CONTINUOUS WORKFLOWS: FROM MODEL TO ENACTMENT SYSTEM

by

Panayiotis Neophytou

B.Sc. in Computer Science, University of Cyprus, 2003

M.Sc. in Computer Science, University of Pittsburgh, 2012

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2013

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Panayiotis Neophytou

It was defended on

December 17, 2012

and approved by

Panos K. Chrysanthis, Professor, University of Pittsburgh

Alexandros Labrinidis, Associate Professor, University of Pittsburgh

Kirk Pruhs, Professor, University of Pittsburgh

Calton Pu, Professor, Georgia Tech

Dissertation Advisors: Panos K. Chrysanthis, Professor, University of Pittsburgh,

Alexandros Labrinidis, Associate Professor, University of Pittsburgh

CONTINUOUS WORKFLOWS: FROM MODEL TO ENACTMENT SYSTEM

Panayiotis Neophytou, PhD

University of Pittsburgh, 2013

Workflows are actively being used in both business and scientific domains to automate processes and facilitate collaboration. A workflow management (or enactment) system (WfMS) defines, creates and manages the execution of workflows on one or more workflow engines, which are able to interpret workflow definitions, allocate resources, interact with workflow participants and, where required, invoke the needed tools (e.g., databases, job schedulers, etc.) and applications. Traditional WfMSs and workflow design processes view the workflow as a one-time interaction with the various data sources, i.e., when a workflow is invoked, its steps are executed once and in-order. The fundamental underlying assumption has been that data sources are passive and all interactions are structured along the request/reply (query) model. Hence, traditional WfMS cannot effectively support business or scientific monitoring applications that require the processing of data streams such as those generated by sensing devices as well as mobile and web applications.

It is the hypothesis of this dissertation that *Workflow Management Systems can be extended to support data stream semantics to enable monitoring applications. This includes the ability to apply flexible bounds on unbounded data streams and the ability to facilitate on-the-fly processing of bounded bundles of data (window semantics).* To support this hypothesis this dissertation has produced new specifications, a design, an implementation and a thorough evaluation of a novel Continuous Workflows (CWf) model, which is backwards compatible with currently available workflow models. The CWf model was implemented in a CONtinuous workFLow ExeCution Engine, CONFLuEnCE, as an extension of Kepler, which is a popular scientific WfMS. The applicability of the CWf model in both scientific and business applications was demonstrated by utilizing CONFLuEnCE in Astroshelf to support live annotations (i.e., monitoring of astronomical data), and to support supply chain monitoring and management. The implementation of CONFLuEnCE led to the realization that different applications have different performance requirements and hence an integrated workflow scheduling framework is essential. Towards meeting this need, STAFiLOS, a Stream FLOw Scheduling framework for Continuous Workflows, was designed and implemented, within CONFLuEnCE. The performance of STAFiLOS was evaluated using the Linear Road Benchmark for continuous workflows.

Keywords Continuous Workflows, Data Stream Processing, Event processing, Workflow Patterns, Workflow Scheduling, Workflow Management Systems.

TABLE OF CONTENTS

PREFACE xii

1.0 INTRODUCTION 1

1.1 Motivation and Challenge 1

1.2 Hypothesis and Methodology 3

1.3 Contributions 5

1.4 Outline 6

2.0 BACKGROUND AND RELATED WORK 8

2.1 Workflows 8

2.2 Data Streams and Data Stream Management Systems 12

2.3 Data-flow and Communication Patterns 13

2.4 Current support for data streams in Workflow Systems 16

2.5 Workflow Scheduling 17

2.6 Workflow Benchmarking 18

2.7 Summary 19

3.0 CONTINUOUS WORKFLOW MODEL 20

3.1 Towards the Continuous Workflow model 20

3.2 Definition of the Continuous Workflow Model 23

3.2.1 Waves 23

3.2.2 Windows 24

3.2.2.1 Window Specifications 24

3.2.2.2 Window operator example 29

3.2.3 Push communication 29

3.3 Continuous Workflow Patterns 31
3.4 Model Evaluation 33
3.4.1 CWf Patterns 33
3.4.2 CWf Model Expressibility and Flexibility 34
3.5 Summary 35
4.0 CONFLuEnCE: CONtinuous workFLow ExeCution Engine 37
4.1 Overview 37
4.2 Kepler's actor-oriented modeling 38
4.3 Continuous Workflow Director 41
4.4 Windowed Receiver 42
4.5 Time Keeping 44
4.6 Push communication 45
4.7 Evaluation 47
4.7.1 Supply Chain Management 48
4.7.2 AstroShelf 50
4.8 Summary 53
5.0 STAFiLOS: STreAm FLOw SCHEDULING FOR CONTINUOUS WORKFLOWS 54
5.1 Scheduler Framework in CONFLuEnCE 54
5.2 Existing schedulers/directors in PtolemyII/Kepler 55
5.3 STAFiLOS Design 58
5.4 Implemented CWF Schedulers 62
5.4.1 The Quantum Priority Based Scheduler (QBS) 63
5.4.2 Round-Robin Scheduler (RR) 65
5.4.3 Rate Based Scheduler (RB) 66
5.5 Evaluation 68
5.5.1 Linear Road Benchmark 68
5.5.1.1 Accident Detection and Notification 70
5.5.1.2 Segment Statistics 71
5.5.1.3 Toll Calculation and Notification 73
5.5.2 Experimental Setup 75

5.5.3 Experimental Results 75
5.6 Summary 78
6.0 CONCLUSIONS AND FUTURE WORK 83
6.1 Summary of Contributions 83
6.2 Impact of this Dissertation 84
6.3 Future Work 85
APPENDIX. MULTI-WORKFLOW EXECUTION 87
BIBLIOGRAPHY 90

LIST OF TABLES

1 Taxonomy of Directors found in Kepler (first group) and ProlemyII (second group) as well as our PNCWF Director (third group) 59

2 Experimental setup 76

LIST OF FIGURES

1 CONFLuEnCE Ecosystem 4

2 The contributions of this dissertation. 6

3 Workflow Control Patterns 10

4 Continuous Workflow enabling architecture 11

5 Data-flow Patterns 14

6 Communication Patterns as shown in [20] 15

7 Abstract Workflow example 21

8 Size and Step for event-based window semantics. The bottom series of boxes represent the stream of events. In this example the size is 5 events and the step is 3 events. The windows produced are represented by the boxes stacked above the queue 25

9 Window operator example 30

10 (a) CWP1: Sequential aggregate and (b) CWP2: Stream-join 31

11 Timed Petri net of “repeat cause-one effect” Supply Chain pattern 34

12 Continuous workflow of “repeat cause-one effect” Supply Chain pattern 35

13 Implementation a continuous workflow for Supply Chain Management reactive application 39

14 The semantics of component interaction is determined by a director, which controls execution and supplies the objects (called receivers) that implement communication 40

15 Modified Kepler form for configuring actor ports. On this form the workflow designer can define in freeform text the size and step of the window associated with specific ports. Shaded cells denote non-editable parameters 43

16 The generic WindowedReceiver as is implemented in CONFLuEnCE 43

17 Timestamp preservation inside CONFLuEnCE 46
18 Implementation a continuous workflow for Supply Chain Management reactive application 47
19 Supply Chain Management application UI: The customer's panel 49
20 Supply Chain Management application UI: Company manager's panel 49
21 High-level design of a continuous workflow for the Astroshelf collaboration platform 51
22 Event flow in the STAFiLOS scheduling framework 60
23 The STAFiLOS scheduler framework in CONFLuEnCE 62
24 The Linear Road Benchmark top level workflow 69
25 The Linear Road Benchmark Stopped Cars Detection sub-workflow 70
26 The Linear Road Benchmark Accident Detection sub-workflow 71
27 The Linear Road Benchmark Accident Notification sub-workflow 72
28 The Linear Road Benchmark car average speed sub-workflow (*Avgsv*) 73
29 The Linear Road Benchmark car count sub-workflow (*cars*) 74
30 Linear Road workload of 0.5 highways 79
31 Response Times of the RR scheduler using varying basic quantum values 80
32 Response Times of the QBS scheduler using varying basic quantum values 81
33 Response Times of all the main schedulers 82
34 Multi-workflow execution framework 88

PREFACE

I would like to thank my advisors, Panos K. Chrysanthis and Alexandros Labrinidis, for their guidance and support throughout my PhD studies. Their knowledge, intelligence, dedication, and personal integrity have helped me set higher standards for myself on the academic, professional, and personal levels. This work would never have been possible without their inspiring advice and continuous encouragement and guidance. I would also like to thank the rest of the members of my committee for their extreme patience in the face of numerous obstacles.

Acknowledgment is also due to the NSF for supporting my research through grants: NSF grant IIS-053453, NSF career award IIS-0746696 and NSF grant OIA-1028162.

I also appreciate the collaborative work I have had with several members of the ADMT Laboratory. at Pitt, especially, my work with Mohamed Sharaf, Shenoda Guirguis, Lory Al Moakar and Christine Chung.

I would also like to thank the staffers at the Computer Science department for all their prompt support and help with any kind of issues I have run into throughout the years.

I have also enjoyed the support of many friends I have made during my stay in Pittsburgh. I would especially like to mention the Greek Mafia team for being there through all the sleepless nights all working together and pushing each other. I would especially like to thank Roxana Gheorghiu and Mohamed Sharaf for their support both as friends as well as colleagues.

Finally, the most appreciation is due to my family: my mother Rita, my father Savvas and my sister-in-law Ody. Special thanks go to my brother Neophytos who has not only stood by me but also guided me throughout my studies from his own experience as a grad student. With gratitude, I dedicate this work to them.

1.0 INTRODUCTION

1.1 MOTIVATION AND CHALLENGE

Workflows are being extensively used by enterprises and scientists to automate their processes, optimize resource utilization as well as productivity and also to enable collaboration. Workflows (also called business processes) in computing[1] were first proposed in the mid 80's to capture and automate an organization's business intelligence that achieves its mission, mostly in a centralized environment [71, 26, 35, 14]. A workflow consists of a sequence of connected steps or tasks, representing the execution of a complex process. Within workflows, tasks are performed cooperatively by either human or computational agents in accordance with their roles in the organizational hierarchy. The proliferation of Business-to-Business (B2B) and Business-to-Client (B2C) interactions, as a result of the booming growth of the internet, and the need for managing greater amounts of data and requests, made workflows and workflow management systems the popular solution to specify and handle these interactions [25]. Furthermore, the inter-business connectivity provided by the internet further enabled B2B interactions for outsourcing and facilitating collaborations beyond the boundaries of a single enterprise, for example in establishing virtual enterprises [16, 7, 51].

The scientific domain did not move towards the use of workflows until recently, since the lab notebook has been viewed as being sufficient in keeping track of the experiment processes and results (i.e., small experiments with few data). This move towards workflows was necessitated by the shift towards data-driven scientific discovery (or eScience) where theory, experiment, and simulation are combined to explore complex phenomena, often referred to as the Fourth Scientific Paradigm [27]. Today, the new highly sensitive instruments and new methods for simulation gen-

[1]Interesting to note that modern-day business processes are based on some of Frederick Winslow Taylor's [1856-1915] principles of management.

erate tremendous amounts of data that need to be analyzed by humans, primarily through the use of computer tools. Workflows make the task of managing (e.g., cleaning, staging), transforming and processing these data, much easier and more effective, while enabling *in-silico* experiments [58, 72]. This led to the development of scientific workflows and also advanced scientific workflow management systems, such as Kepler [36], Taverna [47], and Triana [15].

Although business and scientific workflows have different requirements, both are based, more or less, on the same principles (as defined in [68]). Their main difference is that business workflows focus on *control-flow* while scientific workflows focus on *data-flow*. In control-flow based workflows, the workflow semantics are towards which process can be executed before another, and data access is handled separately. In data-flow based workflows, workflow semantics express the data communication pattern among data producer and data consumer tasks. Both models have advantages and disadvantages, which have to do with the patterns they can handle. For example, conditional execution and exception handling is easier to implement in control-flow based systems, but pipelined-parallel execution is easier to model in data-flow based systems.

The current workflow models, irrespective of whether they support control-flow or data-flow semantics or both, are sufficiently powerful in automating processes, integrating and orchestrating the available resources, processing of large static data sets, keeping track of the provenance of data, enabling reusability, and providing easy to use visual languages for the user's convenience. However, current workflow models and workflow management or enactment systems (WfMS) view a workflow as a one-time interaction with the various data sources, i.e., when a workflow is invoked, its steps are executed once and in-order. The fundamental underlying assumption has been that data sources are passive and all interactions are structured along the request/reply (query) model, i.e., adopting the traditional Database Management System (DBMS) query paradigm. Hence, traditional WfMSs and languages cannot effectively support business or scientific monitoring applications that require the processing of *data streams*. Data streams are ordered sequences of data items (or tuples) that are continuously produced at high rates. Examples of data streams in the business domain include point-of-sale updates, used in applications such as on-the-fly supply chain management, web-search streams and social network updates for on-line marketing strategy decision making applications, etc. In the scientific domain, examples include sensor network data streams (on environmental readings) used in remotely established underwater labs to monitor sea life and conditions in real-time [17], and environmental analysis applications that collect and analyze sensor data to support detection of air and water pollution, etc. [27].

This clearly suggests the need for a new workflow model and workflow management techniques that efficiently handle data streams. Such a workflow model should be designed to transform business intelligence as well as scientific processes from the traditional back-office, report-oriented, historical analysis platform, and to become an enabler for delivering data-intensive, real-time analytics that transform business operations and knowledge discovery in the modern "smart" environment. Traditional database management systems have faced the same challenge in processing data streams which led to a data processing paradigm shift, where Continuous Queries (CQs) [10, 5, 12, 60, 13] are stored and continuously process unbounded data streams looking for data that represent interesting events as data arrives, *on-the-fly*. Even though CQs have static configurations and cannot facilitate user interactions as in the case of workflows, the advanced data processing techniques, such as *window-based processing* developed as part of Data Stream Management Systems (DSMSs), can act as a starting point for the development of an appropriate *Continuous Workflow model*, which is the fundamental objective of this dissertation.

1.2 HYPOTHESIS AND METHODOLOGY

We have identified a major shortcoming in supporting of monitoring applications in the existing Workflow design and execution technologies and our hypothesis is that:

> *Workflow Management Systems both in the scientific and business domains can be extended to support streaming semantics to enable monitoring applications. This includes the ability to apply flexible bounds on unbounded data streams and the ability to facilitate on-the-fly processing of bounded bundles of data (window semantics).*

To support this hypothesis this dissertation provides an extended workflow model and a supporting architecture to satisfy the need for handling data streams published and delivered asynchronously from multiple sources, including DSMSs. We call this class of workflows, *Continuous Workflows* (CWfs). Our investigation began by identifying the limitations of currently available workflow models and exploring ways to extend them to support continuous workflows. The main

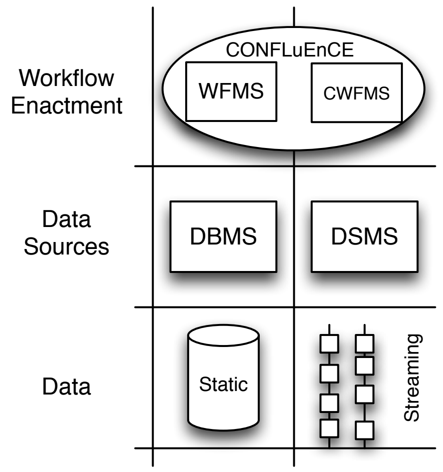


Figure 1: CONFLuEnCE Ecosystem

difference between traditional and continuous workflows is that the latter are continuously (i.e., always) active and continuously integrating and reacting on internal streams of events and external streams of updates from multiple sources, at the same time and in any part of the workflow network.

We have implemented our proposed CWf model as a prototype system, called CONFLuEnCE [43], which is short for CONtinuous workFLow ExeCution Engine. CONFLuEnCE was built on top of Kepler, an existing WfMS [36], and can integrate backwardly with traditional workflows, so that it can support both traditional data sources, such as Database Management Systems (DBMS), and data stream sources, such as Data Stream Management Systems (DSMS) (Figure 1).

Current WfMSs rely on the underlying operating system (OS) to schedule the workflow tasks and hence are oblivious to the users' quality-of-service (QoS) requirements and of their applications. To address this limitation, we developed an integrated scheduling framework within CONFLuEnCE, namely STAFiLOS (STreAm FLOw Scheduling) [44], which enables the development of suitable scheduling policies governing the execution of workflow steps.

As part of this dissertation, we have also experimented with the design and implementation of real-life business and scientific CWfs which can attest to the ease of use and applicability of our system. In particular, we use CONFLuEnCE at the core of *AstroShelf* [4], which is an astrophysics platform that enables scientists to collaboratively annotate sky objects and phenomena, as well as visualize parts of the sky using different algorithms. Another application we have implemented and demonstrated from the business domain is a Supply Chain Management System [42]. The ability to facilitate *on-the-fly* processing of data that arrives at different rates and produces results within predefined time windows mandate better resource management than traditional WfMS. This is precisely the challenge that led us into addressing this problem by means of STAFiLOS.

1.3 CONTRIBUTIONS

In this dissertation, we shift from the traditional step-wise, request-response workflow execution model to a continuous execution model, in order to handle data streams published and delivered asynchronously from multiple sources. This paradigm shift is necessitated by data-intensive, real-time analytics that transform the scientific process and business operations in the modern world of Big Data, and its Volume/Velocity/Variety characteristics.

In summary, the three key contributions of this dissertation which are schematically depicted in Figure 2 are:

1. **CWf**: A formal definition of the Continuous Workflow model [41], specifying the semantics and the characteristics of a WfMS which is capable of carrying out streaming data processing by providing features such as window semantics on data queues, wave-based bundles, support for push-based communications, and proposed new CWF patterns, while at the same time support the current workflow reference model. We evaluate our CWf model by comparing its expressibility and applicability with the Timed Petri-nets workflow model;

2. **CONFLuEnCE**: A fully functional continuous workflow execution engine [43], which fulfills all the requirements defined in the CWf model. We evaluate CONFLuEnCE by developing reference applications built on top of this execution engine, in particular, the *AstroShelf* astrophysics research platform [45] and a Supply Chain Management application [42];

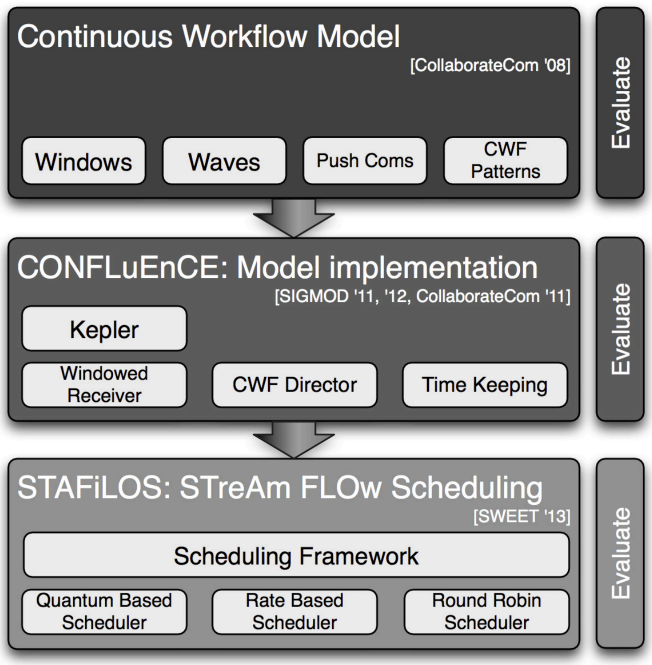


Figure 2: The contributions of this dissertation.

3. **STAFiLOS**: A scheduling framework for implementing scheduling algorithms designed to meet QoS requirements such as response time. We evaluate the suitability and scalability of our framework by implementing three schedulers: Quantum-Based, Rate-Based and Round-Robin and measure their performance by comparing it to a thread-based OS native scheduler, using the Linear Road Benchmark.

1.4 OUTLINE

The rest of this dissertation is structured as follows: In Chapter 2 we provide the necessary background for Workflows and Data streams. We describe the currently available systems and the aspects of these systems that are related to our work, specifically scheduling, QoS requirements and provisioning, and metrics used. In Chapter 3 we describe our Continuous Workflow Model (CWf). In Chapter 4 we describe how our CWf model has been implemented into a prototype, called CONFLuEnCE. Afterwards, we complete the description of our system, by providing the details of our Continuous Workflow Scheduling approach within CONFLuEnCE named STAFi-LOS, in Chapter 5. Finally this dissertation is concluded in Chapter 6 by a guideline for future improvements and feature enhancements of the model and the system, as well as a summary of the overall work.

2.0 BACKGROUND AND RELATED WORK

In this chapter, we first provide the basis of the currently adopted workflow reference model. Then we examine how Workflow management systems are being used as data processing pipelines and discuss their ability to support streaming data processing (which we will elaborate in detail in Section 3.1). Finally we present the basics of job scheduling in generic systems as well as specifically for data stream processing systems, with the goal of optimizing Quality of Service (QoS) metrics.

2.1 WORKFLOWS

As we have already mentioned in the introduction, the workflow model initially emerged from the business domain, and was formally defined by the Workflow Management Coalition (WfMC) in [68]. Broadly speaking, a workflow (also referred to as workflow process) is defined as the automation of a business or scientific process, in whole or part, during which documents, information or tasks are passed for action, from one participant (human or machine) to another, according to a set of procedural rules.

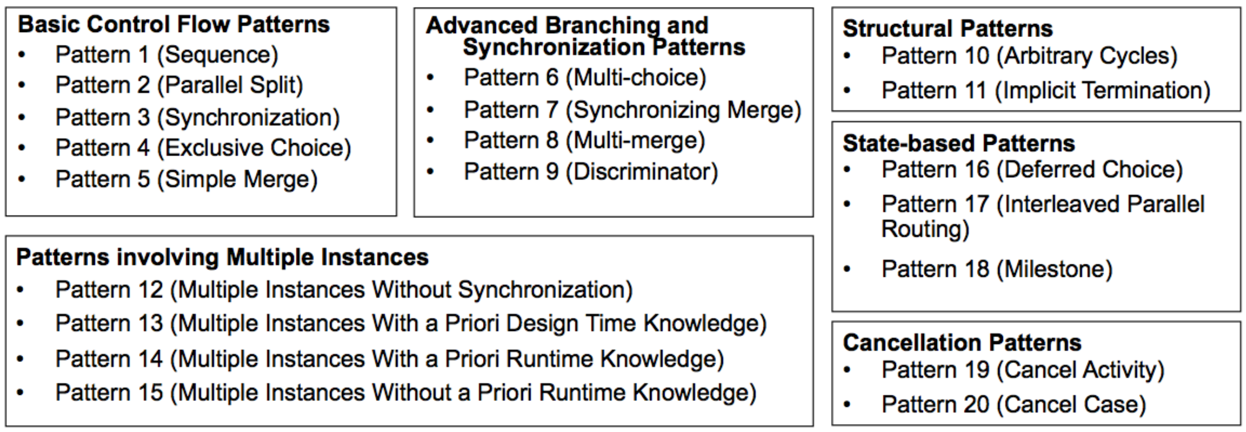
Definition 1. *A* **workflow activity** *is a description of a piece of work that forms one logical step within a process. An activity may be a manual activity, which does not support computer automation, or an automated activity. A workflow activity requires human and/or machine resources(s) to support process execution; if a human resource is required, an activity is allocated to a workflow participant (person). A workflow activity is specified in terms of a* name, preconditions, postconditions, actions, rules of exception handling, completion *and* temporal constraints.

Definition 2. *A* **workflow** *consists of a sequence of connected workflow activities, representing the execution of a complex process. Every workflow specification formalism is built around a set of control flow relationships and concepts, such as those defined in [68]. Examples include simple one-to-one precedence constraints to denote* sequential *execution, or* OR *and* AND *relationships to denote* parallel *execution. These are subdivided into* OR-split *and* AND-split *to specify branching, and* OR-join *and* AND-join *to specify convergence to initiate the next activity in the workflow. A* workflow process *can be defined by set of sub-processes which form part of the overall process. Multiple levels of sub processes may be combined to form a workflow hierarchy.*

Definition 3. *A* **workflow management or enactment system** *(WfMS) defines, creates and manages the execution of workflows on one or more workflow engines, which are able to interpret workflow definitions, allocate resources, interact with workflow participants and, where required, invoke the needed tools (e.g., databases, job schedulers etc.) and applications.*

A comprehensive study presented in [64] enumerates twenty control patterns which are required by workflow applications. A pattern "is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts" [54]. These twenty patterns studied in [64] include more complex control structures, than the ones described by WfMC [68], such as XOR-split, Differed Choice, Multiple Instances etc. These help to define a workflow model in more detail and down to the specific workflow requirements it can support. The study also elaborates on which of these patterns could be realized in workflow management systems and languages, available at the time of the study. Some of the patterns mentioned cannot be realized by these systems because their design did not take them into consideration. They also proposed a new workflow language called YAWL [63], that is Petri-net based and is able to express all twenty patterns. A complete list of all twenty patterns is shown in Figure 3.

Definition 4. **Workflow events** *are distinguished into internal and external events. External events, are relevant input workflow data, pushed into the workflow as a response to a request, from applications, users, databases and other entities external to the workflow. Internal events are workflow control data, as defined in [68], but limited to internally exchanged data between activities. This does not include the engine state data stored in the WfMS database (which are workflow*



Basic Control Flow Patterns
- Pattern 1 (Sequence)
- Pattern 2 (Parallel Split)
- Pattern 3 (Synchronization)
- Pattern 4 (Exclusive Choice)
- Pattern 5 (Simple Merge)

Advanced Branching and Synchronization Patterns
- Pattern 6 (Multi-choice)
- Pattern 7 (Synchronizing Merge)
- Pattern 8 (Multi-merge)
- Pattern 9 (Discriminator)

Structural Patterns
- Pattern 10 (Arbitrary Cycles)
- Pattern 11 (Implicit Termination)

State-based Patterns
- Pattern 16 (Deferred Choice)
- Pattern 17 (Interleaved Parallel Routing)
- Pattern 18 (Milestone)

Patterns involving Multiple Instances
- Pattern 12 (Multiple Instances Without Synchronization)
- Pattern 13 (Multiple Instances With a Priori Design Time Knowledge)
- Pattern 14 (Multiple Instances With a Priori Runtime Knowledge)
- Pattern 15 (Multiple Instances Without a Priori Runtime Knowledge)

Cancellation Patterns
- Pattern 19 (Cancel Activity)
- Pattern 20 (Cancel Case)

Figure 3: Workflow Control Patterns

and runtime meta-data that involve keeping track of the workflow instances, representation of the dynamic state of the workflow system etc.)

Usually internal events mark the completion of an activity and signal the execution of the next one. They are also referred to as *tokens*, which may carry with them some information.

Definition 5. *A* **workflow request** *is the initiating event of a workflow. Once it is received by the WfMS it creates a new instance of the workflow. The request includes relevant data and constraints defined by the requester. This is the first piece of information being fed to a workflow instance.*

Most workflow languages model workflows either as State charts or Petri nets. Figure 4 represents the state chart of a vacation trip booking workflow from [31], where AND-splits (and AND-joins) are implicit when more than one arrow originates from (or is coincident to) a node. For example, *Get Input* represents an AND-split and *Make Trip Decission* represents an AND-join. OR-splits and OR-joins are depicted with arrows annotated with selection conditions. For example, *Make Payments* represents an OR-split with condition **S** (Success) and **F** (Faillure). There is also the case where conditions are not mutually exclusive and more than one branches is activated. The workflow patterns observed in this workflow, are the basic control patterns WP1-WP5 defined in [64], shown as Paterns 1-5 in Figure 3. One can also discern some activities being defined as sub-processes.

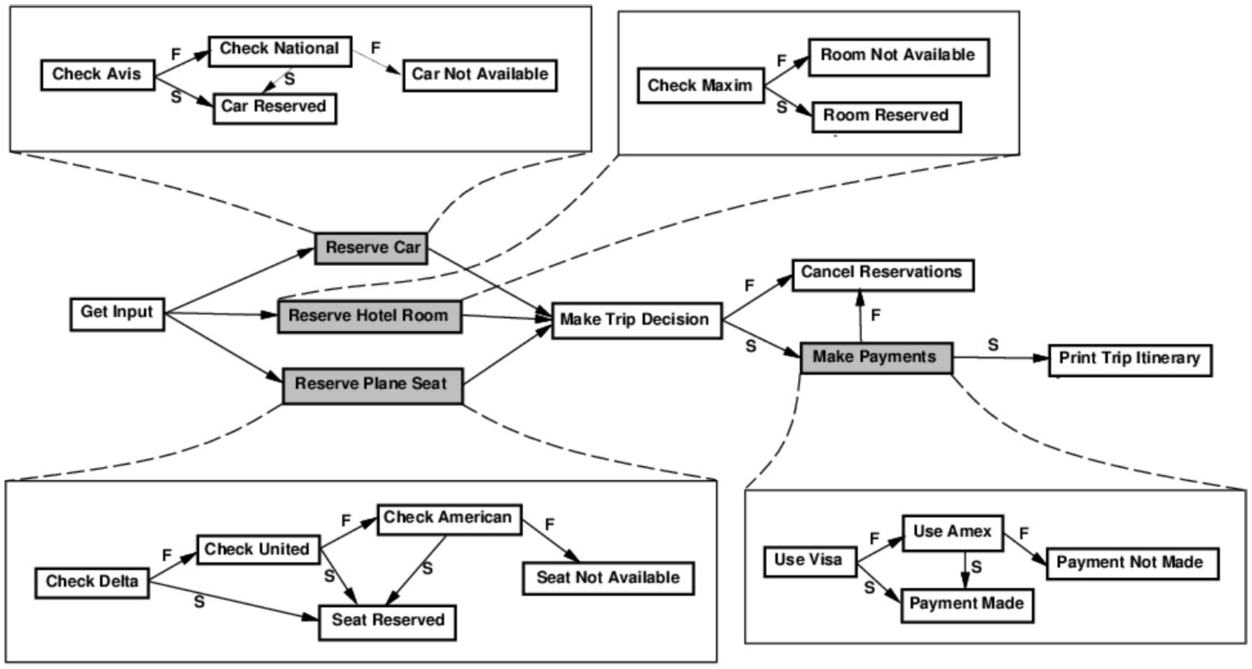


Figure 4: Continuous Workflow enabling architecture

Regarding the execution of activities, according to the transition definition in [68], any two activities in the same sequence cannot run in parallel. That means that if we have two activities A and B where A comes before B in a sequence, then B cannot start running (even on partial results from A) unless A is completely terminated.

Most recent workflow enactment/management systems orchestrate the interactions among activities within a workflow along the lines of web services [65]. Several business process modeling languages have been designed to capture the logic of a composite web service, including WSCI [66], BPML [9], BPEL4WS (with the latest update WS-BPEL 2.0 [46]), BPSS [61] and XPDL [67].

In the scientific workflows domain, the model is quite similar but with a few fundamental differences, noted in [37]. While in business workflows control flow is explicitly defined during design time, in scientific workflows the activation of tasks is implicitly controlled by the availability of data at the inputs of the tasks. This makes the execution model more relaxed and can accommodate parallel execution of tasks and thus pipelined processing of data.

2.2 DATA STREAMS AND DATA STREAM MANAGEMENT SYSTEMS

As mentioned in the Introduction, Continuous Queries (CQs) have been proposed to process continuously arriving data. That is, in the context of a Data Stream Management Systems (DSMSs) [10, 5, 12, 60, 13, 28, 39, 59], users of monitoring applications register Continuous Queries which continuously process unbounded data streams looking for data that represent events of interest to the end user. DSMSs are designed to efficiently handle such large and bursty volumes of data and large number of continuous queries.

Each CQ is usually expressed in an SQL-like format and is translated by the DSMS' query optimizer into a set of interconnected operators. Each operator's output is another operator's input (except the source operator which brings data in, and the output operator which is the query's output). Each operator's input is attached to a queue. In order to process the unbounded datastreams CQs employ window semantics [49] on the input queues of their operators, wherever is necessary. Such operators, which are called *Window Operators*, were developed to facilitate both CQ joins and aggregations [23, 32, 33]. The operators generally emphasize on the latest data items by taking into account some sort of order, usually established through the timestamp attached to each item. A window is defined by a *SLIDE* and a *RANGE* which can be either time-based or tuple-based. The *RANGE* defines the length of the window, while the *SLIDE* defines the rate at which the window moves along the queue. For example, a CQ count aggregation with a RANGE of one hour and a slide of half an hour would report every half hour the count of the last hour.

The use of DSMSs in time-critical monitoring applications has necessitated the need to focus on performance issues. Such performance can be captured by means of Quality of Sevice (QoS) measures as well as Quality of Data (QoD) measures. In [57], Sharaf et al. identify a collection of QoS metrics such as *Response Time* and *Slowdown*, and QoD metrics such as *Freshness*.

The establishment of these metrics, which fit the data-stream processing model, aided in the development of novel algorithms for query scheduling policies. One such policy, which we also consider in the context of CWfs, is the *Highest Rate Policy (HR)* that was designed to improve the average response time. It is based on the *Rate-based (RB)* [62] policy which improved the average response time of single query. HR views a network of multiple queries as a set of operators and at each scheduling point it selects the operator with the highest priority (i.e., output rate) for execution. The output rate of an operator is calculated as the number of output data items over the number of its input data, divided by the cost of the operator. Since the processing of a data item from the perspective of one operator will carry on to all of the downstream operators, the HR policy considers the Global Rate of the operator which takes into account the behavior of all the downstream operators (selectivity and cost).

DSMSs efficiently support the processing of data streams but have limited ability to support interaction involving human and computational agents. This has been the case with traditional DBMSs which led to the use of workflows.

2.3 DATA-FLOW AND COMMUNICATION PATTERNS

In data-flow oriented systems, which include WfMSs, a key characteristic are pipelined execution patterns [50] as well as parallel execution patterns [56], shown in Figure 5. These patterns play a crucial role not only in the design-time modeling but also in the optimization of run-time execution. Parallel execution patterns include: 1) *Simple parallelism*, where tasks lacking control flow dependencies are executed in parallel; and 2) *Data parallelism*, which is a form of single instruction multiple data (SIMD) parallelism.

Pipelined execution patterns, which are mostly governed by data flow include: 1) *Best-effort pipelines*, where intermediate results are dropped if downstream tasks are not ready to process them; 2) *blocking pipelines*, where the upstream tasks block until the downstream tasks are ready to process the upstream task's output; 3) *buffered pipelines* where intermediate results are buffered between tasks; 4) *superscalar pipelines*, where multiple instances of slow tasks are spawned to handle intermediate results and avoid bottlenecks; and finally 5) *streaming pipelines* where intermediate results are fed into continuously running tasks, in a producer/consumer way.

Adding streaming semantics to a pipeline of tasks requires relaxing the basic assumptions of having tasks that support a simple request-response interaction, where a task reads its input as it starts and produces output once it is finished. The goal is for tasks to have the ability to process

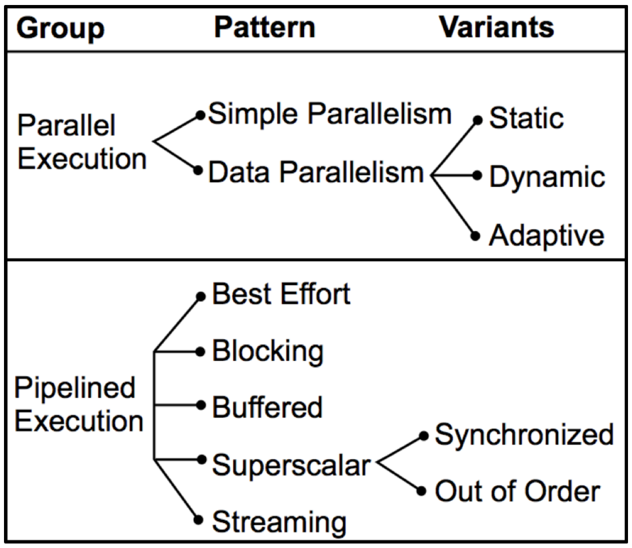


Figure 5: Data-flow Patterns

a data item as soon as it is available in its input port. Thus, the streaming semantics can be fully supported by a workflow language only if tasks allow a more flexible interaction, based on multiple requests and multiple responses [38].

Communication patterns (Figure 6), on which data-flow oriented systems rely in order to receive data, are divided into two categories: *Pull* and *Push*. In the *pull model*, the consumer gets all the data with at most one reply per request. Three patterns from [64] follow this model. (1) *Request/Reply*, where a consumer makes a request to the producer and waits for a reply before continuing execution; (2) *One-Way*, where the producer makes a request to the consumer and waits for an acknowledgment reply before continuing execution; and (3) *Asynchronous Polling*, where a consumer makes a request to a receiver and continues processing. It then periodically checks to see if a reply was sent by the producer. When it detects a reply it stops polling.

In the *push model*, the data consumer receives multiple data items per request. Two patterns that follow this model interest us: (1) *Publish/Subscribe* is a form of asynchronous communication where updates are sent by a data producer and the consumers are determined by a previous declaration of interest. The declaration of interest could also express constraints on the kind of replies each consumer is interested in. (2) *Broadcast*, is a form of asynchronous communication in which

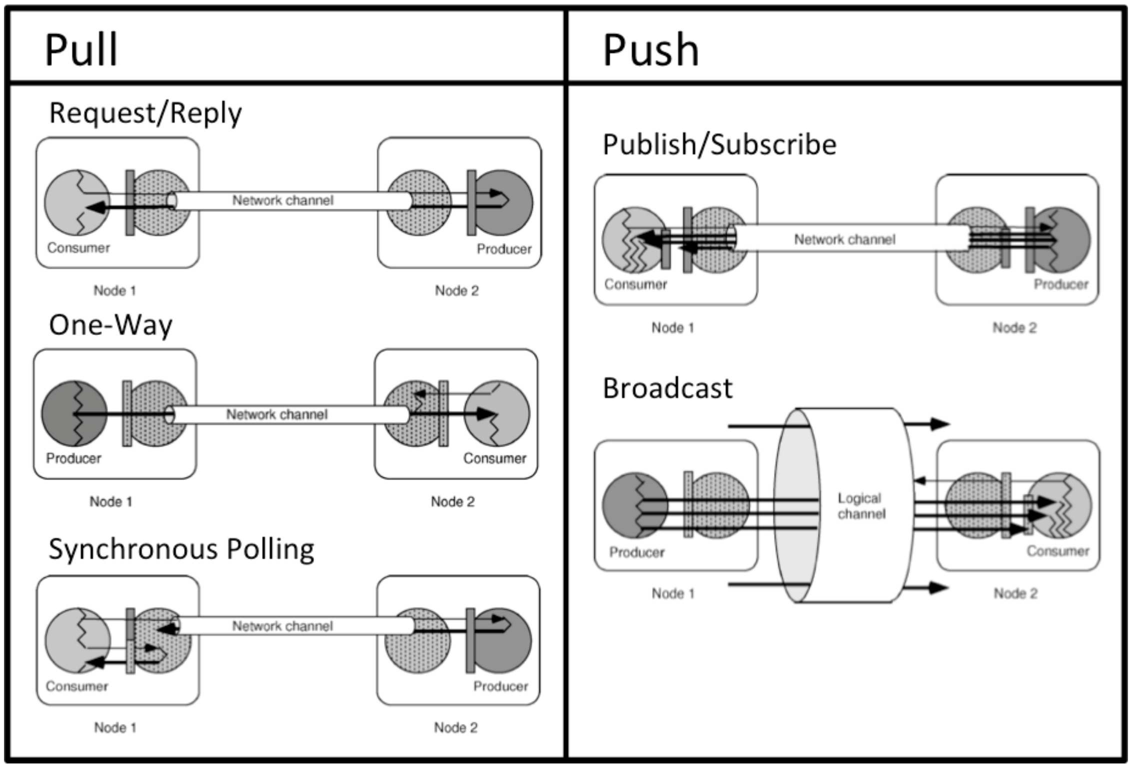


Figure 6: Communication Patterns as shown in [20]

the data is sent by the producers to all participants, having the consumer's role, in the network. Each participant determines whether the data is of interest to them by examining the content.

Parallel to our work, a survey paper which deals with the basic characteristics and requirements of scientific workflow management systems [37], makes the distinction between *stateless* and *stateful* activities. Stateless activities are oblivious to their previous invocations in the same run, as opposed to stateful activities, which are required in Models of Computation (MoCs), that allow loops in the workflow definition, and/or support pipeline parallel execution, which are aware of the previous runs and may even involve data items from previous invocations. We have also identified these requirements in our continuous workflow model [41] which we discuss in Section 3.1.

2.4 CURRENT SUPPORT FOR DATA STREAMS IN WORKFLOW SYSTEMS

Various systems have applied temporal constraints to workflows to enable some form of processing of streaming/push data. One such attempt was by using Timed Petri nets, to apply temporal constraints on events in [34]. The effort covers some cases of workflow patterns for monitoring supply chains and reacting on events such as "Out of stock" and "Order arrived". The Petri net approach is difficult to implement and although is able to capture operations on multiple events, it cannot do it for an arbitrary number of events which is known only at runtime. Also events are consumed when an activity is activated, and they have to be re-instated if there is a need to reprocess them (depending on the consumption model). These are all considerations that the designer has to make before hand. In this thesis we argue that there is a need to develop a continuous workflow model which provides more flexibility and ease of design of these patterns.

A vision for a unified data management and analytics platform called "Live Business Intelligence" (LiveBI) was presented by Castellanos et al. in [11]. LiveBI transforms business intelligence into a platform that supports data-intensive, real-time analytics needed by modern enterprises. They also stress the need for processing both streaming and stored data, structured and unstructured data, and integration of multiple pull and push data sources (much like the CONFLuEnCE ecosystem depicted in Figure 1).

Nova [48] is a system built by Yahoo! which supports stateful incremental processing. It does batch incremental processing for Yahoo's data processing use-cases, which deal with continually arriving data. It deals with data in large batches using disk-based processing, i.e., the data is first stored on a disk. Pig/Hadoop, which is the underlying workflow enactment system for Nova, lacks support of window semantics, extensibility in scheduling policies and it is constrained in the limited number of workflow patterns supported. Also, Pig/Hadoop is short of a high level visual workflow programming language, which makes it not very accessible to other domain experts (e.g., physicists and astronomers). Workflows defined in Pig/Hadoop are also limited in terms of employing user activities in the workflow process.

A similar approach to cloud-based data stream processing using Pig/Hadoop workflows is followed by HStreaming[1]. It has the same limitations in terms of workflow flexibility and high level visual workflow language, and only supports a subset of the window semantics that our proposed continuous workflow model supports.

2.5 WORKFLOW SCHEDULING

In current workflow management systems the scheduler is used to optimize the resource utilization of the system. The scheduler usually defines a static schedule since the data-flow rates are known a priory because each activity has predefined production and consumption rates. Static approaches work well with the traditional workflow model since workflows are considered one-time interactions. In the more complex case of pipelined execution in data-flow based workflows, the activities run on their own threads and are managed by the operating system (usually by employing a Round Robin policy), which is oblivious to any special characteristics of each activity (e.g., token productivity, time to execute etc.)

Many WfMSs consider the task of running a workflow as combining a set of external services, choreographed using the workflow patterns. In order to do that the system has to find appropriate services that carry out the task of each activity, e.g., [29]. The external services, besides a description of their task, also carry a Quality of Service (QoS) characterization. In the context of workflow management systems and Operating Systems in general the QoS metric is measured as the response time. Using this profile the WfMS can compose the workflow instance in a way which satisfies the overall workflow request's QoS requirements (e.g., finish the whole workflow within the time limit). Similarly, [70] breaks the workflow into subsections, by categorizing the activities into branch or synchronization activities. It then distributes the remaining deadline to the subsections making sure that the workflow will finish before the deadline give a minimum execution time for each task.

The challenges here include the dynamic nature of the external resources and the unpredictable nature of the execution of the various patterns which the workflow is composed of. Considering a

[1]http://hstreaming.com/ (2011)

network of tasks composed as a workflow, each task is in a ready state once it has all the necessary data in its input ports that will make it complete one iteration. This decision, of whether to run a task or not, is made by processing a set of task preconditions. Then the scheduler will decide which task to execute next according to a priority function.

In the context of a continuous workflow management system, which shares characteristics from both data stream processing systems and traditional workflow management systems, we have worked towards leveraging scheduling policies to fit this new model.

2.6 WORKFLOW BENCHMARKING

Generally, the workflow community has focused on scalability while neglecting a systematic way to measure the performance of a given system configuration, in order to avoid expensive trial-and-error or guesswork. In [22], a synthetic benchmark for workflow management systems is proposed, based on the TPC-C order-entry benchmark. Although this work is a step towards comparable measurement of WfMS's performance, it is based on the traditional, request-response model of workflows, and thus it would be unsuitable for us to test for our data stream processing continuous workflow model.

Since the focus of this dissertation is on data stream processing in WfMS, we have borrowed a benchmarking suite from the data-stream processing community, called the Linear Road Benchmark [3]. Linear Road simulates a toll system for the motor vehicle expressways of a large metropolitan area. It specifies a variable tolling system and supports accident detection and alerts, traffic congestion measurements, toll calculation and historical queries. The tolling system uses "variable tolling": an increasingly prevalent tolling technique that uses such dynamic factors as traffic congestion and accident proximity to calculate toll charges. The application provides a single feed of car position updates. Each car updates its position every 30 seconds. That includes its position (expressway id, direction, lane, segment of the highway) and current speed. While the implementation processes this feed, it is required to provide notifications to the cars about their toll charges every time they switch a segment, based on a set of conditions. It also needs to alert them of any accidents which happened down the road in order for them to exit the highway and choose another route.

The Linear Road Benchmark has already been implemented in the context of four systems: Aurora [3], an unspecified relational database system [3], IBM Stream Processing Core (SPC) [30], and XQuery[8]. The Linear Road Benchmark implementations measure performance by reporting the L-rating metric, which corresponds on the number of highways the system can handle, while remaining within the response-time boundaries required by the benchmark. So far the highest L-rating reported by a centralized system is L=2.5 by both Aurora and SPC.

In the context of this dissertation our interest in using the Linear Road Benchmark is to define the input rate which can be achieved by CONFLuEnCE, using various scheduling policies that we have implemented within the STAFiLOS framework and how those compare to the operating system based multi-threaded scheduler. This investigation is presented in Chapter 5.

2.7 SUMMARY

In this chapter, we formally defined workflows and workflow management or enactment systems according to the literature. We then described how DSMSs handle data-stream processing, and then explained data-flow processing patterns used in todays systems. We then examined some of the current workflow enactment systems which support data stream processing. We then considered the current scheduling techniques with the goal of optimizing a QoS or QoD metric in the context of DSMSs. We then examined how scheduling has been used within WfMSs and what challenges lie ahead in terms of performance optimization in a system that supports data stream processing through workflow execution. Finally, we looked at approaches used to benchmark the performance of WfMSs as well as DSMSs.

3.0 CONTINUOUS WORKFLOW MODEL

The first main contribution of this dissertation, is the definition of the Continuous Workflows (CWfs) Model which we present in this chapter. We first investigate the drawbacks of the current workflow model in processing data streams in Section 3.1. Our findings from this investigation led us to formally define the Continuous Workflow Model, as described in Section 3.2. Six CWf patterns were derived from the CWf model and are described in Section 3.3. The expressive power of our CWf model as captured by the six additional CWf patterns is evaluated in Section 3.4.

3.1 TOWARDS THE CONTINUOUS WORKFLOW MODEL

In this section we examine the ability of existing workflow models to support monitoring applications. This analysis is based on the communication patterns described in [69] (and in Section 2.3) and how those can be supported in a workflow using the workflow control patterns described in [64] (and in Section 2.1).

From the *communication patterns* described in Section 2.3, current workflow management systems and languages provide support for just the pull model communication patterns. Push patterns are essential for handling data streams and supporting monitoring applications and this is an indication that radical changes need to be made to the current workflow model to support these patterns. At the time when we proposed the CWf model no other WfMS system provided support for either of the two push model communication patterns, namely Publish/Subscribe and Broadcast. In terms of the *data-flow patterns* described in Section 2.3, current workflow management systems

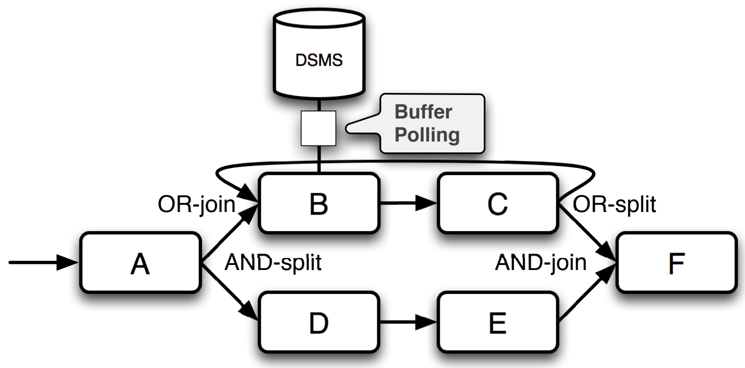


Figure 7: Abstract Workflow example

and languages provide support for all of them besides the *Streaming data-flow* pattern with stateful activities (i.e., events from previous invocations are correlated with events from future invocations), as in [37].

In existing WfMSs the only point in a workflow that is able to handle push data is at the initial activity, which is where the request to instantiate a workflow comes in, along with the necessary parameters. Depending on the model of computation there are two ways to support a single input event stream: With pipelining and without pipelining. Without pipelining, each event belonging to a stream will be individually handled by an instance of the workflow. In this model, there is no clear way of supporting the Streaming data-flow pattern, which in this case would require coordination of activities in-progress across different workflows.

With pipelining, a single stream can be supported by combining the multiple instances of the workflow into a pipelined data-flow execution model. All of the workflow instances (one from each event) share single activity instances (one for each activity in the workflow). The data is staged in between activities and are processed individually as they flow from the initial activity to the other interconnected workflow activities. Correlating events could be done by batch processing events in the buffers, and that functionality should be hardcoded in each activity's semantics (i.e., there is no declarative way of doing it).

A traditional workflow model that supports loops can potentially support *multiple data streams* by using the *Synchronous Polling* communication pattern (Section 2.3). In the example of Figure 7, besides the stream that is pushed into activity A, we also have a second stream connected at activity B. In order for activity B to bring events from the stream to the workflow it needs to continuously poll the buffer. Whenever there is an update B downloads it, processes it, and pushes it to activity C. In order to continuously poll the buffer a loop is used to re-activate B and check the buffer (as in Synchronous Polling). In this way the events produced by the $B - C$ branch is also a data stream.

A problem inherent to this solution is the *Lost Update* problem. Consider the case where the loop is executed at specific time intervals. Assume that activity B wants to find a flight fare that is less than $\$200$. When B queries the buffer finds a price that is more than that. In between two intervals the price goes to $\$180$ and then, just before the next query is submitted to the buffer, it goes back to being over $\$200$. That means the workflow had a lost opportunity of satisfying the user.

Now assume that the stream from the initial activity A is routed towards the $D - E$ branch. This means that we have to join two data streams at the F AND-join point. Joining the events from the two branches as they arrive probably make no sense to the application. Also the two streams could have a big volume difference in number of events per unit of time, thus one of them would either drop events or should have means of buffering them. Introducing queues to the inputs of the joins should workout this problem. Moreover the results would probably make more sense if there was a way to synchronize the two streams in terms of temporal and value based functions on windows of these data, similar to the ones found in continuous queries [49].

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

Another operation that monitoring applications need to be able to make, on workflows processing data streams, is event invalidation. For example, if you have a stream from an airline which publishes fares, and a new fare update comes in that invalidates a previous fare, then the earlier update should be invalidated downstream, in order to avoid processing a fare that is invalid. Invalidation, or otherwise known as cancelation, is used when an upstream event (later event) triggers the invalidation of a downstream event. This is supported in YAWL [63] but it is not possible to selectively invalidate events in the workflow, since they consider each workflow instance independent for each event.

The above observations led us to the idea of a Continuous Workflow model that allows the specification of event stream processing in a declarative manner as in the case of CQs.

3.2 DEFINITION OF THE CONTINUOUS WORKFLOW MODEL

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

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

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

3.2.1 Waves

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

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

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

3.2.2 Windows

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

3.2.2.1 Window Specifications Five parameters are required to define the window semantics for that operator: *window size*, *window step*, *window_formation_timeout*, *group-by* functionality, and *delete_used_events*. A description of these parameters follows.

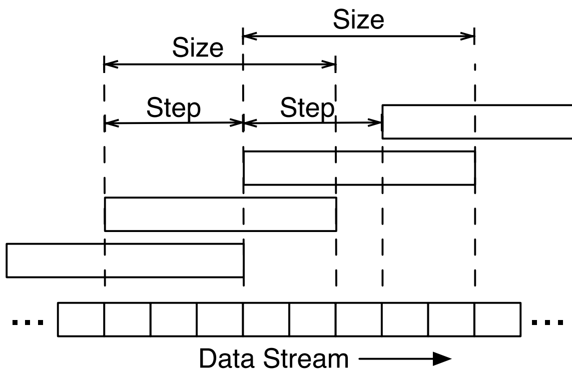


Figure 8: Size and Step for event-based window semantics. The bottom series of boxes represent the stream of events. In this example the size is 5 events and the step is 3 events. The windows produced are represented by the boxes stacked above the queue

Window Size and Window Step: In general, windows in computing are defined in terms of an *upper bound*, *lower bound*, *extend*, and *mode of adjustment* as time advances. The upper and lower bounds are the timestamps of the events at the beginning and the end of the window. The extend is the *size* of the window. When a window is initiated its lower bound is defined and its upper bound is computed using the size. The mode of adjustment, also known as the *window step*, defines the period for updating the window. If a step is not defined, then the window is evaluated every time a new event comes into the queue.

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

(a) *Logical units*: which are time-based, and define the maximum time interval between the upper and lower bound timestamps.

(b) *Physical units*: which are count-based, and define the number of events between the upper and lower bounds.

(c) *Wave-based*: where the upper and lower bounds of a window are defined by the first and last events of a wave currently being processed.

(d) *Semantics-based*: where a general predicate over the data stream can be used to define the window start/end points. This has generally been implemented using punctuations that are embedded in the data stream by the data sources themselves.

A depiction of how the size and step parameters define the windows in terms of physical units is shown in Figure 8.

A window can be defined in term of both step and size in all three types of units (i.e. time, token and wave). We now list all possible combinations and the context where each combination makes sense:

1. *Step: TIME, Size: TIME*: Every x seconds produce a window of size y seconds. For example, an activity calculating an average value for tokens of the last 30 seconds and produces a result every 1 second, should be attached to a window with 1 second step and 30 seconds size.
2. *Step: TIME, Size: TOKEN*: Every x seconds produce a window of size y tokens. If the queue contains more than y tokens, then the window contains the y earliest tokens in the queue. At the next iteration the queue expires x seconds worth of tokens. For example, if the application requires a value every second whenever the first one is available it is advantageous to define a window with step 1 second and size 1 token over defining a window of step 1 second and size 1 second because the later definition requires the window to close before producing it.
3. *Step: TIME, Size: WAVE*: Every x seconds produce a window of size y waves. As we stated above waves are used to separate sets of multiple results, where each set was produced from a single activity execution. This could eventually be seen as a group-by window function, but in this case there are no token based or time based bounds to the windows.
4. *Step: TOKEN, Size: TOKEN*: Every x tokens produce a window of size y tokens. This window specification is used when there is a need for batch processing multiple tokens together where the window size has to be specific. When the step is equal to the size of the window, the tokens are processed only once (tumbling window). In the case where the step is smaller than the size, some tokens interleave executions (sliding window). In the last case, where the step is larger than the size, some tokens are not being used since when the window advances it goes $x - y$ token beyond the last token processed and the rest are discarded.
5. *Step: TOKEN, Size: TIME*: Every x tokens produce a window of size y seconds. For example, if an activity requires the last 5 seconds worth of data, for each token entering the queue, then it would define the step as 1 token and the size as 5 seconds.
6. *Step: TOKEN, Size: WAVE*: Every x tokens produce a window of size y waves. In this case if the step is 1 token and the size is 1 wave then a window will be produced for every token

which entered the queue and it will contain all the tokens from the current wave available at the head of the queue.

7. *Step: WAVE, Size: TOKEN*: For every x waves seen, process the top y tokens of the concatenation of the waves. If the waves' total size is less than x then tokens from the next wave will be included in the window. Another caveat of this specification is that if the wave size is too small then the window will not be produced even when the wave's last event arrives in the queue.
8. *Step: WAVE, Size: TIME*: For every x waves seen, process y seconds worth of data. Similarly to case 7, if the wave does not span a significant amount of time then tokens from the next wave have to be included to close the window. Otherwise, the first y seconds worth of tokens from the current wave will be included in the window and the rest will be discarded.
9. *Step: WAVE, Size: WAVE*: Every x waves produce a window of size y waves. Note that each waves consists of multiple data events. This window semantic combination is useful for synchronizing data items belonging to the same context, e.g., when an activity called *process_order* splits a customer's order into multiple items to be processed by multiple warehouses, and then an *order_completion* activity needs to synchronize the responses from the warehouse belonging to the same order to notify the user of its completion.

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

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

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

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

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

Deleting used events: Events on a queue could either be consumed or only used by an activity. A flag, called "delete_used_events" is used to denote the consumption mode. That is, to denote if events that were used in the window that triggered an activity should be deleted from the queue upon their usage. This is useful in the cases where a complex event is discovered (from a combination of multiple simple events) and the application requires that that set of events should be used to trigger subsquent complex events. The signal to delete used events from queues comes as part of the post-conditions of an activity.

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

The window semantics definition along with the deleted_used_events flag can execute the hybrid window and consumption modes described in [1], such as *unrestricted*, *recent* and *continuous*, as those are combined with *tuple, time*, and *wave-based* windows.

3.2.2.2 Window operator example To provide a better understanding of the window operator, let us consider an example of a window operator that triggers an activity A when two events occur within 5 minutes of each other. For this example consider Figure 9. The window is calculated at every step (of 1 minute). Letters represent events and numbers represent timestamps (in minutes). The window starts at timestamp 0, at which time event a arrives and is enqueued. Between timestamps 1 and 5, on every minute the queue is evaluated and no action is taken, as the precondition is not satisfied since only a is part of the window. At timestamp 5, event a is expired because the current window's upper bound is more than its timestamp plus the size of the window, meaning that a cannot be part of any subsequent windows. At timestamp 6, event b is enqueued. The precondition is evaluated and no action is taken, since only b is part of the window. At timestamp 8, event c is enqueued. This time the precondition is satisfied since both b and c are part of the window (size 2 events). The activity is then triggered and once its execution is completed events b and c are deleted from the queue, since the "delete_used_events" flag is set.

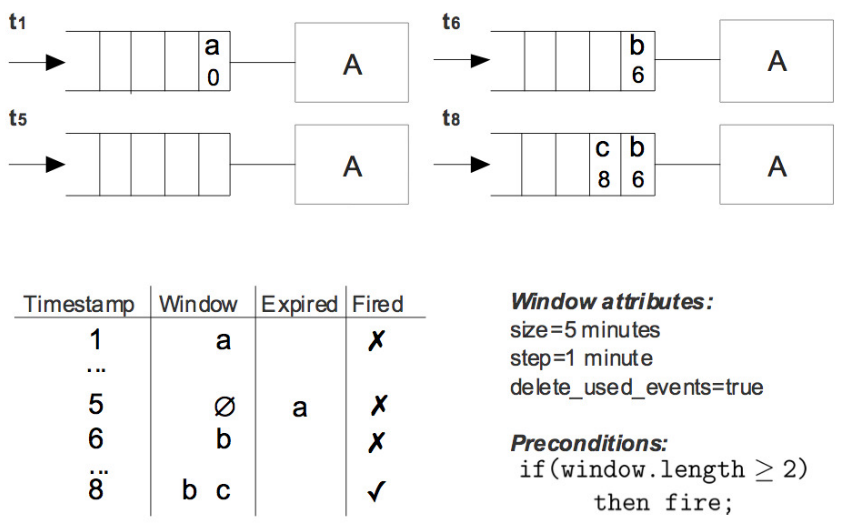
3.2.3 Push communication

In the push communication model, the data consumer receives multiple data items asynchronously. We are interested in two communication patterns which follow this model [55]:

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

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

The push model has not been supported by any workflow system until recently when, parallel to our work, another workflow management system started to support it [48]. The lack of support



Timestamp	Window	Expired	Fired
1	a		✗
...			
5	∅	a	✗
6	b		✗
...			
8	b c		✓

Window attributes:
size=5 minutes
step=1 minute
delete_used_events=true

Preconditions:
```
if(window.length ≥ 2)
    then fire;
```

Figure 9: Window operator example

of these patterns so far has been a direct result of an underlying assumption that data sources in workflows are passive (e.g., data is stored in databases or data files) and data consumers (users, tasks), are the only active entities that can request and synchronously retrieve the data. These two missing communication patterns require that the data sources involved are active as well.

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

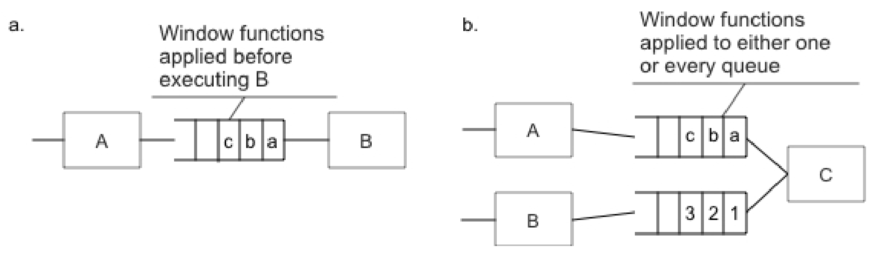

a.
Window functions applied before executing B
A
c
b
a
B
b.
Window functions applied to either one or every queue
A
c
b
a
C
B
3
2
1

Figure 10: (a) CWP1: Sequential aggreate and (b) CWP2: Stream-join

3.3 CONTINUOUS WORKFLOW PATTERNS

We now describe six new workflow patterns that we have identified as required in the context of continuous workflows and are supported by our CWf Model.

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

A sub-pattern of this one involves Group-by windowed operators and it basically does the same thing as above with the added functionality of grouping data items as described in Section 3.2.2.

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

CWP3 Stream-synch: A point in the workflow where two or more different event streams meet to get synchronized. The result is waves of events that are synchronized. This pattern is used to join events belonging to the same wave, which have been independently handled by different branches. **Example**: In a travel agency application, activities *receive_fares* and *receive_hotel_prices* are synchronized according to some window definitions, and split into pairs where the $hotel.price + fare.price < 400$. They are then handled individually on parallel branches but are considered part of the same wave. That wave id is then used to join events from the two branches.

CWP4 Case cancelation: This pattern is applied whenever an event, simple or complex (i.e., resulted as a combination of multiple simple events), occurring upstream needs to cancel some events that have already been propagated downstream, and currently either reside in input queues of downstream activities, or they have already been processed and need to be rolled back. In order to implement this pattern techniques from the WED-flow framework [21] may be applied. **Example**: In a travel agency application, after the user's approval a transaction has been initiated to book a hotel and a flight, but in the meantime a new cheeper price has ben detected upstream. The user wants to cancel/stop the transaction initiated in order to take advantage of the new lower price.

CWP5 Workflow data view: This pattern refers to the ability to extract any kind of data being exchanged inside the continuous workflow and streamlining them into a separate event stream that can be used as an input to another workflow. An example usage of this pattern is to monitor the execution of the workflow and to debug it. Usually the views are not known at design time thus incorporating them into the workflow is not feasible. The view can be expressed as a set of predicates that can be evaluated on any arbitrary set of the data inside the workflow network. **Example**: Somewhere in the workflow an activity produces a result that is above expected values. The designer can add a view that will give her the message/event with the outlier value as soon as it is produced (i.e., $value > 100$). The message is annotated with meta-data regarding the activity it was last processed by.

CWP6 Window Event Expiration: This pattern refers to the existence of a secondary output on the windowed queues. The queues emit events on this queue whenever the step function is applied on the window. Any event that came before the current window starting even is emitted in this queue. If an event was previously deleted as part of the delete_used_events functionality then those are not emitted here. This pattern is used when the applications requires separate handling of unused events. An example of this pattern is described in CWf part of Section 3.4.

3.4 MODEL EVALUATION

We evaluate our model in terms of the applicability of our Continuous Workflow Patterns in Section 3.4.1 and its expressive power in Section 3.4.2.

3.4.1 CWf Patterns

Nova[48] is a system developed to support stateful incremental processing and provides a limited model for continuous workflows. Nova which was proposed four years after our initial proposal of the CWf model and patterns, introduced its own four processing patterns. These pattern can be mapped to our CWf patters as we describe below:

- **Non-incremental**: Process input from scratch; can be realized with CWP1 using a null step and a null size definition. This would produce a window with every item seen since the beginning of the execution, every time new items are being added to the windowed queue.
- **Stateless incremental**: Process just the new input data; can be realized with CWP1 using a null step and null size definition, but setting the *delete_used_items* flag set to true. This way only new items will be processed since whatever has already been processed will be purged, and a window will be produced whenever new items arrive in the windowed queue.
- **Stateless incremental with lookup**: Process just new input data; independently of prior input data items; a side lookup table may be referenced. This is realized using two queues (CWP3). The first one is exactly the same as in the previous pattern, and the other one is the same as the first case.
- **Stateful incremental**: Process just the new input data, but maintain and reference some state about prior data seen on that input. This will again use two input queues. The first one is the same as the second pattern here, and the second one is forming a loop between one output of

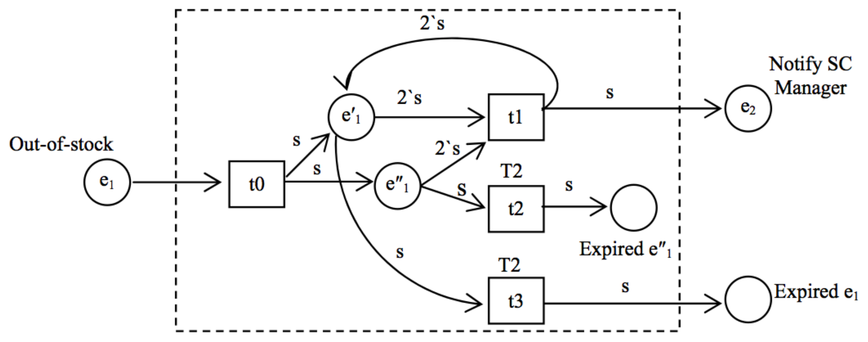


Figure 11: Timed Petri net of "repeat cause-one effect" Supply Chain pattern

the task and the input of the second windowed queue. This input queue is the same as in the first pattern.

3.4.2 CWf Model Expressibility and Flexibility

We have evaluated the expressiveness of our continuous workflow model over a set of patterns which apply to the supply chain monitoring applications introduced in [34]. With the introduction of queues and window operators, the design of those patterns is made much easier and more flexible.

Figures 11 and 12 depict two versions of the same pattern implemented using a Petri net approach and our Continuous Workflow approach respectively. The pattern concerns the case where multiple occurrences of one event within a certain time period cause another event to occur. In the example shown, if two out-of-stock events occur within a time period of T2, then a notification to the Supply Chain manager will be initiated. Figure 11 (which is explained in detail in [34]), transitions $t2$ and $t3$ wait for time interval $T2$ before consuming an event from e'_1 or e''_1 respectively. This is used for expiring events that occurred time T2 ago. A notification by $t1$ is only fired if two events coexist in e'_1 and e''_1.

Using the CWf model (Figure 12) this can be implemented simply by having a queue for `Out-of-stock` events and a window operator on that queue, which constructs windows of size $T2$. No step is defined thus the window is calculated for every new `Out-of-stock` event, and a

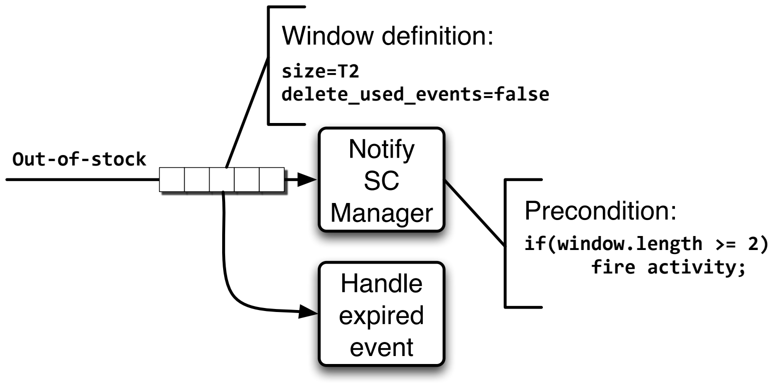


Figure 12: Continuous workflow of “repeat cause-one effect” Supply Chain pattern

notification is fired only if the window has two events in it (according to the precondition). Events are not deleted once used but they are eventually expired and handled by another activity, thus keeping the semantics of the two implementations the same.

It is clear that our implementation is much simpler. Moreover, if the designer wants to change the semantics and requires 3 out-of-stock events to happen before notifying the supply chain manager, then, in the Petri net case she would have to add another transition like $t3$ and another like $t2$ and change the numbers on the arcs going to $t1$, from 2 to 3. In the continuous workflow case she would only have to change the precondition from $window.length >= 2$ to $window.length >= 3$. Because of that, our approach is easier to use and is also more robust since this parameter can be changed during runtime.

3.5 SUMMARY

In this chapter, we formally defined our Continuous Workflow Model. We did that by first examining the drawbacks of the current workflow model, and then by proposing the addition of new primitive constructs that enable the processing of data-streams in the context of workflows. From the defined model we have derived six new Continuous Workflow Patterns. We finally evaluated the expressibillity of our CWf model, by comparing it with other existing workflow models that support some form of data-stream processing. Our CWf model was then realized in the context of CONFLuEnCE (CONtinuous workflow ExeCution Engine), built on top of Kepler [36], and it is described in the next chapter.

4.0 CONFLuEnCE: CONtinuous workFLow ExeCution Engine

In this chapter we describe the implementation of our Continuous Workflow (CWf) Model. In Section 4.1, we give an overview of our approach to our implementation. In Section 4.2, we describe Kepler on top of which we implemented our CONtinuous workFLow ExeCution Engine (CONFLuEnCE). We describe the key components of CONFLuEnCE, in Section 4.3-4.6. We evaluate the applicability of our Continuous Workflow Model and its implementation by building by building two representative monitoring applications in CONFLuEnCE in Section 4.7.

4.1 OVERVIEW

In order to implement our continuous workflow model, we had to implement the three basic primitives presented in the previous chapter in addition to all the primitives provided by a traditional Workflow Management System. Since our model is a superset of the traditional workflow model, we decided to build our implementation on top of an existing workflow management or enactment system instead of building a new system from scratch. We evaluated a number of open sourced workflow systems to find a suitable one. Specifically we evaluated in detail Taverna [47] and Kepler [36]. We chose Kepler as the base for CONFLuEnCE because of its extensibility, modularity and our common aim to support scientific workflows.

The suitability of Kepler, for implementing our CWf model, comes from the fact that it decouples the specification of a workflow and the models of computation that govern the interaction between components of a workflow. This means that a workflow can be specified once and executed under different runtime environments (i.e., models of computations) which Kepler inherited from its underlying PtolemyII system [19]. Also, Kepler's code is inherently extensible, by providing a modular design, and this is also proven by the fact that is being actively developed by nearly twenty different scientific projects.

Furthermore, programming workflows in Kepler is made easy for domain experts without any knowledge of programming structures. It provides a large library of basic actors (i.e., components representing various tasks) as well as specialized actors, for easy reusability and composition of new applications. The library includes actors for database interfacing, data filtering, etc. and it is easily extensible to include domain-specific actors for actions such as automatically annotating astronomical objects. Kepler provides an intuitive high-level visual language for building workflows, where the designer can drag and drop components and connect inputs with outputs quite easily. Configuring parameters is easily done using dialog boxes and it also gives useful displays for debugging the workflows. A screenshot of Kepler with CONFLuEnCE depicting the supply-chain continuous workflow demonstrated is depicted in Figure 13 [42].

Finally, Kepler was implemented in Java which simplifies our implementation of CONFLuEnCE. CONFLuEnCE was implemented within Kepler as a new model of computation (i.e., as another module). This module implements all the necessary constructs which enables Kepler to run continuous workflows.

4.2 KEPLER'S ACTOR-ORIENTED MODELING

A workflow in Kepler is viewed as a composition of independent components called *actors*. Actors have parameters used to configure and customize their behavior, which can be set statically, during the workflow design, as well as dynamically, during runtime. Communication between them happens through interfaces called *ports*. These are distinguished into input ports and output ports and the connection between them is called a *channel*. As part of the communication between the two ports, a data item (referred to as token in Kepler) is propagated from the output port of one actor to the input port of the receiving actor. The receiving point of a channel has a *receiver* object, which controls the communication between the actors. The receiver object is not provided by the actor but by the workflow's controlling entity, called the *director*. The director defines the execution and communication models of the workflow. As such, the communication being synchronous or

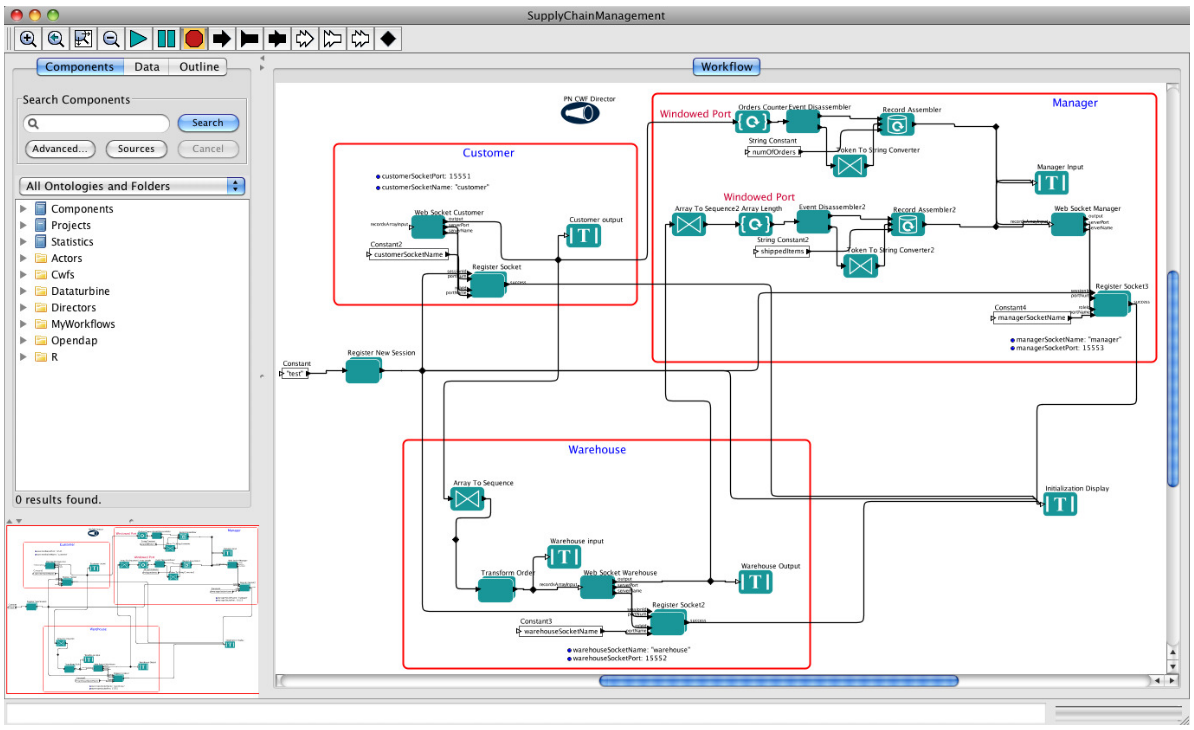


Figure 13: Implementation a continuous workflow for Supply Chain Management reactive application

asynchronous (buffered) is determined by the designer of the director, not by the designer of the actor or of a workflow. Figure 14 [36] shows how all these components are organized as part of a workflow.

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

1) The *Synchronous Data Flow* (SDF) director is designed for sequential and simple workflows with the number of tokens produced by the actors known a-priori, thus the scheduling order of the actors is defined before the execution starts.

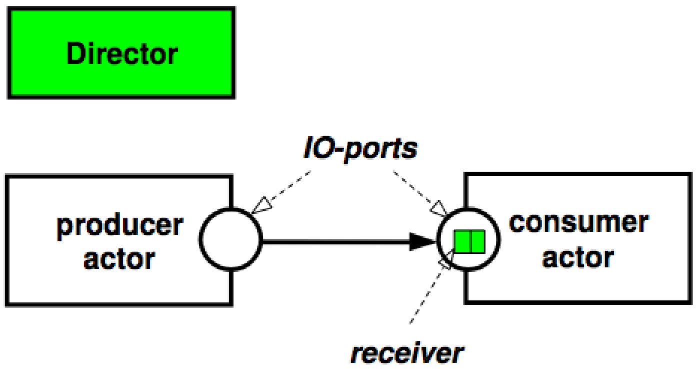


Figure 14: The semantics of component interaction is determined by a director, which controls execution and supplies the objects (called receivers) that implement communication

2) The *Dynamic Data Flow* (DDF) director, like SDF, executes the workflow in a single execution thread. However, unlike SDF it does not use static scheduling, but does so at runtime, since the number of tokens produced by each actor is unknown.

3) The *Process Network* (PN) director is designed for managing workflows that require parallel processing. This director wraps each actor in its own execution thread and the workflow is driven by data availability.

4) The *Continuous Time* (CN) director introduces the notion of time for modeling workflows able to predict how a system evolves over time.

5) The *Discrete Event* (DE) director, which also works with timestamps, measures average wait times and occurrence rates. All the events (data and timestamp pairs) emitted from actors are placed in a global workflow timeline.

A detailed description of these directors can be found in [18].

Throughout the workflow execution, the director goes through a set of phases, summarized as follows (described in more detail in [36]):

1) *Pre-initialize*: Calls the pre-initialize method of all the actors just before starting the workflow execution. This phase is reached only once per workflow execution.

2) *Type-checking*: to make sure that all the data types of tokens between the sending and receiving ends of the actors are compatible.

3) *Initialize*: calls the initialize method of each actor, every time the workflow is run[1]. Initialization tokens may be transmitted from one actor to another, web-service pings may take place or other actions that need to take place before the workflow starts running.

4) *Iteration:* A workflow run may consist of multiple iterations, where in each iteration the director calls the methods: *pre-fire* (actor tests its firing preconditions), *fire* (actor performs its main function, usually by consuming tokens from the input ports and producing results in its output ports) and *post-fire* (where the actor evaluates the postconditions and decides if it should be fired again in the next iteration) of each actor.

Workflows may be reused as part of a larger workflow (parent). They are called sub-workflows. The parent workflow views a sub-workflow as a self contained actor and manages it just like any other actor. Sub-workflows may use different director (i.e., employee different model of computation) than the parent workflow, thus forming a *hierarchical heterogeneity*.

4.3 CONTINUOUS WORKFLOW DIRECTOR

As we have mentioned in the previous section, the director component in Kepler governs the execution model and the communication model. One of the CWf model requirements is the concurrent execution of sequential activities. This can be achieved in a multitude of ways (e.g., multiple thread, parallel processes, single-thread continuous scheduling, etc.). Since this commutation model can be realized in many ways we have defined a generic interface, called ICWFDirector, which needs to be implemented by any director implementing the CWf model. For the communication model, it is clear that the window operator specifications need to be used by any director implementing the ICWFDirector interface. To this extend we implemented an abstract WindowedReceiver (described in the next section), which needs to be extended accordingly for each CWf director.

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

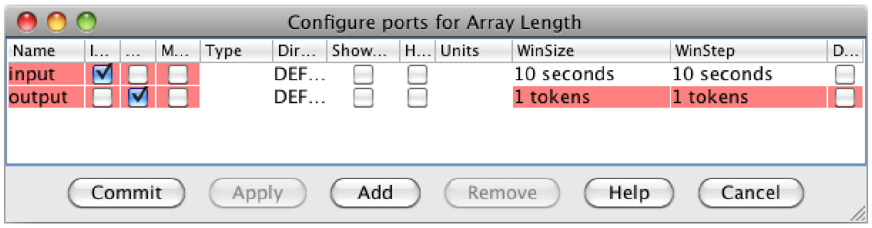
The first CWf director we implemented is called PNCWF Director, because it borrows a lot of traits from Kepler's PN director. Its most important characteristic being that concurrent execution is natively enabled by the PN director, since every actor is executed in parallel in its own thread. During initialization of a workflow each actor is associated with a thread based controller to transition it through the iteration phases (initialize, pre-fire, fire, post-fire). The actor thread blocks, when trying to read from its input ports which have no events available, until a window or event is produced.

However, PN does not support any notion of time needed by the window operator. The notion of time is supported by the CN and DE directors both of which, however, do not support parallel processing. Since none of the existing directors could be used to support the CWf requirements, we decided to implement a new Continuous Workflow director by using time-based techniques used within the DE domain.

To add the notion of timed events to the PN model of computation we encapsulate each data token within an event object (implemented as CWEvent). The event carries its timestamp (either the creation time of the data or the time it entered the system), and its wave-tag. Since all current actors were implemented without being timestamp and wave aware, they cannot output the timestamp and wave of the events to the next receiving actor. To solve this problem the PNCWF director associates a *time-keeper* object to each actor at initialization time. This process is described in Section 4.5.

4.4 WINDOWED RECEIVER

The second requirement is to implement window operators, that is, to add queues on the inputs of actors that are capable of applying window semantics on the stream of ensued events. Although queueing has already been implemented in certain models of computation in Kepler, window semantics on the queues do not exist in any model of computation. We have implemented a new type of abstract receiver, called WindowedReceiver which can be used with continuous workflow directors (e.g., PNCWF director). This new type of receiver implements all the specifications defined in Section 3.2.2, except for the semantics based windows. The parameters that have to be set accord-


Configure ports for Array Length

Name	I...	...	M...	Type	Dir...	Show...	H...	Units	WinSize	WinStep	D...
input	☑	☐	☐		DEF...	☐	☐		10 seconds	10 seconds	☐
output	☐	☑	☐		DEF...	☐	☐		1 tokens	1 tokens	☐

Figure 15: Modified Kepler form for configuring actor ports. On this form the workflow designer can define in freeform text the size and step of the window associated with specific ports. Shaded cells denote non-editable parameters

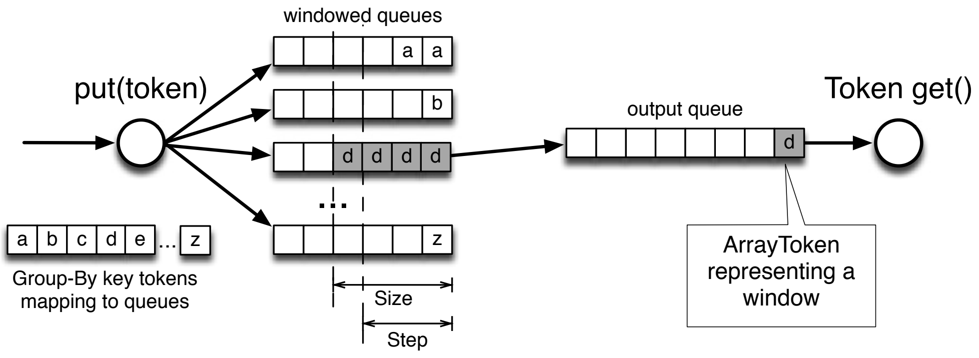


Figure 16: The generic WindowedReceiver as is implemented in CONFLuEnCE

ing to the specifications, such as window size and window step, can be set in freeform text in the modified form which is provided by Kepler for configuring actor ports (Figure 15). Additionally the designer may define the windowed receiver as a ”Group-by”. Since the group-by function may create a lot of groups, if the window definition is time-based, the user may also choose to suppress the empty windows by toggling a checkbox.

Figure 16 portrays how the WindowedReceiver is implemented inside CONFLuEnCE, and following, we will describe how the windowed receiver works. As we have already mentioned in Section 3.2.2, the workflow designer may define a Group-by condition at each windowed port in one or more ways: If the input is of array type the designer may define a set of index position of the input array as a conjunction of values to group the input tokens by. If the input is of record type (which can also be nested by other record tokens) the designer may define a path query which pinpoints a location for objects in the record hierarchy, by which the grouping will be calculated. The windowed receiver keeps a mapping of group-by token keys and their corresponding token queues. When a token is *put* into the receiver (using the `put()` method) then the group-by condition returns a token. That token is used to find the mapped queue to insert the input token to it. If this is the first input token of a group, then a new queue is created and mapped to the group-by token. In the general case, where the windowed receiver is not a group-by, just a single queue is kept and the NullToken is the group-by key.

At each point where a new token is put into a group-by queue, the window semantics of the receiver (based on the defined Size and Step parameters as described in Section 3.2.2) are evaluated upon that queue. If a new window is produced then that window is converted into an Array Token (a native Kepler/PtolemyII data type) and encapsulated into a CWEvent. It is then inserted into the output events queue, which in turn is polled every time the actor, which the WindowedReceiver is attached to, fires and calls the *get()* method.

One of the properties of the WindowedReceiver, regarding the timing of the windows it produces, is that their timestamps may be out-of-order. This is due to the fact that the receiver may have multiple group-by queues which produce windows at different rates depending on the arrival of events into each one of them. In order to partially alleviate this problem, and the problem of a window operator/queue waiting to close a timed window, we introduced the notion of timeouts, on time-based windowed receivers. A timeout factor x can be defined at the director level to enforce window production after the supposed window production time. For example, assume we have a receiver with a step time of s seconds. A window has been produced at time t_1 and now the next window is to be produced at time $t_1 + s$. The timeout enforces the next window to be produced at most after $t_1 + (s \times x)$.

4.5 TIME KEEPING

Part of the requirements of the Continuous Workflow Model is that, in order to apply window semantics on the token streams, these tokens need to be time or wave stamped. As we already mentioned the workflow's internal tokens, in CONFLuEnCE, are encapsulated inside a CWEvent object which keeps details about the wave-id and timestamp of a token. When an actor is fired, it most probably has no notion of timestamp or waves, thus it only processes a token or set of tokens as an input and produces a token as an output. These tokens have no timestamp or wave information. Thus, the responsibility for encapsulating and decapsulating the token into and out of a CWEvent fall to the WindowedReceiver.

During decapsulation, the receiver needs to save the timestamp and wave information of the event which is about to be processed. This information is saved by an object called the TimeKeeper which is associated with each workflow actor. The time-keeper of an actor keeps track of the timestamp information of the latest event processed by the actor. When the actor sends the result on a channel, the receivers at the receiving ends of that channel (which are windowed receivers defined by the CWf director) will ask the time-keeper associated with the producing actor to provide a timestamp for that event. The whole process of encapsulation, decapsulation and saving as well as loading the timestamps and wave information from the TimeKeeper is shown in Figure 17.

With the use of TimeKeepers we also solve the problem of backwards compatibility, to support all the legacy actors available in Kepler's library. Any CWf-specific actors can be implemented with timestamp and wave awareness.

4.6 PUSH COMMUNICATION

We have implemented the push communication patterns described in Section 3.2.3 in three ways:

1. *Web-sockets*: An input actor initializes, within itself, a web socket server listening to a specific port. The application built on top of the specific CWf has knowledge of the port number and whenever it needs to push data to the CWf it connects to the specific port and sends the data. The use of web sockets enabled us to build applications that run on the client's browser.
2. *Direct TCP connection*: An input actor initializes a connection with a specific data stream source. This could be a DSMS, or a generic service providing streaming data (e.g., Twitter streams). The socket connection object runs within the containing actor's thread and blocks

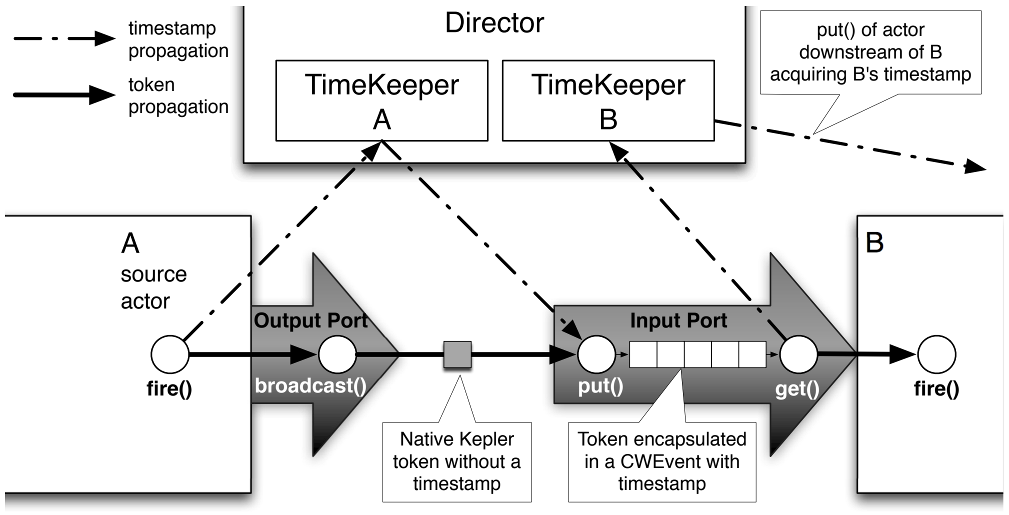

timestamp propagation
token propagation
Director
TimeKeeper A
TimeKeeper B
put() of actor downstream of B acquiring B's timestamp
A source actor
fire()
Output Port
broadcast()
Native Kepler token without a timestamp
Input Port
put()
Token encapsulated in a CWEvent with timestamp
get()
B
fire()

Figure 17: Timestamp preservation inside CONFLuEnCE

whenever there are no data to receive. When new data arrive they are broadcasted to the input ports of the actors connected to that input actor's output port.

3. *Using a mediator*: In order to support more generic data feeds such as RSS streams, we used a mediator platform called PubSubHubub[2] by Google. This service will aggregate data updates from multiple RSS feeds and push them directly to a URL deployed by the workflow server. Once that URL is called, it will, in turn, forward the updates to the CWf using a predefined TCP port, much like what happens in the above case with the web-sockets server within the CWf. This allows us to integrate our workflows with a larger set of data sources.

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

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

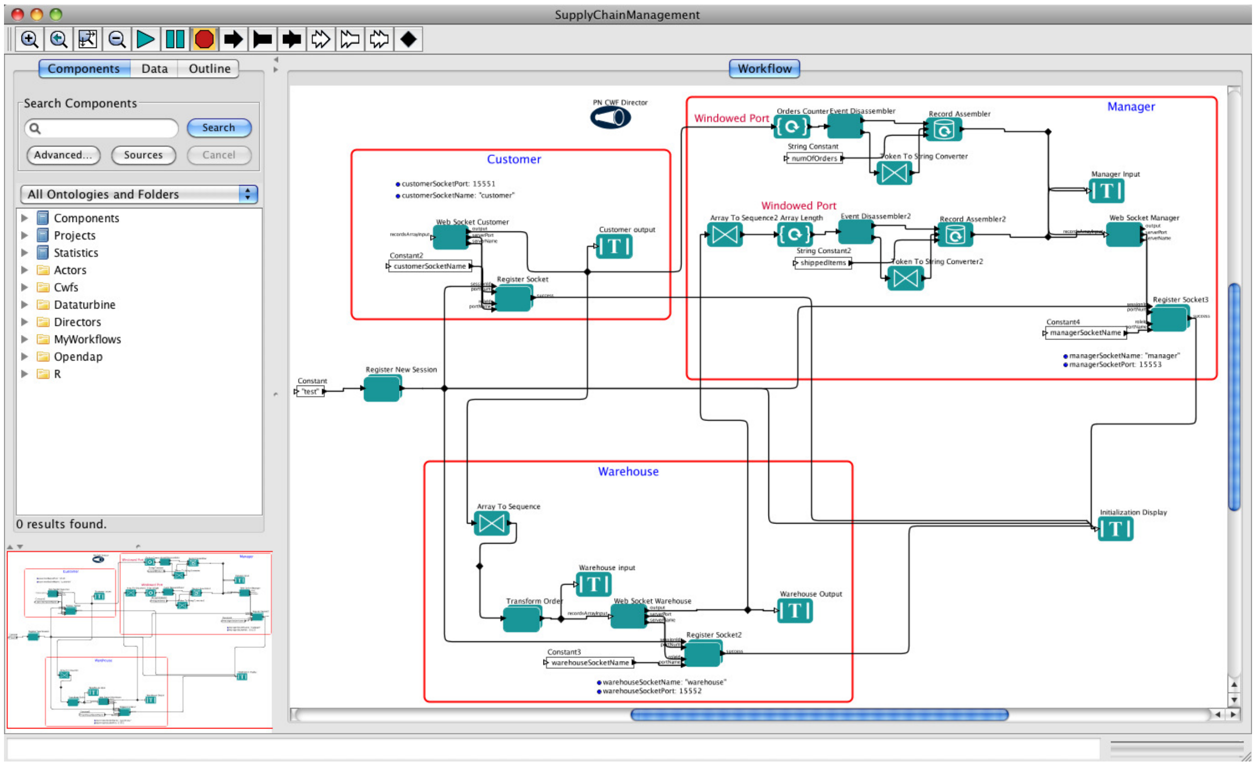


Figure 18: Implementation a continuous workflow for Supply Chain Management reactive application

4.7 EVALUATION

We evaluated the applicability of our Continuous Workflow Model by building two new real world applications which use Continuous Workflows to support their functionality. The first one is an on-line Supply Chain Management application which was demonstrated in SIGMOD '11 [42], and the second one as the monitoring component of a larger system, called AstroShelf, to monitor the interactions of astrophysicists with the system to detect interesting trending areas in the sky, transient astronomical events, and subsequently notify the users of their existence, based on their interests [43]. This application was demonstrated in SIGMOD '12 [45].

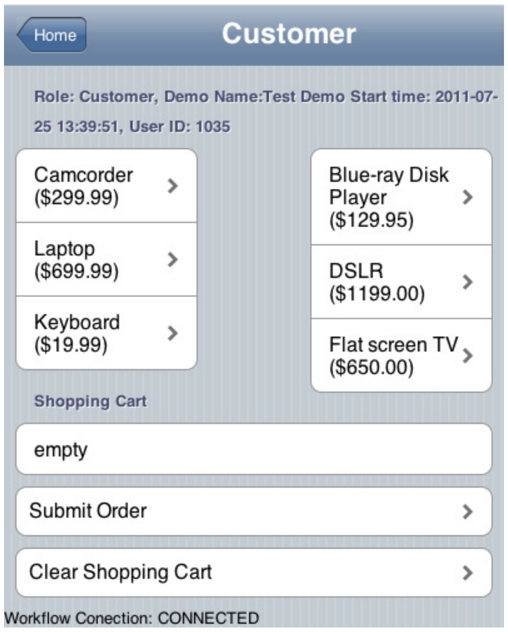
4.7.1 Supply Chain Management

Supply chain management applications are generally built to manage the workload of a business which handles product orders from customers, fulfillment of these orders from the warehouses and seamless bookkeeping of all of the transactions that take place. Ideally, the system should provide analytics on the state of the supply chain. In this scenario the objective was to design a continuous workflow capable of serving as the integration layer between databases in the warehouses, the web server providing the user interface and ordering system, and other administrator interfaces. Additionally, it should provide real-time analytics and event notifications to help the managers alleviate problems as quickly as possible.

The users of this system are split into four categories: (1) Clients, (2) Warehouse manager, (3) Company Manager and (4) Administrator. Roles 1-3 interact with the workflow through a web interface (from a mobile device or a laptop) and role 4 interacts with the workflow directly through the Kepler interface.

A client submits orders with multiple items, using the web interface depicted in Figure 19, and receives a notification once her order has been shipped. A warehouse manager notifies the system when an item is out of stock, and also receives order requests from the system and fulfills them. Note that an order may contain objects that are available in different warehouses. The workflow takes care of routing the order requests to the appropriate warehouse manager. The company manager receives notifications when things go wrong more than once and in more than one way, e.g., when an item is reported out of stock more than once in some specified period, or when multiple orders have been delayed or canceled. The company manager also has a real time view (Figure 20) of the current volume of orders and shipments to customers, updated every second. The statistics are computed using window semantics in certain parts of the workflow. The windows have a size of 1 second and a step of 1 second. The administrator's role is to change parameters, such as window sizes, or tune up settings in the scheduler to make the execution fit the application's requirements.

Figure 18 shows the precise specification of the continuous workflow supporting the Supply Chain Management application. The director can be seen at the top-center of the workflow definition. The workflow is also marked with the three sections, each one responsible for supporting



Role: Customer, Demo Name:Test Demo Start time: 2011-07-25 13:39:51, User ID: 1035

Camcorder ($299.99)
Laptop ($699.99)
Keyboard ($19.99)
Blue-ray Disk Player ($129.95)
DSLR ($1199.00)
Flat screen TV ($650.00)

Shopping Cart

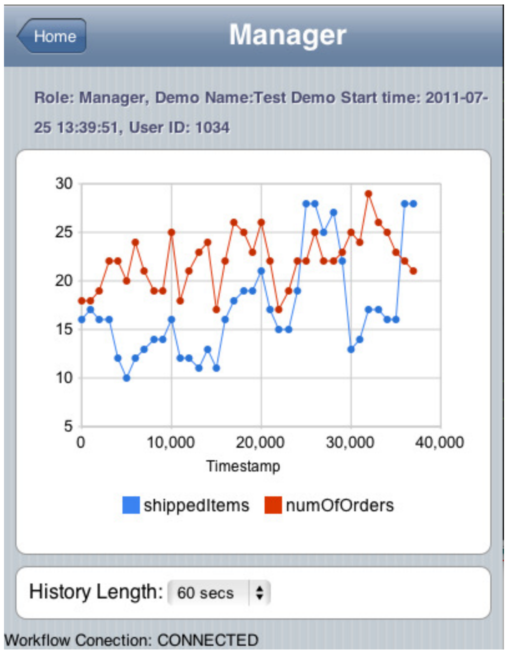
empty

Submit Order

Clear Shopping Cart

Workflow Conection: CONNECTED

Figure 19: Supply Chain Management application UI: The customer's panel



Role: Manager, Demo Name:Test Demo Start time: 2011-07-25 13:39:51, User ID: 1034

History Length: 60 secs

Workflow Conection: CONNECTED

Figure 20: Supply Chain Management application UI: Company manager's panel

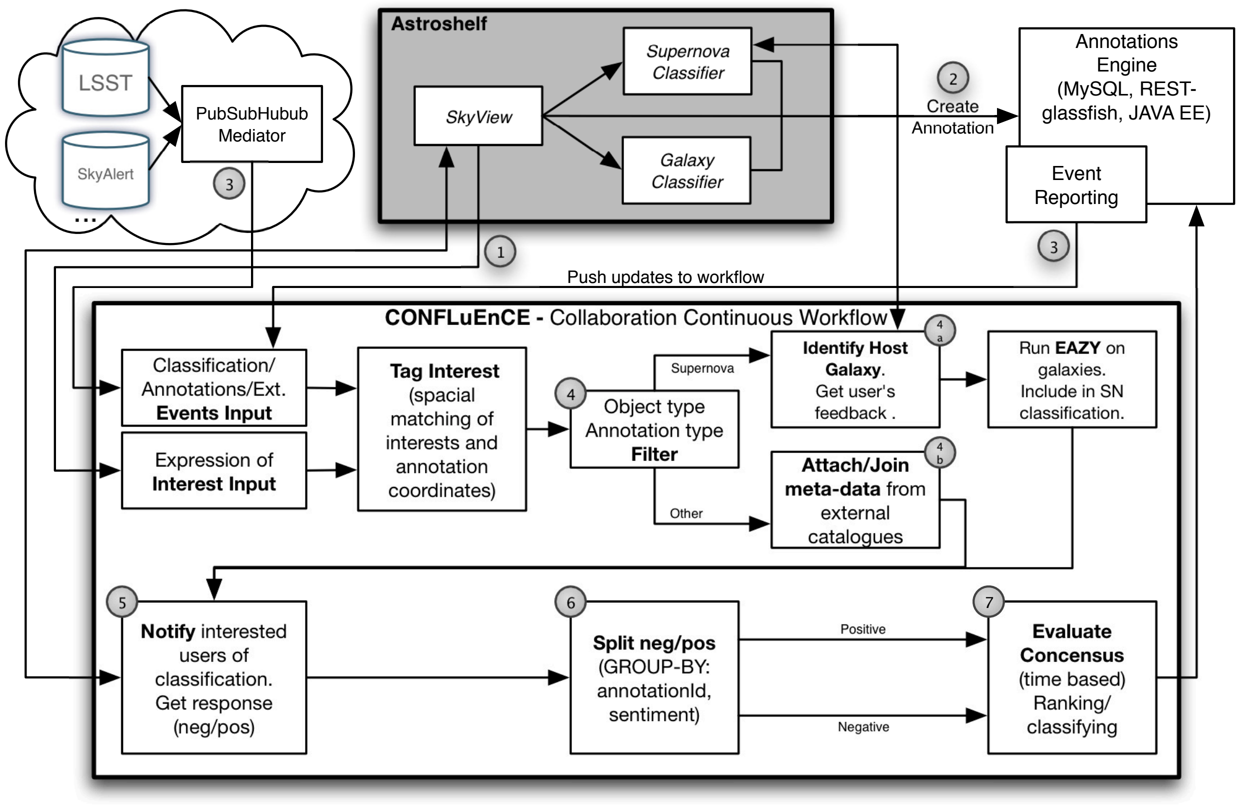
roles 1-3. As part of this application the only actors that we had to implement from scratch were the push communication enabling actors. The rest of the workflow uses “off-the-shelf” actors provided by Kepler. Even for processing window operator results, such as counting the size of the window every second, we used the already available “Array Length” actor, which is inherently compatible with our WindowedReceiver implementation, since produced windows are encapsulated in Array Tokens (Section 4.4).

Furthermore, to test the capacity of the system and its ability to handle high loads, we implemented the roles of customers and warehouse managers as background processes that automatically interacted with the workflow. In the case of the customers the automated process would randomly pick some products from a list and submit them as an order. The warehouse manager process would wake up in intervals and service orders from its work list. As an initial stress test we spawned twenty customers and three warehouse manager processes to see the robustness of our system. The system was running without any problems, until we stopped it after three hours. A more detailed stress test of the system was performed using the Linear Road Benchmark, describe in the next chapter.

4.7.2 AstroShelf

In the context of the NSF project funding the research and development of CONFLuEnCE, we have been working with a group of astrophysicists to develop a complete platform, called *AstroShelf* which will enable them to collaboratively annotate sky objects and phenomena, as well as visualize parts of the sky using different algorithms. This includes a user interface (dashboard) with the ability to display sky images, an annotations management system and a monitoring module for real-time processing of annotations and sky update events. The monitoring module is realized within CONFLuEnCE. A high-level design of the system is shown in Figure 21.

Specifically, we have designed a continuous workflow to run on CONFLuEnCE as part of the monitoring module. The goal of this workflow is to monitor the activity of inserting, updating or deleting annotations as well as integrating the detection of transient events from various sky surveys that are of interest to the users, all in real-time. After processing these events the workflow will ask for feedback from the users. By interacting with the workflow, the users may refine the



LSST
SkyAlert
...
PubSubHubub Mediator
3
Astroshelf
SkyView
Supernova Classifier
Galaxy Classifier
2
Create Annotation
Annotations Engine (MySQL, REST-glassfish, JAVA EE)
Event Reporting
3
1
Push updates to workflow
CONFLuEnCE - Collaboration Continuous Workflow
Classification/ Annotations/Ext. Events Input
Expression of Interest Input
Tag Interest (spacial matching of interests and annotation coordinates)
4
Object type Annotation type Filter
Supernova
Other
4a
Identify Host Galaxy. Get user's feedback .
4b
Attach/Join meta-data from external catalogues
Run EAZY on galaxies. Include in SN classification.
5
Notify interested users of classification. Get response (neg/pos)
6
Split neg/pos (GROUP-BY: annotationId, sentiment)
Positive
Negative
7
Evaluate Concensus (time based) Ranking/ classifying

Figure 21: High-level design of a continuous workflow for the Astroshelf collaboration platform

annotations, iterating over them until they reach a consensus. Manipulating annotations can be done using the SkyView, the Galaxy Classifier or the Supernovae Classifier.

The system interactions and flow of events are as follows (numbered as in Figure 21):

1) Using AstroShelf's user interface the astronomers can define and name areas in the sky that are of interest to them. Additionally, they may define the type of events they are interested in (e.g., new annotation, new supernova, galaxy classification, etc.) This expression of interest is pushed to CONFLuEnCE and it is registered into an R-Tree spacial index [24] which resides inside the actor "Tag Interest". We used an R-Tree for its ability to index multi-dimensional information and quickly match spatial queries with the areas defined by the users.

2) Using the SkyView the users can annotate objects, group of objects or arbitrary points in the sky with any information they deem important to share. Using either the Galaxy or Supernovae clas-

sifiers the users can classify types of galaxies or supernovae, respectively. These classifications are recorded as annotations as well. All of these annotations are inserted into the Annotations Engine through a specialized API.

3) Every time a new annotation is inserted into the Annotations Engine, this event is detected by the Event Reporting module and directly forwards it to the continuous workflow on CONFLuEnCE. Other types of events pushed to the monitoring workflow are transient events detected by various sky surveys (e.g., LSST[3]), which are also available through an aggregation service, called SkyAlert.
4) Once the aforementioned events enter the system they are tagged by the "Tag Interest" actor with the user ids of those who previously expressed interest in the area, and type of the object attached to the annotation (interaction 1). Then they are filtered depending on the event type, and follow different paths in the workflow.
   a. Supernovae events need to be handled differently than other events. Firstly, the supernova object is matched with its host galaxy and this matching is verified by the user through the browser interface. Then the object is run through the EAZY algorithm to calculate the redshift probability distribution.
   b. All other events are joined with data available from various external catalogs. This information will help the users when they provide feedback about an annotation.
5) Once all the necessary data have been attached to the data objects, then the users tagged on those objects are notified directly on their browser (as shown in figure 21) or through email, SMS, twitter etc.
6) The notified users then use the user interface to express their opinion on the annotated objects or classifications. The opinions are tagged as positive or negative and split accordingly. The "Split neg/pos" step groups the opinions according to the object id and the sentiment of the opinion. The window size of the group-by is time based to measure the density of each opinion with respect to temporal bounds.
7) The final step of the workflow is to evaluate the overall consensus on the various opinions (positive or negative). It will then create another annotation on the object that captures the consensus. This new annotation goes back into the workflow and the cycle continues.

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

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

4.8 SUMMARY

In this chapter, we described the implementation of CONFLuEnCE, our Continuous Workflow Execution Engine. We examined Kepler and its actor-oriented modeling, which the implementation of CONFLuEnCE is based upon. Then, we described how the basic constructs of the CWf model were specifically implemented within CONFLuEnCE. These components are the CWF Director, the Windowed Receiver, the Time Keeping methods, and the push communication patterns. Finally we evaluated our implementation by building two representative monitoring applications from the business and scientific domains, respectively.

5.0 STAFiLOS: STreAm FLOw SCHEDULING FOR CONTINUOUS WORKFLOWS

In this chapter, after providing an overview of our scheduling framework in CONFLuEnCE in Section 5.1, we examine the various directors available in the PtolemyII/Kepler suit with respect to their scheduling policies and communication mechanisms in Section 5.2. Based on their characteristics and the way they interact with the data and other workflow components we designed our framework which we describe in Section 5.3. We present four different schedulers that we have designed and implemented to work with our framework in Section 5.4, and we finally evaluate their performance, using the Linear Road Benchmark in Section 5.5.

5.1 SCHEDULER FRAMEWORK IN CONFLUENCE

The PNCWF Director that was described earlier is thread-based, and resource management and allocation to the various threads is handled directly by the Operating System. This leaves no margin for QoS-based optimizations, which are suitable for monitoring applications. Since execution in Kepler is dictated by the Director component, we could have implemented specific scheduling policies in different CWf director implementations. Instead of that, we adopted a slightly different philosophy. We designed STAFiLOS [44], a framework to integrate scheduling through a generic and pluggable scheduled CWf Director that can be populated with different scheduling policies. We applied what we learned from implementing the PNCWF director, the WindowedReceiver, the time keeping method and the token encapsulation inside CWEvents, and reused these components within STAFiLOS, while extending them specifically to work in this new execution model, while at the same time supporting the previous one. We also added a more generic actor statistics module, which can now be used by any CWf scheduler within STAFiLOS, to provide runtime statistics on a number of different metrics (e.g., time per invocation, input rate, output rate, etc.).

In this dissertation we focus on how to enable scheduling of the actors in a workflow, in order to better share the CPU resource and minimize the latency in the production of results. We have designed a scheduling framework within CONFLuEnCE called STAFiLOS. The goal of the framework is to enable the easy application of different scheduling policies, where the developer of a new policy only needs to implement/extend the framework's interface to achieve the desired effect. The framework exposes many types of runtime statistics for the scheduler to use and make smart CPU allocation decisions. At its current version, the framework only supports single core architectures, but this initial design has paved the road to design a multi-core scheduling framework in the future.

5.2 EXISTING SCHEDULERS/DIRECTORS IN PTOLEMYII/KEPLER

During our effort to implement the basic constructs of the Continuous Workflow Model we have already examined the five main directors which are part of Kepler's models of computation domains, as described in Section 4.2. Namely the Synchronous Data Flow Director (SDF), the Dynamic Dataflow Director (DDF), and the Process Network Director (PN), which our own Process Network Continuous Workflow Director is based upon. When we considered a scheduling framework for CONFLuEnCE we have also looked beyond those five, through the list of directors available in PtolemyII (upon which Kepler is built).

Below is the description of each of the directors we have identified as part of PtolemyII:

1) *Component Integration (CI)*: The CI domain supports two styles of interaction between actors, push and pull. In push interaction, the actor that produces data initiates the interaction. The receiving actor reacts to the data. The computation then proceeds in a data-driven manner. In pull interaction, the actor that consumes data decides when the interaction takes place, and the computation proceeds as demand-driven. Actors are divided into active and inactive. Each active Actor is managed by an Actor Manager which is a thread. Inactive actors are controlled by the director, and get to run whenever an active actor does a pull or push request.

2) *Communicating Sequential Processes (CSP)*: This is similar to the PN Director with synchronous message passing. The receiver is the synchronization point of both producer and consumer of data on a channel.

3) *Discrete Time (DT)*: Timed extension of the Synchronous Data Flow (SDF) domain. Employs static scheduling, which means that token consumption rates are defined at design time and the schedule of the actors is defined based on those rates. The director keeps a global time as well as local times at each actor. Period defines the model time spent per iteration. Assumes: Uniform Token Flow, and Causality, i.e., each token produced only depends on the previous tokens. DT is a timed super-set of SDF.

4) *Finite State Machine (FSM)*: Implements and governs the execution of a modal model, which consists of a set of states and transitions. Each state consists of its own sub-workflow model. When the model is fired, the model's input tokens are transferred to the active state. The model proceeds with execution depending on the transitions that are being activated. The model can run in deterministic and non-deterministic modes.

5) *Heterochronous Dataflow (HDF)*: An extension of the Synchronous Dataflow (SDF) domain that implements the HDF model of computation [1], which is a heterogenous composition of SDF and finite state machine (FSM). The semantics of HDF allow rate changes through state transitions of FSM, while within each state the system can be considered as an SDF model. This director recomputes the schedules dynamically. To improve efficiency, this director uses a CachedSDFScheduler. A CachedSDFScheduler caches schedules labeled by their corresponding rate signatures, with the most recently used at the beginning of the queue. Therefore, when a state in HDF is revisited, the schedule identified by its rate signatures in the cache is used, and there is no need to recompute the schedule.

6) *PetriNet*: This is the basic Petri Net model where Places and Transitions form a bipartite graph and enabled Transitions can fire randomly. It also allows Transitions to be replaced by any other Actors in PtolemyII.

7) *Ptera*: Implements the Event Relationship Graph semantics, and can be used by Ptera controllers (instances of PteraController) in Ptera modal models (instances of PteraModalModel).

8) *Synchronous Reactive (SR)*: The SR model of computation has a notion of a global "tick" of a clock, and at each tick of the clock, each port either has a value or is "absent". The job of this

director is to determine what that value is, for each connection between ports. An iteration of this director is one tick of this global clock. Subclasses the StaticSchedulingDirector.

9) *Timing Definition Language (TDL)*: All actions inside a TDL module are executed periodically and the timing information is specified in parameters. This director parses the parameters and builds a schedule for all the TDL actions. The schedule is represented in a graph showing the dependencies between the TDL actions (see TDLActionsGraph). Type of FSM director.

10) *Timed Multi-task (TM)*: A director that implements a priority-driven multitasking model of computation, in simulation mode (i.e., the execution costs are simulated). This model of computation is usually seen in real-time operating systems. Each actor is called a task. It can request an interrupt by calling the *fireAt()* function of the director. Actors have priorities and executionTime parameters. These parameters can be defined per input port, if the actor reacts differently on each port. It assumes a single resource (i.e., CPU) and execution may be preemptive. Scheduling is priority based, and is achieved by dispatching *TMEvent* objects to their corresponding actors, which have a priority and remaining time parameters (among others). Priority is from the destination port, which may get its priority from the containing actor. These events are queued at the event dispatcher and then dequeued and forwarded to the corresponding receiver, after which actor execution is performed. Interrupt events have a timestamp and are handled in a different queue. The execution can be synchronized with real time, but the execution costs are still simulated.

11) *Wireless Domain*: This director is nearly identical to the DE director with the only difference being that it creates instances of WirelessReceiver. It simulates a wireless environment where the ports do not need wired connections.

12) *Timed Process Network*: Same as PN but also introducing the notion of global time. The main difference with the PN model is that active processes, in addition to blocking when trying to read from a channel (read-blocked), and when trying to write to a channel (write-blocked), can also block when waiting for time to progress (time-blocked). Time can progress for an active process in this model of computation only when the process is blocked.

13) *Distributed SDF*: The Distributed-SDF domain is an extended version of the existing SDF Domain that performs a simulation in a distributed manner. A distributed platform is required to perform the simulation. Every node in the distributed platform has to run a server application

(DistributedServerRMIGeneric). A Jini service locator must be running on one of the nodes (peer discovery).

By examining all these domains in PtolemyII, we have learned more about the interactions between the standard components and the domain specific components. The domain most closely related to scheduling of workflow actors based on priorities is the Timed-Multitask domain. We took a closer look at it and designed our scheduling framework based on its director, while at the same time making it generic to accept different schedulers implementing different scheduling policies.

A taxonomy classifying each main director is presented in Table 1. *Actor Interaction* refers to the way data is being passed from one actor to the other and on how the actors react to the data (e.g., actively monitoring the input queues and firing as soon as a new event arrives in the queue, or waiting for the director to fire them, etc.). *Computation Driver* refers to the way a director drives the execution of the workflow (e.g., if it is pre-compiled that means that the schedule is static throughout the execution of the workflow, if it is data-driven then computation is mostly dependent on the existence of data at the actors' input queues). *Scheduling* refers to how the director makes the decision on which actor to fire next (e.g., if it is OS-based the decision is left to the operating system, if it is consumption-based that means that the decision is made based on the availability of data on the actors' input queues in combination with their declared consumption rates). *Time-based* refers to whether or not a director keeps track of time. Lastly, *QoS* refers to whether or not a director is trying to make scheduling decisions based on some sort of priority function, in order to optimize a specific metric.

5.3 STAFiLOS DESIGN

From our evaluation of the existing models of computation in the previous section, we have concluded that the model closest to what we wanted to achieve with STAFiLOS is the Timed Multitask model, which is the one we considered most closely. As we have already mentioned, the TM domain director fires actors based on whether or not they have events waiting to be processed. It keeps a ready queue with events that need to be processed, sorted based on their priorities which

Director	**Actor Interaction**	**Computation Driver**	**Scheduling**	**Time based**	**QoS**
SDF	Director: Topology-driven	Pre-compiled	Pre-compiled	N/A	N/A
DDF	Push	Data-driven	Iterative/ Consumption Based	N/A	N/A
PN	Push	Data-driven	Thread/OS	N/A	N/A
DE	Director: Event Queue	Event-driven	Event Order	Yes (global)	N/A
CN	Director: Topology-driven	Pre-compiled	Pre-compiled	Yes (global)	N/A
CI	Push/Pull	Data-driven	Thread/OS	N/A	N/A
CSP	Push Synchronous	Data-driven	Thread/OS	Yes (global)	N/A
DT	Director: Topology-driven	Pre-compiled	Pre-compiled	Yes(global or local)	N/A
HDF	Director: Topology-driven	Pre-compiled	Multiple Pre-compiled	N/A	N/A
SR	Synchronous Reactive	Pre-compiled	Pre-compiled	Yes(global tick)	N/A
TM	Director: Priority Queue	Priority-based	Pre-emptive Priority-based	N/A	Priority
TPN	Push	Data-Time-driven	Thread/OS	Yes (global)	N/A
PNCWF	Push-Windowed	Data-Windowed-driven	Thread/OS	Yes (local)	N/A

Table 1: Taxonomy of Directors found in Kepler (first group) and ProlemyII (second group) as well as our PNCWF Director (third group)

are derived from their associated actors or ports. Event flow goes from the output port to the TM director, where it is queued, and only when the actor is to be scheduled for firing does the event become available at the firing actor's input port. This is how we designed the event flow within STAFiLOS as well (see Figure 22).

STAFiLOS consists of three main components. The *Scheduled CWF Director* which is the main component that interacts with the workflow model and the management modules ran by Kepler. It is also responsible for initializing the actors, ports, receivers, and the scheduler, as well

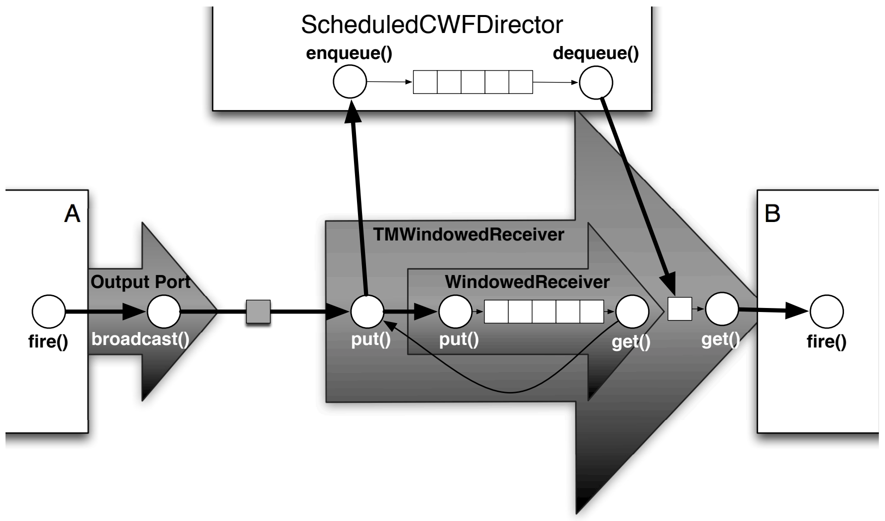


Figure 22: Event flow in the STAFiLOS scheduling framework

as transitioning the workflow model through the various execution stages within each iteration. The second component is the *TM WindowedReceiver*. This component is based upon the TM Receiver of the TM PtolemyII domain, but extends our WindowedReceiver implementation described in Section 4.4. Lastly, the *Abstract Scheduler* is an implementation of all the basic functionality which can be extended by the actual scheduler implementation. All three components and their interactions, are depicted in Figure 23; we describe them in more detail next.

The *Scheduled CWF Director* (SCWF) is the main component that interacts with the workflow model (i.e., actors, ports, sub-workflows) and the management modules ran by Kepler. It is also responsible for initializing the actors, ports, receivers, and the scheduler, as well as transitioning the workflow model through the various execution stages within each iteration. The SCWF Director is also schedule-independent, thus a scheduling policy implementation which extends the Abstract Scheduler is being controlled by it.

The *TM Windowed Receiver* is based upon the TM Receiver of the TM PtolemyII domain, but extends our WindowedReceiver implementation described in Section 4.4. The TM Windowed Receiver interacts with the SCWF Director as shown in Figure 22. When an upstream actor produces an event on its output port it broadcasts it to all the remote downstream receivers connected to it. The TM Windowed Receiver extends the `put()` and `get()` methods of the Windowed Receiver. When an event is passed to the `put()` method, it is propagated to the Windowed Receiver's `put()` method, which in turn is queued in the appropriate group-by queue. During the same call, the window semantics are evaluated on that queue and if a window is produced it is returned to the TM Windowed Receiver `put()` method. The produced window is then enqueued at the actor's ready queue at the Scheduled CWF Director. When the director decides to run that actor (Actor B in the example), it dequeues the event and adds it to a buffer inside the TM Windowed Receiver, rendering it available at the next `get()` call by the `fire()` method of the actor. Besides the regular events being queued at the director, the windowed receivers that compute timed windows also register "window timeout events" which are used to produce timed windows before an event from the next window arrives in order to close and produce the current window.

The *Abstract Scheduler* component implements most of the basic functionality of a scheduler but it is not a complete scheduler. However, it can be extended and made fully functioning by an actual scheduler implementation. The Abstract Scheduler maintains a list of the workflow's actors, and maps them to queues of events (sorted by timestamp) that should be propagated to each actor's corresponding input ports when they are to be scheduled for execution. It also maintains a mapping between actors and their current state as well as a list of flags denoting whether a state is valid or not. Three states are defined: ACTIVE which denotes that the actor can be considered for firing at the current iteration, WAITING which denotes that the actor is waiting for something to happen within the scheduler before it can be run, and INACTIVE which denotes that the actor currently has no events to process. State transition rules are implemented within each actual scheduler implementation. Finally, it keeps two priority queues. One for the active actors and one for the actors who are waiting. Basically when an actor switches state from being ACTIVE to WAITING, it is removed from the active queue and it is placed in the waiting queue. If an actor is INACTIVE it is not placed in any of the priority queues. The `getNextActor()` method returns the next actor from the active priority queue. The priority queues are sorted based on a *Queue Comparator* object which is provided by the scheduler implementation. This comparator could be based on actor priorities defined by the workflow designer or some kind of dynamic priorities calculated at runtime based on the actor statistics. The Abstract Scheduler also provides hooks where the director can signal the scheduler for the director's state changes, such as the start and

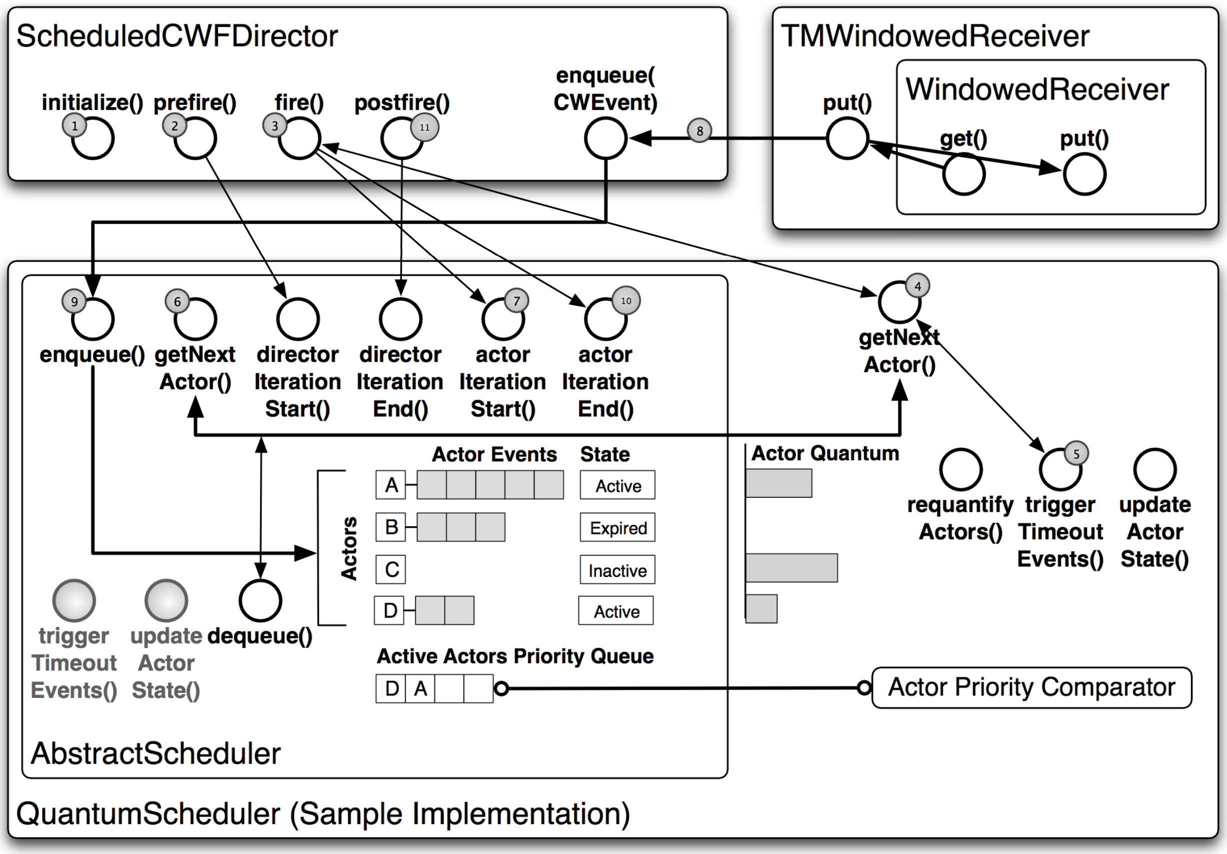


Figure 23: The STAFiLOS scheduler framework in CONFLuEnCE

end of a director's iteration, the start and end of an actor's iteration, etc. In the next section, we describe the implementation of three schedulers showing for each one their transition rules and the implementation of the Abstract Scheduler methods.

5.4 IMPLEMENTED CWF SCHEDULERS

We have used the STAFiLOS scheduling framework to implement three schedulers with different characteristics, which we will describe in this section. The first one is the *Quantum Priority-Based scheduler*, the second one is the traditional fair Round Robin scheduler, and the third one is the

Rate-Based scheduler which is one of the best performing QoS scheduler for CQs, discussed in Section 2.2.

5.4.1 The Quantum Priority Based Scheduler (QBS)

The Quantum Priority Based Scheduler is largely based on the Linux Process scheduler [6]. The actors are assigned priorities by the workflow designer and based on those priorities the scheduler assigns a number of basic quanta, as given by Equation 5.1.

$$q = \begin{cases} (40 - p) \times b, for\ p >= 20 \\ (40 - p) \times 4b, for\ p < 20 \end{cases} \tag{5.1}$$

Where, p is the actor's priority, b is the basic quantum (a scheduler static parameter), and q is the quantum to be given to the actor whenever a re-quantification process is initiated. Source actors are treated independently of the rest of the actors in order to regulate better the flow of data coming into the workflow. Correctly tuning the scheduling policy regarding the source actors can play a significant role in the overall behavior of the QoS metrics. In the case of QBS the source actors are being scheduled in regular intervals (i.e., after x internal actor invocations).

The quantum value for each actor represents its allowance in microseconds that can run before the next re-quantification period. Actors that have events ready to be processed are divided into *active* and *waiting* depending on whether they have a positive quantum or not. The active actors are sorted by ascending priority. If two actors have the same priority then they are treated as FIFO. When an active actor runs for a while and suddenly runs out of time, given its quantum, it is moved into the waiting queue. Once all the actors with events run out of quanta and are moved into the waiting queue, the scheduler initiates a re-quantification process and swaps the two queues (i.e., the waiting queue becomes the active queue and vice-versa). There is a possibility that an actor consumed more than its remaining quantum in its last iteration and ended up having a negative quantum. If that negative value is significant, there is a chance that even after re-quantification, it still has a negative quantum value. In that case it stays in the waiting queue. An actor that processed all of its ready events, transitions into the *inactive* state and its quantum value is preserved until new events are ready to be processed by it.

The state conditions for an actor A in the Quantum Scheduler are:

- ACTIVE:

  A **is not a source actor:** Has events waiting in its queue **AND** has a positive quantum value.

  A **is a source actor:** Has a positive quantum value **AND** has not fired yet in the current director iteration.

- WAITING:

  A **is not a source actor:** Has events waiting in its queue **AND** has a negative quantum value.

  A **is a source actor:** Has a negative quantum value **OR** has fired in the current director iteration.

- INACTIVE:

  A **is not a source actor:** Has no events waiting in its queue.

  A **is a source actor:** *A source actor does not transition into this state.*

The interactions between the components of STAFiLOS when executing QBS are depicted in Figure 23. These are also applicable to any scheduler implemented within the STAFiLOS framework. When the execution of the workflow begins, the director carries out the *initialization* of all the components. It also signals the scheduler, in order for it to carry out its own initialization. As part of this step, the source actors are being registered to the scheduler, which, depending on the implementing policy, decides how to treat them. After that, the director enters the director iteration cycle, first with the *pre-fire* state, again while signaling the scheduler about it. Next, at the *fire* state the director calls the scheduler `getNextActor()` to get the next actor to be fired. At this point the scheduler polls the next actor from the active queue, which is sorted using the Comparator attached to the active queue. The Comparator implements the scheduler's priority function. The actor returned might be a source actor, an internal actor or an output actor. If it is an internal or output actor then an event from the corresponding actor's event queue is dequeued and made available at the actor's input port. Then the director pre-fires the actor and if that returns true, it goes on to fire the actor while starting the necessary timers to measure the cost of the actor.

During the actor's firing, new events will be produced at its output ports. The events go through the flow we explained in Figure 22, and end up being enqueued at the scheduler. An event has a reference to its corresponding actor and, based on that, it is enqueued on the actor it belongs to. At the same time, the actor's input rate as well as the producing actor's output rate statistics are being updated. At this point the actor's state is updated. If it was inactive, the scheduler will re-evaluate its state (e.g., assign a quantum to it and put it in the active queue). Once the actor post-fires, the director notifies the scheduler in order for it to calculate its cost and other statistics it needs to function.

The director's iteration cycle ends when a call to the method `getNextActor()` returns `null`. That is when the director post-fires, notifies the scheduler and restarts the iteration. At this point the scheduler usually performs some maintenance tasks (e.g., re-quantify the actors, recalculate their states, update statistics etc.).

5.4.2 Round-Robin Scheduler (RR)

We implemented the Round-Robin scheduler using the STAFiLOS framework. The Round Robin scheduler works in similar manner with the QPB scheduler. It does not take into account any priorities though. At each scheduling period it gives the active actors a time slice (quantum) on which they are allowed to run. They are then scheduled to process their available events in a round robin manner. If they manage to process all of their current events they transition to the *inactive* state and give up any remaining slice. If they consume their slice, they transition to the *waiting* state, and remain in that state until the next period, to process the remaining of their available events. New events can be added to an actor's ready queue even within the current period. The actor processes them if it has enough time to do so during the current period. If an actor is inactive and new events arrive, a slice is assigned to it and the actor is placed at the end of the Round-Robin queue.

The state conditions for an actor A in the Round-Robin Scheduler are:

- ACTIVE:

  A **is not a source actor:** Has events waiting in its queue **AND** has a positive quantum value.

  A **is a source actor:** Has a positive quantum value **AND** has not fired yet in the current director iteration.

- WAITING:

  A **is not a source actor:** Has events waiting in its queue **AND** has a negative quantum value.

  A **is a source actor:** Has a negative quantum value **OR** has fired in the current director iteration.

- INACTIVE:

  A **is not a source actor:** Has no events waiting in its queue.

  A **is a source actor:** *A source actor does not transition into this state.*

5.4.3 Rate Based Scheduler (RB)

The third scheduler we have implemented is the Rate Based Scheduler which is based on the Rate Based scheduling policy described in [57]. The actors are once again divided into active and waiting, and their priorities are dynamically calculated based on their selectivity and cost as shown in Equation 5.2.

$$Pr(A) = \frac{S_A}{\overline{C}_A} \tag{5.2}$$

$Pr(A)$ is the dynamic priority of actor A. S_A is the actor's global selectivity, and $\overline{C}_A$ is the actor's global average cost, as they are defined in [57]. When an actor is shared among multiple workflow paths (i.e., is connected to more than one downstream actor) then we add up the downstream global costs and global selectivities of each path.

The *cost* (c_x) of an actor is the average amount of time required to process one data item. The *selectivity* (s_x) of an actor is the number of data items produced after processing one tuple for c_x time units (i.e., the output rate divided by the input rate).

The global selectivity of an actor A is the number of data items produced at any downstream output actor once a data item is processed at actor A. It is calculated as the product of all the selectivities of all the downstream actors, as shown in Equation 5.3.

$$S_A = s_x \times s_y \times ... \times s_r \tag{5.3}$$

Where r is the output actor. If an actor splits into multiple branches then the global selectivity of each branch is added together to the actor's global selectivity.

The global cost of an actor A is the expected time to process a data item from actor A until the output. The calculation of the global cost is shown in Equation 5.4.

$$\overline{C}_A = (c_x) + (c_y \times s_x) + ... + (c_r \times s_r - 1 \times ... \times s_x) \tag{5.4}$$

Where r is the output actor. If an actor splits into multiple branches then the global selectivity of each branch is added together to the actor's global cost.

Event processing in this scheduler is divided into periods. At each period the scheduler processes all the events that have been enqueued during the previous period. Any newly enqueued events are kept in a buffer and are put into their corresponding actor's queues once the current period is over. The end of a period is signaled by the director's end of iteration, which happens when the active actors queue becomes empty. The active actors queue is empty when all the actors have no more events to process and all the source actors have executed once during the current period. The dynamic priorities are re-evaluated at the end of each period.

The state conditions for an actor A in the Rate Based Scheduler are:

- ACTIVE:

  A **is not a source actor:** Has events waiting in its queue.

  A **is a source actor:** Has not yet fired in the current period.

- WAITING:

  A **is not a source actor:** Has no events waiting in its queue **AND** has events waiting in the next period buffer.

  A **is a source actor:** Has fired in the current period.

- INACTIVE:

  A **is not a source actor:** Has no events waiting in its queue or buffer.

  A **is a source actor:** *A source actor does not transition into this state.*

5.5 EVALUATION

Given the lack of an appropriate Continuous Workflow benchmark, we evaluated the STAFiLOS framework by running the various schedulers on a continuous workflow implementation of the Linear Road benchmark [3]. The Linear Road benchmark has been established as the standard benchmark for stream processing systems. Linear Road has been endorsed as an DSMS benchmark by the developers of both the Aurora [10] (a collaboration among Brandeis University, Brown University and MIT) and STREAM [40] (from Stanford University) DSMS. The goal of our evaluation was not to compare the performance of STAFiLOS with those of DSMSs, but to evaluate STAFiLOS scalability and overhead compared to the OS schedulers. Any comparison with a DSMS is meaningless given that DSMSs' and STAFiLOS' functionality are different.

5.5.1 Linear Road Benchmark

Linear Road simulates a toll system for the motor vehicle expressways of a large metropolitan area. The tolling system uses "variable tolling": an increasingly prevalent tolling technique that uses such dynamic factors as traffic congestion and accident proximity to calculate toll charges. Linear Road specifies a variable tolling system for a fictional urban area including such features as accident detection and alerts, traffic congestion measurements, toll calculation and historical queries. For the purpose of our evaluation we only focused on the stream processing aspect of the benchmark and thus we excluded the historical queries.

The application provides a single feed of car position updates. Each car updates its position every 30 seconds. That includes its position (expressway id, direction, lane, segment of the highway) and current speed. While the workflow processes this feed, it is required to provide notifications to the cars about their toll charges every time they switch segment, based on a set of conditions. It also needs to alert them of any accidents which happened down the road in order for them to exit the highway and choose another route.

Our workflow implementation of the Linear Road benchmark consists of two levels of workflow hierarchy. The top level consists of all the major tasks and wiring between them, required by the application, and is governed by a Continuous Workflow director (either a STAFiLOS based

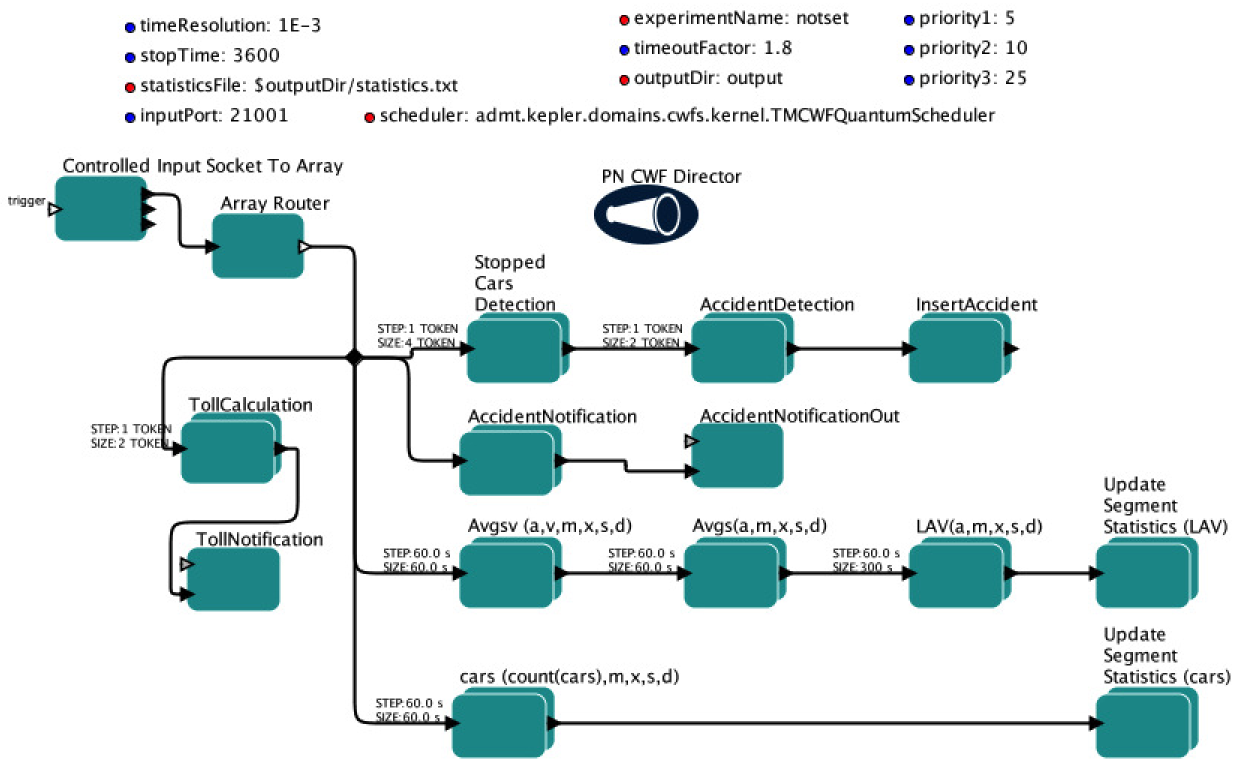


Figure 24: The Linear Road Benchmark top level workflow

one or the thread based PNCWF Director). The second level of the hierarchy consists of sub-workflows of main tasks in the top level and represent tasks like detecting stopped cars, calculating the number of cars in each segments etc. These are all governed either by Kepler's SDF directors (Section 4.2) if the consumption and production rate is constant, or by Kepler's DDF directors if the consumption and production rates are more fluid, e.g., if the sub-workflow includes decision points and does not have constant production rates at the internal actors. Our implementation also requires the support of a relational database to store statistics on the road congestion as well as the recent accidents detected. For this purpose we used MySQL.

The workflow, as shown in Figure 24, is divided into three main areas. One to take care of the accidents, one for calculating the segment statistics, and one for calculating and notifying the cars about their tolling charges.

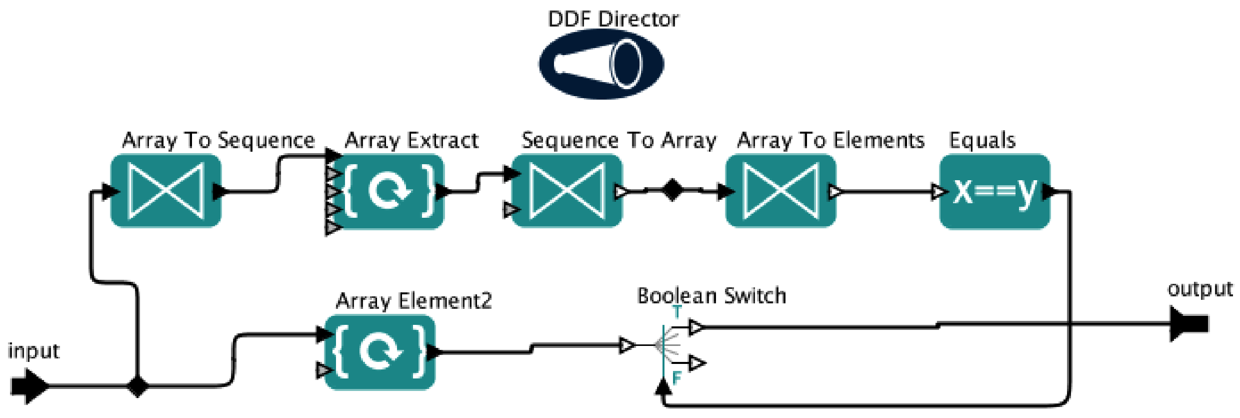


Figure 25: The Linear Road Benchmark Stopped Cars Detection sub-workflow

5.5.1.1 Accident Detection and Notification The accident detection consists of three composite actors. The first one is for detecting stopped cars. If a car reports the same location in 4 consecutive position reports then it is considered stopped. The sub-workflow defining this functionality is depicted in Figure 25. The input port of this actor has the following window semantics: {Size: 4 token, Step: 1 token, Group-by: car ID}. When fired, this actor processes a window of the last four position reports of each car and compares the positions. If the car is stopped then the actor outputs the first of those position reports and sends it to the Accident Detection actor.

The Accident Detection actor is the second one in this pipeline, and the implementation is shown in Figure 26. This actor takes windows of two position reports, which represent the same position, and compares the car IDs. If the car ids are different, and they are not in an exit lane, that means a car accident is in progress. The input port of this actor defines the following window semantics: {Size: 2 tokens, Step: 1 token, Group-by: position}. If an accident is detected, then this is propagated to the Insert Accident actor which records the incident into the relational database. We omit the description of the third actor, because it just consists of constructing an INSERT statement and submitting it to the database.

The application also requires that any car entering a range of segments upwards an accident, be notified within 5 seconds of the position report. The notification is generated by the Accident

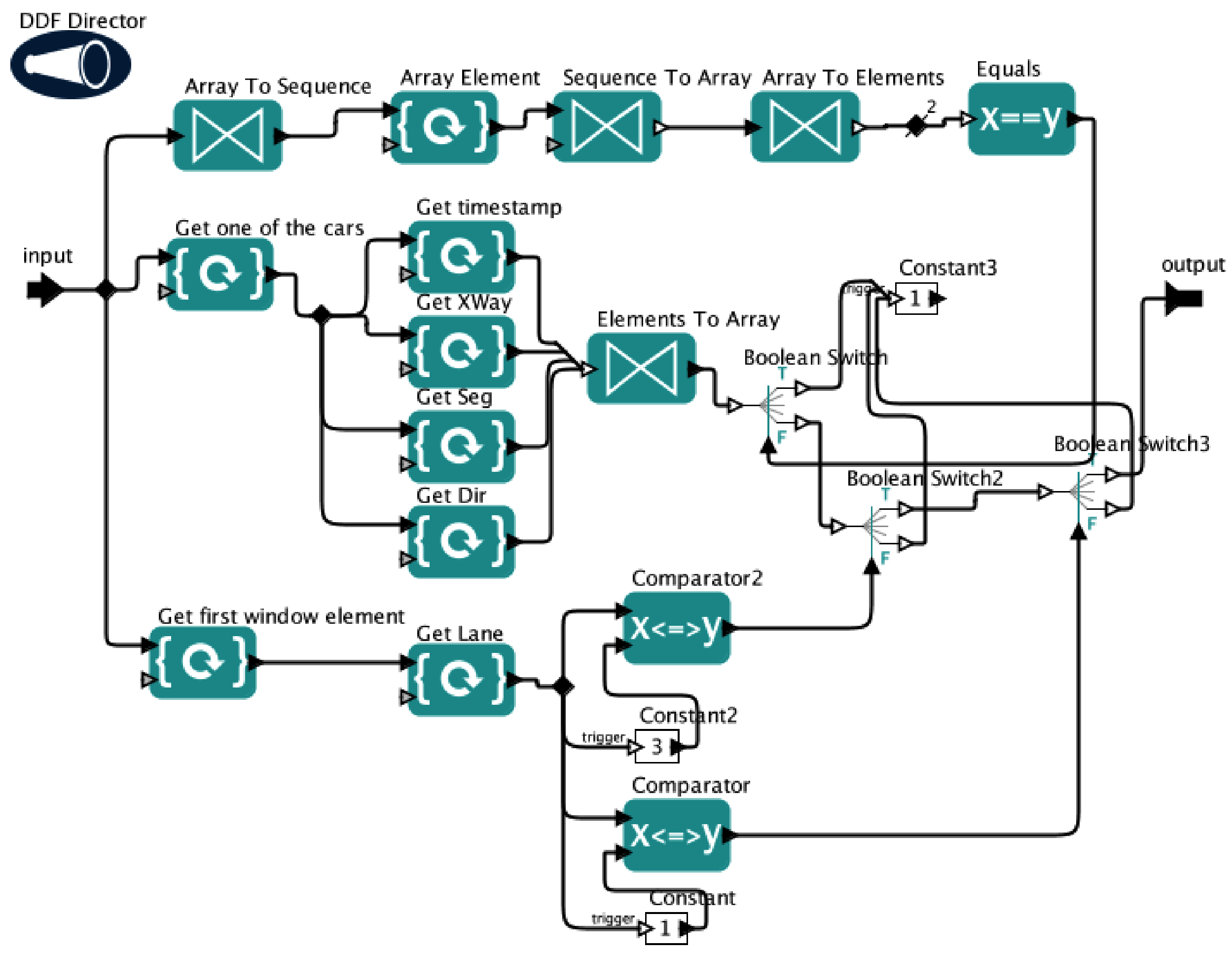


Figure 26: The Linear Road Benchmark Accident Detection sub-workflow

Notification composite actor which, for each position report of a car, checks in the database to see if there is a car accident registered within four segments downstream of each car. The actor is shown in Figure 27.

5.5.1.2 Segment Statistics The Toll Calculation formula relies on the system keeping some statistics regarding each segment of the expressway. Specifically, the tolls depend on the number of cars present in a segment in the previous minute and the "Latest Average Velocity" (*LAV*) value for the segment. LAV is the average of the average speed of all the cars that passed through that

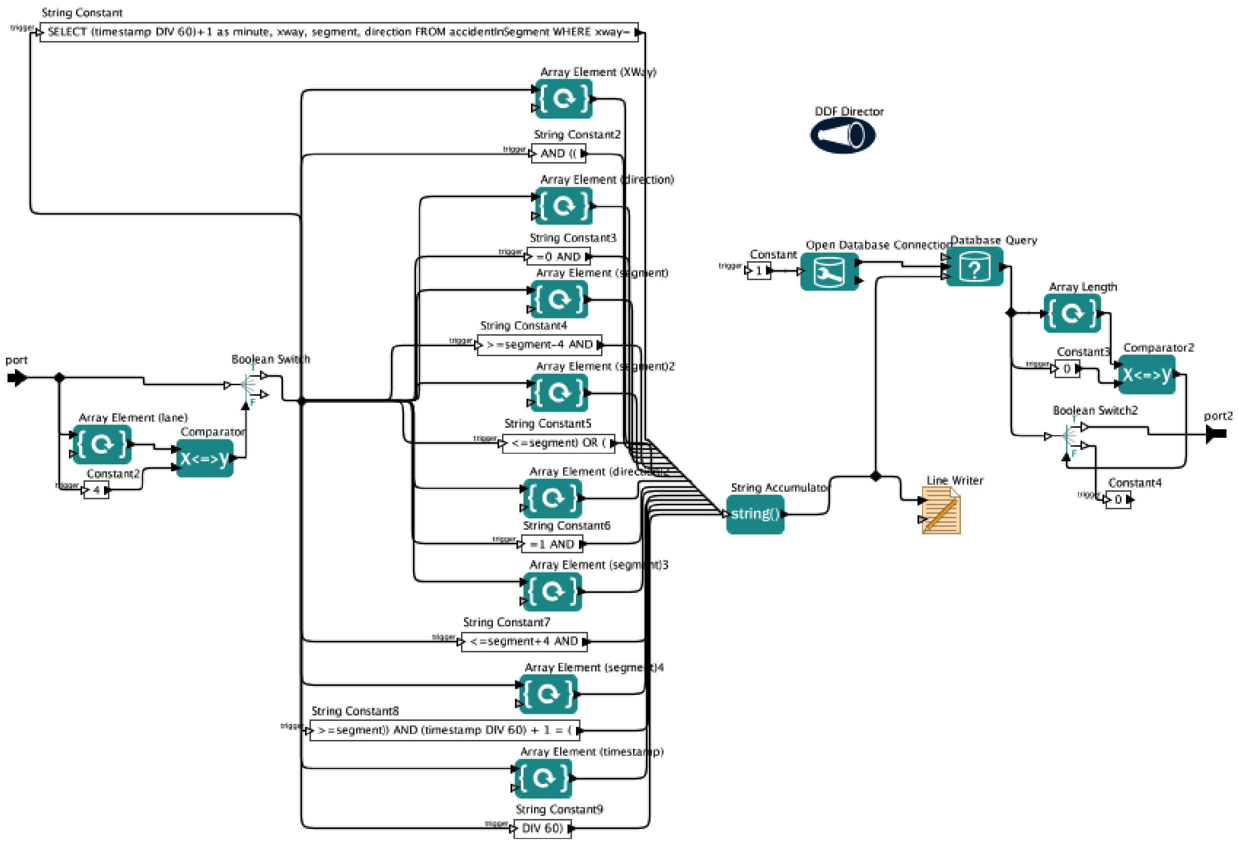


Figure 27: The Linear Road Benchmark Accident Notification sub-workflow

segment every minute, for the past five minutes. In order to calculate the LAV value, we first calculate the average speed per car, for each segment (*Avgsv* composite actor in Figure 28), and then the average speed of all the cars in the segment (*Avgs* composite actor).

The actor that calculates the average speed of a car has the following window semantic definition: {Size: 1 minute, Step: 1 minute, Group-by: Car ID, Expressway, Direction, Segment number}. The output from this actor is propagated to the *Avgs* actor which calculates the overall average speed per segment, per minute, and has the following semantics: {Size: 1 minute, Step: 1 minute, Group-by: Expressway, Direction, Segment number}.

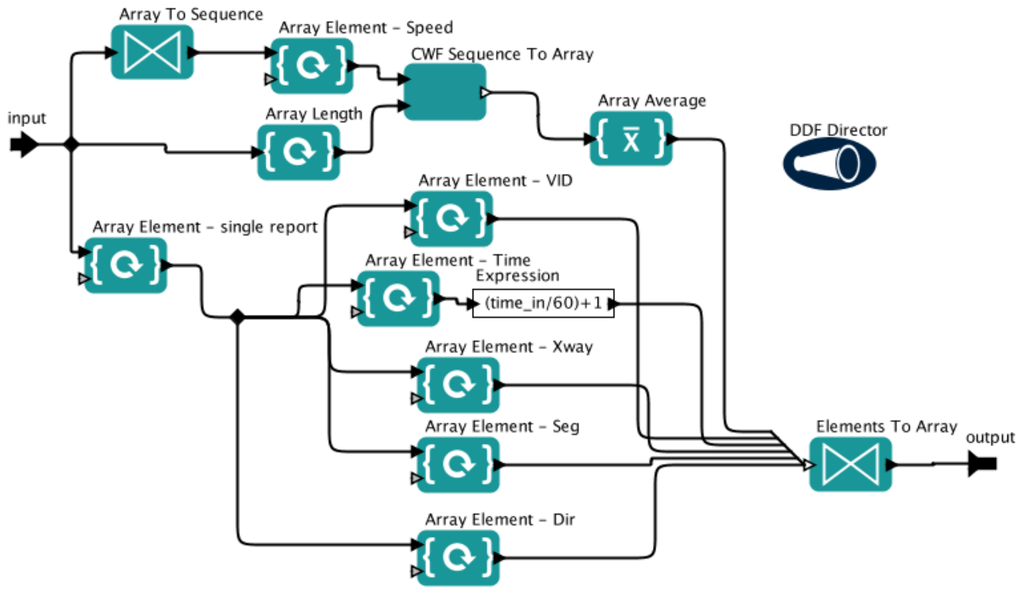


Figure 28: The Linear Road Benchmark car average speed sub-workflow (*Avgsv*)

The actor that calculates the number of cars per segment (*cars*), per minute, has the following window semantic definition: {Size: 1 minute, Step: 1 minute, Group-by: Expressway, Direction, Segment number}.

5.5.1.3 Toll Calculation and Notification The toll calculation is initiated for each car whenever it switches from one segment to the next. To achieve this it has the following window semantics: {Size: 2 tokens, Step: 1 token, Group-by: Car ID}. Each time it is fired it processes a window containing the last two position reports of a car. If those reports have a different segment id then a new toll has to be calculated for that car. The calculation is done by querying the relational database table which keeps the segment statistics.

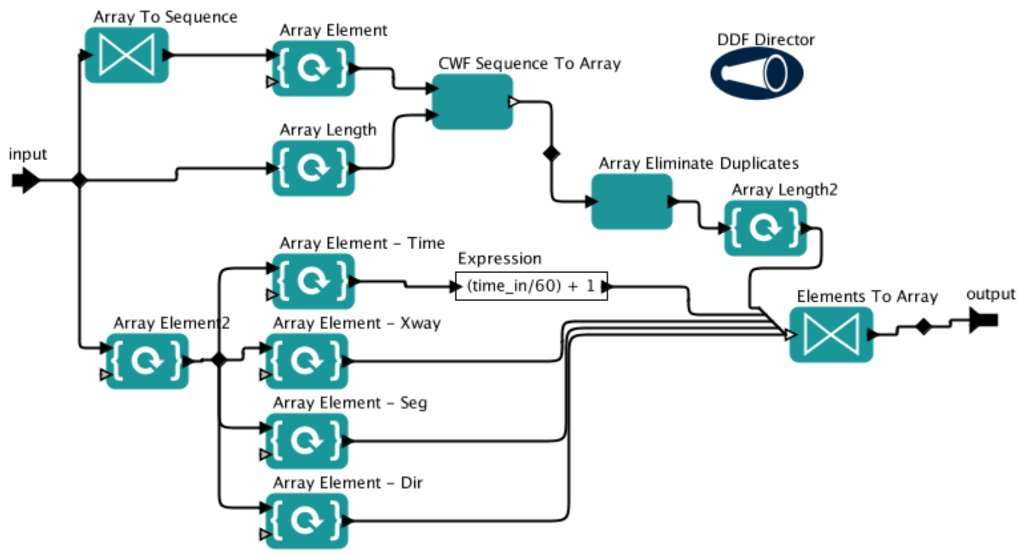


Figure 29: The Linear Road Benchmark car count sub-workflow (*cars*)

The SQL query used to calculate the toll, for a specific car is the following:

```
SELECT
   CASE
      WHEN LAV < 40 AND numOfCars > 50 AND (
         SELECT COUNT(*)
         FROM accidentInSegment AS ais
         WHERE ais.xway = xway AND ais.direction=dir
            AND ((dir=1 AND seg <= ais.segment+4 AND seg >= ais.segment) OR
               (dir=0 AND seg >= ais.segment-4 AND seg <= ais.segment))
            AND ais.timestamp>=330-60
      ) = 0
      THEN 2*POWER((numOfCars - 50),2)
      ELSE
         0
   END as "Toll"
FROM `segmentStatistics`
WHERE xway=$xway
   AND seg=$segment AND dir=$direction
```

5.5.2 Experimental Setup

In our experiments, we used the workload generator provided on the Linear Road website[1] to generate car position reports for an L-rating of 0.5 expressways (Figure 30). All the experiments were ran three times each (results show the average of the three runs) on the same machine configuration, always one at a time with the system being exclusively used for our experiments. The system used was a dual Pentium Intel Xeon E5345 at 2.33GHz with a total of 8 cores of 4MB cache each and 16GB of main memory. Since CONFLuEnCE is implemented in Java, the virtual machine was allocated 8GB of heap space.

The schedulers used in our evaluation are the ones implemented within the STAFiLOS framework and described in the previous section, namely, the Round-Robin (RR), Quantum Based Source (QBS) and the Rate Based (RB) schedulers. As a baseline for our comparison we use the *Thread Based (PNCWF)* scheduler which is implemented in the PNCWF Director described earlier in Section 4.3.

The different parameters we used for configuring the experiments are listed in Table 2. The source scheduling interval listed for the QBS scheduler means that for every five internal actor firings one source actor firing is scheduled. This ensures that the input data is smoothly inserted into the workflow. The basic quantum values listed for the QBS and RR schedulers correspond to the q value and slice values respectively as described in Sections 5.4.2 and 5.4.1. The priorities correspond to individual priorities given to the actors, that are taken into account when the QBS scheduler is running. The highest priority of 5 is given to the actors that handle the immediate output of the workflow. Regarding the tolls those are the *TollCalculation* and *TollNotification*, and regarding the accident notifications those are the *AccidentNotification* and *AccidentNotificationOut*. A priority of 10 was given to the actors relevant to statistics maintenance and accident detection.

5.5.3 Experimental Results

Experiment 1: Sensitivity Analysis of RR. Figure 31 shows how Round Robin behaves when setting different quantum values. Generally the scheduler behaves almost the same for the various

[1]http://www.cs.brandeis.edu/ linearroad/

Workload L-rating	0.5 highways
Experiment duration	600 sec
QBS Source scheduling interval	5 internal actor iterations
Basic Quantum (QBS) (μs)	500, 1000, 5000, 10000, 20000
Basic Quantum (RR) (μs)	5000, 10000, 20000, 40000
Priorities used (QBS)	5, 10

Table 2: Experimental setup

time slots with the best being 20,000μs which keeps a generally lower response time throughout the experiment until eventually thrashes with the 40,000μs case.

Experiment 2: Sensitivity Analysis of QBS. Figure 32 shows how the Quantum Priority Based Source scheduler behaves with different basic quantum values set. As you may recall, the basic quantum is the value of b in Equation 5.1. From the results we see that a basic quantum of 500μs performs the best throughout the experiment compared to the other values. This is due to the fact that having high quantum values given to the actors results in having just a priority based FIFO queue, where each priority class is a FIFO queue with each actor exiting the queue only when it is done processing all of its events. So a small enough value in this case is adequate. What is interesting here is that a basic quantum of 5000μs performs worse than one with 10000μs. We attribute this to the fact that in the case of 5000μs the re-quantification of all actors happens more often, resulting in low priority actors accumulating quantum, such that when it is their turn to run, and having also accumulated many events, they will end up starving the higher priority actors (which are the one we are measuring the average response time for, i.e. the output actor).

Experiment 3: STAFiLOS-based schedulers Vs. OS thread-based scheduler. Figure 33 shows the QBS and RR with the best performing parameters of 500μs and 40,000μs, respectively, as determined in the previous two experiments along with the rest of the schedulers, RB and PNCWF. The figure shows that QBS and RR exhibit the best response times, while the thread-based PNCWF scheduler has much lower capacity in terms of input rates, since it thrashed at second 320 when the input rate is about 120 updates/second, as opposed to the rest of the schedulers which crash at about second 440 where the input rate is 160 updates/second. RB exhibits worst average response times because of the fact that it does not distinguish the source actors as high priority and neither independently schedules them in regular intervals, like the other schedulers. Thus tokens suffer from waiting for a longer period of time to enter the workflow.

These experiments clearly show that the schedulers implemented within the STAFiLOS framework have a higher rate tolerance and generally lower response times than Kepler's own Thread-Based director which relies on the underlying OS.

We generally based our CWf implementation of the Linear Road Benchmark on off-the-shelf actors that come with Kepler, which as can be seen from Figures 24-29 adds a great deal of complexity. Furthermore, the off-the-shelf actors lack any performance optimizations found in the CQ operators. Adding asynchronous I/O calls as well as implementing schedulers which are able to combine priorities with flow information would greatly improve performance. Moreover, providing a set of stream optimized atomic as well as composite actors, which can accumulate and compensate tokens which are added and expired from a sliding window, would help in avoiding redundant multiple aggregate computations and would greatly improve the performance of window-based actors.

Our experiments have also revealed that STAFiLOS offers the scheduling flexibility required by monitoring applications within a Continuous Workflow Management System without compromising performance, as we have seen the STAFiLOS-based schedulers performed better than the Thread-Based one. Moreover, a Continuous Workflow Management System by offering more functionality and flexibility, compared to a DSMS might not exhibit the same scalability, as this was shown by applying the Linear Road Benchmark. However, scalability can be achieved by strategically integrating multiple DSMSs (as in Figure 1), that can be viewed as specialized source actors, to build more complex monitoring solutions, while being able to satisfy any application SLAs. The integrated DSMSs can potentially be tuned to also support load shedding under overloading situations [53, 52].

5.6 SUMMARY

In this chapter, we proposed our stream-flow based scheduling framework for continuous workflows called STAFiLOS. First we examined the existing scheduling policies implemented through Kepler's and PtolemyII's directors. We then presented the components of the STAFiLOS framework, and how we have implemented three schedulers using this framework. Finally, we evaluated the performance of STAFiLOS by comparing the performance of the three schedulers against the native thread-based scheduler, by running them on our CWf implementation of the Linear Road Benchmark. The results have shown that STAFiLOS provides the flexibility required in terms of easily applying different scheduling policies on CWfs, while keeping performance above the native solutions.

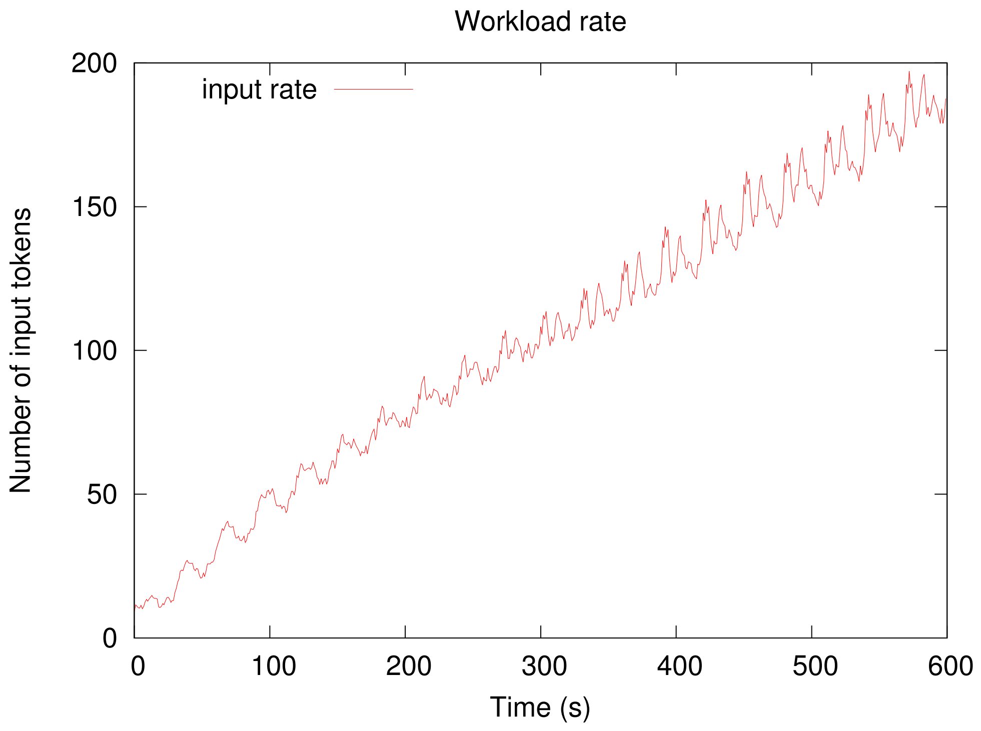


Figure 30: Linear Road workload of 0.5 highways

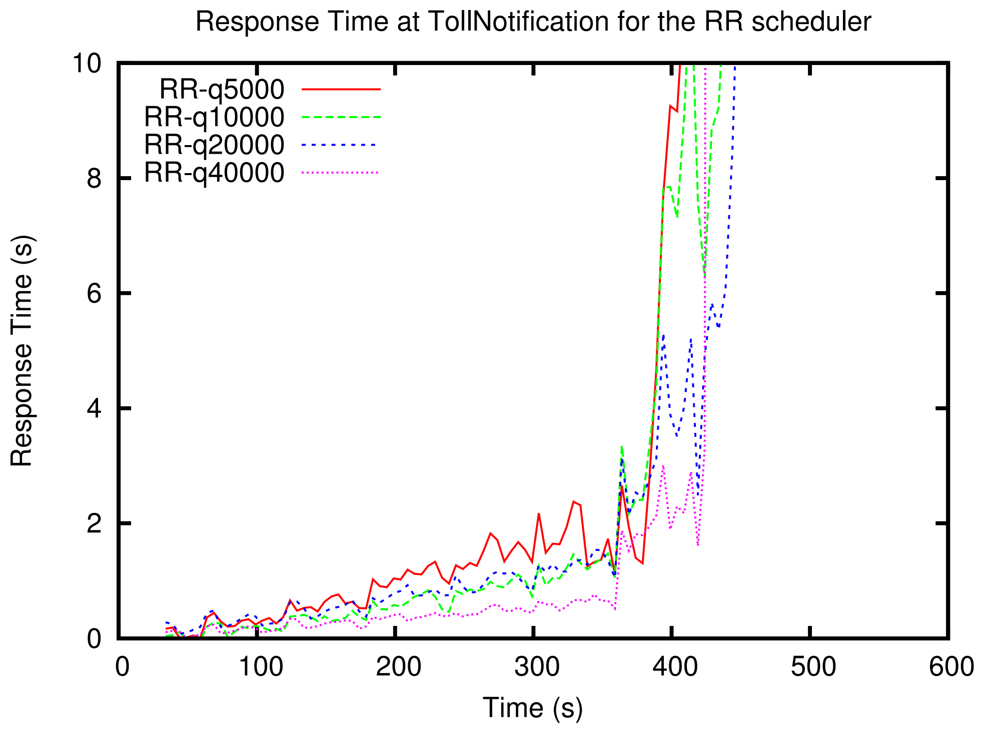


Figure 31: Response Times of the RR scheduler using varying basic quantum values

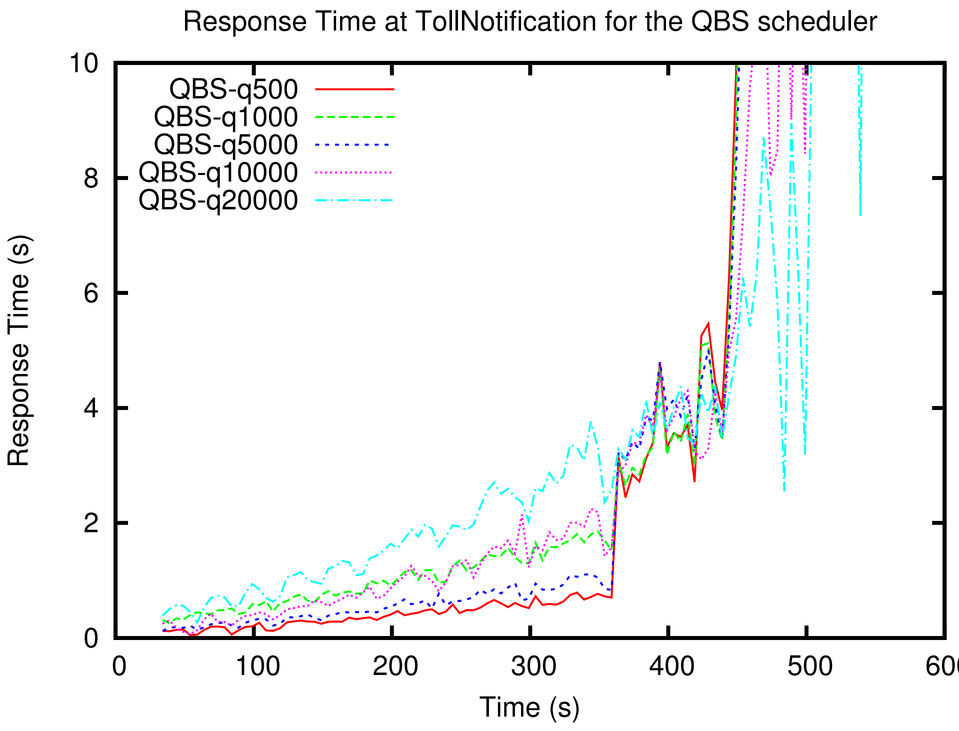


Figure 32: Response Times of the QBS scheduler using varying basic quantum values

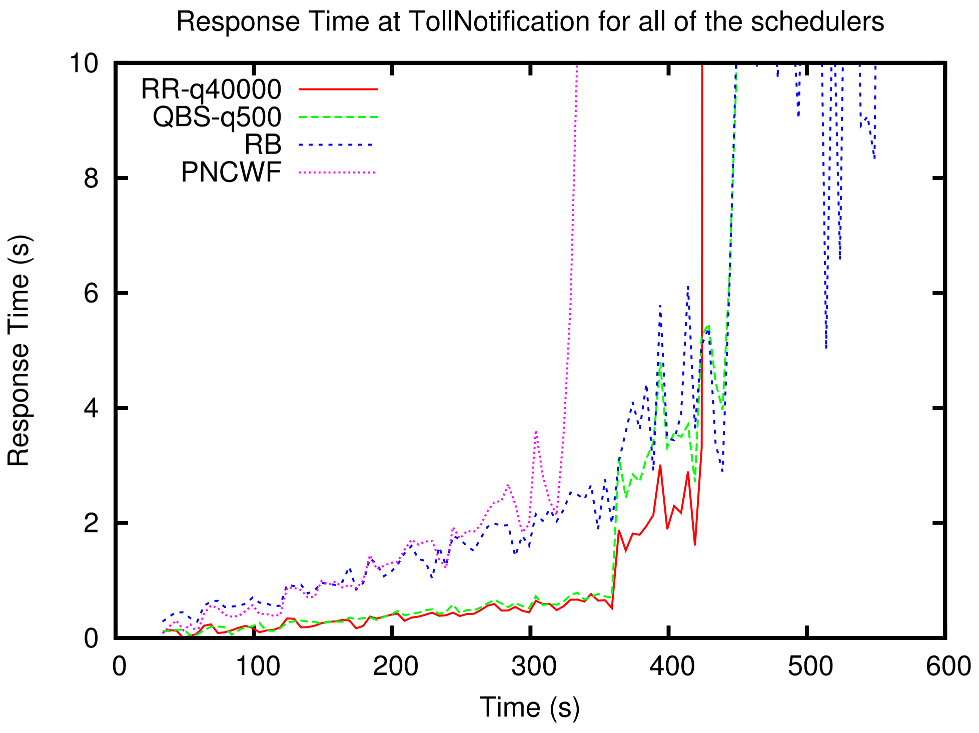


Figure 33: Response Times of all the main schedulers

6.0 CONCLUSIONS AND FUTURE WORK

6.1 SUMMARY OF CONTRIBUTIONS

This dissertation has contributed in setting the building blocks of a new computation model based on the fusion of data stream processing and workflow execution models. We presented the fundamental primitives of a continuous workflow model [41], which laid the basis for us to build CONFLuEnCE, our CONtinuous workFLow ExeCution Engine [43]. Towards this we used Kepler, a complete workflow management system and, in particular, we implemented our continuous workflow model as a new module in Kepler.

As part of the CONFLuEnCE module we first implemented the abstract Continuous Workflow Director with its own communications model. The communications model is realized by a generic Windowed Receiver, capable of setting temporal and logical bounds on unbounded data streams. The receiver can be extended (i.e., sub-classed) accordingly to accommodate the resource model of any CWf director (e.g., multi-threaded, single threaded, multi-core, etc.) We have demonstrated how the resource model can be extended, within a multi-threaded director (PN Director) and a single-threaded director able to apply different scheduling policies (Scheduled CWF Director). The Scheduled CWF Director is part of our STAFiLOS scheduling framework [44], that provides the means to assimilate a wide variety of scheduling policies, in a plug-and-play manner. Three such policies have been described, implemented and evaluated, namely: Quantum-Based, Rate-Based and Round Robin.

Moreover, we have evaluated the applicability of our CWf model and the six new CWf patterns derived from the CWf model by integrating continuous workflows in the heart of both scientific and business applications. Firstly, in the context of the AstroShelf astronomical data exploration platform [45], for processing live annotations by monitoring live data interactions between users and system components. Secondly, as the driving module of a supply chain management business application [42], integrating clients, warehouse operators, and operations managers.

Finally, we have experimentally evaluated the performance of the implemented system over our continuous workflow realization of the Linear Road Benchmark. The results of these experiments, in conjunction with the implementation of the Linear Road Benchmark show that our system balances performance and functionality. It enables extra functionality, ease of use, re-usability, wide deployment and large user-base support, while providing the means to support QoS and QoD requirements by enabling the employment of different scheduling policies.

6.2 IMPACT OF THIS DISSERTATION

This dissertation provides a formal definition of our Continuous Workflow model. This model, is a superset of the traditional workflow model, and describes all the components and processes necessary for building Continuous Workflow Enactment Systems, which can still enact traditional workflows (either entirely or as sub-workflows). The Continuous Workflow (CWf) Patterns that we defined in Chapter 3 extend the workflow patterns framework to include CWf requirements. A system that implements our model has the ability to enact continuous workflows, just like our CONFLuEnCE implementation on top of Kepler. The existence of this concept as a model and as a system, now enables the fulfillment of a vision of unifying traditional static data processing with the dynamic streaming data processing paradigm. Hence, transforming business intelligence and scientific exploration from the traditional back-office, ad-hoc, request/response platform, to an enabler for delivering data-intensive, real-time analytics that transform the scientific process and business operations in the modern world of Big Data, and its Volume/Velocity/Variety characteristics.

Using Kepler as the initial platform to build CONFLuEnCE, enables researchers to readily create, share, re-use, modify and integrate among many different continuous workflows, through the use of social workflow sharing platforms such as myExperiment [2]. Moreover, the open nature of Kepler and its modular design will enable further development of the implementation of CONFLuEnCE towards a more complete, robust, and efficient system. The AstroShelf Live

Annotations platform we have built will be used by astrophysicists to enable collaboration and further exploration of the Universe.

The design and implementation of a Continuous Workflow Scheduling Framework, called STAFiLOS, enables for the first time the use of performance-based schedulers within the context of a workflow management and enactment system, and more specifically a Continuous Workflow Enactment System. Now, workflow designers and data processing performance specialists are able to easily design and implement many different types of plug-and-play schedulers which optimize for specific metrics, depending on the application's requirements. Within the same workflow model implementation, different schedulers may be used and evaluated on various workflows. Finally, our continuous workflow implementation of the Linear Road Benchmark may set the standard for evaluating the various schedulers under different conditions.

6.3 FUTURE WORK

Our future work aims to extend the development of CONFLuEnCE and STAFiLOS further, by providing more features, and by optimizing certain aspects of the platform to increase its performance, scalability, extensibility and usability.

One feature required by any workflow management system is the ability to run and manage multiple workflows at the same time. In the case of Continuous Workflows this entails an extra challenge. Since the workflows are all running continuously at the same time and most probably all are processing data for time-critical applications, a grander scheduling and resource management scheme is required. This scheme should be able to prioritize accordingly with respect to the importance of each workflow running and also depending on the optimization metric. Moreover, the multi-workflow enactment system should be able to not only scale-up but also scale out, within a cloud-based environment in order to facilitate the processing demands of modern day and future applications. An initial sketch of the Multi-workflow execution scheme we propose is described in the Appendix.

When a continuous workflow is communicating with either external data sources (e.g., running ad-hoc queries) or human participants through a specific actor, the rest of the workflow should be able to continue running. In STAFiLOS where the execution control is handled by a single thread, that thread blocks whenever there is an I/O such as the ones described. This fact necessitates for development of asynchronous I/O actors that are specifically supported by the STAFiLOS director in order to handle these types of communication.

A design problem with the windowed receiver implementation, is the fact that each window produced is treated as a new one. The fact is though that consecutive sliding windows intersect with each other in terms of the tokens included in them. The window operator should only update the actor that is processing each window with the new tokens (accumulate) and the expired tokens (compensate). We originally designed our Windowed Receiver to be backwards compatible with the legacy, off-the-shelf actors which were available in Kepler. Such a restriction could be lifted when the CWf support community grows and the actor library is extended with stream-based operators. In particular we propose the development of an actor framework which would support incremental window semantics, i.e., one that accumulates new window tokens and compensates expired window tokens, as compared to the previous window state. Moreover, we propose adapting techniques on optimized processing of multiple aggregate continuous queries such as [23] from the DSMS domain to the CWfs domain.

Finally, we propose implementing schedulers that support multi-core systems. In particular, in order to achieve this, a new multi-core aware STAFiLOS director needs to be designed. The director should be able to manage multiple execution threads, one for each core and be able to balance the processing load while keeping the QoS and QoD targets in order.

APPENDIX

MULTI-WORKFLOW EXECUTION

Multi-workflow execution can be made possible by instantiating multiple Manager instances. Within PtolemyII/Kepler this module serves the purpose of managing the execution of a single workflow. In order to do that we need to implement a new command line interface which has to be based on/extend the `ptolemy.moml.MOMLCommandLineApplication` class.

A manager can execute a model inside the calling thread or by spawning a new thread using one of the following methods:

- `execute()/run()`: starts execution in the current thread. It will return only when the execution is finished or until the finish() method is called by another thread, or by an actor inside the workflow. `run()` has the same behavior as `execute()`; only the Exceptions are handled by a listener interface.
- `startRun()`: start executing the model in a separate thread. This is stored in the Thread object internally, in the Manager object.

A high-level scheduler has to be designed to manage multiple instances of the Manager class. One for each workflow required to run. Since the high-level scheduler is needed to manage the CPU resource then the following Manager methods will be used to handle switching between workflows:

- `initialize()`
- `pause()`
- `resume()`

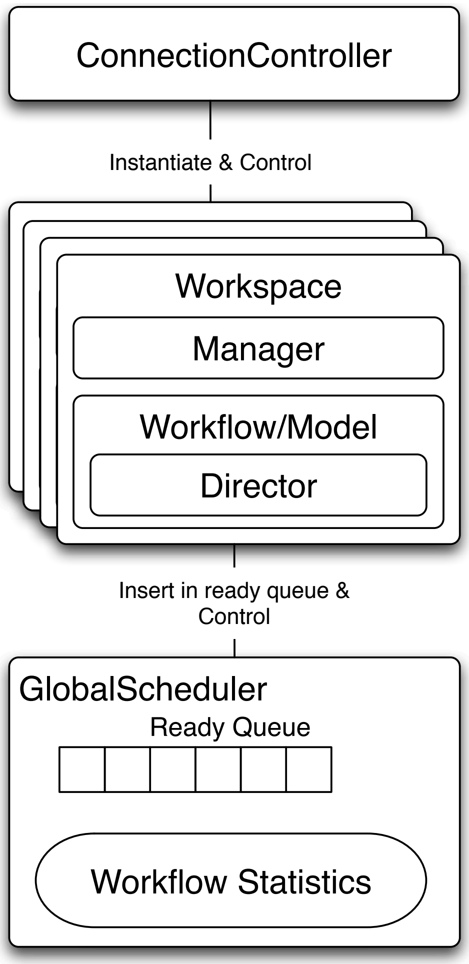


Figure 34: Multi-workflow execution framework

- `stop()`: rarely needed, since continuous workflows are never-ending processes, although existing continuous workflows could be terminated.

Figure 34 provides a general architecture of how the two-level scheduling would work. The ConnectionController is a new module we propose for controlling the execution of multiple workflows externally. When Kepler/Confluence is started in multi-workflow mode from the command line then the ConnectionController is instantiated and is listening for commands to manage running workflows as well as add and remove them from the running list. Here is a list of command line arguments that will be available for `confluence`:

- `confluence register workflow.xml`: Add the workflow to the list of workflows that are continuously running. If the Executive director is not of type ICWFDirector then an exception is thrown.
- `confluence remove workflow.xml`: Remove the workflow from the list of continuous workflows.
- `confluence pause workflow.xml`: Pause the execution of a workflow.
- `confluence resume workflow.xml`: Resume execution of a workflow.
- `confluence list`: Lists the running workflows and their state (using the Manager.state variable.
- `confluence run workflow.xml`: Runs the workflow as an ad-hoc workflow. This will be run concurrently with other workflows in the system, continuous or not.

BIBLIOGRAPHY

[1] R. Adaikkalavan and S. Chakravarthy. Seamless event and data stream processing: Reconciling windows and consumption modes. In J. Yu, M. Kim, and R. Unland, editors, *Database Systems for Advanced Applications*, volume 6587 of *Lecture Notes in Computer Science*, pages 341–356. Springer Berlin Heidelberg, 2011.

[2] A. Ailamaki, V. Kantere, and D. Dash. Managing scientific data. *Communications of the ACM*, 53(6):68–78, 2010.

[3] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[4] AstroShelf. http://db.cs.pitt.edu/group/projects/astroshelf, September 2010.

[5] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3), 2001.

[6] M. Beck. *Linux kernel internals*. Addison-Wesley, 1998.

[7] A. Berfield, P. K. Chrysanthis, I. Tsamardinos, M. E. Pollack, and S. Banerjee. A scheme for integrating e-services in establishing virtual enterprises. In *RIDE*, pages 134–142, 2002.

[8] I. Botan, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending xquery with window functions. In *VLDB*, pages 75–86, 2007.

[9] BPML. Process modeling language (bpml) - http://www.bpmi.org, 2002.

[10] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *VLDB*, 2002.

[11] M. Castellanos, U. Dayal, and M. Hsu. Live business intelligence for the real-time enterprise. In K. Sachs, I. Petrov, and P. Guerrero, editors, *From Active Data Management to Event-Based Systems and More*, volume 6462 of *Lecture Notes in Computer Science*, pages 325–336. Springer Berlin Heidelberg, 2010.

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03*, 2003.

[13] P. K. Chrysanthis. Aqsios - next generation data stream management system. *CONET Newsletter*, June 2010.

[14] P. K. Chrysanthis, D. Stemple, and K. Ramamritham. A logically distributed approach for structuring office systems. *SIGOIS Bull.*, 11:11–20, March 1990.

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

[16] U. Dayal and M.-C. Shan. Issues in operation flow management for long-running acivities. *IEEE Data Eng. Bull.*, 16(2):41–44, 1993.

[17] J. Delaney and R. Barga. A 2020 vision for ocean science. In T. Hey, S. Tansley, and K. M. Tolle, editors, *The Fourth Paradigm: Data-Intensive Scientific Discovery*, pages 27–38. Microsoft Research, 2009.

[18] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[19] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[20] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[21] J. Ferreira, K. Braghetto, O. Takai, and C. Pu. Transactional recovery support for robust exception handling in business process services. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 303 –310, june 2012.

[22] M. Gillmann, R. Mindermann, and G. Weikum. Benchmarking and configuration of workflow management systems. In *In Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*, pages 186–197, 2000.

[23] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Three-level processing of multiple aggregate continuous queries. In *ICDE*, pages 929–940, 2012.

[24] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[25] P. Hart and D. Estrin. Inter-organization computer networks: indications of shifts in interdependence. *SIGOIS Bull.*, 11:79–88, March 1990.

[26] R. L. Haskin. Document processing in an automated office. *IEEE Database Eng. Bull.*, 6(3):31–35, 1983.

[27] T. Hey, S. Tansley, and K. M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[28] IBM. System S - stream computing at ibm research. http://www-01.ibm.com/software/sw-library/en_US/detail/R924335M43279V91.html, 2008.

[29] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. Qos aggregation for web service composition using workflow patterns. In *EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International*, pages 149–159, Washington, DC, USA, 2004. IEEE Computer Society.

[30] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD Conference*, pages 431–442, 2006.

[31] R. K. and P. K. Chrysanthis. Advances in concurrency control and transaction processing. In *IEEE Computer Society Press*, 1997.

[32] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006.

[33] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[34] R. Liu, A. Kumar, and W. van der Aalst. A formal modeling approach for supply chain event management. *Decision Support Systems*, 43(3):761–778, 2007.

[35] F. H. Lochovsky. A knowledge-based approach to supporting office work. *IEEE Database Eng. Bull.*, 6(3):43–51, 1983.

[36] B. Ludäscher et al. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[37] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers. Scientific workflows: Business as usual? In *Business Process Management*, 2009.

[38] T. M. McPhillips and S. Bowers. An approach for pipelining nested collections in scientific workflows. *SIGMOD Rec.*, 34:12–17, September 2005.

[39] Microsoft. Microsoft streaminsight, http://www.microsoft.com/sqlserver/2008/en/us/r2-complex-event.aspx, 2008.

[40] 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 ina data stream management system. Technical Report 2002-41, Stanford InfoLab, 2002.

[41] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis. Towards continuous workflow enactment systems. In *CollaborateCom*, pages 162–178, 2008.

[42] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis. Confluence: Continuous workflow execution engine. In *Proceedings of SIGMOD*, pages 1311–1314, 2011.

[43] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis. Confluence: Implementation and application design. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2011.

[44] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis. A continuous workflow scheduling framework. In *SWEET*, pages 1–12, 2013.

[45] P. Neophytou, R. Gheorghiu, R. Hachey, T. Luciani, D. Bao, A. Labrinidis, E. G. Marai, and P. K. Chrysanthis. Astroshelf: understanding the universe through scalable navigation of a galaxy of annotations. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 713–716, New York, NY, USA, 2012. ACM.

[46] OASIS. Web services business process execution language - http://docs.oasis-open.org/wsbpel/2.0/os/wsbpel-v2.0-os.html.

[47] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[48] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous pig/hadoop workflows. In *Proceedings of SIGMOD*, pages 1081–1090, 2011.

[49] K. Patroumpas and T. K. Sellis. Window specification over data streams. In T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, and J. Wijsen, editors, *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 445–464. Springer, 2006.

[50] C. Pautasso and G. Alonso. Parallel computing patterns for grid workflows. In *WORKS*, June '06.

[51] O. Perrin and C. Godart. A model to support collaborative work in virtual enterprises. *Data & Knowledge Engineering*, 50(1):63 – 86, 2004.

[52] T. Pham, P. K. Chrysanthis, and A. Labrinidis. Self-managing load shedding for data stream management systems. In *In Proc. of 8th International Workshop on Self-Managing Database Systems (SMDB 2013)*, 2013.

[53] T. N. Pham, L. A. Moakar, P. K. Chrysanthis, and A. Labrinidis. Dilos: A dynamic integrated load manager and scheduler for continuous queries. In *The Sixth International Workshop on Self-Managing Database Systems*, pages pp. 10–15, April 2011.

[54] D. Riehle and H. Züllighoven. Understanding and using patterns in software development. *TAPOS*, 2(1):3–13, 1996.

[55] W. A. Ruh, F. X. Maginnis, and W. J. Brown. Enterprise application integration: A wiley tech brief, 2001.

[56] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In L. M. L. Delcambre, C. Kop, H. C. Mayr, J. Mylopoulos, and O. Pastor, editors, *ER*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2005.

[57] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs. Algorithms and metrics for processing multiple heterogeneous continuous queries. *ACM Trans. Database Syst.*, 33(1), 2008.

[58] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34:31–36, September 2005.

[59] I. StreamBase. Streambase: Real-time, low latency data processing with a stream processing engine. http://www.streambase.com, 2006.

[60] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *TKDE*, 2003.

[61] UN/CEFACT and OASIS. ebXML business process specification schema - www.ebxml.org/specs/ebbpss.pdf.

[62] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 501–510. Morgan Kaufmann, 2001.

[63] W. van der Aalst and A. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

[64] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[65] W3C. Web services glossary - http://www.w3.org/tr/ws-gloss/.

[66] W3C. Service choreography interface (wsci) 1.0. www.w3.org/tr/wsci, 2002.

[67] WfMC. Xml process definition language - http://www.wfmc.org/.

[68] WfMC. Workflow management coalition: Terminology & glossary (wfmc- tc-1011), 1999.

[69] P. Wohed, W. van der Aalst, M. Dumas, and A. ter Hofstede. Analysis of web services composition languages: The case of BPEL4WS. In *ER*, pages 200–215, 2003.

[70] J. Yu, R. Buyya, and C.-K. Tham. Cost-based scheduling of scientific workflow application on utility grids. In *e-Science*, pages 140–147, 2005.

[71] J. A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3):276 –292, 1987.

[72] Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *SIGMOD Rec.*, 34:37–43, September 2005.