

ality because the stream join node could facilitate any other combination of different input streams.

4.2.2 Filter Node

The filter node V3 has the effect of reducing the stream rate by discarding events. A stream of events with 1 second inter-arrival rate and a filter that filters every other event would output a stream with a 2 second inter-arrival rate to a downstream node. Filter Nodes do not exhibit a 1-to-1 mapping between input events and output events; that is they exhibit properties of a mathematical partial function, being single valued, but not necessarily total.

In workflow context, the partial function behavior is a side effect of publishing all the events in the event stream to a CEP system and running a CEP query against them. Some events in the event stream would satisfy the CEP query and others would not, thus giving the behavior of a filter or a partial function.

The filtering logic for filter node needs to be specified using a CEP language, EPL (Event Processing Language). An example of a filtering configuration used in context of a Filter node could look like the following declarative query. It maintains an event window of five events of type RadarDataStream and this event type has a property called reflectivity. Events are selected that have reflectivity higher than 3.2. The query produces a combined data event containing five RadarDataStream events.

Select * From RadarDataStream(reflectivity > 3.2).win:length_batch(5)

Most complex event processing languages provide a means for selection, projection, joins, event windows and event window operators, grouping, reordering, pattern matching and other useful declarative semantics.

4.2.3 Stream sink

Stream sink may be viewed as a sink for every stream in Streamflows where the event stream would be published to the CEP Engine. This would exhibit functionality similar to that of an output node in a workflow. Output nodes in Streamflows

are not a single output but rather a time series of events thus it is difficult to quantify an output as such. Thus for Streamflows the outputs are normally published back to the CEP Engine; interested parties can subscribe to this particular event stream.

4.2.4 Stream generator

Stream generator is a special service call that initiates a time series of data as its output, so output of the Stream generator always needs to be connected to a Streamflow type node, i.e. an active node or a filter node and need to have a viral effect on the downstream. The sensors that produce data streams would be abstracted using Stream generators.

4.2.5 Aggregator Node

The Aggregator node makes uses of a sliding window to aggregate the events of a given stream to one event within a finite history. It could behave as bundling the last n events together or bundling the events occurred in the last t time together. If a sliding window length is l and the input event stream is E1, E2, . En the output event stream will be event of the form E1,E2, El, E2,E3, El+1,

4.2.6 Merge Node

The merge node merges event in different streams into a single event stream. This does not require nor does join events, rather inserts all the incoming events into a single event stream in the order the different events arrived at the node. If there a n event streams where first stream is E11, E12, E13 , second stream is E21, E22, E23, and so on the output events will be some order of these events one after another with time ordering based on arrival time of the events at the node.

4.2.7 Join Node

The Join node joins events from different streams to a single stream by selecting one or more events from each of the stream and building a composite event. The

output stream can be clocked to one of the event streams, i.e. the output events will be produced when one of the streams receive an event. If there are n streams getting joined there need to be caching of events of at least n-1 streams to some defined history window, so when the join is triggered the values for the composite event can be found in the history.

4.2.8 CEP Node

The CEP node is a generic Complex event processing node that has a EPL query associated with it. EPL is a declarative Stream processing query language and the behavior of the node will entirely be governed by this query.

The Streamflow framework deals with other intricacies like promoting Edge Weight of a Workflow-Edge to match the edge weight of a Stream-Edge. For example, the V2 active node in Figure 41 has two inputs: one is a Stream-Edge and the other is a static workflow input. During the execution, the Streamflow framework ensures the input set to V2 takes the form of e1,d, e2,d ...e6,d, thus the Edge Weight of the static input may appear to have been replaced by a data stream of repeating elements that would match the data stream of the Stream-Edge. So if the data stream is modeled as time series I1, I2, In , it would produce a sequence of outputs O1, O2, On.

Although we define the different streaming nodes in this section, the rules that govern the composition of these components into a Streamflow need to be defined separately. In programming language theory the programming syntax defines the rules that need to be adhered to when writing a program. Similarly, there are grammar rules that define how these components can be used with each other and with conventional workflow components. The formal listing of these grammar rules is presented in Figure 40. For the completion we would articulate the rules qualitatively in this section. We would use the term pure workflow to identify the workflow nodes that are not intended for the stream and are used with static input data in conventional scientific workflows.

- Pure workflows components with static data can be part of any Streamflow

and they would run as a pure workflow if all its upstream inputs are static.

- Pure workflow components will be promoted to Streamflow nodes depending on the input data context the pure workflow components may appear. For example, if the workflow user composes a pure workflow and to one input he connects a stream generator as the input to the workflow, all the downstream nodes from that stream generator will be promoted to streaming nodes.

- Implicit promotion of a pure workflow web service node because of stream promotion action by the user will always be treated as a promotion to a Streamflow node.

- Streamflow nodes, Filter nodes, Stream aggregator nodes and CEP nodes can be arbitrarily connected to each other while preserving the Directed Acyclic Graph rules and type matching rules.

- Stream sink acts as termination point by simply publishing the inputs it receives to a CEP Engine.

- Stream generators represent sensors or other type of event streams upon which the Streamflow will act.

4.3 Architecture

The architectural framework of the Streamflow framework facilitates programmability of the data streams with scientific workflows. The architecture of the framework can be decomposed into three architectural layers as shown in Figure 10. The framework proposed consists of the workflow composition tool which makes use of composition and configuration modules in layer 1, the execution and orchestration components in layer 2, and the pluggable service components and stream operators in layer 3. An analogy can be drawn from programming languages where there is the language itself and then the language runtime. The first layer of the framework provides Streamflow composition related modules that allows, the scientific workflow users to build their experiments and this could be seen as

analogous to writing and compiling a program. Once the Streamflow graph is compiled and ready to run it will be schedule to different runtimes depending on the runtime characteristics of the sub-graphs and this is captured by the second layer . Layer 2 is responsible for orchestrating the execution of components and managing the event streams, and is analogous to the programming language runtime. The orchestration engines in the middle layer interact with service components and stream components of the bottom layer, which are pluggable application wrappers for scientific applications.

Composition and configuration layer provides a Graphical User Interface based programming abstraction very similar to the Scientific Workflow programming abstraction. This can be used by the domain science user to formulate their experiments. The Streamflow composition module makes use of other modules like service discovery, stream data binding and hot deployment to provide a richer workflow composition experience for the scientific workflow user. In addition that the responsibility to compile and deploy the composed Streamflows is done through the framework components at this layer. The compilation of a given Streamflow involved using partitioning graphs, such that the sub-graphs would have coherent runtime characteristics based on throughput/ event rates and resource consumption. It is also necessary to identify the most suitable runtime engine that a given sub-graph should be deployed to and to manage the deployment order of the sub-graphs such that there will not be inconsistent sub-graphs in an event of partial compilation failure. Also once a given Streamflow is deployed and running it is necessary for the scientific user to monitor the health of the system; this is also handled by layer 1 using a publish/subscribe system [37], particularly using Web Service Notification [117] based asynchronous messaging system.

The static nature of workflow inputs makes it challenging to integrate the data streams directly into the workflows, thus the Streamflow framework in layer 2 uses complex event processing systems in conjunction with the scientific workflow system to make this integration more flexible. The Streamflow [57] [99] framework uses Esper [35] complex event processing system to manage high volumes of data

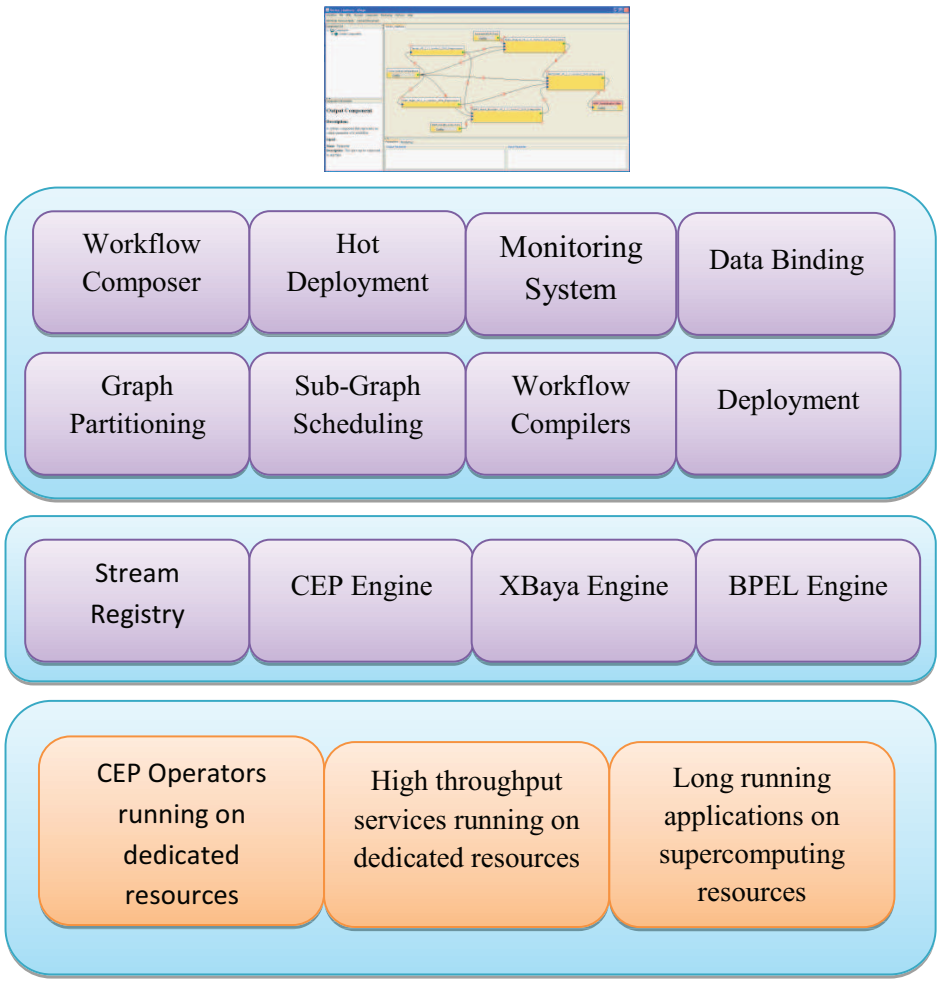

Figure 10: Streamflow Architecture with architectural components showing different layers. Layer 1 resource layer. Layer 2 server components layer. Layer 3 runtime tools and composition layer.

events coming in from different event sources. The workflow sub system is purely SOA based; thus the complex event processing system is wrapped in web services with extra functionality to manage the different streams and this service is referred to as the CEP Engine. The CEP Engine is built on the reasoning that data streams have increased value when centralized decision logic exists, and its not just limited to localized decision logic [56]. It is a global repository for all the data streams system-wide, and all the data streams are channeled through the CEP Engine which makes it a single stream repository against which all the stream queries could be run against. On the other hand, the CEP Engine sits in between workflows and data streams and thus allows the flexibility of letting data streams be pre-processed and filtered before they reach the workflows. For example a weather forecasting workflow may be interested in radar data streams in a certain region to do a local weather forecast. But the CEP Engine may receive radar events that are specially distributed all across the country. So it is necessary to filter out the radar events that fall out of the interested region: because the CEP Engine sits between the data streams and the workflows, allow unnecessary events can be filtered out. Thus workflows will not be run on unnecessary data sets and this will save compute cycles.

XBaya engine and the BPEL workflow engine both provide similar orchestration functionality for the frameworks runtime yet they differ based on the runtime event throughputs that they are able to handle and the reliability of the service that they offer. BPEL [4] based workflow engines, particularly the Apache ODE workflow engine [5] used in this framework, supports features like persistent business processes, fault tolerant and transaction based business process execution which has supports for check pointing, making it a good candidate for long running compute intensive scientific workflows. But the event rates supported by Apache ODE is relatively low as shown in Figure 51 later. The XBaya engine is a scientific workflow enactment engine that is used in multiple science gateway projects and is capable of handling higher event rates in the streaming mode. The tradeoff is it does not support the check pointing and failure recovery features like

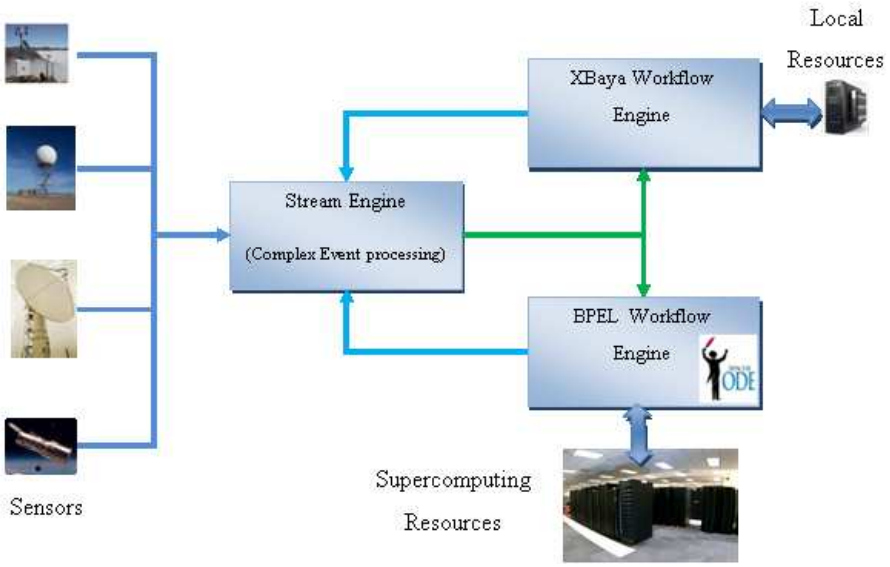

Figure 11: Streamflow event interaction showing stream interaction from event sources, CEP Engine and the workflow engines.

that of Apache ODE.

Figure 11 shows an interaction between the runtime orchestration engines and CEP engine. Once the user composes the Streamflow using the available semantics, depending on the different semantics that were used, the system would compile the necessary workflow scripts and queries to actually produce an executable Streamflow.

Besides being used for filtering, the CEP Engine provides a means to facilitate data binding of streams to workflows. Assume that there exists a data stream S and there exists a type T to which every event in the data stream conforms. If this data stream needs to be processed using a workflow, it is necessary that the workflow be able to process inputs of type T. THis problem needs to be resolved and typed checked during the process of deployment, as explained in the following section.

4.3.1 Workflow Composer

Similar to most scientific workflow systems, the Streamflow system provides a composition interface for the science domain user to define and interact with the workflows. The composition allows users to use workflow activities, Stream operators and event sources to compose scientific experiments by defining data dependencies among the workflow activities and stream operators. This essentially builds a workflow Directed Acyclic Graph (DAG) similar to that of a workflow but having flexibility to process event streams if necessary.

The workflow Composition interface provides the necessary functionality to make the job of the domain scientist much easier. In any workflow system it is essential for the composition tools to provide means for service discovery. This requires the system to find service registries and process service descriptions and understand those descriptions so that those services can be used in the workflow during composition. The XBaya composer provides service discovery interfaces by allowing the users to add service registries as well as a event stream registry so the user can load available services as well as currently active event streams.

From the perspective of the workflow composition, when dealing with live observational data feeds, a researcher will continue to use a workflow composer as the tool for experimental setup. If the experiment involves a data stream of particular data type, in the simplest case the scientist would focus on setting up the workflow as if this workflow is going to process a single event of the data stream. XBaya builds a mechanism to replaces an static workflow inputs to one of its nodes and connect a data stream of the same type to be connected to that node. At the time of the execution of the workflow it is necessary to understand that this workflow has different execution semantics because of the stream nature of the inputs and XBaya workflow system manages all that execution complexity in a way that is transparent to the user.

To accomplish this we introduce new workflow semantics to the workflow system in the form of a streaming data source. From the perspective of the experimenter the stream data source is an abstract concept representing a continuous

data stream. From an architectural standpoint it represents a service that ingests events posted by the data stream service to a workflow that is waiting for an external event to continue execution.

This new node type is configured through setting a query, an EPR and a time range. The query represents the events within the data stream that are of interest to the researcher. Since a workflow executes once from beginning to end in a linear fashion, yet the data stream is continuous, the time range defines the period of time for which the data stream should be monitored. Finally the EPR indicates to the event processing service where to send data events matching the given query.

Many scientific sensors produce large continuously generated observations and, in some cases, much of the data is not very interesting. Weather radar, for example, produces events occurring regardless of weather conditions. Occasionally an event occurs that needs urgent attention and evaluation: for example, a tornado may starts forming in the atmosphere. Thisrequires urgent forecasts to determine the strength and path of the weather system. In such a situation, it is necessary to trigger much larger experiments to evaluate the event. The pattern matching and other declarative programming constructs supported by complex event processing systems provide a sufficient programming model to capture such scenarios. A rule based system is another possible way of detecting such trigger scenarios.

In the simplest case, the programming model allows the user to compose a workflow as a normal static input workflow and then add a streaming data source to the workflow. This is done by identifying the particular input that will be fed from an event stream and configuring it to be interfaced with an event stream, via the convienient drag and drop composition functionality provided by the composition tool. Once such a configuration is done it is represented during compilation, and deploying the Streamflow framework would initiate the execution of a graph instance for every incoming event in the event stream. This simple programming model extension makes it easy for the scientist to adopt because it is very similar to composing conventional scientific workflow. This section will first focus on how such a simple use case is handled by the framework and later it will move on to

explain how generic graph structures with unforeseen complexity can be compiled and scheduled using the Streamflow framework.

For simplicity, we will consider an example where a user will use an existing pure workflow which runs on static data and tries to interface it with an event stream. This is very likely scenario whenever the user has setup a workflow to evaluate a single sensor event, and then decides to plug the stream into the workflow and run it for every event that the sensor generates. Figure 12 shows how this interaction will take place. Figure 12(a) shows the existing pure workflow and in Figure 12 action (1) the user would simply drag and drop a stream source to the workbench and drag the workflow input to the stream source: what could be read as the workflow input now comes from the newly added stream source. This stream source is now a stream generator node type and thus all the down stream nodes would become implicit Streamflow nodes. During the compilation of this Streamflow, the framework compiles the workflow, and generates two workflows as follows:

- Control Streamflow that initiates the execution and receipts of messages from a CEP system and dispatch to the child workflow as needed.
- Child workflow representing the actual scientific workflow without any streaming in it. This is the same as the static data set workflow.

Once these two workflows are deployed the system exists in a runnable state. The dataflow of a conventional workflow is normally represented by an edge from one workflow node to another and represents a single message or event flowing from data out-port to a data in-port. It is important to note that when a streaming data source is introduced to a workflow the cardinality of the dataflow edge changes and edges connected to stream data sources and all subsequent data dependencies represent data streams instead of a single data event. These changes are reflected in the workflow system using the thick blue edges in the control Streamflow 3 and can be clearly seen in Figure 12. These workflows are then compiled to WS-BPEL scripts and be deployed to the workflow engine used by the framework, Apache

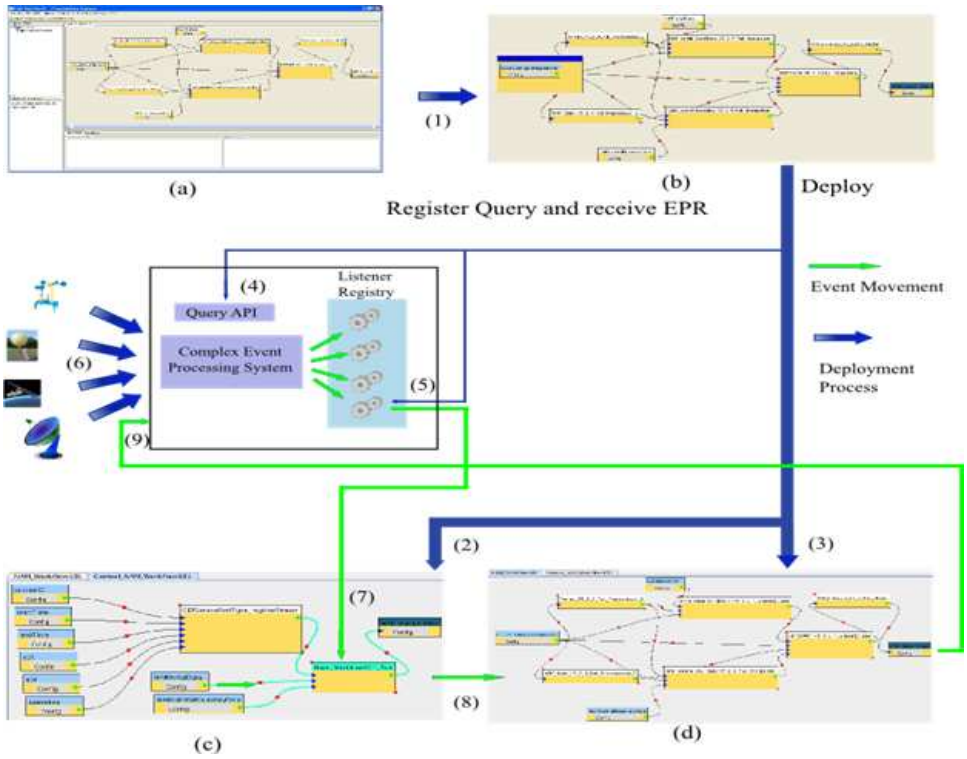

Figure 12: Conversion of workflow to Streamflow and how deployment of Streamflow connects events with processing components

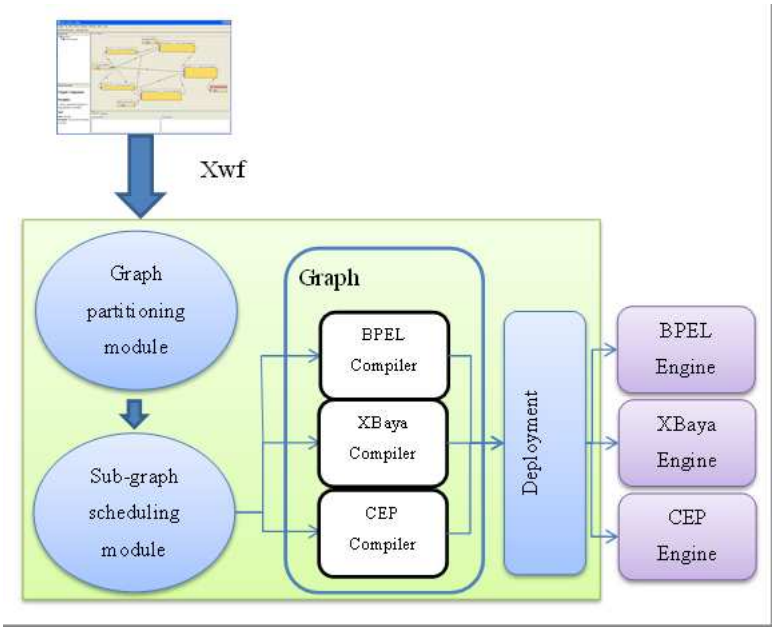

Figure 13: Compilation and deployment Streamflow DAG. A DAG may be compiled to any of the three target runtimes using the workflow DAG abstraction provided by XBaya

ODE [5].

4.3.2 Streamflow compiler

Streamflow composer allows the users to compose experiments using the Graphical User Interface as DAGs. Using these DAGs as a running system, requires compiling the DAG structures into runnable scripts specific to the runtime orchestration and execution engines. Given a particular Streamflow DAG, its different sub-graphs may have different runtime properties such as different runtime throughputs and runtime resource requirements. Depending on these aspects the Streamflow framework will partition the user composed DAG into different sub-graphs. The algorithms and justification of such partitioning is described in the following subsection. This section focus on illustrating how a given sub-graph will be compiled and deployed into to a particular runtime engine.

Figure 13 shows the architectural aspects of the compilation of a given Streamflow DAG. During the compilation XBaya will provide the Streamflow DAG to the

graph partitioning module and it will identify the biggest deployable sub-graphs. The runtime scheduling module will determine which runtime engine will be the best fit for a given sub-graph. This is based on number of factors such as the semantics of the activities in the sub-graph, expected runtime throughput, and expected resource requirements. Once a particular sub-graph is assigned to a runtime engine, it will be fed into a compiler for that particular runtime engine and the compiler will transform the sub-graph structure into an executable workflow script. As shown in Figure 12 graph a partitioning algorithm will produce multiple workflow graphs and they will be fed into the scheduling module that will try to identify which target runtime is the best fit for executing the sub-graph. The sub-graphs are described in the high level workflow description language described in previously and workflow compilers in XBaya are capable of generating executables for target runtime for a given graph or a sub-graph. The Streamflow framework will use three graph compilers, BPEL Compiler, XBaya Compiler or CEP Compiler depending on the scheduling decision. Once the executables for all the sub-graphs are generated the deployment model will interact with the deployment mechanism of the runtime engines. There are aspects like static input registering, hot deployment and protocol resolution that need to be resolved in the deployment module.

Compilation of BPEL sub-graphs The compilation of BPEL scripts and XBaya Engine scripts are functionally similar apart from the syntactic differences in the scripts. Both the algorithms are based on topological sorting algorithm [64][63] that will produce the ordering of the tasks that will be executed based on the dependencies in the graph. Such ordering would produce the maximum level of parallelism within the workflow graph that is being compiled [9]. It is important to note once a particular sub-graph is compiled and it starts executing it will behave as a running instance of a conventional scientific workflow.

Compilation of CEP sub-graphs If a particular sub-graph entirely consists of CEP nodes, it is defined as "CEP reducible" and it can be compiled entirely

in the CEP engine. The Esper Complex Event Processing system used in this framework does not provide means for graph structures;thus, the CEP compiler must convert the graph structure into a linear set of queries that would provide the functional dependencies exhibited in the DAG. This subsection illustrates how a DAG will be transformed into a set of queries while preserving the structural dependency shown in the DAG.

Normally orchestration of DAG activities is done using a workflow engine and workflow engines are well equipped for that. One choice in this case is to run the events through the CEP engine to process against the query, and then route that event to a workflow engine for orchestration and resolving dependencies. But if all those events need to be routed to a workflow engine, either XBaya engine or a BPEL engine, the throughput of those event streams will be upper bounded by the rates handled by the workflow orchestration engines. The event rates that Complex Event Processing systems could handle are much higher than the throughputs of the workflow systems. In such an architecture, the workflow engines will become a bottleneck and the high throughput event rates that the Complex Event Processing Systems are able to handle will go under-utilized. Therefore if the event flows are kept within the Complex Event Processing system when possible, it will significantly increase the maximum throughput that particular "CEP reducible" sub-graph could handle.

If the Stream related event flow can be handled within the Complex Event Processing CEP Engine itself, then the bottleneck that was described can be removed and the event that only involve CEP can run at the throughput supported by the Complex Event Processing system. There are certain nodes that would definitely warrant the orchestration of a workflow engine and there is also a class of nodes that can do without the workflow engine. In particular, the CEP node and its child nodes such as join node, split node and filter node, can be made to work in a runtime without a workflow engine. Compiling a single CEP node to work in an environment without a workflow engine is fairly straightforward. Consider a sub-graph or even an entire graph of CEP nodes haw to make all of them

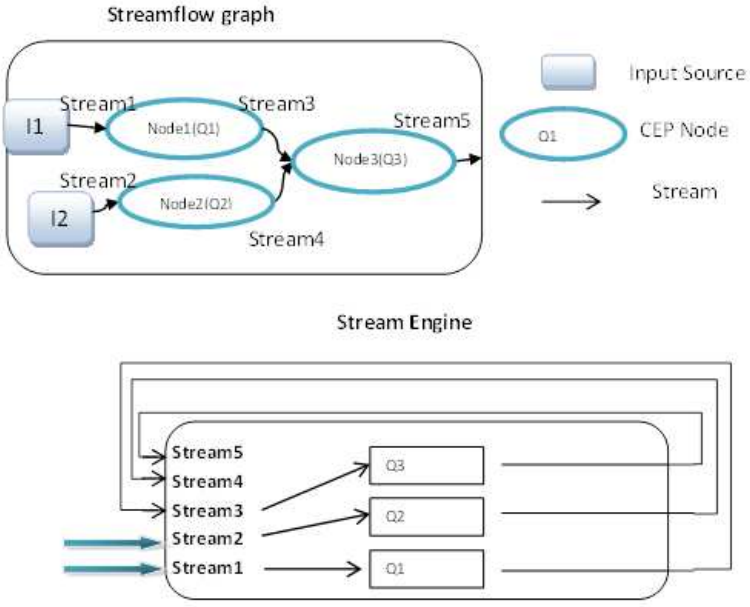

Figure 14: CEP Graph compilation to CEP Engine using the CEP queries that listen to the streams depicted by the edge dependencies in the graph

enforce the event flow dependencies is much more challenging. The conventional Complex event processing systems do not provide a means for dependency flow management; thus, there is a need for such dependencies to be incorporated to the Complex Event Processing system using existing EPL [36] functionalities.

Given a sub-graph in a Streamflow there can be a property that can be identified as "CEP reducible" defined as all the nodes in the sub-graph that are CEP nodes or a CEP child nodeslike join node, filter node. In otherwords, the processing tasks or kernels in the sub-graph are such that they can be expressed only using Complex Event Processing queries and do not require external service or component interactions such as web services. If a given sub-graph of a Streamflow is "CEP reducible" it can be shown that there is an algorithmic way of defining Complex Event Processing queries in such a way that it would preserve all the relevant event flow dependencies associated with the edges in the sub-graph.

Figure 14 shows a CEP node only Streamflow sub-graph and illustrates how

the components shown in the sub-graph will be reduced into queries and streams in the CEP Engine.

Stream1 ⟶ Node1(Q1) ⟶ Stream3

Above notation states that there is a Complex Event Processing node named Node 1 which has a CEP query, Q1, associated with it; this node will have an incoming stream Stream1 and will produce an output stream Stream3.

The query will be of the form **Select * from Stream1 where ...**

Now consider a downstream Node3 that uses Stream3 as an input to that node.

Stream3 ⟶ CEP Node3(Q3) ⟶ Stream5

To make this stream dependencies to be orchestrated at the runtime one option is registering a listener to Stream3 and sending them through a workflow engine so those events can be orchestrated to the Node3. But this is exactly the throughput bottleneck that was mentioned earlier, that was discussed to be undesirable. The other approach is to publish all the events back to the CEP Engine and get CEP Node 3 to listen to that stream. This would include slightly changing the Event Processing Language query associated with CEP Node1 to publish the event back to the CEP Engine.

Insert Into Stream3 Select * from Stream1 where ...

As shown above, the introduction of a new stream name and adding an Insert clause will signal the CEP Engine to publish the events back to the stream. The selection of the output stream name will be done during the configuration of the node in the XBaya composer and all the downstream nodes that would use this output stream need to utilize that stream name.

It should be noted that there is a strong possibility that two output streams may be given the same name. This could be seen as a feature or an issue that could lead to nondeterministic behavior. The default mode would be to produce a Universally Unique Identifier (UUID) [71] which would be relatively unique. To guarantee the uniqueness of the stream name chosen at the end of configuration of a given CEP node the Stream will check with the CEP Engine to ensure the selected name is not in conflict with an already existing stream name. On the

other hand the user might be interested in publishing events to an existing event stream in which case the user would be required to confirm his/her action to make sure it is not an accidental selection.

Once the naming issue is resolved, the Complex Event Processing queries can be registered with the CEP Engine. It will be necessary to resolve the required flow dependencies that are represented by the edges in the graph. Because the uniqueness of the output stream name is guaranteed unless otherwise is intended, the fact that the downstream nodes listen to that particular stream name ensures the semantics implied by the edges in the Streamflow graph are met. This allows the notion of a dependency graph to be encoded into a set of EPL queries in a CEP Engine.

stream1 $\longrightarrow$ Node1(Query1(stream1)) $\longrightarrow$ stream3

stream2 $\longrightarrow$ Node2(Query2(stream2)) $\longrightarrow$ stream4

Above set of graph edges will produce the following queries in the CEP Engine; the CEP compilation Algorithm explains exactly how this matching of streams is done:

Insert into stream1

Insert into stream3 Query1(stream1)

Insert into stream4 Query2(stream2)

Such dependency based stream insertions and matching stream names would facilitate the implied semantics shown in the graph structure of the Streamflow. This also removes the bottlenecks introduced by the workflow engines because the entire event interaction happens within the CEP Engine so the event throughput rates are bounded by the rates of the CEP Engine which are significantly higher, relative to workflow systems. Figure 15 lists the compilation algorithm for sub-graphs that consist entirely of CEP components. The Algorithm makes sure that the dependency edges shown in the DAG are enforced when the sub-graph is compiled into set of queries.

The algorithm would first topologically sort the graph [69] to obtain the nodes in the order of the dependency graph. For example if Node 3 is dependent on

```
1  compileToCEP(Graph g) {
2    Vertice[] sortedNodeOrder = topologicallySort(g);
3    Map portStreamLookupMap = new HashMap();
4    LinkedList inputStreamComponents = g.getInputSources();
5    for (Vertice source : inputStreamComponents) {
6      portStreamLookupMap.put(source.getOutputPort(),
7          source.getStreamName());
8    }
9    for (Vertice nodei : sortedNodeOrder) {
10     Edge[] inputEdges = nodei.getinputEdges();
11     String[] nodeInputStreams = new String[inputEdges.length];
12     for (int i = 0; i < inputEdges.length; i++) {
13       Edge edgei = inputEdges[i];
14       OutputPort fromPort = edgei.getFromPort();
15       String portStreaName = portStreamLookupMap
16           .get(fromPort);
17       nodeInputStreams[i] = portStreaName;
18       validate(nodeInputStreams, nodei.getCEPQuery());
19     }
20     String newStreamName = generateUniqueStreamName(nodei
21         .getOutputPort());
22     registerWithCEPEngine(newStreamName, nodei.getCEPQuery());
23     portStreamLookupMap.put(nodei.getOutputPort(), newStreamName);
24   }
25 }
```

Figure 15: CEP compilation Algorithm

Node2 then it is necessary for the Node 2 related queries to be registered with the CEP Engine before the Node 3 can be registered. Also, if the CEP graph contains dependency cycles, there will be event build up in the loop which would eventually crash the system, so the graph is strictly a Directed Acyclic Graph. The topological sort on a graph also allows the identification of cycles and thus the line2 topological sort serves two purposes.

It should be noted that the Algorithm is designed to work with nodes with the same cardinality and it is much easier to identify subsets of nodes with the same cardinality by using a technique similar to Information Flow control [109] labeling.

Next the loop in line 18 makes sure the query in the CEP node is in fact working with the input stream names and not referring to unbounded streams. This is necessary to guarantee that the visual semantics of the edges in the graph are preserved. It is necessary to assign unique stream names to the output streams of

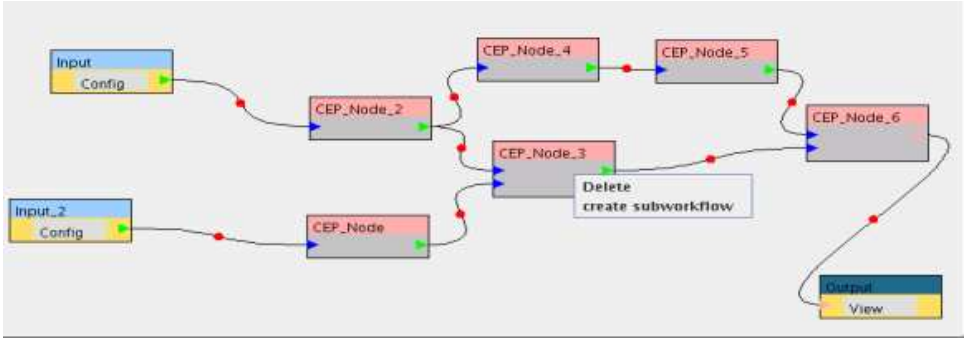

Figure 16: Streamflow CEP graph that can be configured to any EPL query supported by the CEP engine.

each CEP component and this is achieved by assigning by default trying to assign the output port name of the CEP name to the particular output stream name; this name will be looked up with the CEP Engine which will act as a register for all the stream names and will return a unique stream name if there is a conflict. In fact that the CEP Engine assigning the Stream name will remove any possible synchronization issues that might occur. Next, in line 22, the system will register the Query as well as the Stream name with the CEP Engine. The time complexity of the CEP compilation is O (N+E) based on the visitation of the nodes and edges during the algorithm.

Figure 16 shows a typical CEP graph and this graph can be entirely reduced to single as sub-workflow component assuming the event cardinalities of the nodes are the same. It should be noted this entire graph can be reduced to single Sub Workflow if necessary.

4.3.3 Graph Partitioning

The overall Streamflow DAG composed by the domain scientist consists of activities of different semantics and different runtime characteristics. The underlying execution runtime consists of CEP engine and workflow engines which can handle different runtime characteristics. The goal of the graph partitioning is to partition the user composed DAG into sub-graphss such that each of the sub-graphs runtime characteristics will match to the underlying execution environments. Sometimes

the DAG as it is may exhibit matching characteristics to a particular execution environment and in such cases graph partitioning is not necessary. But as mentioned in the introduction there are many scientific applications that require different phases of computations that will warrant partitioning of the DAG into sub-graphs.

When considering the compilation of generic Streamflows the framework will use several strategies to compile workflow and query scripts that would orchestrate the running instance of a Streamflow of a specific graph. Figure 17 shows an example Streamflow and how it would be compiles into different sub workflows that are ready for execution. Sub workflow 1 was mined based on the fact that it consists of pure workflow nodes consisting of a connected sub-graph. The Stream input to node 5 made the all the rest of the downstream nodes to be Streamflow nodes and Cardinality of the nodes 5, 6, 7, 8 and 9 are the same, 1 to be precise thus sub workflow 2 is considered as a unit of execution. Node 10 is a filter node thus it has cardinality less than 1 this stops node 10 being included in sub workflow 2 because of two reasons. First being a filter node it forces involvement of queries thus it needs to be considered as an isolated case. Considering nodes 11, 12 and 13 they have the same cardinality again cardinality of 1 to be precise thus would qualify to be a considered as a separate sub workflow. But there is some further complication that needs considering if sub workflow 3 is considered as a separate sub-graph the input streams to the sub workflow 3 has different input rates thus it would require a stream join. This means to upgrade the slower input rate to a higher rate by repeating event values or downgrade the higher input rate to a lower input rate by throwing away extra events and synchronizing with the slower event stream or some other combination of join. In theory id pure dataflow model is used similar implicit join would happen at some point of time so such assertion of an explicit join will not curtail the expressiveness in any significant way. Once the sub workflows are identified the Streamflow compiler would generate the workflow scripts using the workflow language compilers introduced in section 3. The extra effort that was put into the having an abstract workflow graph and

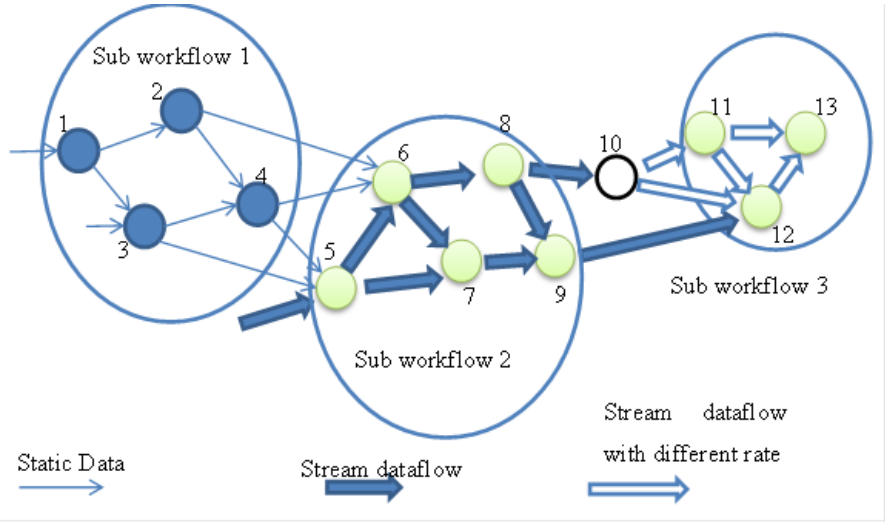

Figure 17: Sub Workflow mining during Streamflow compilation. The seperation of sub-graphs are governed by the cardinality of the nodes and edges

having compilers that generate different workflow language scripts for the same abstract graph becomes very useful during the Streamflow compilation process.

Whether the sub graph mining algorithm finds the optimal sub-graph is an important theoretical question that needs to be raised. This thesis argues that the query language predicates the change of rates in the dataflow and the data rates drives the optimal sub workflow identification. Given a query which can very well include arbitrary user defined functions which could be written in any Turing complete language, the question of asking how it affects the event rates is an extensional problem thus Rice's theorem [108] would make it an un-decidable problem.

But in most cases users may be able to provide clues as to how rates may change. By forcing explicit joins the need to programmatically analyze how queries affect the event rate can be eliminated for most of the cases. There is an incentive for the users to identify these event rates if possible because that would give much more abstract view of the dataflow, for example Figure 17 Streamflow would be reduced to a Streamflow of three sub workflows which would much better perspective to the user about the control flow as well as it would be easier to

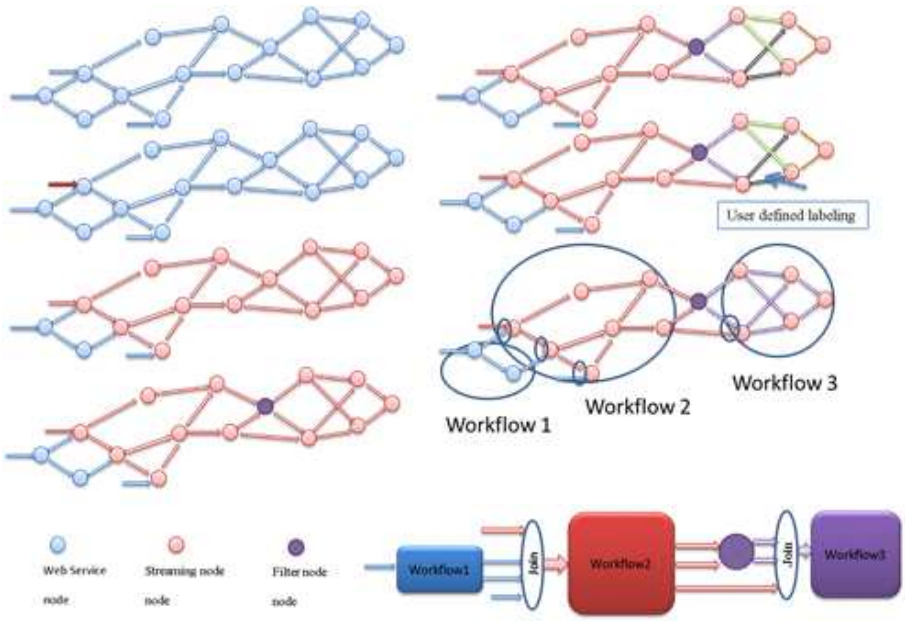

Figure 18: Labeling Algorithm that is used to partition the Streamflow graph based on the event streams that the edges carry. The graph structure shows what happens to an already partitioned graph if one of the nodes change from a web service node to a filter node

estimate resources for such a structured graph.

Consider the Streamflow that is shown in Figure 17. Assume the graph was initially a graph of web service components of three inputs and at one point the user selected one of the inputs to be streamed in from an event stream(shown in yellow input arrow). Such a change would imply that for each event in the event stream the workflow graph should be recalculated at least a portion of it.

A closer examination of the implications of such stream input to a workflow reveal that part of the workflow will be recalculated just to satisfy the data dependencies of the computation instead of it having anything to do with the new events arriving at the event stream. To understand how to identify which of the nodes need to be recalculated the Streamflow framework uses labeling algorithm. The labels correspond to the cardinality of the stream in the edges of the graph. The intuition behind the labeling is if a stream source is connected to a node, all its downstream nodes will have to handle the time series of outputs produced by

```
1  labelGraph(Graph g){
2    def List<Node> streamInputs = getStreamInputs(g);
3    def List<Node> staticInputs = getStaticInputs(g);
4    def Queue<Node> queue //all inputs are labeled
5
6    //Intialize static inputs to a one label
7    foreach(inputNode in staticInputs){
8      inputNode.setStaticLabel();
9      addInputLabeledDownstreamNodesToQueue(inputNode, queue);
10   }
11   //invent label for stream sources
12   foreach(inputNode in streamInputs){
13     inputNode.assignUniqueueLabel();
14     addInputLabeledDownstreamNodesToQueue(inputNode, queue);
15   }
16   //queue invarient: all nodes have already labeled inputs
17   while(queue not empty){
18     Node node = queue.dequeue();
19     if(node.hasSameInputLabels(){
20       if(node.isCanonicalOneTOOneNode()){
21         node.setSameLabelAsInput();
22       }else{
23         node.inventLabel();
24       }
25     }else{
26       node.inventLabel();
27       if(node.isCanonicalOneTOOneNode()){
28         node.requireJoin();
29       }
30     }
31     node.checAndAddAllLabeledDownStreamNodes(node, queue);
32
33   }
34 }
```

Figure 19: Graph Cardinality Labeling Algorithm

their previous components. The labeling algorithm labels the edges in the graph in such a way that the edges with same label are expected to carry same number of events over the lifetime of the Streamflow run.

Out of the node types there are some that would present themselves as the canonical one to one input-output correspondence and these can be programmatically labeled. For other types of nodes it is not always straightforward to define the output label because the query associated with the node can change the output rate. Optionally the user can specify the input-output correspondence for nodes without canonical input output correspondence and it will be taken into consider-

ation to refine the labeling. At the point when the algorithm cannot determine a label mapping from the already assigned labels, it will invent a new label. Reusing the labels where possible lead to identification of bigger sub-graphs improves the effectiveness of the partitioning algorithm.

The significance of the labeling of the edges is, once the graph is labeled it identifies the sub-graphs that can be considered as a single one computational unit or a single sub-workflow. For such sub-workflows at each invocation it would behave as a pure workflow where the control-flow is deterministic and well defined. These sub-graphs essentially have different stream rates, although the stream rate within a sub-graph is the same. Figure 17 shows a graph that is labeled and the different labels are shown in different colored edges graphs. Based on this labeling the Framework will partition the original graph in to sub-graphs of workflows. Although when considering the irregularities of the control-flow in the entire graph, the individual sub-graphs behave more like workflow invocations for input set in the event stream. This is the reasoning behind considering these sub-graphs as single computational units. Although the entire graph shown in Figure 17 cannot be deployed to an off the shelf workflow engine, the individual sub-graphs can be deployed to standard workflow engines because they have regular control-flow.

The graph partitioning identifies the biggest sub-graph that qualifies as a pure workflow deployable to a workflow engine. The Figure 20 shows the algorithm that partitions a Streamflow graph into sub-graphs. Topological sorting [64][63] is used to orders the nodes of the graphs in an order that they can be scheduled based on the dependencies of the nodes. There are certain graphs that will not produce such an order because the graph contain cycles. The topological sorting also identifies these cycles and thus it can be used to parse a given graph for its validity and to check whether it is a Directed Acyclic Graph (DAG). The algorithm identifies cycles by traversing the graph in near topological order and clustering sub-graphs with connected nodes with same labels assigned by labeling algorithm. Once the sub-workflows are identified a high level graph abstraction of the original workflow can be reconstructed as shown in Figure 17.

The identification of the sub-graphs of an original graph is used to reveal the boundaries where the event stream rates do not match. This would mean that there need to be some kind of event matching at the boundaries. For example consider the input edges to the workflow 3 in Figure 17. Two of the three inputs are of same cardinality and other edge is of different cardinality. It means the event streams the edges represent would produce events at different rates, for example in Figure 17 workflow3 operates at 1 event per second and the second event stream producing events at 2 events per second. There needs to be an explicit decision made on how the two stream sets should be joined. Normally for practical circumstances the slower event stream will be matched to the faster event stream by caching the most recent event of the slower event stream. If stream1 is represented by S1T1,S1T2, S1T3 time series and stream2 is represented by S2T1, S2T1.5, S2T2, S2T2.5, S2T2, where the subscripts represent the clock reading. There need to be explicit join that decides how these two event streams can be joined and fed into workflow3. If the user wants to match the faster stream by caching the slower stream which would produce (S1TI,S2T1), (S1T1, S2T1.5), (S1T2,S2T2) , (S1T2, S2T2.5), stream. Or the user could match the faster stream with the slower stream by throwing away events which would produce (S1TI,S2T1), (S1T2,S2T2), stream.

4.3.4 Inter-sub-graphs Event Orchestration

The graph partitioning module and the run-time schedule mole will partition the Streamflow graph and deploy it to different orchestration run-times. When the events start flowing through the system there needs to be transition between these sub-graphs running in one system to another. This sub-section explains how it will be done canonically.

In Figure 21 consider that workflow 1, workflow 2, workflow3 got scheduled to CEP Engine, XBaya engine and BPEL engine respectively. Workflow1 and Workflow2 has two input event streams and produce two stream event streams as output. Since the origins to the two input event streams are not the same the two

```
1  partitionStreamflow(Graph g){
2
3    def Map<Label,List<Node>> partitionMap;
4    topologicalSort(g);
5    labelGraph(g);
6
7    Foreach( node in g){
8      partitionMap.add(node.label, node);
9      if(node.requireJoin()){
10       introduceJoin(node, g);
11     }
12   }
13   def Set<Graph> partitions
14   Foreach(key in partitionMap){
15     List<Node> subGraphNodes = partitionMap.get(key)
16     Graph partition =  createSubGraph(g, subGraphNodes);
17     partitions.add(partition);
18   }
19   return partition;
20 }
```

Figure 20: Graph Partitioning ALgorithm

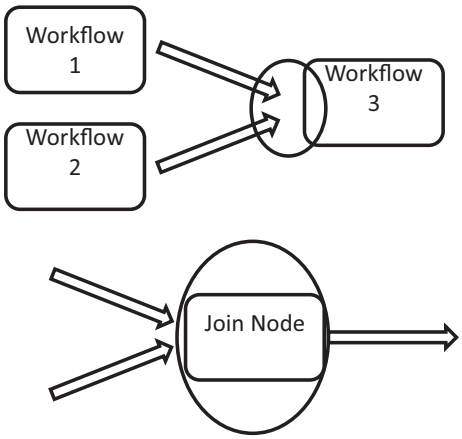

Figure 21: Sub-graph boundary join insertion. If a particular subgraph has streams of different cardinality as inputs a join node is explicitly inserted to preserve the correctness of the graph

output streams generally will have different cardinality. The outputs of a given sub-workflow need to be fed to the sub-workflows that have stream dependency to it. One approach is to slightly modify each workflow to include a publisher at the end of the sub-workflow to the dependent sub-graph. Other approach is to publish all the outputs of the producing sub-workflow to the CEP engine and the consuming sub-workflow to listen to the events published by the producer. There is reasoning for and against whether all the events produced by such sub-workflows should go through the CEP engine.

Obviously going through the CEP Engine would introduce extra latency to the graph pipeline where as direct approach would by-pass such latency. Often there is a need for joins at the boundaries of such sub-workflows because one of the reasons the sub-workflows got partitioned is because of their discrepancies in the event rates that would prompt stream joins. In such cases going through CEP engine is mandatory like in the case of workflows shown in Figure 21. Further the outputs of these sub-workflows are derived event streams and could be of some significance and having it published to the CEP engine would allow it to be used or monitored by a different process. Also this will allow the Streamflow monitoring system to intercept the output streams of a given sub-workflow thus allowing the monitoring system to be coherent with the rest of the CEP events. Going through the CEP engine will provide a means for lose coupling between the sub-graphs.

A given sub-workflow would produce all the outputs at the same time so all of those outputs can be published to the CEP Engine as a composite event which will capture the correlation between those outputs. Such events can be published as a stream with a unique name so the subsequent stream connections or the stream joins have unique stream reference. Once the partitioning identifies there need to be joins that has to be resolved because of different stream cardinalities, the deployment time will prompt the user to introduce a join node in place. Once the joins are resolved the resulting stream is registered with endpoint reference which would indicate the endpoint reference that the stream events should be sent to. For example the Figure 21 's Workflow3 once deployed would have a web service

EPR which will accept events.

Assume that the output of Workflow1 is stream1 and workflow 1 has two outputs and this would mean each event in stream1 is a composite event of the two outputs denoted by s1.p11, s1,p12. Assume a similar case for Workflow2 where the composite event is denoted by s2.p21,s2p22. Say the input to WF3 has input p11 and p22 from the two streams, and assume that the event rates of stream1 and event rated of stream2 has some disparity. This would mean that inputs to workflow 3 have to be prepared by joining the two streams. This would prompt some join condition that needs to be specified by the user. If the join condition is captured by Query1 this would provide much cleaner abstraction in terms of how the join transitions take place in a given Streamflow. Following Query1 shows a possible joining of the two streams where there will be pair of events be generated matching the past phased stream.

Select s1.p11,s2.p22 From stream1 as s1, stream2 as s2;

The CEP node that captures Query1 would have two input event streams and will produce one output event stream. Given the fact that stream1 and stream2 stream names are unique and they are outputs of Workflow1 and Workflow2 respectively, Query1 will produce the correct input event stream for workflow 3. To complete this connection at the time of the deployment of the CEP join node need to be aware that every output event from Query1 need to be sent to Workflow3. This is achieved by associating a publish EPR with the query during the deployment to the CEP engine. The expectation is the CEP engine will honor the contract that every output event from the query will be published to the EPR. Esper Complex event processing system allows an API to register event listeners with streams with the contract that event listener will be invoked for every event output in the stream. These listeners could have invoker hooks that would publish events to the EPR registered during the deployment of the join node.

```
1  <radar>
2
3    <location>
4
5      <centerlat>40</centerlat>
6
7      <centerlon>-86</centerlon>
8
9    </location>
10
11 </radar>
```

Figure 22: Example metadata event form an instrument

4.3.5 Data Binding

The workflow system primarily deal with web service based activities and the events are Xml messaged. The Complex Event Processing queries make references to the properties of the events and this brings up the need to bind the xml event tags with the properties referred by the CEP queries. This subsection explains how the framework allows users to map event data to queries and this is referred to as data binding from here fourth.

Most of the Event that are focused in this framework are Xml events and the Queries defined during composition refer to property values of the events. These event properties are usually the values in the Xml TextNodes and the EPL specification allows reference to such properties to be done independent of the structure of the properties inside the xml messages. This is achieved by defining references to the event properties inside the xml message event using Xpath expressions and assigning property names that can be referred in the EPL query.

Consider query that would filter event based on the location properties of the radar event. Above xml shows a possible metadata event from a radar and assume that the query that run over the stream of events will produce a event stream with radar events which falls in a particular special area. This will mean the query would have to refer to the location properties in the event and filter out the events that will not fall within the required spatial bounds. The query for such a setup would look like the EPL query given below.

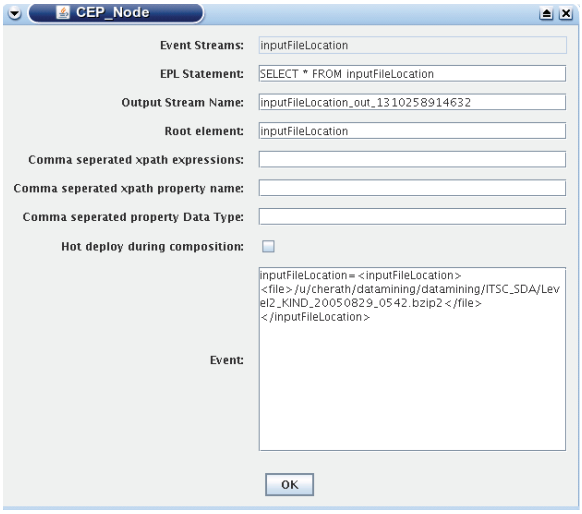

Figure 23: CEP Node Configuration where the declarative query can be declared for as the processing component. It shows the sample of the last fetched event. Variables used in the query can be data bound using xpaths.

Select * from RadarEvent where lat > 38 and lat < 42 and lon > -87 and lon < -85

The variables lat and lon referred in the query need to map to the cernterlat and centerlon Xml element values in the Xml event. This is achieved by specifying aliases for Xpath properties that will map the variable names used in the query to the Xml element values found in the Xml Event.

```
Query Variable Name :=  XPath expression   :Type
lat  := /radar/location/centerlat  :NUMBER
lon  := /radar/location/centerlon  :NUMBER
```

Mapping shown in the above would allow the CEP Engine to write queries with the ability to be data bound to the values in the Xml event. Besides the event properties it is also necessary for the user to specify the root event so the CEP Engine can associate the stream name with the xml events that that gets published to the system.

Figure 23 provides a configuration options available for a given CEP Node and

these include the immutable input stream name and then configurable parameters like the EPL statement and other data binding related aspects.

4.3.6 Hot Deployment and Stream registry

When a domain scientist is composing a Streamflow it would be extremely useful to know what will be the event rates that a particular node will have to handle. The runtime throughput depends on many factors so it is hard to calculate this in general. But if the framework is capable of deploying Streamflow components as and when the user composes it, the system can monitor the affect of adding a particular node to the Streamflow in real-timeand notify the user about the throughputs it is dealing with. The hot deployment model allows users to do exactly this where they can examine the stream rates of the newly added nodes to the Streamflow during the composition phase so the user can incrementally build the experiment while keeping an eye on the event rates.

There are two aspects that a particular CEP node has to focus during composition. They are the functional aspect and the runtime aspects of the node. The functional aspect deals with identifying the input data streams, the EPL query associated with the node, data bound parameters and other aspects that are required to deploy the query associated with the node to the CEP Engine. The other aspect is how the node would perform during the runtime. That is to say what will be the input event rates the node has to handle and the output event rate the node would produce. This would become particularly important when a particular node has an EPL query that would increase the event rates and whether the system will be able to handle such rates. More frequent example is where output stream of the particular node is connected to a web service node which has much lower maximum throughput rates. Given this it can be argued that it will be useful to know the effect of adding a particular node to the system.

XBaya Streamflow provides two modes of composition which would be referred to as online and offline. The offline mode is the conventional workflow or dataflow composition mode where the entire graph is composed and completed before any

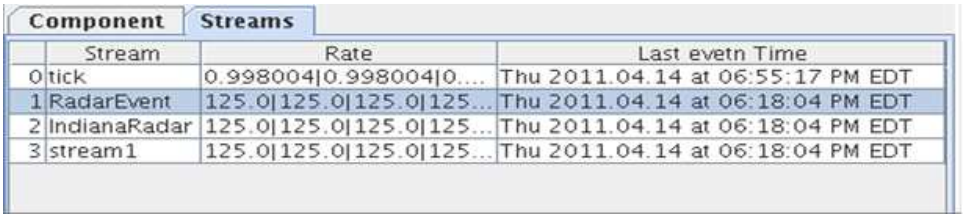

Figure 24: Listing of the available streams by quering the stream registry.

deployment and running is done. But the fact that the CEP Engine and Complex Event Processing systems are real-time systems allow the users to deploy the components as they compose the graph and this would allow the user to see the real-time effect of that component and the output stream rates that particular node would produce. It should be noted there is a possibility that the output stream rate can be predicted by only looking at the input rates and the nature of the query, but the general case this would reduce to an extensional problem[ref Rice] and will be undecidable.

XBaya composition tool allows the user to connect to the CEP Engine and load the available streams in the CEP Engine that are currently active. Figure 24 shows the interface where it lists the streams and the stream related metadata. The metadata include the sequence of event rates calculated at the time interval of the event arrival and by providing multiple event rates the system allows the users to get an idea of short term fluctuations of the event rates of the stream. It also provides a time stamp of the last known event that arrived at the stream. This would allow the user to determine whether the data is current or is it a zombie stream that has not been active for a while. Interface also provide management functionalities like refreshing stream metadata and also stopping removing streams from the CEP Engine.

The most useful functionality the stream listing provides is the ability for the users to use a stream in a Streamflow. User during the Streamflow composition can use a stream from the listing and add it to the workspace and one such stream sources are shown in Figure 25 as a Stream_Source. Now this source can be used for composition of Streamflow and normally the composition happen in such

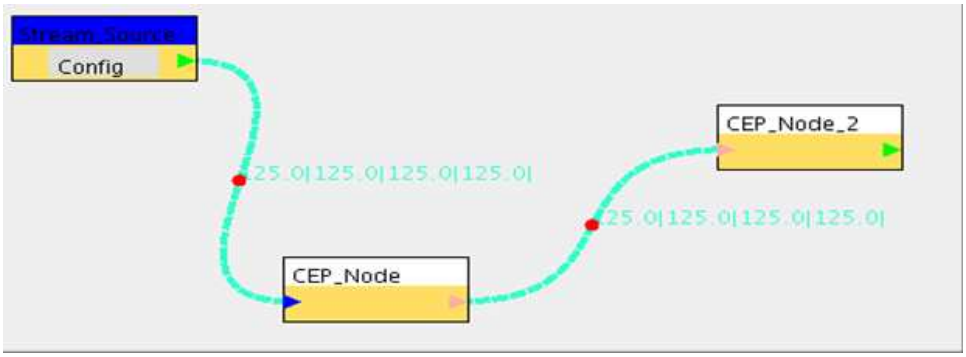

Figure 25: XBaya Dynamic composition mode where the cep nodes are deployed during composition and the XBaya will show some metadata about the new output stream

incremental manner. During composition as shown in Figure 25 the Stream_Source is connected to the CEP_Node which is obviously an CEP Node and XBaya will immediately show the event rate that the particular edge implies by listing a set of last known event rates so the user would have an immediate understand what would be the rate at the CEP Node.

Once the CEP node is connected the user has to configure the node using the Figure X3 configuration window and if the user selects the "Hot deploy during composition" option the EPL associated with the Node will be immediately deployed and the EPL will go into effect in the CEP Engine. This Is the online composition mode that was referred to earlier. The enhancement that this system provides is that when user tries to connect another CEP component to the output of this node, that particular edge will also be annotated with the event rate and thus the user can make an judgment on whether the node down stream could handle the rate produce by the node just got deployed.

Also the CEP nodes are known as dynamic nodes where user can drag and drop multiple input stream to the node and node will create a port dynamically and accommodate the docking of the edge. This allows the user to configure the number of stream inputs to a given CEP node.

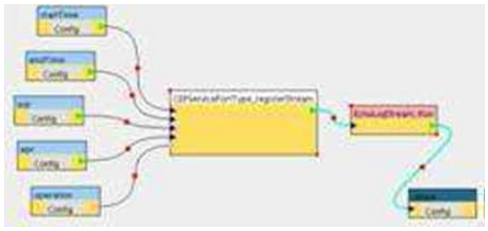

Figure 26: Simple Streamflow showing a web service component that initiates a stream feeding a down stream Streamflow node

4.3.7 Streamflow Runtime

Once a give Streamflow DAG is partitioned into the respective sub-graphs and the sub-graphs are deployed to the most suitable runtime. Figure 10 shows the three possible runtime orchestration engines a given subgraph can be scheduled to run. The CEP Engine is different from the other two engines because the sub-graphs that get scheduled to CEP Engine need to be entirely of CEP nodes. The XBaya Engine or BPEL engine support similar workflow semantics and this mean that a given sub-graph can either be deployed into the XBaya Engine or BPEL Engine. This decision is mainly based on the runtime characteristics of the sub-graph.

The execution of the composed and deployed workflow begins by invoking the control Streamflow and as shown in Figure 12 the first activity of the control Streamflow is to contact the complex event processing system to register a query that was specified by the users about the type of events that this workflow is interested in and the duration that the continuous query should be valid. This is shown as step 4 in Figure 12.

The workflow then waits for the first message to arrive from the CEP Engine and when the CEP Engine finds an event that satisfies the query, it sends a message to the waiting workflow process. Once the workflow receives the message and invokes the child workflow that it starts executing. Meanwhile the control Streamflow's Active node would loops back and waits for another observational event and the process continues.

A given workflow component may be viewed as a function with or without side effects that is invoked upon receiving the data events from the stream. The

workflow takes the input and a global state and produces an output and depending on whether the workflow is side effect free or not it may not or may change the global state, respectively. The static nature of the workflow inputs can be observed in this notation because of the workflow execution begins only when all the inputs are available. Once the workflow starts executing the inputs cannot be changed nor is lazy evaluation [70] of inputs allowed.

The edges in the workflow graph are denoted as:

Workflow-Edge = ((Service1, outputPort1) $\longrightarrow$(Service2, inputPort1))

where Service1 has outputPort1 which is connected to the inputPort1 of Service 2. This represents a single data event over the lifetime of the workflow passed from Service1 to Service2. The Streamflows supports the Workflow-Edge and also introduces the Stream-Edge denoted by:

Stream-Edge= ((Service1, outputPort1) $\underrightarrow{stream}$ (Service2, inputPort1))

where Stream-Edge represent a stream of data that may pass through the edge over the lifetime of the workflow. In other words Service1 would produce a data stream and that stream is channeled to Service2 via the Stream-Edge.

For the purpose of completion we would allow Input nodes and Output nodes be connected to services input ports and output ports respectively and as a convention they will be names Input-Node$_i$ or Output-Node$_i$. These will only be mentioned when necessary and it is assumed the workflows always have Input nodes and Output nodes connected to the remainder of the data ports when not specified. Some services have one output port and one input port in such cases we would adopt the notion where Service1 $\longrightarrow$ Service2 represents the edge and when there is no ambiguity we would use this notation in a sequence such as S1 $\longrightarrow$ S2 $\longrightarrow$ S3.

Sometimes the number of events that may be channeled via a different Streamflow edges may vary and where possible we try to quantify the events that may pass through an edge. We define Weight of a Streamflow edge as follows to capture this information.

- Weight(E) = 1 if Workflow-Edge

- Weight(E) ≥ 1 if it s a Stream Edge and would be a number of events or a time in units depending on how the stream is bounded or simply unbounded if the Stream in indefinite.

4.3.8 CEP Engine Interface

The CEP Engine is a persistent service that would provide a web service interface to existing Complex event processing system. The CEP Engine provides interfaces that would facilitate four major functionalities.

- Query registration and adding listeners to query results. - The CEP Engine allows users to register EPL queries and at the event of queries are satisfied the user can register an web service Endpoint Reference or a stream name to redirect the resulting events.

- Available streams. - The interface provides a mechanism for the user to query existing streams that are active in the CEP Engine. These can be used during the composition phase to develop the Streamflow.

- Publish(push) API or pluggable pull extension - CEP Engine provides a web service interface for publishers to publish events. If the event rates are higher than the rates supported by the publish web service interface the CEP Engine provide a pluggable extension mechanism to pull event from other source and be published to the engine against the existing queries.

- Unregister query - A Stream that has timed out or due to other reasons it is necessary to unregister streams and queries, and CEP Engine provides a API call to unregister queries. Another use case that the unregistering of the queries that are produced as outputs of Nodes. So it is important to unregister the streams when the node is no longer active.

Once a sub-graph is deployed it is ready for execution whenever the driving event source has incoming events. One particularly significant transition point to note is graph edges that cross the different engine boundaries. For example if

the Streamflow DAG has an edge in such a way that From node of the edge got scheduled to the CEP Engine and the To node of the Edge got scheduled to the XBaya Engine. Now there needs to be routing of messages from the output stream of the From node to the input of the To node in the XBaya engine. During the deployment process all such cross boundary dependencies will be resolved and End Point Reference (EPR) of the To node workflow will be sent to the CEP Engine when the sungraphs are getting deployed. There is a publish mechanism in the CEP engine that will honor the contract that if there is a EPR associated with a given CEP query, it will ensure all the output events of the query will be published to that EPR. This arrangement ensured flow of events from one runtime engine to another.

4.3.9 Iterative programming approach

Event processing is an evolving process unlike compiling and running programs. It could be discovery oriented or simply iterations are performed to match the processing requirements with the available resources. Streamflow framework try to facilitate such iterative approach to programming Streamflows. System provides interfaces for hot deployment as well as monitoring of the system state due to the last change. Figure 27 shows the interaction between the user and the Streamflow graph where use will make changes to the existing Streamflow graph and hot deploy those changes and monitor the effects on the system to determine whether he did the right change or whether he needs to undo those changes and try a different approach. The monitoring will give statistics such as execution rates, internal queue length and workflow notifications that will aid the users decission.

4.4 Weather use case

The use case presented in this section involves in processing large number of radar events from weather radars and processing them in a coherent manner, using both the Complex Event Process and Workflow processing techniques. It will use the graph partitioning techniques presented in the earlier section to pin the sub-graphs

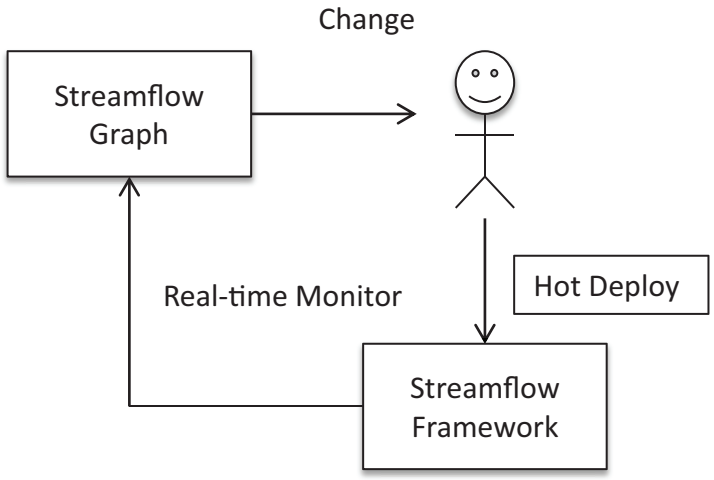

Figure 27: Iterative programming approach

to the right runtime framework based on runtime characteristics.

There are 121 NEXRAD Level II [55] weather radars distributed across United States. These radars produce radar data for every sweep thus publishing events per every sweep. These events are distributed using a binary publish subscribe framework named LDM [20].

The latest weather forecasting models proposed by Weather Research Forecasting model WRF [87] allows mesoscale forecasts with higher accuracy. But the computational requirements for such high resolution forecast are significant. A 36 hour forecast for a 800kmx800km spatial region with 5km grid spacing would run for around 2 hours with 1024 nodes in Indiana University BigRed super computing resources.

Even with access to the supercomputing launching high resolution compute intensive forecasts for all the data sets produced by the weather radars is infeasible. The severe weather conditions happen occasionally and most of the time the light weather readings do not require highly compute intensive forecasts. The approached proposed in this case study is to data mine the data sets using computationally feasible techniques to identify the severe weather datasets and selectively

```
1 <radar>
2   <file>/ldm/datamining/datamining/ITSC_SDA/Level2_KLIX_20050829_0542.bzip2</file>
3 </radar>
```

Figure 28: Weather radar event for a NEXRAD Level 2 data

launch computationally intensive high resolution forecasts which would make it manageable weather monitoring system. The data mining algorithms to identify possible severe weather also requires execution of a small workflow, but this is relatively less computationally intensive thus being able to keep up with the real time radar event throughput.

The use case focuses on three phases of computation, CEP phase, high throughput low computationally intensive phase and low throughput computationally intensive phase. Each of these phases will have intermediate CEP phase as necessary.

It is assumed that the radar events are published to the CEP Engine as an xml event as shown. Although it will be a metadata event of the location of the radar data, it has sufficient metadata in the event including the encoding of the file name to do meaningful processing.

The composition would start with this stream which is named "radar" and is shown as the `Stream_Source_radar` in Figure 2A. This stream consist of all the radar events distributed across United States. To find this event source XBaya allows a registry lookup which lists the existing active streams and displays some metadata about it. Figure 29 shows such a registry table and it shows metadata like, last event, last event time, last few known event rates.

Assume the case study involve in monitoring weather in Indiana. It is necessary to select the radar events from Indiana. There are three NEXRAD radar station in Indiana (Evansville, Indianapolis and Fort Wayne). In the first phase the objective is to produce a single stream of event that has radar events from Indiana. This is done by defining three event streams for the three radar stations by matching regular expressions on radar event name that identifies the radar station. For example KIND is radar id for Indianapolis radar station and this is encoded in the event name of the xmlradar event that was introduced earlier.

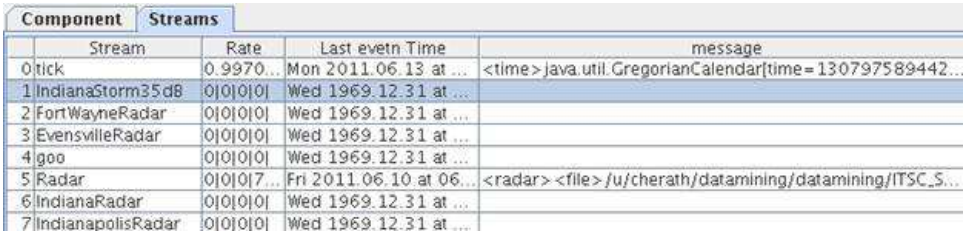

Figure 29: Stream Registry showing the currently deployed streams. It shows the status of hot deployed streams generated by CEP Nodes

TO achieve this there need to be definition of stream properties. For example the EPL query make reference to the file element value with the alias radarfilename. There need to be data binding of the alias name to the element value. This is done using xpath [24] based referincing to identify a property value and binding it to a property name. For the following CEP query to be valid it is necessary to bind the /radar/file xpath expression to radarfilename property. Once bound Esper is capable of executing the following query where it will select the events from the "radar" stream which has a radarfilename property matching the given regular expression. Defining the query, defining the properties defined by the query and choosing the output stream name is part of configuring the CEP node as shown in 30.

```
SELECT * FROM radar WHERE radarfilename REGEXP './*Level2_KIND_./*'
```

The 31 shows the overall Streamflow graph starting from the "radar" Stream Source and identifying the three radar streams that belong to Indiana called Indianapolis,Evensville and FOrtWayne processed by CEP nodes `CEP_NODE_Indianapolis`, `CEP_NODE_Evensville` and `CEP_NODE_FOrtWayne` respectively. Once the three streams are identified it will be merged to one stream called IndianaRadar. It is important to note that this is not join where three events Ai, Bi, Ci from three streams are joined together to form composite events of the type (Ai,Bi,Ci). But rather the events from the three streams will be inserted to the new stream independently in the order they arrive. This is shown in Figure 2A as `Combine_Stream_indianaRadar`. Althose user need not worry about the under-

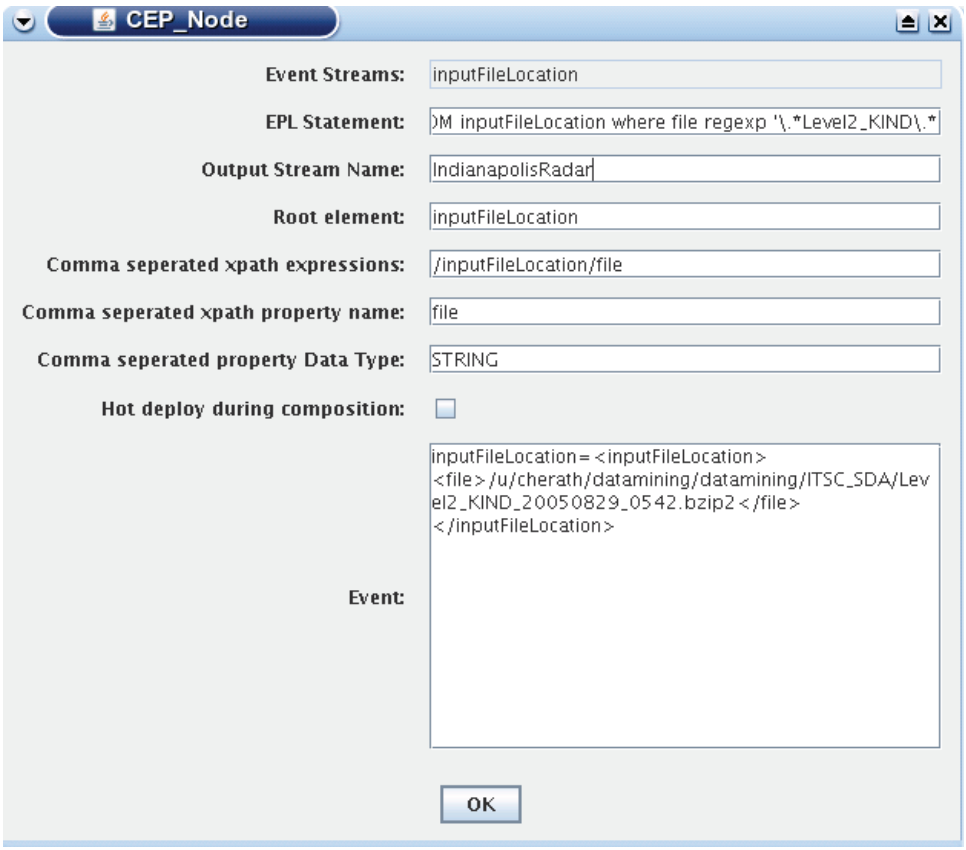

Figure 30: Configure a CEP node with data binding to the file element in the event and query that does a regular expression match against the defined property name

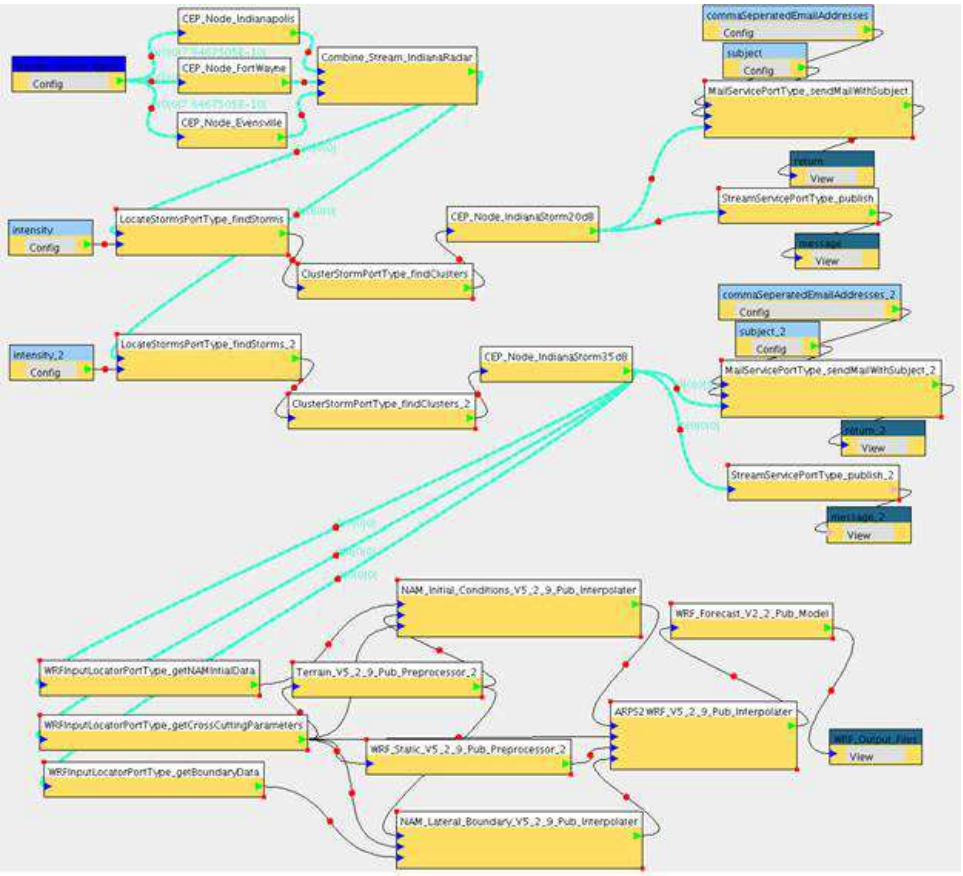

Figure 31: Weather Streamflow that does event filtering for the Indiana state and running storm detection workflow against that stream and running a scientific workflow against that stream

lying characteristics

Once IndianaRadar Stream is in place next step is to identify the possible storms in this stream. This is done by running a Storm Detection Algorithm published as a web service `LocateStormsPortType_findStorm` that identifies locations with high radar intensity. Once these high intensity radar locations are identified it will be sent to a clustering algorithm service named `ClusterStormPortType_cluster`, that would try to identify storm clusters and Clustering Algorithm that would cluster the possible storm locations together to identify a possible storms. This can be viewed as a simple sequential workflow. Looking at the resource consumption for each of these services and the rates that it will have to deal with it is well suited for low resource consuming yet high throughput, thus would fit in the operating range of XBaya Engine. This is further confirmed by the dynamic queue monitoring interface provided by XBaya shown in Figure 33. The computational requirement for such a workflow is less than 20 seconds in a 2.4 GHz Linux node. The IndianaRadar stream will be fed into two such workflows that would look for two different intensities 20 dB representing light rain and 35dB representing severe weather conditions. These two web service nodes are one to one nodes thus its control flow can be clearly identified and during the partitioning these two nodes will be labeled into one workflow.

The expectation of the storm identifier is to identify storm and it will return an array of possible storm locations. Most of the times there will be no storms and this workflow would return an empty array. It is necessary to filter out those event that do not have storms. The predicate whether there are storm coordinates in the cluster output will be a filtering predicate. The output from the cluster output would be of the form shown below.

To filter out event that do not have storms it is necessary to define a property that we will call numberOfStorms and the xpath expression that will be data bound to this property will be count(/Cordinates/Cordinate). The XPath expression will simply count the number of Cordinate elements in the event where each event represent a possible storm location. This filter nodes is shown in Figure 31 as

```
<Cordinates>
  <Cordinate>
    <lat>-89.827118</lat>
    <lon>30.337002</lon>
  </Cordinate>
  <Cordinate>
    <lat>-88.481018</lat>
    <lon>29.529257</lon>
  </Cordinate>
   ...
  <kml>http://pagodatree.cs.indina.edu:8081/kmls/kml-893028539834.kml</kml>
</Cordinates>
```

Figure 32: Possible Storm Locater responce

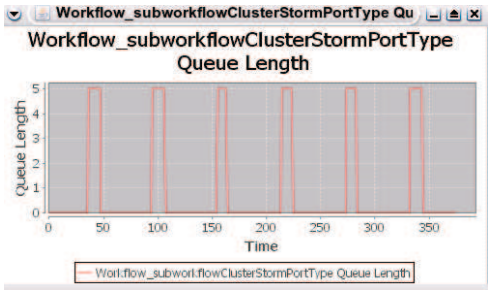

Figure 33: Dynamic Queue Monitoring for ClusterStorm workflow

`CEP_NODE_IndianaStorm20dB` and `CEP_NODE_IndianaStorm35dB` respectively for possible storms of different intensities. Figure 34 shows the visualization of the kml file using Google Earth showing one storm location.

There are two reactions to these two streams. Events in the stream where it shows intensity over 35dB the experiment not only issues a email alert but also will launch a high resolution WRF forecast. For possible storm over 20dB we will only issue an email alert. Since this is a long running forecast during to composition we will force the WRF forecast workflow to run in BPEL environment. Once the selected labeling is done user can deploy the Streamflow at which point it will be partitioned to different graph components. Once it starts running it will send out notifications showing the current execution. The output can be visualized using tools like Grads [31] or Integrated Data Viewer [90].

The other approached is to publish the IndianaStorm35dB stream back into

Figure 34: Google Earth visualization of the kml that is generated by the storm detection algorithm

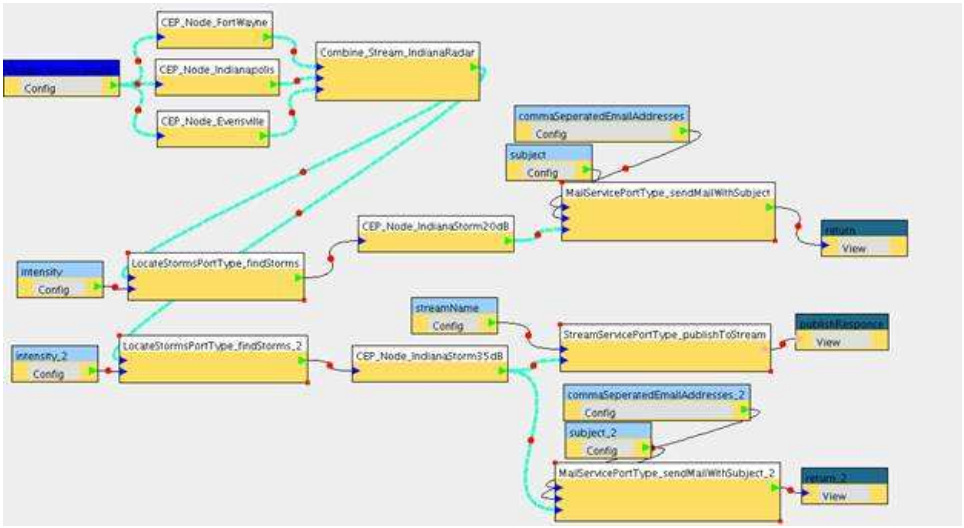

Figure 35: Phase 1 and Phase 2 of weather Streamflow where event filtering storm detection and notifications are done

the CEP Engine as shown in Figure 35 and the high resolution WRF forecast can be coupled with that stream in a separate Streamflow as shown in Figure 36.

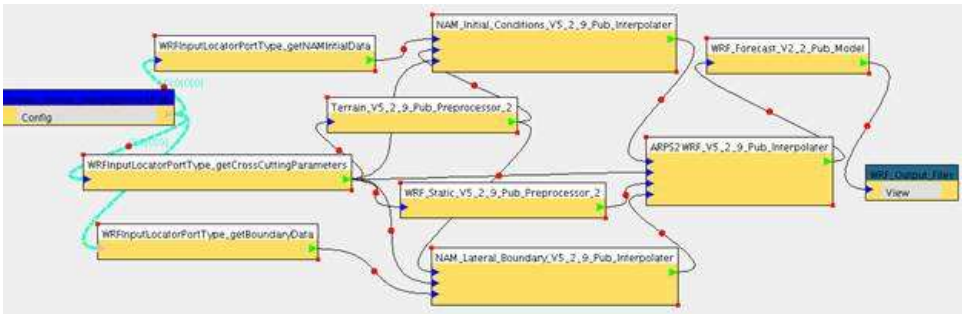

Figure 36: Phase 3 of weather Streamflow where ther previously published stream is used to launch a scientific workflow

5 Formal model for Stream semantics

The programming model proposed in the earlier section integrates two programming paradigms, event stream processing and scientific workflow systems, defining Streamflow semantics and a framework that allows a user to compose different Streamflows and have the framework compile it into runnable system. The resource requirements of the various components in a particular configuration may differ from one another. The ability to compose any Streamflow and compile them into a running event processing system does not mean the system is able to sustain the event rates through the system given the resource requirement. So it is important to analyze whether a given Streamflow is able to handle a given input event rate given the available resources. The two paradigms have different resource requirements and flow rates. Scientific workflows are long running with high resource requirements whereas the stream processing tend to have high flow rates and low resource requirements. The schedulability of the two systems are quite different and there have been studies to evaluate resource requirements of the two programming paradigms independently [89] [124] [119]. The schedulability of the unified programming model introduces new challenges.

Since the long running complex scientific workflows require significant resources, and given the limited computing resource available for scientific workflows, there is a rate at which the workflows can be launched without running out of resources and queuing up jobs indefinitely which will eventually crash the sys-

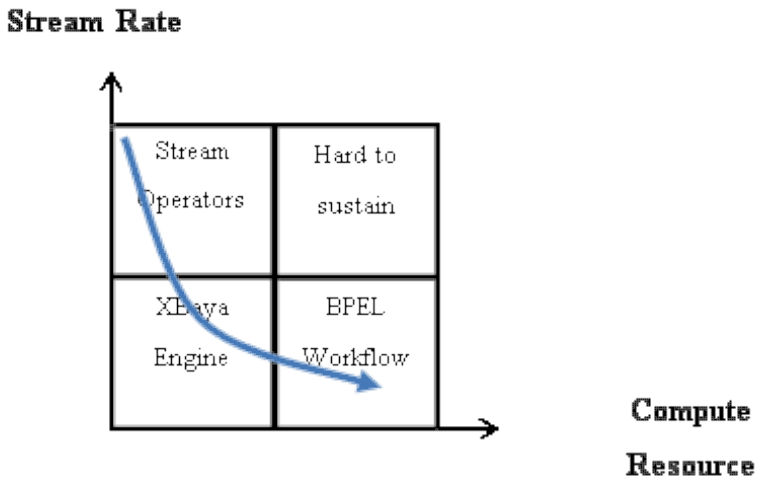

Figure 37: Resource consumption and event rate showing different operating areas that the sub-graphs of the Streamflow is classified to

tem. So a balance needs to be achieved that minimizes compute intensive tasks in combination with stream processing to reduce the event flow rate to an acceptable level that will not exhaust available resources. Experimental evaluation of such system will be discussed later in this section.

From the perspective of trying to understand how a given Streamflow can be compiled in to a runnable framework and when trying to analyze a given such setup can sustain over the event rates that the Streamflow would have to support there are two key aspects that require close attention. They are availability of compute resources for each node in the Streamflow and the different event rates that particular node required to support. Figure 37 shows how a compute resource requirement and event rate affect the computability of a running Streamflow. For simplicity we consider high and low quantities of each of these aspect and that would produce four combinations shown in the following listing.

- Low resource consuming high event rates - Event processing systems provide in-memory stream operators such as selects, aggregate, join and pattern matching and these operators can support high event rates from few thousand events per second to in some cases tens of thousands of events

per second [121].Significant portion of such continuous queries that occur in practice can be computed using a bounded memory [8] thus would fit well into the low resource consuming category that can support very high event rates. The computability of stream processing systems will in many cases not restricted the primitives mentioned earlier because of the extensibilities they provide to plug in user defined functionality.

- Low resource consuming low event rates - ere are certain activities that require more processing than is provided by traditional query operators. For instance activities need access to scientific libraries or may require embedded processing such as data mining an incoming event. In these cases service component is introduced that encapsulates functionality for inclusion in Streamflow The use cases that fall into this category will have a upper bound imposed upon the event rates supportable by the web service engines at the least.

- High resource consuming low event rates - The type of jobs falling within this category are the multiprocessor long running jobs that require high performance compute resources. The WRF model is a good example of such long running high resource consuming jobs that may take an hour to run a 24 hour forecast over a 800kmx800km at 5km resolution with 512 processors. Even if a particular system has access to Teragrid supercomputing resources it is not sustainable to launch such a job each second, each minute or each five minutes.

- High resource consuming high event rates - From a practical point of view this is unsustainable because it is hard to meet the resource requirements.

By closely observing the classification mentioned above leads us to the conclusion that the operational semantics of Streamflows need to be based on the event rates and the resource needs. Figure 38 shows a fully spectrum Streamflow from operational point of view where different stages of the stream processing will require different sub systems of the Streamflow framework to be active and brought

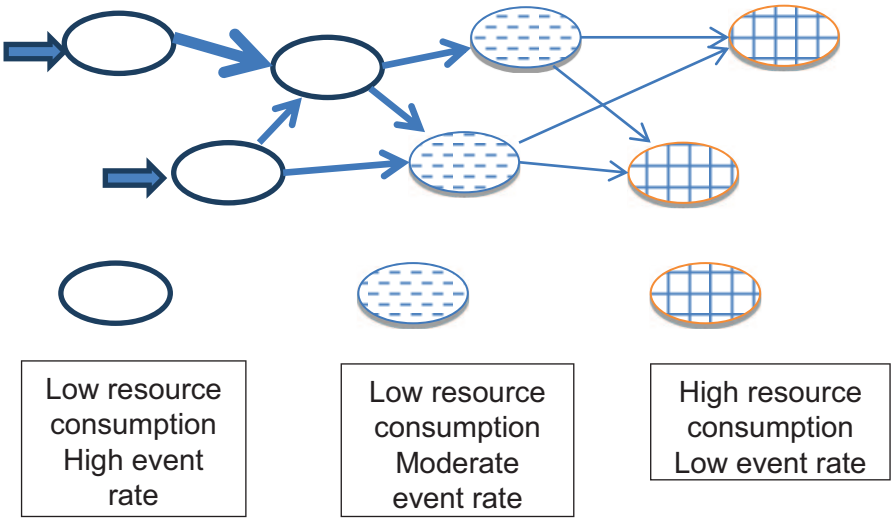

Figure 38: Operational sub-graphs of a Streamflow based on different event rates

to bear. The canonical use case is where the Streamflow will have inputs with high event rates and the first phase of the Streamflow would utilize Stream Operators such as selections, joins, aggregations and pattern matching to reduce the event rates to a manageable level for high throughput workflows without losing the interesting event in the process. Once the event rates are at a rate that is acceptable for the high throughput web service based workflows the XBaya workflow engine can handle the event rates. The activities at this stage are able to make much more fine grain analysis because these activities can launch low resource consuming applications. But this stage assumes that the activities resources are already allocated and when launched the job need not sit in a queue of a supercomputing resource. These kind of jobs could run in a dedicated machine or a cloud computing resource. Such activities would further prune the event rates to a level, if necessary to launch long running scientific workflows to further the findings. The long running scientific workflow would normally mean running scientific applications on supercomputing resources and the rate at which launching such workflows can be sustained is relatively low.

This thesis proposal propose to define semantics that would not only formalize

the workflow semantics that were presented in chapter 4 but also to propose a formal model that is the basis for calculating the schedulability of a given Streamflow setup given the flow rate are the resource availability. An example Streamflow that make use of the strengths of the semantics is shown in Figure 38 demonstrating the reduction of events rates to a level that can promote meaningful scientific experiments with available resources.

5.1 Abstract Syntax

Figure 39 presents the abstract syntax production rules for the Streamflows which is defined by a set of edges connecting two nodes. This constitutes a complete description of the Streamflow because all the Streamflow graphs are connected graphs. The grammar rules are defined such that the part of the Streamflow could be conventional scientific workflows and there are transition edges that would change the streaming nature of the dataflow in the edges. Because the StreamflowEdge is between the nodes that support the streaming dataflow nature and the rules are such that a TransitionEdge may sit in the boundary of scientific workflow and Strict Streamflows and once the boundary is crossed all the subsequent downstream Edges are StreamflowEdges and all the downstream nodes belong to the StreamingNode node set. The StreamingNode node set is the building block of the Streamflows and they are designed to handle the streaming semantics which will be discussed in detail in the Operational Semantics section.

5.2 Model of Computation (MOC)

There is a need for clearly identifying the semantics of each individual node that is introduced in the Streamflow framework without ambiguity. The Actor model [58] offer a widely accepted formal model that defines semantics of processing components in a distributed system. Actors are abstractions in a distributed computing environment that encapsulate computations that have input parameters and output parameters. Each of their input parameters are received from external messages known as tokens and the Actors expose input ports to receive these

Streamflow :: StrictWorkflow | StrictStreamflow | TransitionEdges
TransitionEdges :: ϵ | TransitionEdges, TransitionEdge
StrictWorkflow :: ϵ | StrictWorkflow , WorkflowEdge
StrictStreamflow :: ϵ | StrictStreamflow , StreamflowEdge
TransitionEdge :: (WorkflowNode , port) $\longrightarrow$ (StreamGeneratorNode , port)
WorkflowEdge :: (WorkflowNode , port) $\longrightarrow$ (WorkflowNode , port)
StreamflowEdge :: (StreamingNode , port) $\underrightarrow{stream}$ (StreamingNode , port) |
(StreamingNode , port) $\underrightarrow{stream}$(StreamSinkNode , port) |
(StreamingGeneratorNode , port) $\underrightarrow{stream}$(StreamingNode , port)
WorkflowNode :: WebserviceNode | IfElseNode | ReceiverNode
StreamingNode :: StreamflowNode | FilterNode | StreamJoinNode |
AggregatorNode | CEPNode

Figure 39: Streamflow grammer

tokens. The result produced by Actors computation will be sent out as a token from its output port to other Actors. In this thesis the Actors are considered not to have side-effects on the global system and some actor definitions may carry internal state. The firing mechanism of the Actors based on the availability of the input tokens makes It a appropriate model to be considered for Streamflow and Simple Actor Language (SAL) [2].Further there are known Models of Computations (MOC) built on top of the Actor model that can be used to identify how the actors are orchestrated and the interaction between them can be managed.

The Model of Computation defines how the individual components in a distributed computing graph may interact with each other. There will be few Models of Computations that will be considered to define the interaction between the actors and they are based on the MOCs defined by Goderis et al in [50]. The models of computations that will be considered are Synchronous Dataflow (SDF), Dynamic Dataflow (DDF), Process Networks (PN), Finite State Machines (FSM) /Modal Models. The SDF is a MOC where the firing schedule of the Actor graph is statically decided and its different from DDF because DDF has a dynamically scheduled Actor firing schedule. Process network is a asynchronous MOC where each Actor runs in its own thread. FSM/Modal model is different from actors because these Actors maintain state and Modal model defines the MOC that updates the state of the Actor and firing of the Actor is based on the inter state.

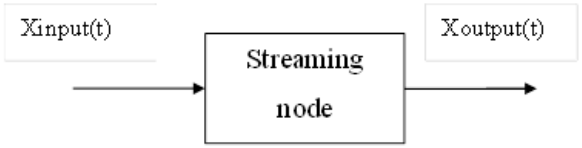

Figure 40: Input stream Vs Output stream of a streaming node

5.3 Operational semantics

For the purpose of modeling a data stream we define a standard inter arrival rate, made arbitrary small to ensure that no two events occur within the same time interval. Th following random variable X(t) is defined such that:

$$z(t) = \begin{cases} 0 & \text{if event did not occur at time t} \\ 1 & \text{if event occured at time t} \end{cases}$$

An example sequence may be defined ($1_{t=0}$, $0_{t=1}$, $1_{t=2}$, $0_{t=3}$, $1_{t=4}$, $0_{t=5}$, ...) when an event exists at time o but not at time 1 and so on. Although this discrete time sampling may appear to be too coarse grain, the discretization could be made arbitrarily small. The event sequence above implies that an event arrives at every alternate timestep. This notation is useful when considering how a particular node as shown in Figure 40 may alter a given input event sequence. This notation will be referred to during the evaluation of event queues later in this section.

Some Streaming nodes behave as true mathematical functions and for each individual event in the input stream, one and only one output event is produced. Some nodes may filter out event thus they may behave as mathematical partial functions and produce an output event for some input events. There are other possibilities where the node may produce two output events for each input event. It is important to understand this relationship and we define a quantity Cardinality of a node defined as shown in equation 1. Cardinality of a node is the ratio between

the output event rate and input event rate of a given node. The limit $n \rightarrow \infty$ is necessary because there are event delays and computational delays that may produce local deviations thus producing inaccurate readings.

$$Cardinality = \lim_{n \rightarrow \infty} \frac{\sum_{t=0}^{n} X_{output}(t)}{\sum_{t=0}^{n} X_{input}(t)} \tag{1}$$

Following is an example of a Filtering node cardinality that filters two out of three events: Input sequence: $\mathrm{X}_{input}(\mathrm{t}) = (1_{t=0}, 1_{t=1}, 1_{t=2}, 1_{t=3}, 1_{t=4}, 1_{t=5}, ...)$ Output Sequence:$\mathrm{X}_{output}(\mathrm{t}) = (1_{t=0}, 0_{t=1}, 0_{t=2}, 1_{t=3}, 0_{t=4}, 0_{t=5}, ...)$)

$$Cardinality = \lim_{n \rightarrow \infty} \frac{\sum_{t=0}^{n} X_{output}(t)}{\sum_{t=0}^{n} X_{input}(t)}$$

$$Cardinality = \lim_{n \rightarrow \infty} \frac{\sum_{t=0}^{n} 1/3}{\sum_{t=0}^{n} 1}$$

$$Cardinality = \lim_{n \rightarrow \infty} \frac{n/3}{n}$$

$$Cardinality = \lim_{n \rightarrow \infty} \frac{1}{3}$$

So the cardinality of this node is 1/3 meaning that the output event rate of the node is one third of the input event rate, over a long period of time. It is useful to identify the cardinalities of the different nodes available in the Streamflow programming model. Figure 41 shows the input event stream and respective output streams of different Streamflow nodes and it is important to note that cardinality is quantitative analysis of how a given event stream is affected when it is sent through a given Streamflow node.

- Streamflow node cardinality = 1
- Filter node cardinality = m/n < 1
- Event aggregator cardinality = 1/aggregation factor < 1.

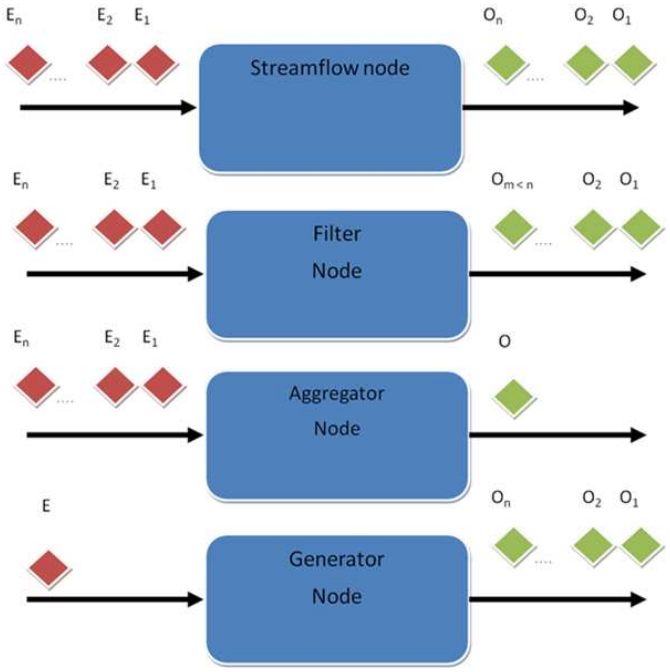

Figure 41: Stream cardinality of different nodes

- CEP node cardinality depends on the particular query

In the subsequent sections we identify the operational semantics of some important programming nodes available in Streamflow programming model.

5.3.1 Streamflow Node

The Streamflow node is implemented in the BPEL engine and the XBaya Workflow engine. Streamflow node is always associated with an underlying web service that is invoked for each event that arrives at the Streamflow node. Streamflow node exhibits a cardinality of 1 where it behaves as a pure mathematical function and thus every output event's provenance can be traced backed to a one and only one event from the input event stream. So in essence Streamflow Node is a receive-invoke loop with a web service associated with it. That is to say it is a never ending control structure which would be used to wait for an incoming event and at an arrival of such an event it would invoke some web service/component and loop back and wait for another event to arrive. The nodes exhibit push behavior

```
1  def streamflow-node (m1,m)
2        [sender , arg]
3    let k = f(arg)
4    {send [k] to m}
5    send ready to m1
6    become streamflow-node (m1,m2)
7  end def
8
9  //m1  input port
10 //m2 - output port
```

Figure 42: SAL Actor definition for Streamflow node

```
1 <bpelns:while>
2   <bpelns:condition/>
3   <bpelns:receive/>
4   <bpelns:assign/>
5   <bpelns:invoke/>
6 </bpelns:while>
```

Figure 43: BPEL structure for Streamflow node

where the events will be pushed to the workflow process instead of a pull model. This control structure will be reused again and again in the rest of the streaming nodes in the Streamflow framework.Figure 42 shows the definition of Streamflow node written using SAL language used in Actor theory. The model of computation that is required to execute a Stremflow node can be Synchronous Dataflow.

The BPEL implementation uses existing BPEL semantics (Figure 43)to achieve the repetitive nature of the stream processing where the BPEL process has to wait till the arrival of an event and then it may invoke another web service which could be a web service or another sub workflow.

Streamflow node is a defined as a loop activity that contains a receive activity followed by a invoke activity. The receive activity receives an event from the time series and invokes some service with the incoming event as the argument.

BPEL correlation - During the execution of above BPEL construct it is necessary for a particular even to be delivered to the right running instance of the workflow. This achieved using BPEL correlation where a set of domain specific variable set is defined and the values of these variables at the creation of the BPEL

```
1  Def filter (m1,m) [sender , arg]
2    let p = predicate(m1)
3      if p then {send [m1] to m}
4    send ready to m1
5    become filter (m1)
6  end def
7
8  //m1  input port
9  //m  output port
```

Figure 44: SAL Actor definition for Filter node

process is matched against the variable values of the incoming messages to do the correlation of incoming events with the proper BPEL process.

XBaya workflow engine supports similar constructs that support the pipeline behavior that allows the Streamflow node semantics.

5.3.2 Filter Node

Filter node exhibits operational semantics similar to that of the Streamflow node when it comes to the repetitive nature of the execution driven by the incoming events. The Filter Node has a cardinality <= 1 and it exhibits the properties of a partial function. There is a predicate associated with a given Filter Node and a continuation which is mostly a web service. Whether a given input event is be passed on to the web service depends on whether the predicate returns true for that particular event. Binding of a web service with the node is optional. Figure 44 shows the Filter node definition using SAL Actor language. Model of Computation that is used for execution of Filter Nodes is Process Networks.

The association of a web service component within the filter node is optional and is put in place as a convenience API, Figure 45 provides a view of internal flow control of a Filter node where the input event sequence is filtered out based on the predicate outcome. Filter node without the web service components is more fundamental and important than the other because it provides a means for the much needed event rate reduction down to a manageable level.

The predicate is compiled using the functionalities of the Complex Event Pro-

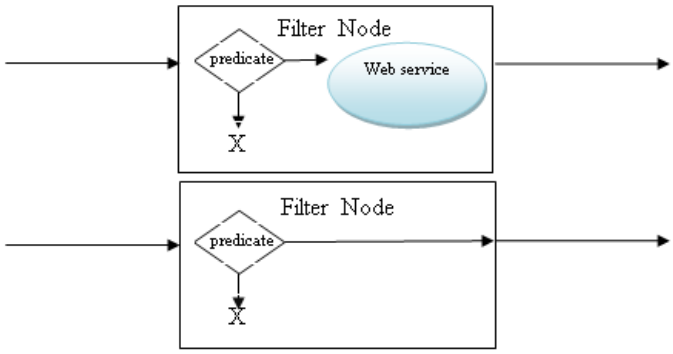

Figure 45: Filter node operational semantics, filter node without web service for mere filtering

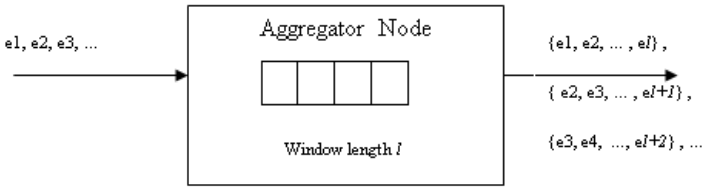

Figure 46: Aggregate node of batch length l aggregating events in the stream to events of length l

cessing system. Complex event processing systems allow predicate constructs using SQL where clauses can refer to the properties of the event. For example if a particular event E consist properties xi then a CEP query can be written to define the predicate as follows where predicateFunction(x1, x2, .. xn) is predicate function that refers to the event properties during its predicate calculation.

Select * From E Where predicateFunction(x1, x2, ..xn);

5.3.3 Event Aggregator node

The necessity for events to be bundled together to produce composite events or produce event batch based on a sliding window maintained on a continuous event stream, is useful for many applications for staging datasets for scientific experiments.

Aggregation is realized using the sliding window Complex event processing

```
1  Def aggregate-1 (m1,q, m) [sender  arg]
2    q.remove()
3    q.add(arg)
4    {send [content(q)] m}
5    become aggregare-1(m1,q,m)
6      end def
7  Def  aggregate-2 (m1,q, m) [sender , arg]
8    q.add(arg)
9    become aggregare-1(m1,q,m)
10 end def
```

Figure 47: SAL Actor definition for Aggregator node

semantics as shown in Figure 46. Aggregation of an even stream of E events into batches of l events can be realized using a CEP query of the following form. Figure 47 shows the definition of Aggregator node using SAL Actor language. Singe aggregator need to maintain the state of the window it requires Model of Computation FSM/Modal model for its execution.

Select * From E.win:length_batch(l)

5.3.4 Merge Node

Merge Node merges one or more streams to one output stream. The event ordering is decided by the arrival timestamp of the event at the merge node. Merge node with n input streams names I1, I2... In will yield following CEP queries during compilations given its output stream name is MergeOut. Figure 48 shows the SAL Actor definition of a merge node. The Model of Computation required for execution of Merge Node is Process Network.

Insert INTO MergeOut Select * From I1

Insert INTO MergeOut Select * From I2

...

Insert INTO MergeOut Select * From In

5.3.5 Join Node

Join Node joins one or more streams to one output stream of composite events. The trigger mechanism is often synchronised by one of the input stream. This is

```
1  Def n-merge (m1, ,mn, m) [sender, arg]
2      {send [arg] to m}
3    if sender = mi
4      then send ready to mi fi
5       become n-merge(m1, ,mn, m)
6  end def
```

Figure 48: SAL Actor definition for Merge node

```
1  def two-inputs-needed (m1,m2,m) [sender, arg]
2    if sender = m1
3      then become one-input-needed (m1,m2,second,arg)
4      else become one-input-needed (m1,m2,first,arg)
5    fi end def
6  def one-input-needed (m1,m2,m,new-arg-position, old-arg) [sender , new-arg]
7    let k = (if new-arg-positio = second then f(old-arg , new-arg) else f(new-arg , old
8      {send [k] to m}
9    send ready to m1
10   send ready to m2
11   become two-inputs-needed (m1,m2)
12 end def
```

Figure 49: SAL Actor definition for Merge node

done used the "unidirectional" keyword in CEP language. Join node with n input streams names I1, I2... In will yield following CEP queries during compilations given its output stream name is JoinOut with output stream synchronized with I2. Figure 49 shows the SAL Actor definition of a merge node. The Model of Computation required for execution of Join Node is Sybchronous Dataflow.

Insert INTO JoiOut Select * From I1, unidirectional I2, ... In

5.3.6 CEP Node

CEP Node is a generic node that capture any kind of Complex Event Processing query that one may come up with. It is difficult to make any kind of assessment of the cardinality of this node in a quantitative way because the query may result in a spectrum of output stream behaviors (Figure 50).

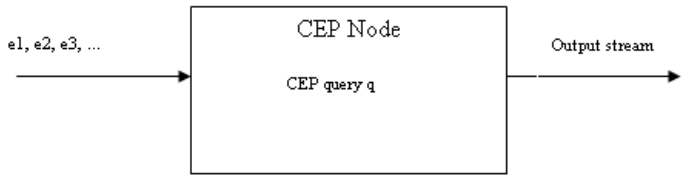

Figure 50: CEP node with query q

5.4 Evaluation

The evaluation of the Streamflow presented in this section focuses on the compile time, deployment time and runtime performance of the Streamflow framework. Most of them are organized as micro-benchmark evaluations of important components of the framework.

5.4.1 Throughput

One of the motivating reasons behind the graph partitioning algorithm is to identify the sub-graphs of different runtime characteristics and deploy them to the different runtimes. The reasoning behind the selection of the right runtime is based on the nature of the activities of the Streamflow and the event rates the sub-graph will serve. The nature of the activities is largely qualitative and hard to quantify, although a metric is proposed later in the section to partially quantify it in terms of resource consumption. The other aspect, event rate, is measured in this section to identify the operating range of the different runtime engine, BPEL, XBaya Engine or CEP Engine. Figure 51 plots the behavior of the event latency with the event throughput. The throughput is measured in the setup shown in Figure 53 where the Streamflow1, Streamflow2, Streamflow3 is deployed to measure the latencies of BPEL Engine, XBaya Engine and CEP Engine respectively.

Streamflow1 is defined as:

CEP Node1 $\underrightarrow{stream}$ Workflow1 Workflow1 is workflow with single web service node WS Node1

Streamflow2 is defined as:

CEP Node2 $\underrightarrow{stream}$ Workflow2 Workflow2 is workflow with single web service node WS Node2

Streamflow3 is defined as:

CEP Node3 $\underrightarrow{stream}$ CEP Node 4

The latency measurement in micro benchmarks are (T2-T1), (T4-T3) and (T6-T5) as shown in Figure 53 for BPEL Engine, XBaya Engine and CEP Engine respectively. These measurements were done in Environment2.

Figure 53 shows CEP engine operates in maximum throughput of 1114 events per second and BPEL engine's message correlation system cannot sustain Event rates beyond 2 events per second. XBaya engine has an operating range in between the other two systems. This falls within the thesis premise presented earlier in graph partitioning. Thus the sub-graphs of different event rates should be deployed into different runtime engines depending on the rates that the sub-graphs are expected to operate.

It should be noted that the BPEL engine have high workflow launch rate than 2 events per second, but here the measurement is driven by the ability of the BPEL process to correlate and sustain event into a receive-invoke-loop activity for the CEP Node in BPEL.

This latency throught graph provides different insight to why a Continuous Time Model of Computation defined by Goderis et al [50] will not be a suitable model for Streamflows and why applications that exhibit Continuous Time Model of Computation will not suit the Streamflow model. Given a task in a Streamflow node that takes time T to compute. This T is purely the compute time for the computation associated with the node. In a workflow it would be a time taked forrunning the services, for a CEP it would mean time taken for evaluation of the stream operator. Assume this Streamflow node is connected to an input stream. The model of computation implies the nodes computation will be triggered for each event in the stream. Figure 53 shows the latencies of each of the execution runtimes and given the latencies it is possible to analyze the how the overall efficiency of the computation will behave. Efficiency can be defined by:

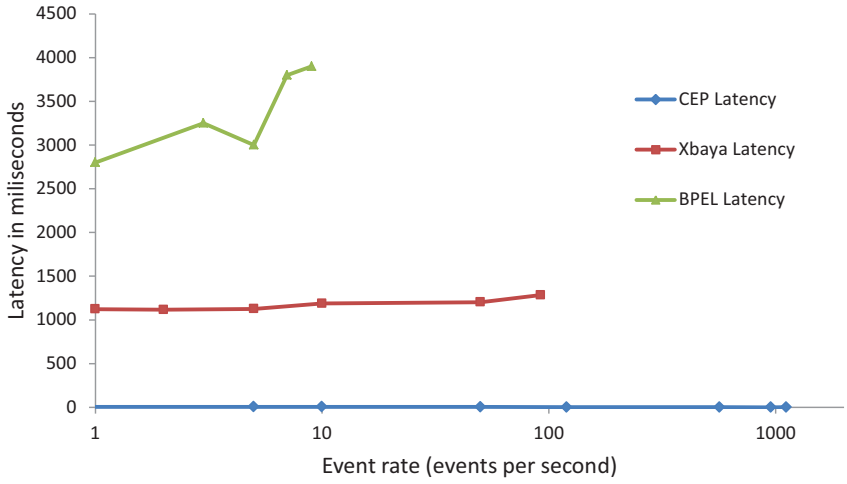

Figure 51: Throughput Vs Latency in different Streamflow runtimes showing the Operating areas of CEP, XBaya and BPEL

$$NodeComputaionalEfficiency = \frac{NodeComputetime}{NodeComputetime + Frameworklatency}$$

Figure 52 shows the plot of efficiency variation with node compute time for three different runtimes. This shows the minimum compute time the node need to send in its computation for it to achieve at least 90 percent efficiency. Higher latency runtime, BPEL would require the node to have minimum 20 seconds to achieve 90 percent efficiency. On the other hand CEP engine would reach 90 percent efficiency with 50 miliseconds. This allows the users to pick the right target runtime depending on the computation that they are trying to compose.

Figure 54 and Figure 55 shows the saturation of event latencies with the event rates for CEP Engine and XBaya Engine repectiely.

5.4.2 CEP Engine Web Service Interface

The CEP engine introduced earlier in the section provides a Web Service API for event sources to publish events to the system so they could participate in

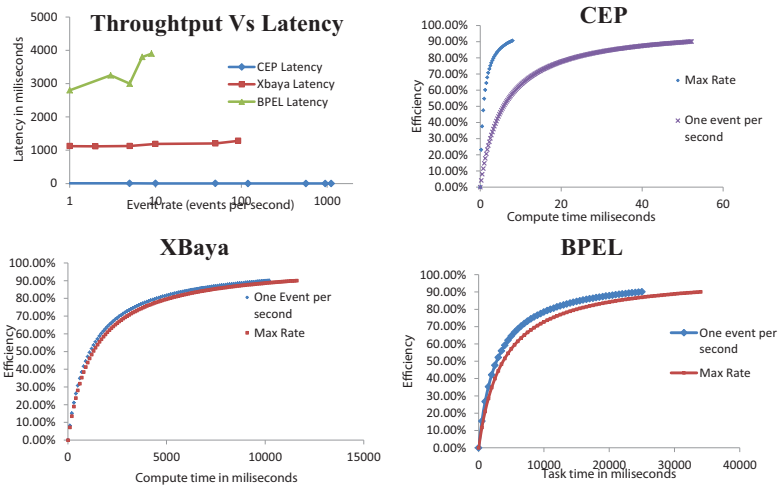

Figure 52: Synthetic Computational Efficiency given the processing time per event for different Streamflow runtimes.

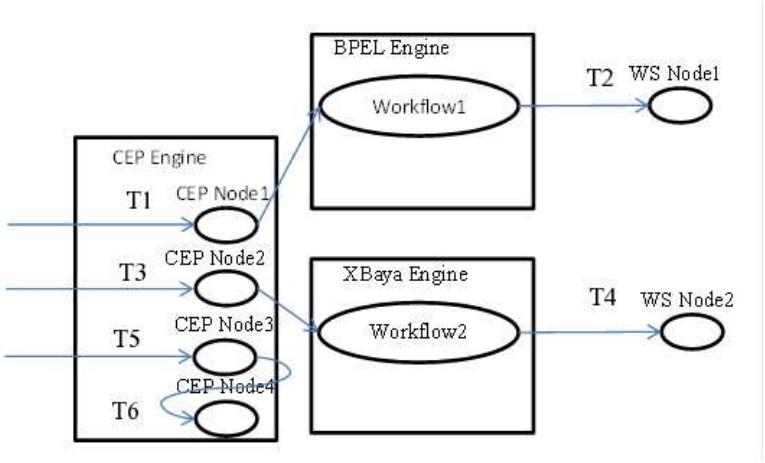

Figure 53: Throughput Measurement setup where one streamflow consist of CEP node and BPEL workflow node, another consist of CEP node and XBaya workflow node and another consist of two CEP nodes

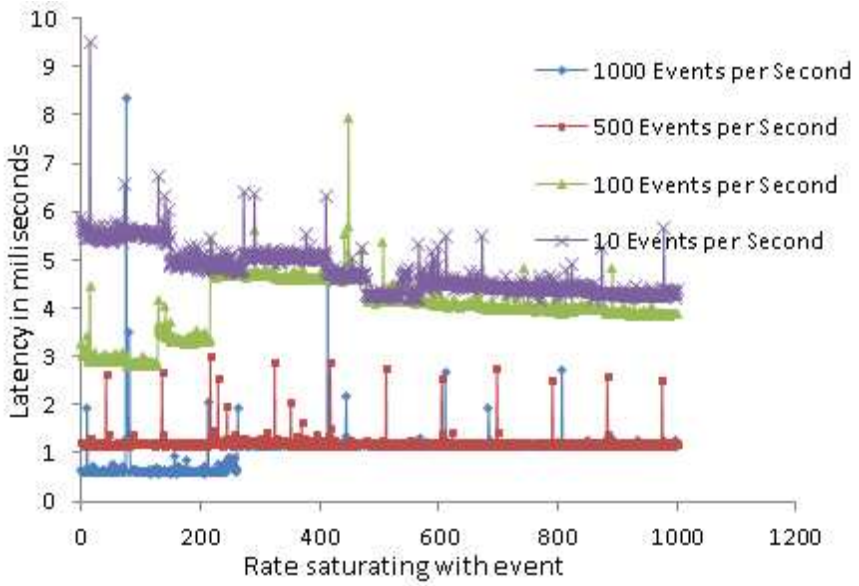

Figure 54: The latency between nodes in the CEP engine is measured as Latency Vs Throughput

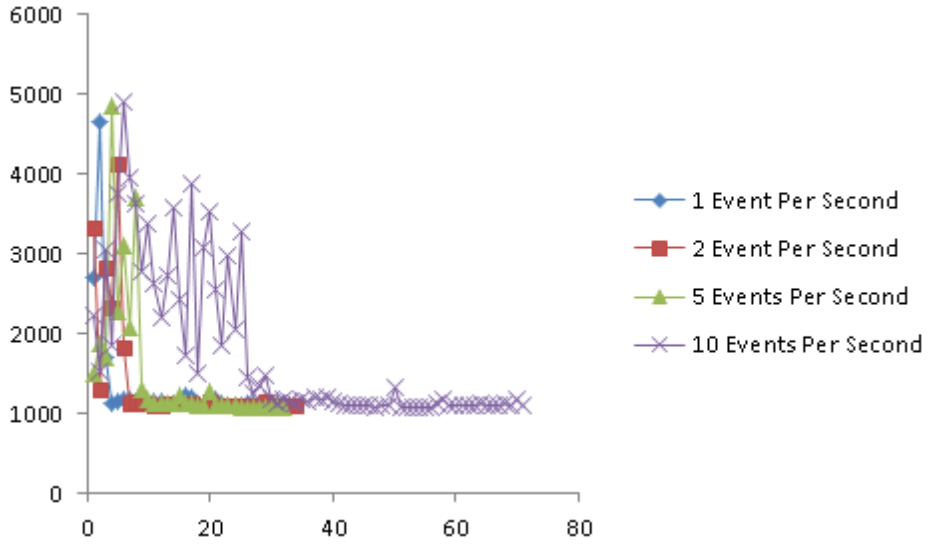

Figure 55: The Latency between a CEP node and XBaya workflow node measured as Latency Vs Throughput

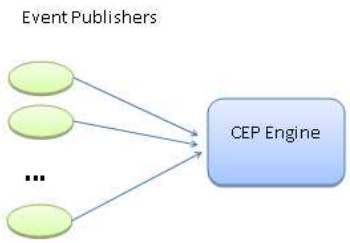

Figure 56: CEP Engine performance setup for multi threaded event publish perfromance on the CEP engine web service interface

the larger Streamflow execution framework. The experimental setup is shown in Figure 56 where multiple event sources are publishing events to the CEP engine in an attempt to find the external event publishing throughput.

Figure 57 shows the performance evaluation done with using up to 128 nodes and the publish rate saturates just over 2000 messages per second. The x-axis shows the number of parallel publishers for the CEP Engine and for a given number of such clients the total sustainable throughput at the CEP engine is measured. The y-axis shows the median messages per second for a given number of parallel publishers. System shows degradation of the performance after that but also shows a throttling effect on the publishing clients so that the system would operate in much lower message rate yet continue to show healthy operation at the CEP Engine.

5.4.3 Graph Partitioning

This section focuses on quantifying the costs associated with the graph partitioning algorithm presented in the previous section. It should be noted, measured cost included both labeling as well as partitioning yet it does not include the deployment time. The graph partitioning is largely dependent on the complexity of the graph. The time complexity of the graph partitioning algorithms are calculated

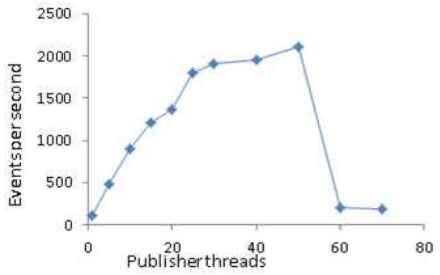

Figure 57: CEP Engine external event publishing performance measured as throughput as the publishing threads are increased

to be Θ(V+E) where V and E and Vertices and Edges of the graph. This section evaluates a linear graph with single edged connections and intree graph with double edged connections.

The analysis of time complexity of the graph partitioning algorithm is relatively straight forward. Since the graph partitioning algorithm make use of the labeling algorithm it is convenient to analyze the time complexity of the graph partitioning algorithm. Since it is an iterative algorithm, the key to unlocking the time complexity of the labeling algorithm is to understand the queue invariant in the iteration. The initialization in lines 7 to 10 and lines 12 to 15 are bounded by the input nodes so it may be absorbed to any Θ(V). The iteration in line 17 uses a queue which has a invariant, that is all the nodes in the queue are ready to be labeled and a node will be added to the queue only once. Once the node is added to the queue it will read all its outgoing edged to identify possible candidates for the queue. This process will read all the out edges from a given node thus the entire iteration will have Θ(V+E) complexity.

Graph Partition algorithm on the other hand is much more straight forward for complexity analysis. The toplogical sorting and labelGrap methods exhibit ΘV+E) individually and loop in line 1-11 exhibits Θ(V) . The partiotionMap will have at most Θ(V) keys and it will at most read the entire graph and edges which yield Θ(V+E). Thus overall time complexity still adds up to Θ(V+E)

Figure 59 shows the behavior of graph partitioning as the number of nodes grows in the two graph structures. It shows the linear behavior as anticipated.

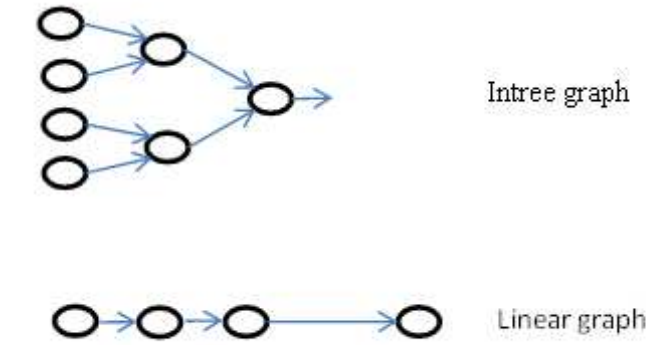

Figure 58: Graph structures used for partitioning performance measurement. One is a sequential workflow and other is a intree graph workflow

The two graphs one linear graph and the other Intree (shown in Figure 58) , both with the same number of nodes, show different compilation times, because Intree graph has higher edge density than the linear graph.

5.4.4 Deployment

The deployment of Streamflow involve in compiling the Streamflows into target runtime and calling the deployment APIs of the target runtime. This section quantifies how the deployment of a given graph would vary based on the size of the Streamflow. It should be noted the structure of the graph has a significant bearing on the compilation time of the Streamflow. This is captured by the graph partitioning evaluation and this section merely focuses on the deployment aspect of the Streamflow. The Streamflow WF1 that is used to measure deployment time is defined as:.

S1 $\longrightarrow$ S2 $\longrightarrow$... Sn-1 $\longrightarrow$Sn

This is a sequential workflow of n nodes and one of the parameters of the measurement is to vary the number of nodes and to see how the deployment time is affected by the variation. The partition algorithm would partition this Streamflow as shown in Figure 12 c) with a Stream Generator SG and an Streamflow node Active-WF1 which would dispatch events to the scientific workflow WF1. Finally

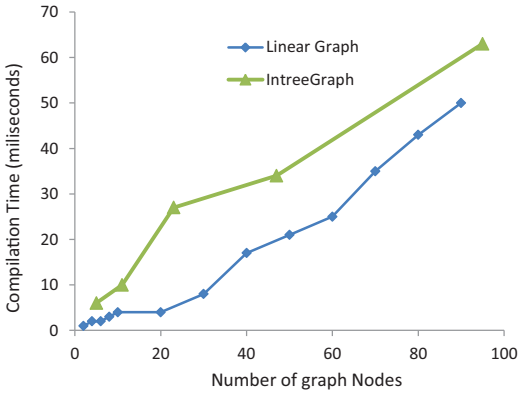

Figure 59: Graph Partitioning time Vs the size of the graph. This include the partitioning time as well as the labeling algorithm time. The linear bahavior is inline with the time complexity O(V+E)

the resulting output event stream from the Streamflow node is sent to a Stream-Sink. The control Streamflow would look like what is shown in Figure 12. There will be extra deployment overhead for generating and deploying this Streamflow that could be defined as follows.

SG $\underrightarrow{stream}$ Active-WF1

Active-WF1 $\underrightarrow{stream}$ Stream-Sink

Figure 60 shows the cost of deployment of the Stramflow by getting using median statistical measurements. It also tries to isolate the overhead added by the Streamflow framework itself by comparing it with a deployment time of a conventional scientific workflow to the same system. The deployment time of the Streamflow increases linearly with the number of nodes in the Streamflow graph but the micro benchmark measurement of overhead introduced by the Streamflow framework itself remain constant as anticipated. It can be concluded that the deployment overhead of the Streamflow remain less than one second which is an acceptable measurement because this is triggered by an interactive user action.

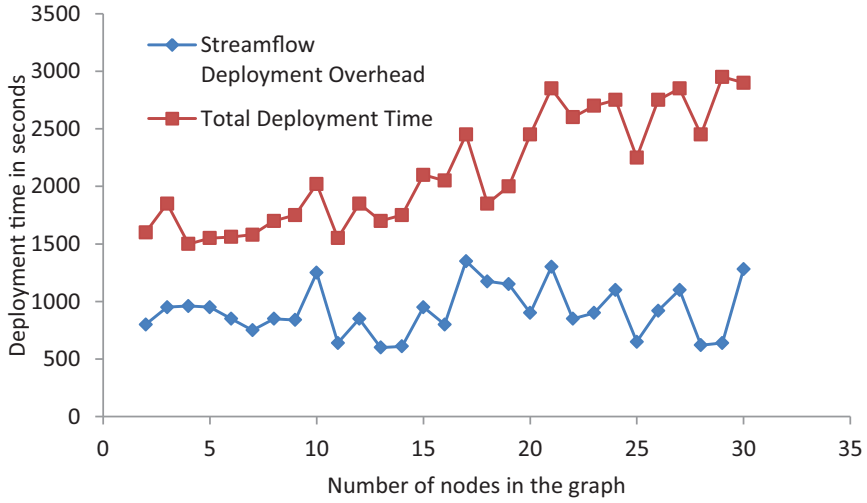

Figure 60: Deployment time comparison between Streamflow and Workflow

5.4.5 Computational gain use case

Triggered workflow systems discussed in the Related Works section, are triggered by certain events and entire workflow is executed for each event in the stream. There are situations where part of the workflow is independent of the incoming events and the partial results of the stream independent sub graph of the workflow will be recalculated nonetheless. The Streamflow systems allow possibility of not recalculating the section of the graph that is independent of the incoming events.

The evaluation is setup with the Streamflow shown in Figure 61 where thick input array means a input fed by a stream. The Streamflow is defined as follows where SVSi and Si are web Services. Input-Node$_1$ $\longrightarrow$ SVS1 Input-Node$_2$ $\longrightarrow$ (S1, input-port2) SVS1 $\longrightarrow$ SVS2 $\longrightarrow$ ... SVSm-1 $\longrightarrow$SVSm SVSm$\longrightarrow$ (S1, input-port1) S1 $\longrightarrow$ S2 $\longrightarrow$ ... Sn-1 $\longrightarrow$Sn

Because the Input-Node1 is a static input and all the SVSi services are side effect free web services once it is computed there is no need for this section of the

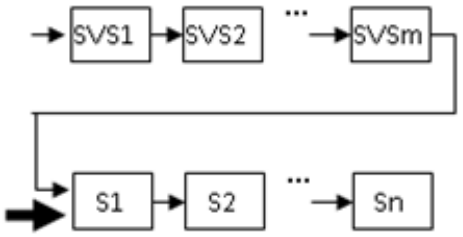

Figure 61: Sample workflow needing stream integration

graph to be recalculated. It is this save in computation that will be focused on in this evaluation. The Input-Node2 is a event stream and the semantics of the Streamflow is the workflow sill be recalculated every time an event occur in the input event stream. Assume the input event is defined as E1,E2, ,Ek and thus the workflow input event in a triggered workflow scenario would be (I1,E1), (I1,E2), ... (I1,Ek).

In case of a triggered workflow system the sub workflow shown in [SVS1⟶ SVS2 ⟶ ... SVSm-1 ⟶SVSm] is repeated for every event Ei but always produces the same output because Input-Node1 is static and SVSi services are side effect free. It would be computationally efficient to calculate this section of the workflow thus saving compute cycles as the events arrive in the input event stream. If the running time for a given workflow W for k events is defined as Tk(W). The running time of WF0can be defined as follows where T(Si) and T(SVSi) are time taken by services Si and SVSi respectively.

$$\mathrm{T}_k(\mathrm{WF0}) = \sum_{j=1}^{k} \left[\sum_{i=1}^{m} T(SVSi) + \sum_{i=1}^{n} T(Si) \right]$$

The evaluation removes the inter-arrival time between the events in the stream from the calculation and introduces an event number and the measurements are made to track and register the times at which the event arrives at each sub-graph. This is deliberate because the time from one event to another is a subjective measurement. When the Streamflow graph is partitioned it will yield a a Streamflow as shown in Figure 62.

The Workflow WF1 is defined as follows where Si are web services and without

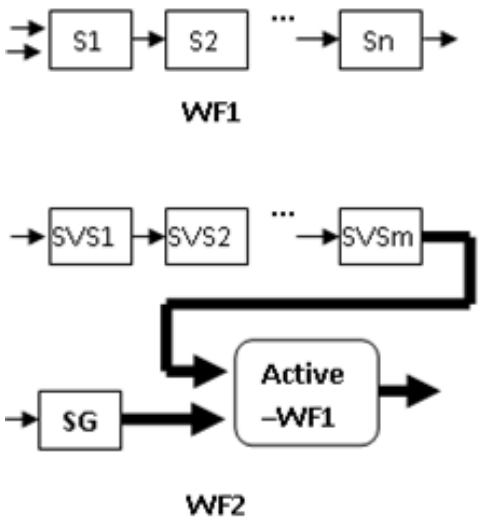

Figure 62: Streamflow approach for stream integration

the loss of generality the S1 activity is selected to have two inputs and rest of the services have one input and one output.

S1 $\longrightarrow$ S2 $\longrightarrow$ ... Sn-1 $\longrightarrow$ Sn

Input-Node$_1$ $\longrightarrow$ (S1, input-port$_1$)

Input-Node$_2$ $\longrightarrow$ (S1, input-port$_2$)

Streamflow WF2 shown in Figure 62 where SG is the Stream Generator, Active-WF1 is a Active that represent WF1, SVSi are again pure web services with single input port and single output port. It should be noted that S1 and Active-WF1 both have the same number of input ports.

SVS1 $\longrightarrow$ SVS2 $\longrightarrow$ ... SVSm-1 $\longrightarrow$ SVSm

SVSm $\underrightarrow{stream}$ (Active-WF1, port$_2$)

SG $\underrightarrow{stream}$ (Active-WF1, port$_1$)

Active-WF1 $\underrightarrow{stream}$ Stream-Sink

This partitioning allows the WF2's services components SVSi's to be executed only once and thus all the subsequent streams would reuse that result. But this would introduce extra overhead for setting up SG - Stream generator and as calculated in the earlier performance analysis Streamflow introduces latency for event

dispatching, the Active-Node-Latency which would be referred to as ANL. Under the assumptions since WF0 can be functionally replaced by WF1 and WF2, we can use this setup to compare the time taken by the Streamflow approach against the WF0 approach for the same event stream E1, E2, ...Ek. The time taken by the new approach to compute this event stream is as follows.

$$\mathrm{T}_k(\mathrm{WF1} + \mathrm{WF2}) = T(SG) + \sum_{i=1}^{m} T(SVSi) + \sum_{j=1}^{k} \left[\sum_{i=1}^{n} [T(Si) + ANL] \right]$$

It should be noted that the $\sum_{i=1}^{m} T(SVSi)$ is independent of the event stream now, which is a clear performance gain and it would undoubtedly improve the performance of the Streamflow based solution. But there is a constant time ANL which occurs for every event and finally a constant setup time of T(SG). The critical question would be what is be the minimum number of m required to offset the overhead added by the ANL and the following performance analysis shows that at m=1 the Streamflow solution offsets the initial setup cost T(SG) and ANL just after four events.

Figure 63 shows the realization of above experiment with m=1 and n=1 so it gives a lower bound on the system performance. The higher the value of m, the faster the Streamflow would outperform WF0. Higher the value of n, T(SG) cost becomes less significant.

Environment1 Evaluations tagged Environment1 was run on Indiana University Odin cluster of 128 nodes of 64bit Dual Core AMD Opteron Processors (2000MHz) with 4GB of memory for each node. The cluster machines were used for clients that were executed in parallel with a job manager and the server was running on an eight core AMD Opteron(tm) Processor 8218(1000MHz) machine with 32GB of memory.

Environment2 Evaluations tagged Environment2 is done with server components running in a 8 CPU machine running Duel-core AMD Opteron processors

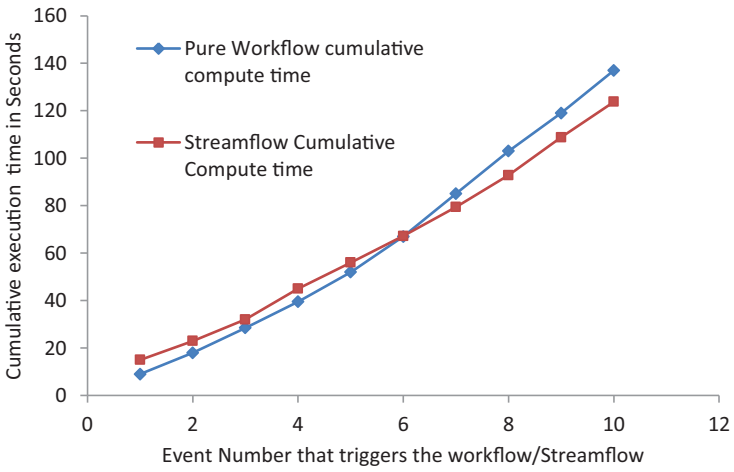

Figure 63: Throughput measurement that with the Streamflow optimizations and conventional triggered workflow system

at 1000MHz, 32 GB memory and a client machine where the workflow composer was run had a configuration of 2 CPUs with Intel Pentium 4 CPU 3.20GHz, 2GB memory.

5.5 Sustainability of Streamflow Execution

Operational Semantics of the Streamflow describe the way a given Streamflow can be compiled into the underlying framework components like workflow engines and Complex Event Processing systems. Once a Streamflow is launched and coupled with the input event streams there are many things that need to be in place for the system to continue to operate in a sustainable manner. There are few factors that affect the continuing execution of a particular Streamflow. They are;

- Structure of the Streamflow
- Resource allocations for each activity of the Streamflow
- External event rates to the Streamflow

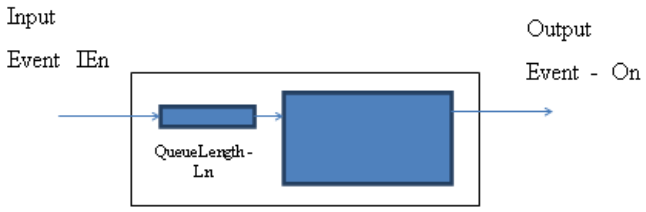

Figure 64: Event Queuing at the activities where the queue can be conbination of job queue, operating system queue and application server queue

The external event rates is a the driving factor in overall system sustainability. With available resource consumption and rates for a given workflow, it could be difficult to determine an analytical solution to the sustainability of a Streamflow. We will explore an analytical solution to this problem. We will also explore a real-time monitoring of the system focusing on the queue sizes initially.

5.5.1 Analytical solution

We would use queuing theory to analytically solve the sustainability of a given Streamflow composition. It should be noted that the result of this analytical solution is constrained by several assumptions. The nature of the Streamflow nodes is such that, if an event arrives at a Streamflow node it will trigger a job in some backend resource. In most of the Streamflow use cases the jobs will be launched to a Teragrid site and thus ending up in a queue. The nature of the resource allocation in the particular resource site affects the modeling that can be used.

The queuing systems use standard notation for its classification such as A/B/C/D/E where A represents the probability distribution for the arrival process, B denoted the probability distribution of service process, C represents the number of servers, D represents the maximum number of customers in the queue and E represent the number of customers in total. Most of the analytical solutions for queuing systems in steady state uses Poisson distributions to model arrival process. So M is used to denote Poisson arrival distribution as well as exponential

service time. Also we may only specify first three configurations when latter two are ∞. In other words M/M/c would be same as M/M/c/∞/∞ and denotes a multi-server queuing system with c servers and with Poisson arrival rate and exponential service time. Infact this would be Queuing model that would be used in this analysis.

Most of the grid/cloud based resource allocations can be modeled using queuing theory with certain constraining assumptions. For example if a single job requires n nodes and the reservation at the site consist of 3n nodes that could be looked upon as a multiple server queuing system. If the reservation is not shared with any other process or node it can be modeled as a multi-server system. Before an analytical solution can be reached it is necessary to model the arrival rate and the service time. The available analytical solution for a network of queues also known as tandem or multistage queuing systems has the roots on Burke's Theorem [19] and Jackson's Theorem [62].

The Streamflow can be modeled as network of queues at each nodes as shown in Figure 65. Even though there are no explicit queues in the Streamflow framework, during execution the components like back end job queues and workflow engines will act as implicit queuing systems. Thus each node will act as a queuing system, in fact it would be multi-server queuing systems because each node would service its request using a pool of resources.

The Burke's Theorem specifies that M/M/c systems when arrival process is Poisson and is in steady state the departure process is also Poisson with same rate parameter as input rate parameter. Jackson's Theorem shows that if the arrival of jobs in the network is Poisson and service time is exponentially distributed the joint probability of the state of all the jobs is product of individual probabilities, in other words they can be calculated as independent systems. Also [66] extended the Jackson's Theorem to more generalized networks of queues that handle multiple event classes where the arrival rates and processing times could differ for each class. Availability of more general forms of queuing models are discussed in [52].

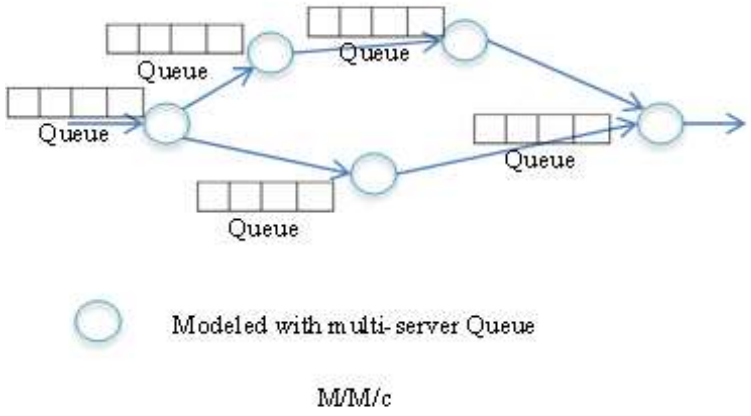

Figure 65: Queuing model for Streamflow using a queuing at each of the activities in the graph

$$\text{Utilization: fraction of time facility servers are busy } \rho = \frac{\lambda T_s}{N} \tag{2}$$

$$\text{Poisson ratio } K = \frac{\sum_{I=0}^{N-1} \frac{(N\rho)^I}{I!}}{\sum_{I=0}^{N} \frac{(N\rho)^I}{I!}} \tag{3}$$

$$\text{Probability that all servers are busy } C = \frac{1-K}{1-\rho K} \tag{4}$$

$$\text{Average length of Queue } w = C\frac{\rho}{1-\rho} \tag{5}$$

The analytical solution for M/M/c queuing model that can be applied for a single node is given by equation 5[111]. Jackson's Theorem can be used to apply the equation 5 to explore the steady state queue length of all the nodes starting from input nodes. Such lengths would give an indication of the theoretical stability of the system.

There can be several cases where the analytical solutions do not give valid solutions and the most significant of them is the assumption of exponential service time. Most tasks in a workflow have some kind of a complexity analysis that can

be assumed. Thus the service time of the tasks in most cases depends on the input event thus it may very well not be behaving to exponential service time rule. Also Jackson's Theorem assumes that the event exiting the system, such as in the case of the Filter node would have finite and constant probability to become an exit event. In the Filter node the predicate that decides whether a given event needs to be filtered out will not produce a constant probability density function in most cases. Another anomaly that can be observed is that when resources are used sometimes multiple nodes are served from the same resource allocation. This breaks down the multi-server model upon which the analytical solution is Jackson's Theorem is based.

5.5.2 Dynamic solution

The Streaming nodes in Streamflow have multiple execution contexts alive at a given time, that is to say that multiple event have arrived at the node and are in different stages of execution within the node. There is always finite amount of resources that available for the processing of events that arrive at the streaming node. Thus if the event arrival rate at a particular node is greater than the amount of events that the Streaming node could handle, the service requests could either be rejected or there event would be queued until such time resources will be available. It should be noted that queuing is not part of the Streamflow framework explicitly but implicitly queues are present in Streamflows because most of the HPC and supercomputing resources implement job queues when job requests exceed the allocation. Studies like [104] does explicit queues to buffer requests for streaming. We will model an abstract queue and try to maintain the length of the queue by book keeping the number of unfiltered events that were delivered to the streaming node and the number of event that finished processing at the node and emitted.

Every time a new event arrives at a streaming node a new queue length is calculated Figure 66 shows the queue length matrix kept n streaming nodes for past m event activities. History is important to giving an insight into normal behavior of the system as to determine anomalous behavior.

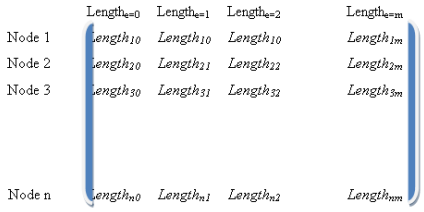

Figure 66: Event-Queue length matrix used by the Stream registry to measure the current length of the queue and finite history of the behavior of the queue

$L = \lambda W$

An acceptable value of the queue length is an important estimate and if the current queue length is significantly larger than the expected queue length chances are the system crash due to the queue buildup. To estimate the expected queue length we propose the Little's theorem [75] stated above, where L is the expected queue length, λ is the arrival rate and W is the average job duration. The queue length matrix will be updated periodically and if the queue length values exceed the expected value and monotonically increasing that is a indication where the Streamflow in consideration cannot be sustained at the given streaming node. Such a scenario calls for dynamic resource allocation to stop system from crashing as well as to reduce load on existing service, although a promising research but will not be explored in this thesis.

The modeling of the queues in the framework level shows allows much simpler modeling and because it will abstract the different queuing behaviors in different layers, namely operating system, web server and application layer. It is hard to model the effects of these queuing models individually thus defining an abstract queue at the level of the Streamflow nodes would make it more manageable as well as easier to measure.

Each CEP Node that gets deployed has two streams associated with it and these two shall be referred to as source and the sink. if the node has one to one input output correspondence like a web service node or workflow then measuring the number of events that went through event source and number of events that

went through event sink will give an estimate of how many events are in transit within that particular Streamflow node. If the node does show one to one input output correspondence it is safe to assume the difference between the source events and sink events will be the queue length.

The evaluation done using these queuing models discussed earlier does show coherent results. The following experimental setup is used to demonstrate the nature of the queues of the systems and its ability to grow and shrink as the load on the system vary. The Poisson processes that generate events produce analytical solutions to single server or multi-server queues with Poisson waiting time. But it is hard to look at such results to identify the event signatures and correlate it graphically.

The input event stream is modeled such that the event rates in the event stream varies according to the following formula and the workflow that is launched computes the Linpack benchmark[30], which is a compute intensive CPU benchmark. The Linpack input size is selected such that it produces service time delays that would reveal the event rate signatures in the output stream. This evaluation uses three stream rate distribution setup run on Environment2 and measure the variation of the queue lengths.

Stream1:

Strem1 is defined as:

$y = Max[0,\ 3sin(\frac{\pi x}{4}) + 1]$

which will produce the following sequence for y that will be used as event rate in the input stream. It will be a repeating sequence of the form 1, 3, 4, 3, 1, 0, 0, 0, 1, 2, 4 ... . This sequence represents the number of events that will be published in x^{th} second from the start of the experiment. Figure 67 shows the event rate distribution in Stream1 and the queue length measured by the Streamflow framework. It should be noted the tests are done in an experimental setup where the input values to the Linpack workflow is selected to show the correlation of the queue length to event rates. Figure 67 shows a clear building up of the queue which will lead to eventual crashing of the workflow engine. The Asynchronous nature

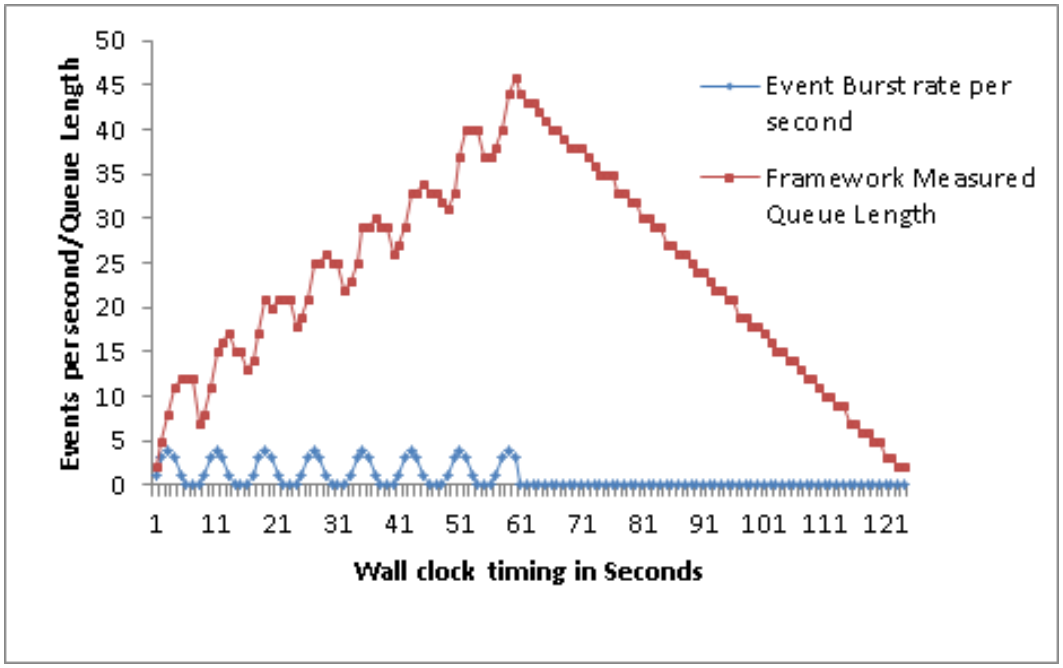

Figure 67: Queue Length behavior of for Event Stream1 where queue growth shows the characteristic event rates that were modeled by the publishing setup and the queue length continue to grow

of the launching the workflows by the Streamflow makes sure the Streamflow will not build up queues internally.

Stream2:

Stream2 is defined as:

$y = Max[0,\ 3sin(\frac{\pi x}{8}) + 1]$

which will produce the following sequence for y that will be used as event rate in the input stream. It will be a repeating sequence of the form 1, 2, 3, 3, 4, 3, 3, 2, 1, 0, 0, 0, 0, 0, 0, 0, 1, 2, 3 ... . Figure 68 shows the event rate distribution in Stream2 and the measured Queue length variation. The distribution has different harmonics and the queue length growths follow the new harmonics.

Stream3:

Stream3 is defined as

$y = Max[0,\ 3sin(\frac{\pi z(x)}{4}) + 1]$

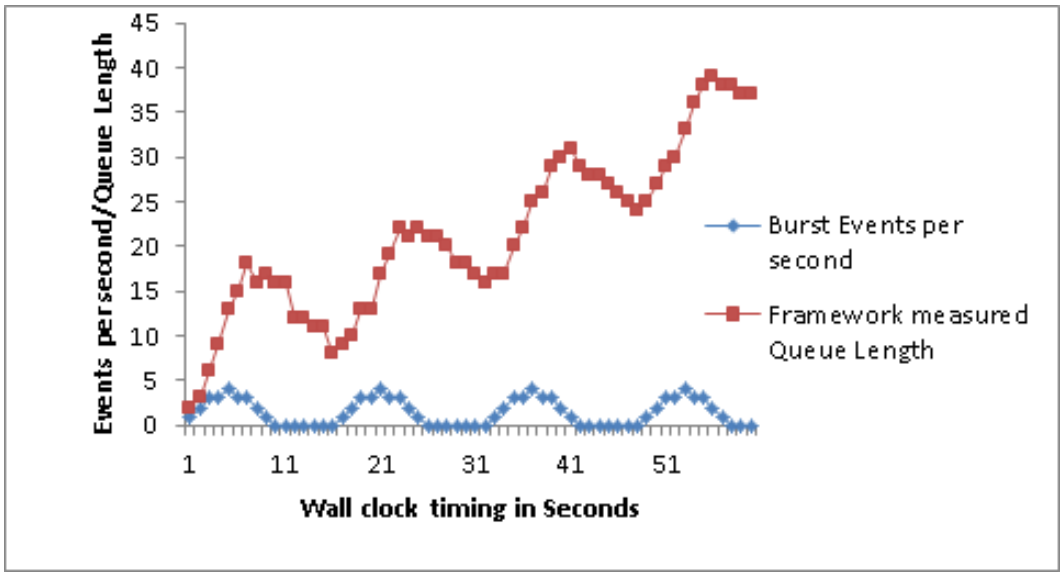

Figure 68: Queue Length for Event Stream2 where the average rate is less than Stream1. Although the queue lengths still ingrease, the rate of increase has reduced.

$$z(x) = \begin{cases} 0 & \text{if x even} \\ x & \text{if x odd} \end{cases}$$

which will produce the following sequence for y that will be used as event rate in the input stream. It will be a repeating sequence of the form 1, 0, 3, 0, 4, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 4, 0 ... . The rates are published every other second thus the events are sparsely distributed than in Stream1 or Stream2. Also the average event rate in this Stream is less than the average events in Stream1 and Stream2. . Figure 69 shows the system working with perfect expected throughput model where the queue lengths grow and shrink as the event rates in the stream increase and decrease.

Figure 70 shows the queue length fluctuation at much higher event bursts and how system behaves in such burst conditions. The experimental setup involves a workflow that runs a linpack application in a single node thus simulating a bottleneck in the application by running compute intensive application in a single node. As the event stream receives the event bursts the workflows are launched

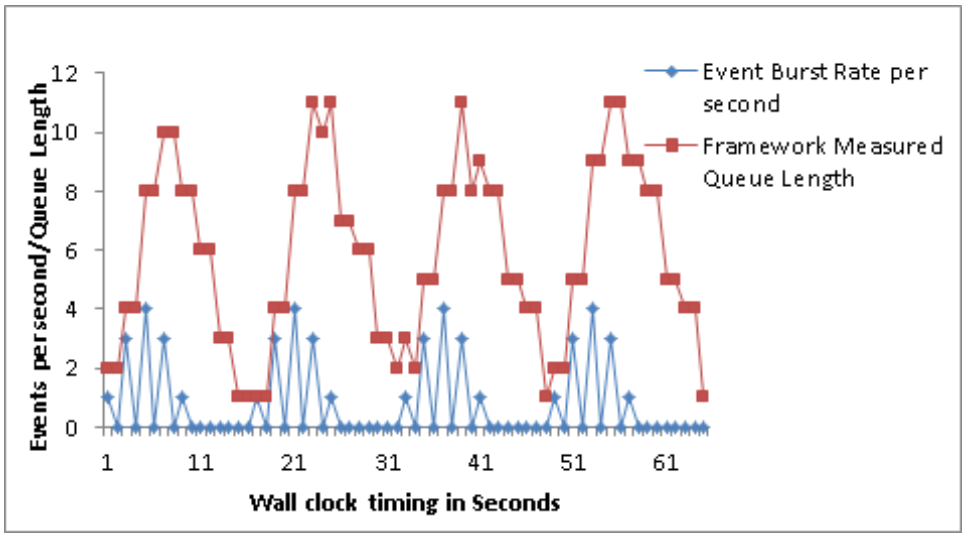

Figure 69: Queue Length for Event Stream3 where the system has a healthy queue where queue lengths grow and shrink with the event rates of the input stream

but the resource will for the workflows to queue up because the compute intensive linpack application will consume much of the compute resources in the node. System show queuing up of requests and gradually queue getting reduced as the compute resource gets freed up and subsequent workflows are run.

Cascading Queue behavior is observed in the experiment shown in Figure 71 where a stream source is connected with six sub-workflows connected in the given topology. It should be noted that A,B,C,D,E,F are workflow produces by the partitioning algorithm. and G is a join node that joins the two output streams from workflow C and F. Figure **??** show the measured queue length for this experimental setup and it shows the cascading behavior of the queues which are grown and shrink with the input event stream. The signature of the input event rate is visible down to the third sequential component in the Streamflow

5.6 Type validation

Type safety in workflow systems allows verification of the correctness during composition. This sub-section discuss the possibility of validating type safety in a stream processing environment where strong typing is not a prerequisite. The

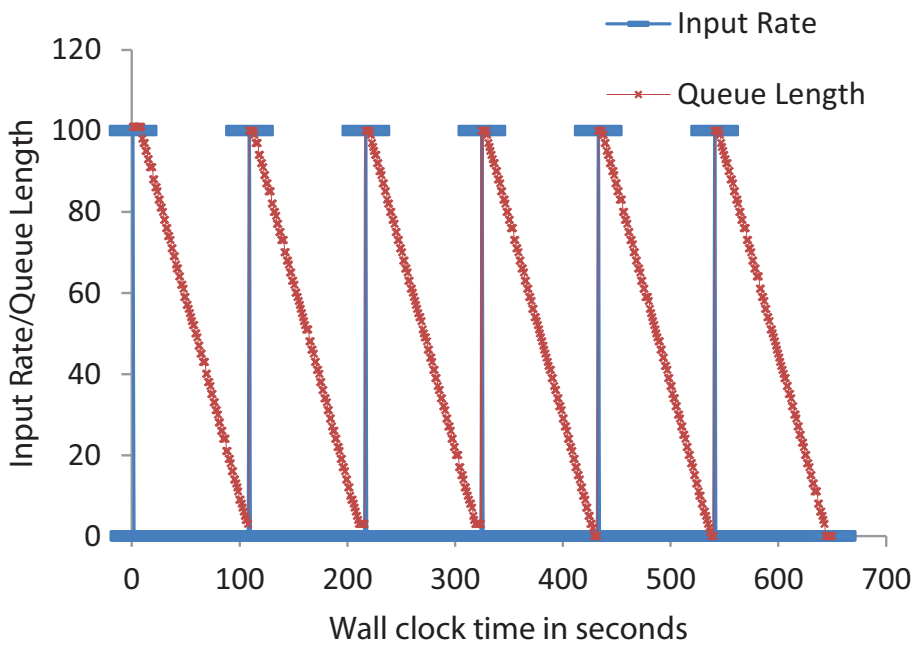

Figure 70: Queue Length behavior for event bursts. Workflow node's input event stream is published with high event bursts and queue length is monitored.

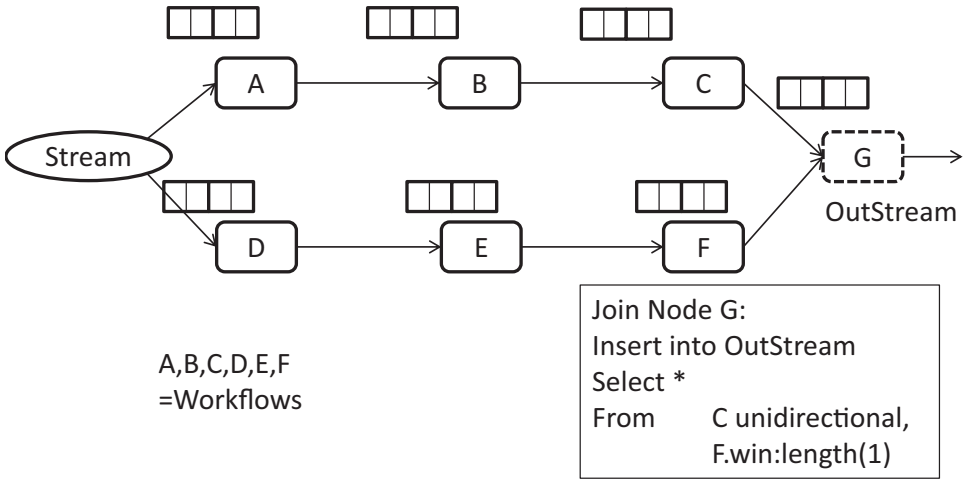

Figure 71: Cascading queue length experimental setup with six workflow nodes one stream source and one join node.

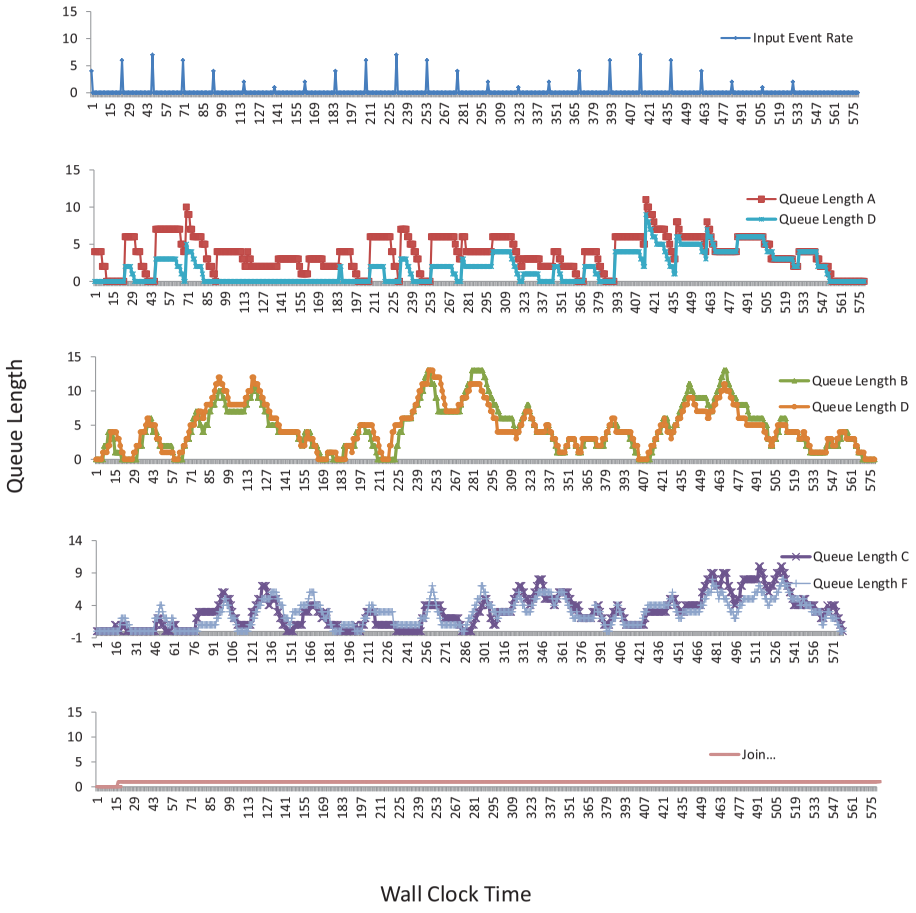

Figure 72: Cascading queue length measurements of different Workflow nodes in the experimental setup

typing is based on the Xml message types because the data flow edges in the Streamflows are based on Xml message based events. The control flow of the Streamflow is strongly types based on the WSDL type definitions which provide Xml Schema Definition and these can be used for checking type safety. Type matching of pure control flow sub-graphs is based on the guidelines in BPEL specification and XBaya is BPEL compliant thus providing type safety for pure SOA control flows. The focus in this will be on the type safety of CEP based stream processing. The Streamflow composition does not enforce strong type safety because it is intended to be flexible and at times it is allowed to do CEP based processing without knowing the full structure of the event. Sometime the CEP processing only focus on few elements of the event and system is expected to operate without having the full knowledge of the event but have partial knowledge of the structure. Also some streams may consist of events of different structure and it will still continue to operate because all the event of the stream satisfy the structure required by the propertied defined on the stream. This is similar to duck typing in type theory. First it is useful to identify where the type systems play a role in the stream processing. The CEP nodes make reference to properties in the events in the stream. They will be data bound using XPath expressions and it is not necessary for the system to be fully aware of the complete schema of the event. Yet the stream properties will be referred to in the CEP query and it would be useful to validate where possible that the property refer to a valid element in the event. Invalid property definition, for example, due to a typographical error, would not throw an error yet continue to search for an nonexisting element and will continue to return null results. It useful to validate that such errors will not occur during the time the property is defined so the debug cycle will be much shorter. There are cases where the system can guarantee the type safety given certain conditions are met. It is important to understand that this is not an" if and only if" condition. In other words there may be instances where the system will not be able to guarantee the type safety yet the system is entirely safe, just that it does not have sufficient information to prove type safety. The type verifi-

cation in this framework is inspired by type systems for multiple inheritance and record concatenation [118].

Definition: Structurally Unaltered Derivation of the stream Stream A is structurally unaltered derivation of stream B if Stream A is produced by query without projection, without join or aggregation.

The query would take the form of:

```
INSERT INTO A SELECT * FROM B [predication|pattern]
```

The intuition behind the Structurally Unaltered Derivation stream is the events in the derived stream will have the same structure as the parent stream events.

Following recursive property validation rules will be used to infer the validity of a property:

- A property definition on a stream is valid: if stream has defined schema for its events and elements referred to in the property are valid elements in the schema graph.

- A property definition on a stream is valid: if property is an already validated property on the same stream or Structurally Unaltered Derivation of the stream.

  eg: Given property definition for B `reflectivity= /radar/maximumreflectivity` `INSERT INTO A SELECT * FROM B WHERE maxreflectivity>50` And given property definition reflectivityis valid in B Then property definition of the following form is valid for A `<propertyname>=/radar/maximumreflectivity`

- A property definition on a stream is valid: if property is a substructure of already validated property on the same stream or Structurally Unaltered Derivation of the stream.

  eg: Given property definition for B `reflectivity= /radar/maximumreflectivity` `INSERT INTO A SELECT * FROM B WHERE reflectivity>50` And given property definition reflectivity is valid in B Then property definition of the following form is valid for A `<propertyname>=/radar`

- A property definition on a stream is valid: if stream is generated by merging of n streams and the property definition is valid in all the n streams. eg: Given property definition for A1

  `maxreflectivity= /radar/maximumreflectivity`

  `INSERT INTO A SELECT * FROM A1 WHERE maxreflectivity>50` And given property definition for A2 `reflectivity= /radar/reflectivity`

  `INSERT INTO A SELECT * FROM A1 WHERE reflectivity>50` And given property definition reflectivity and maxreflectivityis valid in A1 and A2 respectively Then property definition of the following form is valid for A

  `<propertyname>=/radar`

During user definition of the stream properties above inference rules can be used to give an indication if the newly defined property can be guaranteed to be valid. Other than the above rules user can examine the last event in the stream using the stream metadata to verify the Xpath is valid on an example event. It should be noted that the inability to validate a given property does not imply an error. It could very well mean the structure of the event is not fully understood or this could be the base case where the first valid property for the stream is defined.

6 Map Reduce Streaming

The Map Reduce programming model is mainly focused some of the applications that follow the Single Instruction Multiple Data model yet by carefully constraining the control structure it has the ability to build a simple yet scalable parallel computing systems. One of the key aspect to note about the current Map Reduce frameworks is they support static input datasets, that is the input data files to the Map Reduce framework need to be present at the beginning of the execution. This premise will be revisited in this chapter and we will evaluate the ability to relax this constraint to allow simple programming paradigm Map Reduce to be coupled with data streams without losing the scalability, and simplicity. Further investigations will be made on the prototyped streaming version of Map Reduce built on Apache Hadoop on how to improve compute or data performance as well as theoretical analysis programming abstraction.

6.1 Background

The Map reduce exposes task parallelism in a way that has proved to be used broadly. Over the years research has been done to enhance the data parallelism aspect of this and other parallel programming models [122] [88] [86]. It is useful to examine the emerging programming models in in light of the broad parallel programming models proposed by Flynn [39]. The challenges in parallel computing is made complicated because of the big data problems presented in scientific applications so this thesis will focus on "Multiple Data" aspects of the taxonomy. (Single Instruction Single Data) is a parallelization strategy when the data are partitioned and same instruction applied to each partition. (Multiple Instruction Multiple Data) MIMD model the single instruction constraint is lifted and datasets can be bound to multiple programs thus allowing better flexibility. The relative simplicity of SIMD model makes scheduling relatively easy and replicating the application seems to be a probable approach among others, this has received acceptance especially in the big data applications and providing for affinity where

moving the application required significantly less bandwidth than moving the data around. The MIMD model presents much more challenging resource allocation problems and in most MIMD frameworks adopt a pipelined approach. One of the significant issues with the modern day scientific computing is that they are very data intensive application. So pipelining although a viable approach for MIMD architecture the complexities associated with assigning affinity of application to data makes the pipelined approach not so attractive. SIMD model on the other hand is tractable enough to develop frameworks that can facilitate affinity of data and instructions. The parallel programming techniques suits well for data streaming and high throughput applications and MIMD model when applied in pipeline fashion in fact relies on the fact that application has streaming data to achieve high throughput.

The Map Reduce programming framework which on high volume data intensive applications. The simplicity of the programming semantics allows frameworks be designed that allow data affinity when code is moved closest to the big whenever possible. Map Reduce programming semantics was not designed with data streaming in mind and it is intended more for single Map Reduce run. The parallel nature, availability of a barrier as synchronization mechanism (similar to MPI barrier) as well as simplicity of the programming makes Map Reduce a good candidate for data intensive parallel applications.

6.2 Streaming Map Reduce

Given the pipelined and streaming approach that may allow high throughput, along with the data affinity based task scheduling, a framework auch as Map Reduce may show potential for data intensive scientific application. It is important to keep the aberrations to a minimum because it is important to keep the new streaming Map Reduce programming paradigm as simple as possible, unlike the MIMD pipeline approach with its complicated data affinity and resource allocation needs.

The clearest most straight-forward way to use the existing Map Reduce frame-

works in a streaming context is to write a component that will listen to an event stream and launch map reduce jobs for each even in the event stream. Each Map Reduce run would have multiple maps and in most cases a single reduce, thus each Map Reduce run is independent of previous runs. These different Map Reduce runs in such a situation although independent, are triggered by events from the same event stream thus may exhibit significant correlation with each other because most event streams are a discretization of some kind of continuous process. The Streaming Map Reduce architecture proposed in this paper tries to facilitate this continuity by using sliding windows and adopting a pipeline like model to produce outputs.

In the proposed approach the Map Reduce Job will accept inputs from a data stream and new map tasks will be added to the Job as an when new data events arrive in the input stream. The mapping of new data events to the number of map tasks has no particular relationship, but rather will be determined by a input formatter interface that will determine the split files for the input data events, thus determining the number of map tasks. For simplicity, assume that a single event launches a single map task, so each event in the input event stream will trigger a map task. Figure 73 shows the creation of such map tasks where Map0, Map1, and Map2 correspond to the map task triggered by events D0, D1, and D2 respectively. This would produce a stream of map outputs and using a windowing mechanism used in complex event processing, a current map output set can be identified. It is this current map output set that will be used when the framework is going to trigger a reduce task.

The approached proposed is a setup where the Map Reduce Job that would accept inputs from a data stream and new map tasks will be added to Job as an when new data events arrive in the input stream the mapping of new data event to the number of map task has no particular relationship but rather will be determined by an input formatter interface that determines the split files for the input data events thus determining the number of map tasks (Figure 73).

In any given production system, the resource consumption cannot grow linearly

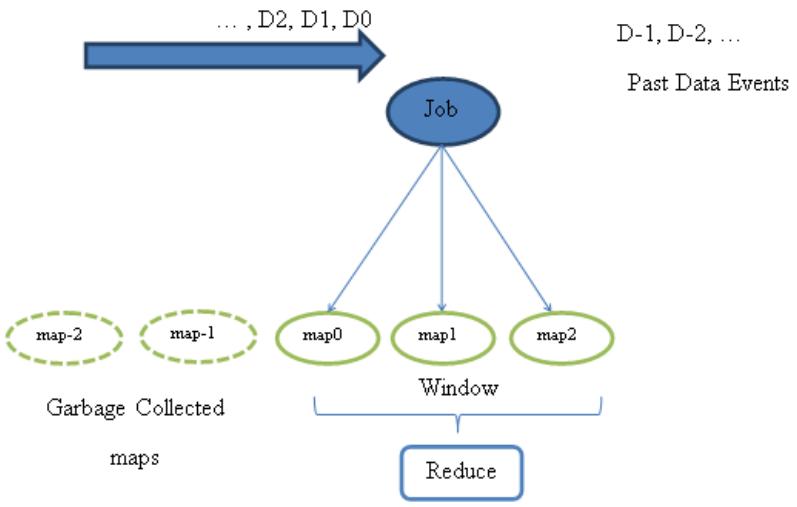

Figure 73: Streaming Map Reduce mapping of events to maps and defining a window that caches map outputs. The window can be a length window or a time window

with time or with a input data stream because the system will not achieve a steady state, but it would run to the point until it consumed all the available resources and caused a eventual system crash. By mapping the map tasks to input data events, the system becomes vulnerable to such short comings, but this can be mitigated by such techniques as garbage collection mechanism.

Figure 74 shows the window mechanism and how the window of map outputs are used not only for garbage collection but also for triggering the reduce tasks. Sliding windows on an event stream is, in most cases, defined by time or length. The time windows maintain events within a certain time frame and any event in the window would be removed after its time stamp expires that of the window. The length windows are such that the window would collect events up to certain parameter, called the window length parameter, and after that when a new event arrives at the window, the oldest event will be removed keeping the window size within the length parameter. In Figure 73 Map_1 and Map_2 show a few such expired events removed from the window.

The triggering of the Reduce task is triggered by changes to the map output

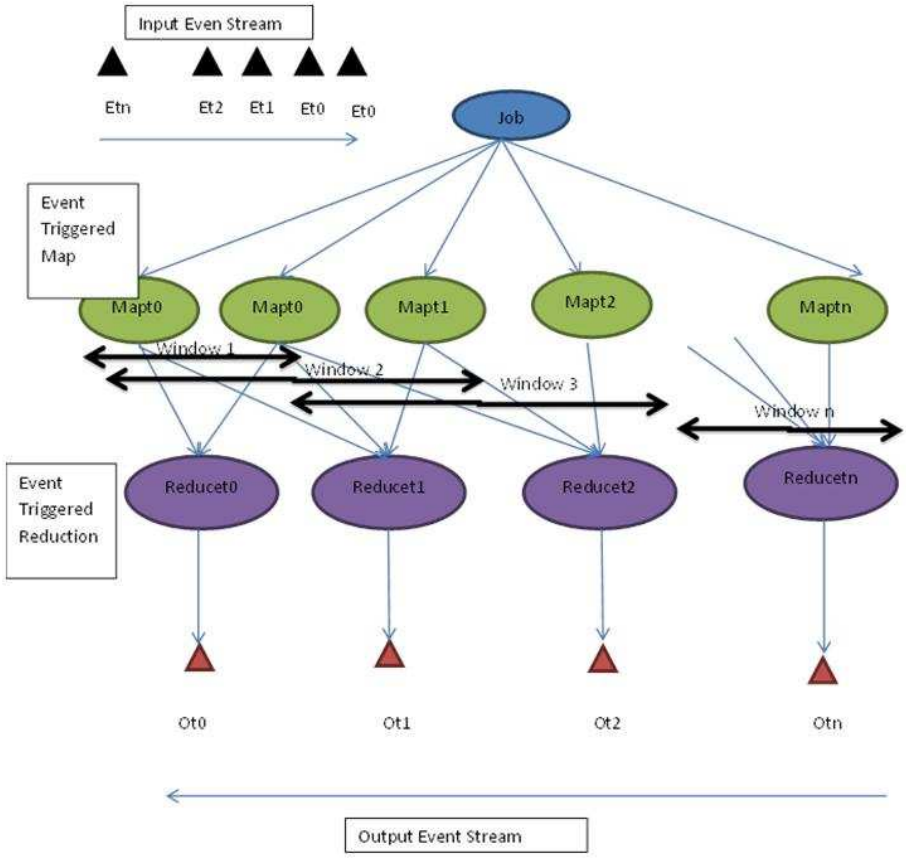

Figure 74: Streaming Map Reduce map output sliding window and reduce tasks getting triggered by changes to the window

window, so the definition of the map output window in streaming Map Reduce defines how the reduce tasks are triggered. This framework will support constant length and time windows. Figure 74 shows how the reduce tasks are scheduled based on a constant length window of size 3. As mentioned earlier the reduce tasks are triggered by the changes to the window and Map_0 values trigger the $Reduce_0$ and addition of Map_1 triggers $Reduce_1$. Addition of Map_2 to the window make one of the Map_0 outputs to be removed from the window because the window is a constant length window of 3 and that change triggers the $Reduce_2$. As these reductions produce outputs they will be produce another event stream Ot0, Ot1

A Streaming Map Reduce Job keeps a current map window, as a sliding window, which acts like a cache for map outputs that are currently in consideration; any new additions to this window may trigger a reduce with the map outputs currently calculated and resident in the window. The window can be a time window or a length window allowing limited number of maps to be active. The maps that are expired, either because they were pushed out of the time window or pushed out of a length window of fixed number of items, will be added to the garbage collection and those resources can then be reallocated for new incoming event maps. As the input data stream delivers data to the Streaming Map Reduce job, it will compute new maps and that will trigger updating of the windows, thus, triggering reduce tasks.

The implementation of Streaming Map Reduce is built on top of the Apache Hadoop Map Reduce framework [6] which is an open source Map Reduce implementation. Figure 75 shows architectural changes to the Hadoop framework to accommodate continuous input event streams as well as providing the sliding window of intermediate output cache.

The client interaction with the Hadoop system was changed to accommodate the data event stream, where all the data files will not be available at the time of the launch of the Map Reduce job but become available as time progresses and events are delivered by the event stream. The client API allows launching Map Reduce jobs similar to the conventional Map Reduce jobs. In addition the client API will allow subsequent data events that arrive in the event stream to be added to an existing Stream Map Reduce job that is currently running. These subsequent events that are published to the existing Streaming Map Reduce Job will go through the same InputFormatter interfaces and split files will be generated using the existing Hadoop architecture. The Map tasks will be scheduled as and when the split files arrive at the Streaming Map Reduce job, and the scheduling policy will be similar to that of a conventional Map Reduce job based on factors such as load, data affinity within the same rack and switch.

The input data stream in Figure 75 consists of discrete input events such as

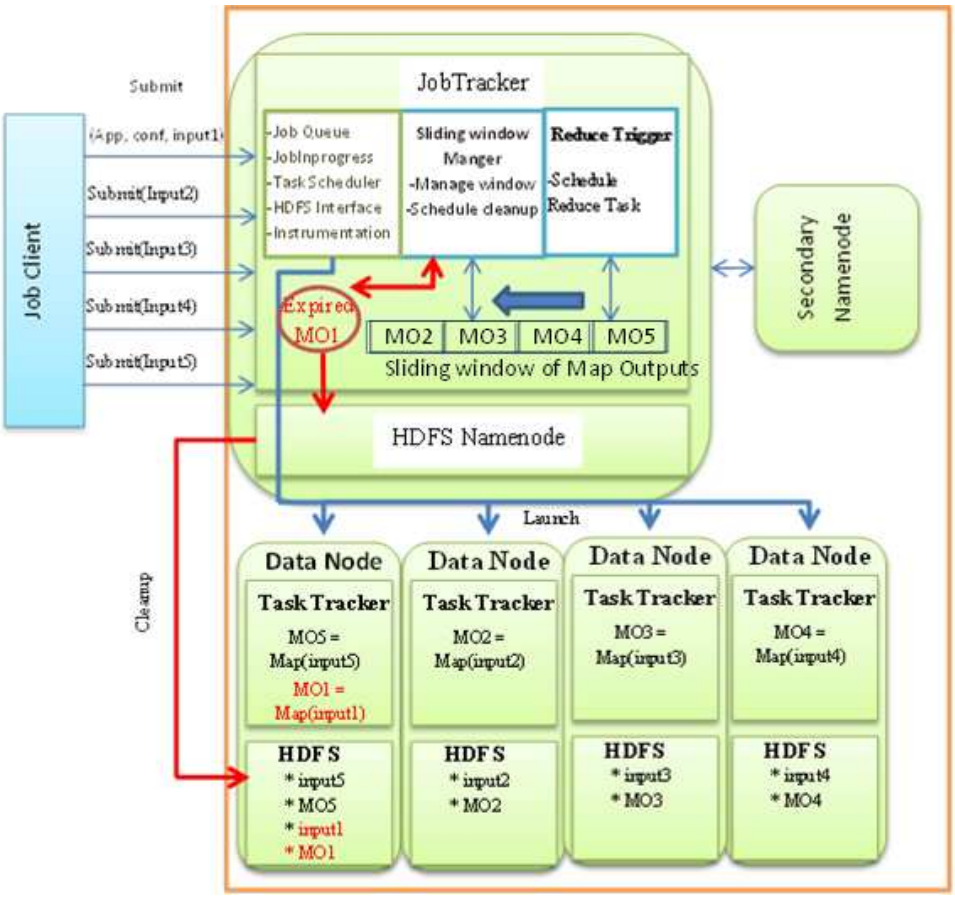

Figure 75: Architecture for Streaming Map Reduce showing the map output window and other Map Reduce framework components

$Input_1$, $Input_2$ and $Input_3$, and for simplicity it is assumed that each input file will produce a single split file which will be processed by a single Map task. For example, in Figure 75 input1 was scheduled to the tasktracker in Node 1, Input2 is scheduled to the tasktracker in Node 1, and so on. As the Map tasks are launched for the incoming input event stream, the outputs produced by these Map tasks when they finish executing will also produce a logical stream, such as MO1, MO2, MO3, and so on. Management of such Map outputs are done using a sliding window which is managed at the master node. In Figure 75, the sliding window is of constant length 4 and as can be seen, 5 Map tasks are already finished executing, so the oldest Map output MO1 is removed from the window and marked for cleanup. Thus the sliding window of Map outputs will be maintained as a cache in the current Streaming Map Reduce job. The Reduce tasks are triggered when they are changes to the sliding window, or every time a new Map output is added to the sliding window. The Reduce task will be scheduled to a node depending on the load and where most of the Map outputs are located.

The programming pattern for the Streaming Map Reduce is deliberately kept similar to that of the Map Reduce because the model proved to be a relatively efficient mechanism to handle tasks of parallel data intensive applications with very manageable resource allocation. Ground breaking changes to the Map Reduce map push the framework closer to the MIMD pipeline approach that would make the resource allocation as well as the data affinity issues becomes infeasible.

The deployment mechanism is solely based on moving files into a monitored input directory. The client API is very similar to the conventional Map Reduce jar client where the user would specify the application as a binary jar file and the inputs to that application. The application is supposed to set the directory, and the Hadoop stream is suppose to monitor which is mandatory. Other parameters that need to be set are either the length window size or time window size and the optional notification topic and notification broker URL.

The directory location can be any directory accessible by the Hadoop head-node and outputs will be written to the output location that is specified by the

application. Once the application is launched the system will continue to monitor the input directory for changes and any new files added will launch new map tasks and will become part of the streaming job. The system will continue to monitor the input directory until the user will copies a shutdown.txt into the input directory.

6.3 Evaluation

We evaluate the Streaming Map Reduce in comparison to Apache Hadoop in an attempt to identify the overheads introduced by the new programming model. The caching of the map outputs allows such intermediary results to be reused for different reductions. For example, say an output result of a particular map reduce is needed not only for the Reduce for which it was originally intended, but also for subsequent reductions. Since map outputs are kept in a window they can be reused in multiple reductions instead of having to recalculate the map outputs. Following is a theoretical graph of how map calculations can be reduced by reusing map outputs from the window. The nature of this will entirely depend on the application and how many reduces will reuse already calculated map output without recalculating the Map task. If a given map output is relevant to l subsequent reduce tasks we would define a sliding window of length l so it may be available till l more maps be calculated in which time the earlier map output will not be useful anymore and it will be thrown out of the window and garbage collected. Figure 76 shows the theoretical case where the window length change from 1 to 10 and thus the total number of map tasks that need to run per reduce changes as some map outputs are shared among reduce tasks.

The experimental evaluation presents a jobs that would make use of incoming data streams that consists of data events and map reduce jobs may be run on a window of the incoming data files. For example, assume the incoming data event stream has files F1,F2,F3,F4,F5, and assume the Map Reduce job works on a window of inputs. Assuming the window size is 3 the first job will be launched with file F1 second job with files(F1, F2) , the third (F1, F2, F3) and fourth (F2, F3, F4) and so on.

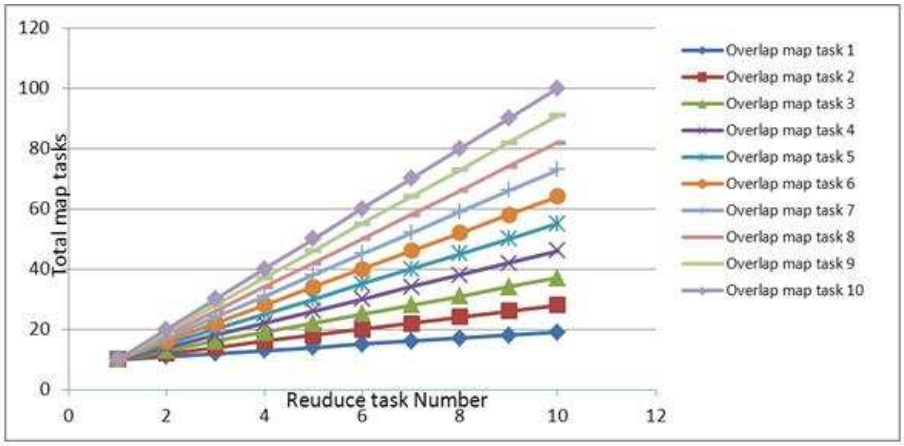

Figure 76: Total map tasks when map output are overlapped in the job

It is important to understand that file F1 is used is three jobs because of the window size being 3 and in conventional Map Reduce, jobs may run map function on F1 three times. If the Streaming framework presented in this thesis is used, it will only run the map for file F1 once and it will be cached in the window and be reused in the different reduce phases that depend on the map output of F1. Figure 77 shows n comparison of the two Map Reduce systems where the Jobs involve sharing the data files in a window of three. The Jobs run a word count benchmark with input files of size 32 MB and it was run on a eight processor environment with 1 GHz clock speed with 32 GB of memory.

Both the jobs take similar amount of time to finish the initial job and as more and as more files are streamed in, reusing of the map outputs in the window tends to pay off and cumulative time for job completion clearly shows the Streaming Map Reduce taking less time to finish, thus showing less cumulative time. The cumulative time is computed such that it removes the interarrival time of the files. It is as if jobs were launched right after one another so the cumulative time does not include event latencies.

Figure 78 shows a better distinction between the two systems where the file sizes were 256 MB thus showing a much bigger gradient difference and hence showing better performance when the computations of the maps are reused.

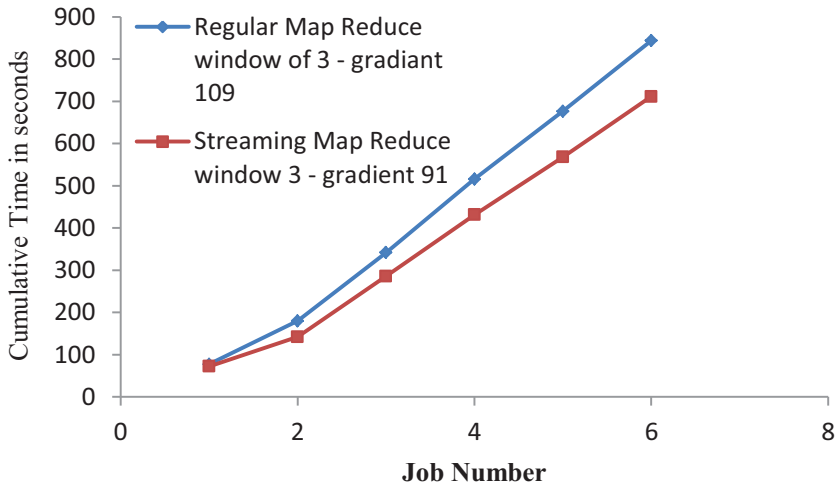

Figure 77: Cumulative job time when the job consist of inputs to the previous run and job size is 32 Mb of text file processing. When window of leangth 3 is used in the Streaming Map reduce Vs nowindows used in conventional Map Reduce

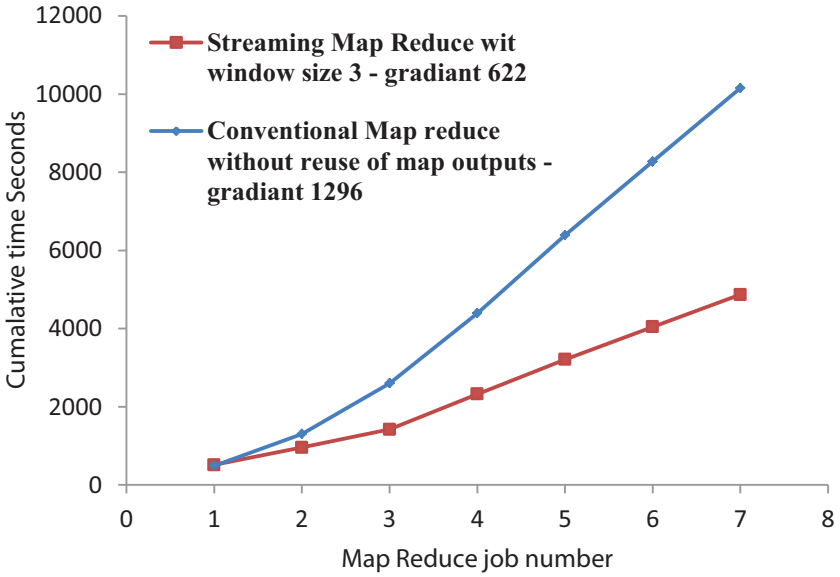

Figure 78: Cumulative job time when the job consist of inputs to the previous run and job size is 256 Mb of text file processing. When window of leangth 3 is used in the Streaming Map reduce Vs nowindows used in conventional Map Reduce

7 Conclusion

This thesis focuses on a programming abstraction that better enables event processing by allowing a graph-based computational model that builds upon the scientific workflow model and the declarative query based processing system. The semantics defined in this programming model include workflow semantics, streaming semantics, and Map Reduce semantics. One of the main focuses of this thesis is not only to provide a programming abstraction, but also to make it as close to a deterministic control-flow model as possible. Unlike workflow systems or batch processing systems, real-time event processing systems have an implied notation of a deadline associated with processing an event in the event stream to keep up with the incoming events. Failure to do so will result in building up of job queues and will lead to the eventual crash of the processing instance. This thesis also focuses on the runtime sustainability of a given Streamflow graph given its processing activities and the event rates needed to service in each phase of the Streamflow.

The Streamflow programming model builds a graph structure that can be used as a global view for a given event processing application that may allow the scientific user to have an over all view of a given event processing application with emphasis on event flow and dependencies. The severe weather use case presented in chapter 4 makes use of this global view to present much a clearer view of the overall event processing experiment. The graph structure consists of declarative stream processing components as well as scientific workflow components. The declarative query based processing components were useful in the event stream processing use case presented in chapter 4. Much of the filtering and selection of the initial high throughput event stream is reduced down to a manageable level, using the declarative filtering of the event streams to datamine the events that are important to the experiment. Also the workflow semantics that can be used interchangeably in the graph allow the scientific processing components to be encapsulated in SOA based activities, that may submit scientific computing jobs

to HPC resource when necessary. The severe weather use case shown in chapter 4 does use SOA activities that locate storms and launched weather forecasting models.

The graph partitioning algorithm that is presented in chapter 4 partitions graphs in a way that it identifies the sub-graphs that operate with the same stream cardinality and thus allows such graphs to be deployed as a single processing unit to a workflow orchestration engine. The complexity of such portioning is within the linear time complexity, making it an efficient algorithm and graph partitioning provides many benefits to the programming abstraction. It helps identify the different possible phases of the Streamflow graph that identifies the biggest orchestration graphs with the same cardinality. Once such sub-graphs are identified their orchestration can be delegated to a workflow orchestration engine. Once a sub-graph is identified to have the same cardinality in a given Steamflow the internal joins in the graphs do not require stream joins within the sub-graphs. This will improve the deterministic nature of the overall processing structure and this is done while preserving the correctness of the Streamflow graph. This is shown in the weather use case where the system launches the complicated WRF forecast towards the end of the Streamflow as shown in Figure 35. If this sub-graph is not partitioned all the intermediary activities, such as NamLateral and Arps2WRF, etc, would require a stream join because all those nodes have more than one input stream. Further partitioning may provide a high level view of how the event rates change in the Streamflow because the post partitioned graph would show activities that have different event cardinalities.

Most of the scientific event processing applications need to process high volumes of events, and once the interesting event that matches certain characteristics is found, it will lead to a high amount of processing that may requiring significant compute resources. Thus the characteristic evolution of the event rates of a event processing application have high event volumes at the beginning of the application and a phase of data mining that reduces event rates to a manageable level. The different phases identified in a Streamflow as shown in Figure 37 have different

resource requirements, thus different quality of service requirements and have different event rates. The partitioning algorithm identifies possible sub-graphs that may be deployed to different runtime systems. The evaluation shown in Figure 51 shows that the CEP engine is able to handle very high event throughput rates of simple event operators and the BPEL workflow engine is able to handle computationally intensive workflows with higher quality of service, yet support lower event rates. The XBaya workflow engine supports intermediary event rates and is able to provide some level of quality of service requirements. Thus the evaluation of the target processing runtimes match with the thesis premise. The use case presented in chapter 4 has all three of these processing phases that gets compiled into the three different runtime systems that match their processing requirements. The iterative programming paradigm introduced in this thesis is well suited for event processing systems because at the time of compilation and deployment the user may not have a full understanding of what would the resource requirement will be and existing resources will be able to handle it. The hot deployment and queue monitoring APIs provided by the Streamflow workbench allows user to do incremental changes while monitoring the changes done to the

A Streamflow graph can be viewed as a high level program that gets executed continuously. A program may be compiled to a target runtime, but that does not mean it will run within the expected time for a given input set. In conventional programs this is governed by the Time complexity of the program. Similarly a given Streamflow can be compiled into the target runtimes, but to ascertain whether a given Stremflow will continue to sustain given its input streams there need to be similar analysis that has stricter deadlines. For example, consider a Streamflow with single node which requires one second of compute time on an available resource. If the input stream to the streamflow is at one event per second or less, the Streamflow can continue to operate indefinitely in the given setup. But an input event rate of more than one event per second at a sustained rate will mean the previous event processing at the node will not finish before the arrival of the current event, so the system will build up queues and eventually the

application will crash. This thesis discusses the possibility of achieving a steady state and the constraints needed to be achieve that. Yet the parameters that need to be estimated for sucha a analytical model makes it difficult to make use of these models. The Streamflow framework uses a much more pragmatic process to monitor the building up of the queues. The evaluation shows the reading of the measured queue length when the event stream is modeled using a signature event rate distribution. The queue length distributions closely follow the input event rate distributions and as expected the queue lengths grow and shrink when the event rate is less than the event rate that target runtime could handle. When the input event rates are greater than the event rate handled by the target runtime, the system is observed to build queues of increasing length which will lead to failure at a later point. Features like hot deployment and queue length monitoring allows the Streamflow users to be aware of the resource consumption and to build Streamflows that will always be sustainable in the runtime.

The Map reduce framework presented in chapter four provides a means for Map Reduce jobs to be launched to events in a stream. It has windowing concepts built into the system which allows intermediary outputs to be reused when possible and in cases where such window techniques are used, the system shows better compute performance and reduces the requirement for the computer resource that will be utilized.

References

[1] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] G. Agha, I. Mason, S. Smith, and C. Talcott. Towards a theory of actor computation. In *CONCUR'92*, pages 565–579. Springer, 1992.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business process execution language for web services, version 1.1. *Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation*, 2003.

[5] Apache. Apache ODE, 2003.

[6] Apache. Hadoop.

[7] XML Apache. Project. Java XMLBeans, 2003.

[8] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems (TODS)*, 29(1):162–194, 2004.

[9] P.C. Attie, M.P. Singh, E.A. Emerson, A. Sheth, and M. Rusinkiewicz. Scheduling workflows by enforcing intertask dependencies. *Distributed Systems Engineering*, 3:222, 1996.

[10] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[11] P. Bajcsy, R. Kooper, L. Marini, B. Minsker, and J. Myers. CyberIntegrator: A meta-workflow system designed for solving complex scientific problems using heterogeneous tools. In *14th International Conference on Geoinformatics, May*, pages 10–12. IEEE, 2006.

[12] P.B. Bedient, B.C. Hoblit, D.C. Gladwell, and B.E. Vieux. Nexrad radar for flood prediction in houston. *Journal of Hydrologic Engineering*, 5:269, 2000.

[13] Biorn Biornstad, Cesare Pautasso, and Gustavo Alonso. Control the flow: How to safely compose streaming services into business processes. *IEEE International Conference on, Services Computing*, 0:206–213, 2006.

[14] B.J. Biornstad. *A workflow approach to stream processing.* Ph. D. Thesis, ETH Zurich, 2008.

[15] J. Blower, K. Haines, and E. Llewellin. Data streaming, workflow and firewall-friendly Grid Services with Styx. In *Proceedings of the UK e-Science All Hands Meeting*, pages 19–22, 2005.

[16] D. Box, L.F. Cabrera, C. Critchley, F. Curbera, D. Ferguson, A. Geller, S. Graham, D. Hull, G. Kakivaya, A. Lewis, et al. Web Services Eventing (WS-Eventing). *W3C member submission*, 15, 2006.

[17] G. Brettlecker. *Efficient and reliable data stream management.* Ph. D. Thesis, University of Basel, Faculty of Science., 2008.

[18] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Readings in hardware/software co-design*, pages 527–543, 2002.

[19] P.J. Burke. The output of a queuing system. *Operations Research*, 4(6):699–704, 1956.

[20] D. Campbell and R. Rew. Design issues in the unidata local data management system. *Preprints of the Fourth*, pages 208–212, 1988.

[21] L. Chen, K. Reddy, and G. Agrawal. GATES: A grid-based middleware for processing distributed data streams. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 192–201. IEEE, 2004.

[22] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of CIDR*. ACM, 2003.

[23] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[24] J. Clark, S. DeRose, et al. Xml path language (xpath) version 1.0. *W3C recommendation*, 16:1999, 1999.

[25] K. Cooper, A. Dasgupta, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, et al. New Grid scheduling and rescheduling methods in the GrADS project. In *Proceedings of 18th International, Parallel and Distributed Processing Symposium*, page 199a. IEEE, 2004.

[26] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 131–140. Springer, 2004.

[27] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, et al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003.

[28] E. Deelman, G. Singh, M.H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[29] G. Dong, R. Hull, B. Kumar, J. Su, and G. Zhou. A framework for optimizing distributed workflow executions. *Research Issues in Structured and Semistructured Database Programming*, pages 152–167, 2000.

[30] J. Dongarra. The linpack benchmark: An explanation. In *Supercomputing*, pages 456–474. Springer, 1988.

[31] BE Doty and JL Kinter III. Geophysical data and visualization using grads. *Visualization Techniques Space and Atmospheric Sciences, EP Szuszczewicz and Bredekamp, Eds., NASA*, pages 209–219, 1995.

[32] K.K. Droegemeier, V. Chandrasekar, R. Clark, D. Gannon, S. Graves, E. Joseph, M. Ramamurthy, R. Wilhelmson, K. Brewster, B. Domenico, et al. Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. In *20th Conf. on Interactive Information Processing Systems for Meteorology, Oceanography, and Hydrology*. AMS, 2004.

[33] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[34] D. Erwin and D. Snelling. Unicore: A grid computing environment. In Rizos Sakellariou, John Gurd, Len Freeman, and John Keane, editors, *Euro-Par 2001 Parallel Processing*, volume 2150 of *Lecture Notes in Computer Science*, pages 825–834. Springer Berlin / Heidelberg, 2001.

[35] Esper. Event Processing LanguageProject. 2003.

[36] O. Etzion and P. Niblett. *Event Processing in Action*. Manning, 2011.

[37] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[38] K. Ezekiel and F. Marir. Monitoring information and data flows using triggers in a dynamic workflow environment. In *2nd International Conference on Information Technology: Research and Education, 2004. ITRE 2004.*, pages 175–178. IEEE, 2005.

[39] M.J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 2009.

[40] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35:37–46, June 2002.

[41] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200, 2001.

[42] T. Fountain, S. Tilak, P. Hubbard, P. Shin, and L. Freudinger. The Open Source DataTurbine Initiative: Streaming data middleware for environmental observing systems. In *International Symposium on Remote Sensing of Environment*. ISRSE, 2009.

[43] G. Fox, G. Aydin, H. Bulut, H. Gadgil, S. Pallickara, M. Pierce, and W. Wu. Management of real-time streaming data Grid services. *Concurrency and Computation: Practice and Experience*, 19(7):983–998, 2007.

[44] G. Fox, H. Gadgil, S. Pallickara, M. Pierce, R.L. Grossman, Y. Gu, D. Hanley, and X. Hong. High Performance Data Streaming in Service Architecture. Technical report, Computer Science Department, Indiana University., July 2004.

[45] G. Fox, S.H. Ko, M. Pierce, O. Balsoy, J. Kim, S. Lee, K. Kim, S. Oh, X. Rao, M. Varank, et al. Grid services for earthquake science. *Concurrency and Computation: Practice and Experience*, 14(6-7):371–393, 2002.

[46] J. Frey. Condor DAGMan: Handling inter-job dependencies, 2002.

[47] B. Gedik, H. Andrade, K.L. Wu, P.S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134. ACM, 2008.

[48] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3(2):119–153, 1995.

[49] T. Glatard and J. Montagnat. Implementation of Turing machines with the Scufl data-flow language. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 663–668. IEEE, 2008.

[50] A. Goderis, C. Brooks, I. Altintas, E. Lee, and C. Goble. Composing different models of computation in kepler and ptolemy ii. *Computational Science–ICCS 2007*, pages 182–190, 2007.

[51] L. Golab and M.T.
"Ozsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[52] M.K. Govil and M.C. Fu. Queueing theory in manufacturing: A survey. *Journal of manufacturing systems*, 18(3):214, 1999.

[53] RL Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 205–217. ACM, 1972.

[54] T. Gunarathne, C. Herath, E. Chinthaka, and S. Marru. Experience with adapting a WS-BPEL runtime for eScience workflows. In *Proceedings of the 5th Grid Computing Environments Workshop*, pages 1–10. ACM, 2009.

[55] W.H. Heiss, D.L. McGrew, and D. Sirmans. Nexrad-next generation weather radar (wsr-88d). *Microwave Journal*, 33:79, 1990.

[56] C. Herath, J. Cox, S. Perera, D. Gannon, and B. Plale. Events Processing Programming Models for Reactive e-Science Workflows. *technical report*, 2008.

[57] C. Herath and B. Plale. Streamflow - Programming Model for Data Streaming in Scientific Workflows . In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing 2010*, pages 302–311. IEEE, 2010.

[58] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[59] A.J.G. Hey, S. Tansley, and K.M. Tolle. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research, 2009.

[60] Y. Huang, A. Slominski, C. Herath, and D. Gannon. Ws-messenger: A web services-based messaging system for service-oriented grid computing. In *Proceedings of the 6th IEEE/ACM Int'l Symposium on Cluster, Cloud and Grid Computing CCGRID*, pages 1–10. ACM, 2006.

[61] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[62] J.R. Jackson. Jobshop-like queueing systems. *Management science*, 50(12):1796–1802, 2004.

[63] MP Jarnagin and Naval Weapons Laboratory (US). *Automatic machine methods of testing pert networks for consistency*. US Govt. Print. Off., 1961.

[64] AB Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[65] N. Kartha, S. Commerce, P. Yendluri, and O. Alex Yiu. Web Services Business Process Execution Language Version 2.0. *WS-BPEL TC OASIS*, 2007.

[66] F.P. Kelly. Networks of queues with customers of different types. *Journal of Applied Probability*, 12(3):542–554, 1975.

[67] B. Kienhuis and E.F. Deprettere. Modeling stream-based applications using the SBF model of computation. *The Journal of VLSI Signal Processing*, 34(3):291–300, 2003.

[68] K. Klingenstein, D. Gannon, K. Hazelton, P. Hill, C. Goble, B. Ludaescher, E. Deelman, C. Lynch, et al. Improving interoperability, sustainability and platform convergence in scientific and scholarly workflow. *Technical report, NSF and Mellon Foundation*, 2007.

[69] D.E. Knuth. The Art of Computer Programming: Fundamental Algorithms, volume 1. *Addison-Wesley,*, 5:1–4, 1973.

[70] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 144–154. ACM, 1993.

[71] P. Leach, M. Mealling, and R. Salz. RFC 4122: a Universally Unique IDentifier (UUID) URN Namespace. *Retrieved from http://www. ietf. org/rfc/rfc4122. txt*, 2005.

[72] E. LEE and S. Neuendorffer. MoML-A Modeling Markup Language in XML-Version 4.0. *ICCAD, March*, 14, 2000.

[73] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 2005.

[74] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 2002.

[75] A. Leon-Garcia. *Probability, statistics, and random processes for electrical engineering*. Prentice Hall, 2008.

[76] R.M. Lhermitte and D. Atlas. Precipitation motion by pulse doppler radar. In *Proc. Ninth Weather Radar Conf*, pages 218–223, 1961.

[77] X. Li, B. Plale, N. Vijayakumar, R. Ramachandran, S. Graves, and H. Conover. Real-time storm detection and weather forecast activation through data mining and events processing. *Earth Science Informatics*, 1(2):49–57, 2008.

[78] Y. Liu, D. Hill, A. Rodriguez, L. Marini, R. Kooper, J. Myers, X. Wu, and B. Minsker. A new framework for on-demand virtualization, repurposing and fusion of heterogeneous sensors. In *Collaborative Technologies and Systems, 2009. CTS'09. International Symposium on*, pages 54–63. IEEE, 2009.

[79] Y. Liu, N. Vijayakumar, and B. Plale. Stream processing in data-driven computational science. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 160–167. IEEE Computer Society, 2006.

[80] D.C. Luckham. *The power of events: an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001.

[81] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[82] L. Marini, R. Kooper, P. Bajcsy, and J. Myers. Supporting exploration and collaboration in scientific workflow systems. In *AGU Fall Meeting Abstracts*, volume 1, page 07. AGU, 2007.

[83] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. ICENI dataflow and workflow: Composition and scheduling in space and time. In *UK e-Science All Hands Meeting*, volume 634. IOP Publishing Ltd, 2003.

[84] A.S. McGough, J. Cohen, J. Darlington, E. Katsiri, W. Lee, S. Panagiotidi, and Y. Patel. An end-to-end workflow pipeline for large-scale grid computing. *Journal of Grid Computing*, 3(3):259–281, 2005.

[85] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington. Workflow enactment in ICENI. In *UK e-Science All Hands Meeting*, pages 894–900. IOP Publishing Ltd, 2004.

[86] C. Miceli, M. Miceli, S. Jha, H. Kaiser, and A. Merzky. Programming abstractions for data intensive computing on clouds and grids. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 478–483. IEEE Computer Society, 2009.

[87] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. The weather research and forecast model: Software architecture and performance. In *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*, pages 156–168. Citeseer, 2004.

[88] C. Moretti, J. Bulosan, D. Thain, and P.J. Flynn. All-pairs: An abstraction for data-intensive cloud computing. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11. IEEE, 2008.

[89] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*. ACM, 2003.

[90] D. Murray, J. McWhirter, S. Wier, and S. Emmerson. The integrated data viewer-a web-enabled application for scientific analysis and visualization. *Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology*, 2003.

[91] P. Neophytou, P.K. Chrysanthis, and A. Labrinidis. Towards Continuous Workflow Enactment Systems. In *Collaborative Computing: Networking, Applications and Worksharing: 4th International Conference, CollaborateCom 2008, Orlando, FL, USA, November 13-16, 2008, Revised Selected Papers*, page 162. Springer-Verlag New York Inc, 2009.

[92] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Greenwood, T. Carver, M.R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:3045–3054, November 2004.

[93] T. Oinn, P. Li, D.B. Kell, C. Goble, A. Goderis, M. Greenwood, D. Hull, R. Stevens, D. Turi, and J. Zhao. Taverna/my Grid: Aligning a Workflow System with the Life Sciences Community. *Workflows for e-Science*, pages 300–319, 2007.

[94] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[95] S. Pallickara and G. Fox. Naradabrokering: A distributed middleware framework and architecture for enabling durable peer-to-peer grids. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 41–61. Springer-Verlag New York, Inc., 2003.

[96] S. Perera, C. Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, and G. Daniels. Axis2, middleware for next generation web services. In *Web Services, 2006. ICWS'06. International Conference on*, pages 833–840. IEEE, 2006.

[97] S. Perera, S. Marru, and C. Herath. Workflow Infrastructure for Multi-scale Science Gateways. In *Proceedings of 1st TeraGrid Conference, June*, 2008.

[98] B. Plale. Framework for bringing data streams to the grid. *Scientific Programming*, 12(4):213–223, 2004.

[99] B. Plale, B. Cao, C. Herath, and Y. Sun. Data provenance for preservation of digital geoscience data. 2008.

[100] B. Plale, D. Gannon, J. Brotzge, K. Droegemeier, J. Kurose, D. McLaughlin, R. Wilhelmson, S. Graves, M. Ramamurthy, R.D. Clark, et al. CASA and LEAD: Adaptive cyberinfrastructure for real-time multiscale weather forecasting. *Computer*, 39(11):56–64, 2006.

[101] B. Plale, D. Gannon, Y. Huang, G. Kandaswamy, S.L. Pallickara, and A. Slominski. Cooperating services for data-driven computational experimentation. *Computing in science and engineering*, pages 34–43, 2005.

[102] B. Plale, C. Herath, and R. Ramachandran. Unified Programming Model for Instrument-driven e-Science Workflows? 2008.

[103] B. Plale and K. Schwan. Dynamic querying of streaming data with the dquob system. *IEEE Transactions on Parallel and Distributed Systems*, pages 422–432, 2003.

[104] Beth Plale and Nithya Vijayakumar. Evaluation of rate-based adaptivity in asynchronous data stream joins. *Proceedings 19th IEEE International Parallel and Distributed Processing Symposium, 2005*, 1:69b, 2005.

[105] FIX Protocol. The financial information exchange protocol (fix), version 4.3. *At: http://www. fixprotocol. org/specification/fix-43-pdf. zip*, 2001.

[106] I. Raicu, I. Foster, A. Szalay, and G. Turcu. AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis. In *TeraGrid Conference*. NASA GSRP Final Report, 2006.

[107] L. Ramakrishnan and B. Plale. A multi-dimensional classification model for scientific workflow characteristics. In *Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science*, pages 1–12. ACM, 2010.

[108] H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[109] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[110] S. Shirasuna. *A dynamic scientific workflow system for the Web services architecture*. Ph. D. Thesis, Indiana University, 2007.

[111] W. Stallings. Queuing analysis notes. *http://www.electronicsteacher.com/download/queuing-analysis.pdf*, 2000.

[112] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana applications within grid computing and peer to peer environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[113] I.J. Taylor. *Workflows for e-science: scientific workflows for grids*. Springer-Verlag New York Inc, 2007.

[114] D. Turi, P. Missier, C. Goble, D. De Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *e-Science and Grid Computing, IEEE International Conference on*, pages 441–448. IEEE, 2008.

[115] Nithya N. Vijayakumar and Beth Plale. Dynamic filtering and mining triggers in mesoscale meteorology forecasting. In *In International Geoscience and Remote Sensing Symposium*, 2006.

[116] N.N. Vijayakumar, B. Plale, R. Ramachandran, and X. Li. Dynamic filtering and mining triggers in mesoscale meteorology forecasting. In *International Geoscience and Remote Sensing Symposium*. IEEE, 2006.

[117] S. Vinoski. Web services notifications. *Internet Computing, IEEE*, 8(2):86–90, 2004.

[118] M. Wand. Type inference for record concatenation and multiple inheritance* 1. *Information and Computation*, 93(1):1–15, 1991.

[119] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the ASKALON grid environment. *ACM SIGMOD Record*, 34(3):56–62, 2005.

[120] N. Wilkins-Diehr, D. Gannon, G. Klimeck, S. Oster, and S. Pamidighantam. Teragrid science gateways and their impact on science. *Computer*, 41(11):32–41, 2008.

[121] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.

[122] L. Yi, C. Moretti, S. Emrich, K. Judd, and D. Thain. Harnessing parallelism in multicore clusters with the all-pairs and wavefront abstractions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 1–10. ACM, 2009.

[123] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM Sigmod Record*, 34(3):44–49, 2005.

[124] J. Yu, R. Buyya, and C.K. Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *First International Conference on e-Science and Grid Computing, 2005.*, pages 8–147. IEEE, 2006.

[125] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14. USENIX Association, 2008.

[126] W. Zhang, J. Cao, L. Liu, and C. Wu. Grid Data Streaming. *Grid Computing Research Progress, J. Wong, Editor*, 2008.

[127] D. Zinn, S. Bowers, S. Kohler, and B. Ludascher. Parallelizing XML data-streaming workflows via MapReduce. *Journal of Computer and System Sciences*, 76(6):447–463, 2010.