University of Piraeus

Master Thesis

Real-Time Workflow Management Service based on an Event-driven Computational Cloud Storage

Author:
Athanasios Tsitsipas
ME11072

Supervisor:
Assoc. Prof. Marinos
Themistocleous

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Digital Systems

June 2014

Abstract

Applications built in a strongly decoupled, event-based interaction style have many commendable characteristics, including ease of dynamic configuration, accommodation of platform heterogeneity, and ease of distribution over a network. It is not always easy, however, to humanly grasp the dynamic behavior of such applications, since many threads are active and events are asynchronously transmitted. With the growing complexity of industrial use case requirements, independent and isolated triggers cannot fulfil the demands anymore. Fortunately, an independent trigger can be triggered by the result produced by other triggered actions and enables the modelling of complex use cases. The chains or graphs that consist of triggers, are called workflows. Therefore, workflow construction and manipulation become a must for implementing the complex use cases. This study presents an approach that aids in the discovery and visualization of the behaviors of event-based applications. It applies to real, implemented systems, without requiring the presence of component source code, and supports partial or incomplete, heuristic behavior specifications. A prototype implementation of this study approach, was applied to VISION Cloud, a computational storage system that executes small programs, called storlets, as independent computation units in the storage. Similar to the trigger mechanism in database systems, storlets are triggered by specific events and then execute computations. Consequently, one storlet can also be triggered by the result produced by other storlets, and that is called connections between storlets. Due to the growing complexity of use case requirements, an urgent demand is to have real-time storlet workflow management supported in VISION system. Furthermore, because of the existence of connections between storlets in VISION, problems such as, non-termination triggering and unexpected overwriting, appear as side-effects.

The thesis is part of the European Project VISION Cloud and it was written in collaboration with Swedish Institute of Compute Science (SICS), a partner of the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Event-Driven Computing Systems

Event-driven architectural styles, are styles in which software components communicate with each other via explicit events or messages, as their sole basis for communication. Because there is no basic assumption of a global clock ordering of execution among components, event-driven systems are fundamentally asynchronous; component may send an event at any time, and may receive an event at any time. Such architectural style is VISION Cloud, a storage system that supports event-driven computing. Computations in VISION can be chained as workflows at runtime. At the beggining of this chapter, the event-driven computing systems are introduced among with the general objectives and scope that will be explored, and then an overall discussion of the related work takes place

1.1.1 Introduction

Event-driven computing [1] systems are typically based on Event-Condition-Action rules (ECA) [2]. ECA rules consist of events, conditions and actions. The event specifies the signal that triggers the invocation, the condition is a logical test to determine whether the action is to be carried out, and finally the action defines the operations on data. When an event is received, the system evaluates a set of rules interested in the events, and if the conditions are satisfied the actions are performed. In ADBMS [3], a detected system behavior leads to the triggering of an ECA (event-condition-action) rule. ECA rules are used in order to support reactive behavior in database management system functions, as well as, external applications. Although distributed database systems are becoming trite, research in active databases has to be focused on centralized systems. In distributed debugging systems, a detected system behavior is compared with the expected system behavior. Differences illustrate erroneous behavior.

1.1.2 ECA-rule based Workflow Visualization Model

The result of actions can match other rules and thus trigger a graph of activities. As an important feature, ECA rule-based systems support workflow construction[4]. However, it is not convenient to augment, modify and test workflows without visualization. One challenge of workflow visualization is to distinguish one triggering from another. Moreover, when the workflow is ECA rule-based, the visualization model should support the distinction between different events. Besides that, visualize the behaviour of event-based systems augment the fact that can understand and debug such systems. The two common visualization models of ECA rule-based workflows are Coloured Petri Net (CPN) model[5] and Event-Driven Process chain (EPC) model[6].

1.2 Cloud Storage

Cloud Storage is a model for an on-line networked storage delivery and management, where storage is provided in a 'storage-as-a-service' pay-per-use manner. The storage infrastructure is hosted remotely by the storage infrastructure provider on multiple data centers. The storage cloud paradigm addresses the needs to host increasingly large volumes of data, for workloads which require relatively lower performance than the typical enterprise application workloads.

The technological foundation which is at the heart of the cloud storage infrastructure is based on two building blocks: storage virtualization and high scalability. The physical resources of the cloud are spread over many servers and disks located in multiple data centers across a few geographical locations. The storage service operates by means of virtualizing these physical resources as storage pools in a manner that is transparent to the users. These pools are available anywhere over the web via simple http protocols; the data is replicated transparently across the data centers.

Cloud Storage employs the newly emerging cloud-based data management structures in order to store and manage the metadata associated with the data. In the following subsections are reviewed the state of art in distributed file systems, distributed data management, data mobility and content centric storage.

1.2.1 Distributed File Systems

File systems evolved over time. Starting with local file systems over time additional file systems appeared focusing on specialized requirements such as data sharing, remote file access, distributed file access, parallel files access etc. A distributed file system is a network file system whose clients, servers, and storage devices are dispersed

among the machines of a distributed system or intranet. Using Ethernet as a networking protocol between nodes, a DFS allows a single file system to span across all nodes in the DFS cluster, effectively creating a unified Global Namespace for all files, unifying files on different computers into a single namespace. Also, files are distributed across multiple servers appearing to users as being stored at a unique place on the network and users no longer need to know and specify the actual physical location of files in order to access them.

File systems evolved over time. Starting with local file systems over time additional file systems appeared focusing on specialized requirements such as data sharing, remote file access, distributed file access, parallel files access etc. A distributed file system is a network file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system or intranet. Using Ethernet as a networking protocol between nodes, a DFS allows a single file system to span across all nodes in the DFS cluster, effectively creating a unified Global Namespace for all files, unifying files on different computers into a single namespace. Also, files are distributed across multiple servers appearing to users as being stored at a unique place on the network and users no longer need to know and specify the actual physical location of files in order to access them.

Distributed file systems may include facilities for transparent replication and fault tolerance. When a limited number of nodes in a file system go offline, the system continues to work without any data loss or unavailability.

Distributed file systems (DFS) maintain control of file and data layout across the nodes and employ metadata and locking mechanisms that are fully distributed and cohesively maintained across the cluster, enabling the creation of a very large global pool of storage. A DFS can seamlessly scale to petabytes of storage. A fully distributed file system can handle metadata operations, file locking and cache management tasks by distributing operations across all the nodes in the cluster. Notable approaches and implementations are: General Parallel File System (GPFS)[7], Lustre[8], Google File System (GFS or GoogleFS)[9], Hadoop Distributed File System (HDFS)[10] and Parallel Virtual File System (PVFS)[11].

1.2.2 Distributed Object Storage Systems

Distributed storage systems are becoming the method of data storage for the new generation of applications - web applications by companies like Google, Amazon and Yahoo!.There are several reasons for distributed processing. On one hand, programs should be scalable and should take advantage of multiple systems as well as multi-core CPU architectures. On the other end, web servers have to be globally distributed for low latency and failover. Object systems differ from file systems in the data model

and access semantics they provide. Distributed object stores support wide distribution of data and access to data with high availability, even in presence of frequent node failures. They support data storage cloud services in which data is read and written by a wide variety of client applications running anywhere and typically using HTTP interfaces to access the storage service and their data. Providing storage as a cloud service introduces a new ICT delivery model for storage. Businesses and individual can consume flexible, variable and unlimited amount of storage without acquiring hardware, managing and maintaining it, paying per use for storage consumed "on demand". Noteworthy existing commercial distributed object storage systems are Amazon Simple Storage Service (S3)[12], Google Storage[13], Windows Azure Storage[14] and EMC Atmos[15].

1.2.3 Distributed Data Management Systems

Distributed database systems are becoming the method of massive data storage (in the order of petabytes) for the new generation of web applications storing large amount of metadata information. Distributed database systems use an architecture that distributes storage and processing across multiple servers to address performance, scalability requirements, caching, flexible graph handling and easy query manipulation. All these requirements cannot be met by traditional relational databases mainly because of their non-stateless, namely, requests inside the same transaction have to be run by a single node of the database. Some companies, with the rise of Internet, made the choice to skip relational databases to simpler but highly scalable "NoSQL" databases[16–18]. This kind of databases allow simple request like single row transaction, but do not allow complex SQL request like join operation. Notable production implementations include Google's BigTable[19], Amazon's Dynamo[20] and Cassandra[21, 22].

1.3 Introducing: VISION Cloud Project

1.3.1 Introduction

VISION Cloud[23] was built as a cloud storage system which was joined together with an event-driven computational infrastructure. A critical ingredient in delivering converged data-intensive services, is the tightly-coupled storage and computational infrastructure, providing comprehensive data interoperability, mobility, QoS and means of massive data processing[24].

1.3.2 Storage Aspect

VISION Cloud is raising the abstraction level of storage beyond blocks or files to data associated with a sophisticated metadata schema, enabling the user to quickly index and find them. The data are being resided in a unit of placement, called "container", and a user can store the same data objects to multiple containers. However, differs from the global access in a NoSQL data store, where he cannot have unauthorized access in a container. It is noteworthy to say, that containers not only separate the objects based on the owner (user or groups) but also, form the boundary of trigger computation because triggers can only be invoked by the events occurring in the same container.

1.3.3 Event-driven Computing in VISION

The event-driven computational infrastructure of VISION is also based on ECA rules. However, the rules are not held by a centric management service like DBMS, but carried by light-weight computation units, called storlets, which are placed in containers. At the beginning, a storlet that contains a set of ECA rules is created and injected into the container as a static object by the user. Until it is triggered by a specific event under a certain condition, it remains as an ordinary object. If it is triggered, it becomes an active process to execute the specified action on behalf of the user. When the execution finishes, it becomes passive again.

VISION Cloud is designed for data-intensive services. The usecases of data processing are getting inevitably more complex. Therefore, there is a vital requirement for VISION that is the feature of workflow creation and manipulation. In a storlet workflow, the interactions between rules are indirect. A rule is not triggered by the actions of other rules, but from the result events produced by these actions. Especially, the events produced by storlets are diverse because storlets can perform complex actions. For example, a media file transcoding storlet can produce the events, such as,

file format update, file size update, last modified time update and delete the original file.

Although both of VISION and DBMS are ECA-rule based architecture, the storlet mechanism is quite different from the normal triggers in DBMSs. In a DBMS, rules are usually managed by database engineers. In contrast, storlets in VISION are created from a template by the empowered users who may have no programming experience. In terms of maintenance, triggers in an ordinary database can hardly be changed unless new tables or views are created. Whereas, new rules in VISION are added with the creation of new storlets. Moreover, when a storlet is running within a container, it cannot be activated by the events occurred in other containers. It is different from the situation of DBMS, where database triggers are activated by events globally in the whole system. The rule format of a storlet is explained in section 2.3.3.

1.4 Motivation and Problem Statement

VISION Cloud is concerned to be an event-based system. But it is slightly different from these systems, because the basic components of the system (storlets) don't know about each other and they tend to behave on their own. Whenever a predefined trigger condition, which is located in storlet definition, becomes true, the storlet is executed. However, the output of a storlet might trigger another storlet and they are somehow chained together indirectly formulating a workflow. For example, a chain might have began with an insertion or creation of a text object. Then a storlet that does a text analysis to classify the topic is triggered. Next, another storlet is triggered that does a specialized analysis to find what kind of science the text is about (Physics, Chemistry, Computer Science etc.) and last another more specialized storlet classify the text if it is about Cloud Computing.

In VISION there is a simplicity in storlet creation by a user who is expected to be non-programmer. The user sets a few parameters when creating storlets and the storlet templates are assumed to be correct. Using the previous example, the second storlet only outputs Physics and Chemistry tag, whereas the third storlet expects a Computer Science tag. Hence, there is a broken chain, it is not the behavior the user was expecting to happen. Therefore, the user needs to easily monitor his storlets' execution and interact with a simple interface illustrating a well annotated graph of storlets' actions; like a debugging tool for his own information in the system.

The aim of this project is to develop a workflow management service helping the user to understand the actual observed workflows, to ensure that they are working correctly and help him to augment existing workflows with additional storlets. Also, focusing on unexpected behavior in storlet runtime such us, existing non-termination storlet triggering and unexpected activation of some storlets. Due to the distributed architecture of VISION, everything tends to work asynchronously, thus any message can be delayed in runtime making the tracing of interactions between storlets a more complicated task.

1.5 Problem Scope

All problems mentioned above are significant to VISION Cloud. Problems can be categorised into dynamic and static. The dynamic problems are basically related to storlet's runtime problems whereas, static problems are related to the design and creation of a single or set of new storlets for an existing environment completed with existing storlets. The possible new connections and problems such as cycles, could be predicted before the new storlets are injected.

This study is focused on the dynamic problems. Due to the fact that the storlet code is a potential black-box and storlets' definition gives a partial information, there is a need for runtime information. For example, when a workflow of storlets has been activated once or multiple times. This is an information that could not be derived statically. It is a benefit to find out potential issues before execution like a compiler, but the runtime information has to be provided. In spite of the static problems being interesting as well, due to their potential nature of problems and prevention of happening, they are excluded from the study.

1.6 Related Work

The area of workflow management service is vast and distributed solutions have been around for a while [25, 26]. Moreover, tools that follows a centralised [27, 28] or a decentralized architecture [29, 30]. In [31], the authors investigate a rescheduling mechanism for workflow applications in multi-cluster Grids. The scheduling algorithm proposed uses real-time information during the enactment of a workflow to re-estimate the predictions used to map the different tasks of a workflow to the available resources.

This project approach has been influenced by many types of analysis, understanding, testing, and debugging tools. These include architecture-based specification and analysis, state-based and specification-based analysis, notions of causality from the distributed systems and parallel computing communities, and debuggers for message passing systems.

Rapide [32] is an architecture description language and event pattern language that is compiled and executed as a simulation to detect event sequences, causalities, and constraint violations. It is designed for prototyping system architectures. Rapide works with a causal event history and a set of events and relationships by creating a partial ordering of sets of events called *posets* [33]. Relationships can be defined using maps (aggregators) that list the input and output event patterns. Rapide's analysis of causality is based on complete component behavioral specifications, and is decoupled from running systems (i.e. there are no tools for matching a running system to its specification).

Complex Event Processing (CEP) [34, 35], an extension to Rapide, assists in understanding of a system by organizing the activities of a system in an event abstraction hierarchy. CEP focuses on understanding a system by aggregating system-level events into higher-level events. Unlike our approach, it does not specifically take into account architectural topology. Furthermore, it is not heuristic; it expects complete component specifications and does not work well with incomplete information.

Some approaches produce finite-state models from implemented programs such as Java PathFinder 2 [36], JCAT [37], and Bandera [38], which can be analyzed using

model checkers. These tools differ from this project approach because they do not address causal relationships between events dynamically created at runtime with other parts of a system, but rather analyze static descriptions of systems to ensure various properties.

The distributed systems and parallel computing communities have addressed issues of event causality in concurrent, distributed programs [39]. In these domains, processes are roughly equivalent to components in project approach. In this work, causality can help to establish a global system snapshot, which is important for certain kinds of analysis. Results from these domains concur that, in the general case, it is difficult to determine exact causality. Determining potential causality, as project approach does, is feasible. This approach to program understanding and visualization works with black-box components and allows a user to follow event traces that occurred at any time during a run.

Chapter 2

VISION Cloud Computational Storage

This study is based on the platform of VISION Cloud Computational Storage. Chapter 2 introduces the features and innovations of both storage and computational infrastructure in VISION.

2.1 Introduction

VISION Cloud is a powerful ICT infrastructure for reliable and effective delivery of data-intensive storage services, facilitating the convergence of ICT, media and telecommunications. This goal is achieved by two innovations. Firstly, the data model of storage that has the objects stored and is associated with metadata, can be specified in a rich and flexible schema. Secondly, the powerful computational storage. The rich metadata schema can facilitate data query and operations, as well as massive computation abilities by the users.

2.2 Data Model

2.2.1 Data Object

The concept of data object is the fundamental of innovative storage features in VISION Cloud. When a file is stored in VISION, it is regarded as a data object that contains data of arbitrary type and size. A data object can be accessed with the user specified unique ID (i.e. tenant, container, object) and normally cannot be partially updated. If an object is overwritten, the whole content is replaced.

2.2.2 Metadata

Metadata takes the responsibility to store the description of data objects. Because there is no file type in VISION, one simple use of metadata is to keep the format of data objects. Due to the rich schema of metadata content, metadata also determines the access policy, placement restrictions and some operation specifications. Each data object is associated with its own metadata. Metadata is normally specified in a simple list of key-value string format. Particularly, metadata can be classified into two categories: user metadata and system metadata.

- **User Metadata**

  User metadata is specified by the user. It contains the user-defined description about the objects, but some of the content is transparent to the system. In a specific case, some user metadata are set by other executing storlets automatically instead of the object owners.

- **System Metadata**

  System metadata is used to inform the system the credentials, locations, attributes of the object, and provide the object attributes to the client. This kind of metadata is strictly typed and valid values are specified in VISION API specification. Examples include access control policy, reliability, object size, creation time, etc.

2.2.3 Data Operations

Metadata can be modified without updating its associated object, however, the user cannot update the object without updating the metadata. When a user retrieves the object together with its metadata, the system guarantees the strong consistency to metadata. Also, the user can also retrieve the metadata individually.

In the data model, typical operations of data objects and metadata are:

- Create an object with associated metadata
- Modify an existing object with new data and metadata
- Read an object's data
- Read an object's metadata
- Set an object's metadata
- Delete an object

2.3 Computational Storage

2.3.1 Storlet

Storlets are computation units that reside in VISION Cloud containers and can be triggered to react on events occurring within the same container. This mechanism allows different actions to be executed in response to different trigger events. Moreover, storlets are able to produce new events while consuming events. As a result, it is possible to chain storlets and create workflows at runtime. Through the introduction of storlets, it enables the system to automatically react to diverse object events, so that reduces the need of frequent user intervention. Conceptually, storlets are a special kind of data object, hence they inherit all capabilities of any data object and like them they are stored in a container, have metadata in the catalog and can be searched, stored, moved and placed by the same mechanisms. Storlets differ from ordinary data objects in that, they can interact with the storlet runtime environment, and spawn an active computation when triggered. A storlet can be seen as an extension of a user, it behaves like any user, it has to authenticate and authorized to do an operation.

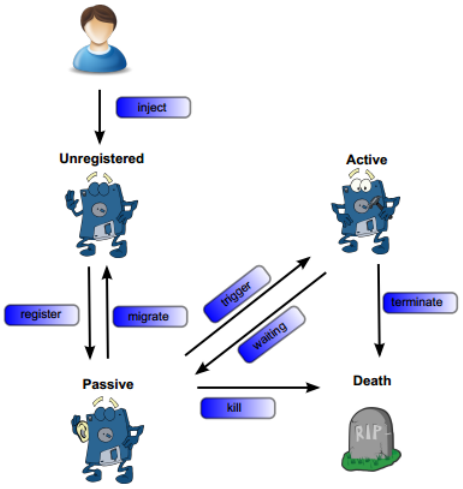


FIGURE 2.1: Storlet Lifecycle

The storlet lifecycle is illustrated in figure 2.1, a storlet is created by the tenant and stored in the storlet library as a wrapped component. In this unregistered state the storlet is moved (e.g stored) close to the data that it will process. Via the action of the storlet runtime the storlet will become registered in order that it may be appropriately triggered. Whenever a user performs computations in the cloud, he fetches the appropriate storlets, specifies the trigger conditions and injects them into the cloud storage. The trigger event could be any state change of the storage state, which is reflected by the update of data object metadata for example. A storlet might remain in passive state until it is triggered by certain event, then it becomes active and starts operation. The storlet will then either terminate, or go back into a passive state, waiting to be triggered again. Through various control mechanisms storlets may also be killed when passive.

A storlet can be referred as an "agent factory", because multiple parallel activations of it might happen; for instance a storlet that do transcoding in different video formats, if there is a large media company this storlet will be activated multiple times and in parallel probably.

2.3.2 Storlet Template

Normally, users create storlets using templates rather than start from scratch. The users are expected to be non-programmers, thus they use storlet templates, which are provided by tenants and third parties for different purposes, including compressing/decompressing, transcoding or classifying and summarizing files. Those functionalities are predefined and hard-coded in the template, it's a black-box for the user. Based on the template, users are only required to set several parameters, such as trigger conditions as desired, to create a storlet. Filling storlet parameters is as simple as setting configurations with well-known variables in the environment. The programming model spectrum is all the way from code programming to something that is more configurable, covering all the range of user needs.

2.3.3 Programming Enviroment

In the programming model of storlet, the trigger event and the result event expression are held by a single trigger handler of storlet and storlet is capable to have multiple handlers.

The trigger event, also referred as trigger evaluator, of a storlet is limited to metadata change on a single data object within the same container that is located. Typically, trigger events are presented by logical expressions, a simple expression presents one of the following *statements*, "appearance", "disappearance", "presence" and "absence", and contains a key-value string pair constraint, which will be explained afterwards in this section. Composite expressions are connected by logical operators(e.g. and(&&), or(||)). For example, if a constraint is associated with the "appearance" statement, and declared as "appearance (constraint)", that means the constraint is not satisfied before the trigger event, but becomes satisfied after the occurrence. On the other hand, "presence" statement means that the constraint keeps being satisfied both before and after the event. On the contrary, the statements "disappearance" and "absence" represent the negation of "appearance" and "presence".

When the storlet finishes its execution, there will be a result invoked on a data object. It can do arbitrary computations on a data object, such as modifying metadata or content on object. Certainly, the operations require proper authorization to take place. The target objects can be either the same object that executes on or different objects. Furthermore, the result event expression makes a prediction of metadata change when operation finished. This expression is presented also like trigger evaluators, but in addition, there is a flag (i.e [same], [diff]) identifying whether the change happens on the event object or different objects. For example, the result expression "[same] appearance (constraint)" means that when the operation finished, the specified

constraint becomes satisfied from unsatisfied on the event object (event object matches the [same] flag).

Key-value String Pair Constraint

Key-value string pair constraints are the components of trigger conditions. The keys in constraints are used to match the same keys in metadata, but the values in constraints represent the restrictions on the value in metadata which is mapped by the same key as a constraint. The constraint values are in one of the three different forms : constant form, alphabetical order form, and regular expression form.

- The *constant form* is indicated by the equal sign "=". Normally the constraint is declared as Key, ="Value". It claims that in the metadata, there must be a key-value pair attribute contains the same key as the constraint, and this key-value attribute of metadata also contains the same value as the value of constraint.
- The *alphabetical order form* support inequality operators like ">", "≥", "<" and "≤". The constraint "Key", <"Value" requires the metadata contains a key-value attribute that has the same key, in which the value that is alphabetically less than the value of constraint.
- The value of a *regular expression form* constraint is enforced to be in the standard regular expression format succeeded by the identifier symbol "∼". Again the constraint demands there is a key-value attribute in the metadata that has the same key as the constraint and the value of this attribute meets the regular expression rule specified as the value of the constraint.
- In a special case, if the value of a constraint appears as NULL, it means that the constraint is met if there is a key-value attribute with the same key existing in the metadata, regardless of the value.

2.3.4 Usecase Example of Storlet Creation

Storlet mechanism is a type of parameterized computation. Usually, users do not create storlets from scratch but only need to fill in a few parameters in the storlet templates. For example, a storlet template is created for text analysis to figure out if the text is about science or art, users only need to flexibly specify key words as parameters for the trigger conditions, without modifying the template of the storlet. However, the metadata attribute (section 2.2.2) scheme can be different between different containers. The text files can be tagged as "text" in one container but "txt" in another. As a result, the parameters to be filled in, rely on the specified scheme of the target container. The storlet will be triggered by an insertion or creation of a text object, if the trigger conditions are met the storlet is activated and do the analysis. The idea is, as soon as,

a text object is inserted into the cloud, it will be classified in order to be found easily in the cloud.

Let's say that an employee works for a big IT organization, uses VISION Cloud to manage articles and he would like to classify those articles in levels, meaning that from a general topic e.g. Science, extract more information for a specialized topic, e.g. Computer Science, creating an analysis tree chain. Hence, he creates three storlets called "Topic Classifier", "Science Classifier" and "Computer Science Classifier". So he injects the storlets into VISION Cloud, placing them, as plain data objects, near the data that he wants to process. In runtime there will be events and messages flowing through the system, if log callbacks were installed we could capture the messages and have the following structure simulating a fragment in the log:

```
[2013-12-06 16:15:30.205 , ACTIVATION ,TopicClassifier,Obj1, '('"
    FileType"="'txt'"')']
[2013-12-06 16:17:15.509 , PUT , TopicClassifier, Obj1, '('"Topic
    "="'Science'"')']
[2013-12-06 16:17:50.256 , ACTIVATION ,ScienceClassifier, Obj1,
    '('"Topic"="'Science'"')']
[2013-12-06 16:25:40.653 , PUT , ScienceClassifier, Obj1, '('"
    SpecialTopic"="'ComputerScience'"')']
[2013-12-06 16:25:59.243 , ACTIVATION , ComputerScienceClassifier,
    Obj1, '('"SpecialTopic"="'ComputerScience'"')']
[2013-12-06 16:26:10.447 , PUT , ComputerScienceClassifier, Obj1,
    '('"MoreSpecialTopic"="'CloudComputing'"')']
```

Immediately can understand, that an activation of "Topic Classifier" storlet took place because the Obj1 file type was 'txt'. And later the same storlet put in the system the tag 'Topic' on Obj1 with value "Science". The "Science Classifier" storlet was activated on that tag on Obj1 and after its execution put on Obj1 the tag 'SpecialTopic' with value "Computer Science". At last the "Computer Science Classifier" was triggered on the value of tag 'Special Topic' and finally after its job done places on Obj1 the tag 'MoreSpecialTopic' with value "Cloud Computing". Consequently, we have a text analysis tree chain, referring to as text analysis workflow.

2.4 Execution Environment

Storlets are running in an environment as shown in figure 2.2, consists of three subcomponents as well as a set of communication channels between them. These components are, the Object Service (OS), the Notification Service (NS) and the Storlet Runtime Environment (SRE). Conceptually, a single container has one Object Service, one Notification Service and one Storlet Runtime Environment in each node. Whenever a new storlet is injected into the cloud, its trigger information is extracted and registered in the Notification Service (NS). Figure 2.2 illustrates the runtime of VISION Cloud. When a state change of a data object happens, the Object Service (OS) sends the event to the NS. The NS then check its registration, if some trigger conditions match the event, the registered storlet is triggered to become active. There are many use cases relies on the storlets mentioned above, e.g. Media Transcoding, Telco and Health Care.

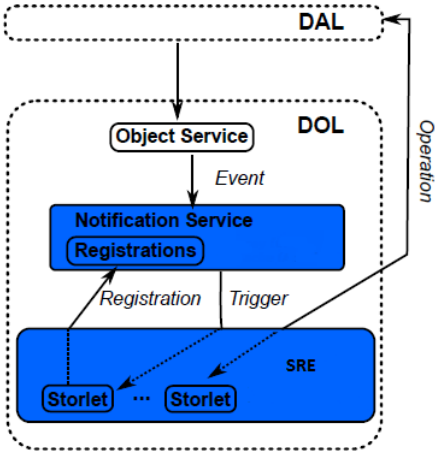


FIGURE 2.2: Storlet Execution Environment

Chapter 3

Solution Approaches

This chapter clarifies the purposes of discovering the workflow formulations and related problems. After that, the solution approach is being discussed.

3.1 Workflow Discovering Purposes

Generally, there are two main purposes for users to understand the observed workflows and moreover understand the behavior in the system. The first purpose is that if the user wants to inject a storlet workflow in the system to augment an existing workflow or isolated storlets and the other one is that he wants to keep the injection of a new storlet workflow independent from existing storlets. The injection of a single storlet to the system is a specific case of the two purposes.

3.2 Workflow Validation

A user needs to know how often a storlet workflow or individual execution of storlets are taking place in the environment. Also, he needs to find out if something is wrong with the execution of a storlet or a workflow of storlets. A wrong execution of a storlet might happen from a bad design, because it is tricky to properly set the trigger condition and result of each storlet, even though there is a template of storlet. Users are unaware of whether the connection types between storlets are as it is expected without an analysis tool during runtime, even if all the storlets seem to be fine before injection that considered individually, thus they need to find out these implications. For example, if two storlets are designed to execute in sequential but accidentally they are executed in parallel; serious side effects could be caused. Moreover, cycles in the workflow graph might produce endless triggering loop, whereas simultaneous operations on the same object are risky in ordinary storage systems.

In conclusion, storlet workflow problems are formulated and classified into three primary categories; the incorrect connection, the non-termination and the unexpected overwriting.

- **Incorrect Connection**

  The incorrect connection type problem implies the situation that the actual connection type between two storlets is different from the intention. Assuming that the existing storlets in the container are well designed, this problem might exists within both the newly created storlet workflows and the connections between the new workflows and existing storlets. For example, the user does not expect that the new storlet workflow connects with other existing storlets. Unfortunately, it does connects with some existing ones. This problem is regarded as an incorrect connection problem.

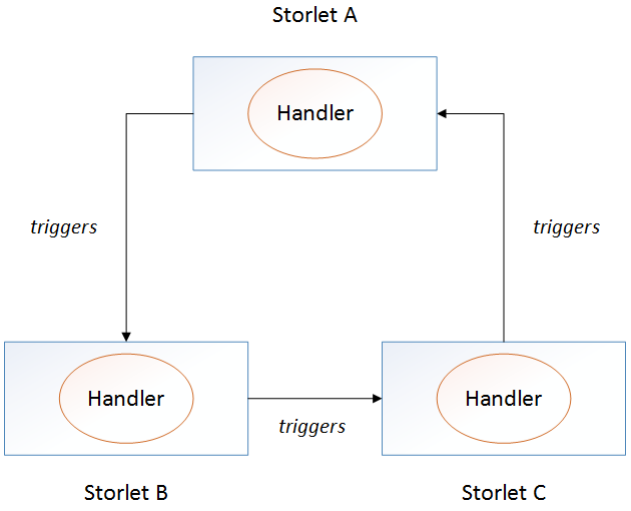


FIGURE 3.1: Non-Termination Case Example

- **Non- termination**

  Among storlets execution there might be a problem of non-termination of execution. For example, in figure 3.1 the non-termination could happen on a chain of storlets. Imagine that Storlet A got triggered by X and wrote in the system Y, but storlet B got triggered on Y and wrote in the system Z and last a storlet C got triggered on Z and wrote in the system X, so again storlet A got triggered. This is an endless cycle. Moreover, as we mentioned above, a storlet may contain multiple trigger handlers and if those handlers form a cycle, non- termination of execution problems would appear.

- **Unexpected Overwriting**

  In VISION Cloud, storlets are allowed to simultaneously operate on the same data object. Operations on objects follow the rule of last write wins. Nevertheless, if

an event can trigger one storlet, it also has a chance to trigger other storlets at the same time. When some of the running storlets operate on the same object in parallel, due to the last write wins policy, the operations that have finished early would always be overwritten by the latest one. As a result, the former one becomes useless and unexpected overwriting problems are caused.

3.3 Approaches

The most common method in debugging and management tools are the visualization and well-formatted information of a snapshot of the system. However, the suitable content to display depends on the user's requirements for the tool to support. For example, if the user wants to augment an existing workflow with storlets, he would be more interested more to see how often this relationship between storlets happens or when did it last happened. Additionally, these numbers might have groups(per day, per week etc.). Hence, there has to be a proper annotation on the graph itself with meaningful information for the user. On the other hand, if the user wants to trace a problem of storlet workflow, the best option would be to show the wrong behavior to the user and provide him with proper actions to eliminate those problems.

The contribution of this work is a novel, complete approach that aids in understanding, debugging, and visualizing the workflow formulations. The approach has four main parts:

Event capture

The implemented system is instrumented and executed. During execution, events are captured and stored in log files. Events are linked to the system through the use of log callbacks are introduced in the system.

Event classification

Based on the available knowledge of event content, the events would be classified and logged (section 4.1) aiding in sorting, filtering, and determining causal relationships.

Event causality

Based on the available knowledge about the behavior of individual components(storlets), algorithms are implemented to determine the connections by the logical consequences between a storlet trigger condition and a result event of each storlet handler. This assists the event visualization by helping the user understand the relationships between events.

Event visualization

Data in the event log are presented to the user in an organized and browseable manner. A view can be employed, to incorporate graphical models of the system

that generated the events. Although the simple organizing of this mass of data provides tremendous value, the visualization is becoming especially effective when the system engineers determine the maximum time distortion between log entry timestamps as an independent parameter.

Chapter 4

Design and Implementation

This chapter explains the solutions based on the approaches discussed in last chapter. As discussed before, in runtime, events and messages flowing through the system containing information about storlet activity. Having them captured into logs, would be analysed in order to detect chains of storlets. Log parsing and proper specification of recognition criteria of message types, in order to explore the workflow formulations, are explained. Mentioned before that storlets can be connected implicilty, these connections are classified into two types: *complete* and *partial*. Classification related algorithms are presented step by step afterwards. In normal use cases, storlets usually have only one trigger handler, and it is easier to interpret definitions and algorithms on storlet level rather than handler level. For example, it is easier to follow "interactions between storlets" than "interactions between handlers". Therefore, the explanations in this chapter assume one handler per storlet, whereas all the analysis and visualizations are actually done in handler level.

4.1 Log Parsing and Events Classification

4.1.1 Introduction

Because of the vast number of event messages, captured by proper log callbacks in the system, there is a need to define a way to read, process or translate the log file consist of structured text data. To solve the relationships between these messages for this study, initially implemented a log parser based on ANTLR [40], composed of lexical and syntax analysis for log files, that will be integrated with the analysis tool that is presented in this project. As a result, a powerful parser for reading the log files of storlet runtime execution.

4.1.2 Log Structure

As mentioned before, storlets can be connected indirectly and in order to form a workflow, one storlet have to add metadata on a given object and another storlet is activated by that event on object. However, to build the logic of discovering these relationships, the log consists of two event types, *storlet's activation* and *storlet's put*; what caused the storlet to be activated and what wrote into the system. Below, the fields of every message are explained.

- **Storlet Activation Event**

  [Timestamp, Event type, Storlet ID, UUID, Container ID, Machine ID, Tenant name, Object ID, Trigger condition]

Table 4.1: Activation Event Message Definition

Field	Meaning
Timestamp	Time and date when a storlet got activated.
Event Type	The type of the event, e.g. ACTIVATION.
Storlet ID	The id of the storlet. Storlet name.
UUID	Universal Unique ID, used in actual algorithm execution.
Container ID	Actual scope of a storlet execution is within a container.
Machine ID	The IP address where the storlet executed
Tenant name	The name of the tenant that executed a storlet.
Object ID	The object name storlet got invoked on.
Trigger condition	The trigger condition of the storlet (section 2.3.3).

- **Storlet Put Event**

  [Timestamp, Event Type, Storlet ID, UUID, Container ID, Machine ID, Object ID, Result Expression]

Table 4.2: Put Event Message Definition

Field	Meaning
Timestamp	Time and date when a storlet got activated.
Event Type	The type of the event, e.g. PUT.
Storlet ID	The id of the storlet. Storlet name.
UUID	Universal Unique ID, used in actual algorithm execution.
Container ID	Actual scope of a storlet execution is within a container.
Machine ID	The IP address where the storlet executed.
Object ID	The object name storlet got invoked on and made changes or not.
Result Expression	The result expression defining the metadata change on the Object ID when the storlet execution has finished (section 2.3.3).

4.2 Connection Classification

In order to classify the connection types between storlets, it is required to match the result event of one storlet with the trigger condition of another storlet. As introduced in chapter 2, the only component that has continuous information about the storlet actions in the system is the Storlet Runtime Environment(SRE) in each machine of VISION Cloud . Therefore, the log callbacks that will be installed will log all the storlet actions within the container. After the parsing of a log, containing event messages which is discussed above in section 4.1, the next step is to match the trigger condition of one storlet to the result event of another storlet, and determine the connection type with the classification algorithm which will be explained in section 4.3.

4.2.1 Motivation

The context of this study is the real-time monitoring of the system, hence the connections between the storlets are derived from their actual behaviour in the system. Consequently, the connections between storlets are classified into two types mentioned above, which are distinguised by the strength of connections. Analysis of connection types is following.

4.2.2 Connection Types

The connection types distinction is as follows:

Complete Trigger

A *complete trigger* denotes that there was a strong connection between two storlets. If a source storlet has a complete trigger to the target storlet, that means the action result of the source storlet was able to trigger the target storlet by itself. Particularly, there are strong and weak cases among complete triggers. Strong complete triggers ensure that if there was a result produced by correct execution of the source storlet, it was indeed triggered the target storlet. For example, if the trigger condition of target storlet is "Appearance(key, = A)", as well as the result of source storlet "[same]Appearance(key, = A)", it infers that there is a strong complete trigger connect them. On the other hand, the weak case, the source storlet has a chance to produce a result that triggers the target storlet alone. For example, if the target storlet has the trigger condition "Appearance(key1, = A)", but the result of source storlet is "[same]Appearance(key1, = A)||[same]Appearance(key2, = B)", only when the result comes out as "[same]Appearance(key1, = A)", the trigger condition of source storlet is matched. These two cases of complete trigger are not separated in the analysis result. Because that there is no connection between two storlets that guarantees triggering for

every time, due to the chance of execution failure and unpredictable user intervention. For example, if a user performs an external execution on the object which the source storlet is operating on, the result has a chance to be overwritten, thereby the target storlet cannot be triggered anymore.

Partial Trigger

If the "strength" of a connection is not sufficient to make it a complete trigger, the connection still has a chance to be a *partial trigger*. If a source storlet connects to a target storlet with a partial trigger, it means the action result of source storlet can never trigger the target storlet itself, but can form a complete trigger by complementing with other storlets results or user behaviour. For example, if the trigger condition of target storlet is "Appearance(key1 = A) && Appearance(key2, = B)", but the source storlet only produce the result "[same]Appearance(key1, = A)", it requires another storlet or user behaviour to complement the rest part of condition "Appearance(key2, = B)", and these two results should be on the same object. Therefore, it is a partial trigger. A user can easily modify the storlet parameters to upgrade a partial trigger to complete trigger or eliminate the unexpected possible connections between storlets.

In conclusion, the connection type classification follows the "strongest win" principle. That means even the connection between two storlets is a complete trigger, it is possible that the result event sometimes only produces a connection type as partial trigger. To formulate the relations between different connection types with logical consequences, it is presented as the "complete trigger" implies the "partial trigger".

4.2.3 Underlying Methodologies

This subsection introduces the mathematical basis that applied in the classification related algorithm. Conjunctive(disjunctive) normal forms are used to break the expressions into proper fractions and help find out the "strongest" connection as the final connection type.

Conjunctive and Disjunctive Normal Form

In boolean logic, a conjunctive normal form(CNF)[41] is a conjunction of clauses, where each clause is a disjunction of literals. They can be seen as conjunctions of one-literal clauses and conjunctions of a single clause, respectively. As in the disjunctive normal form (DNF)[42] the only propositional connectives a formula in CNF can contain are "and", "or", and "not". The not operator can only be used as part of a literal, which means that it can only precede a propositional variable or a predicate symbol.

A propositional formula of conjunctive normal form

$$\bigwedge_{i=1}^{n} \vee_{j=1}^{m_i} C_{ij} \tag{4.1}$$

where each $C_{ij}, i = 1, \ldots, n; j = 1, \ldots, m_i$, is either an atomic formula (a variable or constant) or the negation of an atomic formula. The conjunctive normal form 4.1 is a tautology if and only if for any i one can find both formulas p and $\neg p$ among the $C_{i1}, \ldots, C_i m_i$, for some atomic formula p. Given any propositional formula A, one can construct a conjunctive normal form B equivalent to it and containing the same variables and constants as A. This B is called the conjunctive normal form of A.

On the contrary, a disjunctive normal form (DNF) is a standardization of a logical formula which is a disjunction of conjunctive clauses. The formula of DNF is displayed as

$$\bigvee_{i=1}^{n} \bigwedge\nolimits_{j=1}^{m_i} C_{ij}, \tag{4.2}$$

4.2.4 Algorithm

To classify the connection type between two storlets is to figure out the logical consequence of the result event of a source storlet to the trigger condition of a target storlet. The classification algorithm 1 is used to distinguish a connection of two different types: complete trigger or partial trigger. In the algorithm a result expression and a condition expression are initialized as input data. Due to the "strongest win" policy of connection types, the result expression is transformed to a DNF and split into fractions as *pure conjunctions of atoms* (i.e. $\bigwedge_{i=1}^{n} Atom_i$) in order to find out the "strongest" connection types produced by fractions. And then the algorithm checks each conjunction with the condition expression of target storlet. If there is a logical implication from the current conjunction to the condition expression, the connection type of current conjunction is determined as "complete trigger" and algorithm enters the next loop. On the other hand, if there is no implication exists, the algorithm

Algorithm 1 Connection Classification

Input: $resExpr \leftarrow ResultExpression(SourceStorlet)$ ▷ resExpr is the result expression of source storlet
1: $condExpr \leftarrow ConditionExpression(TargetStorlet)$ ▷ condExpr is the condition expression of the target storlet
Output: Complete trigger, Partial trigger

2: $resExprInDNF \leftarrow TransformToDNF(resExpr)$ ▷ Transform the result expression to DNF
3: $resExprFracs \leftarrow SplitExpressionByOr(resExprInDNF)$ ▷ Split the DNF expression into fractions by 'or' operators,each fraction is a pure conjunction of atoms
4: **for all** $resExprFrac \in resExprFracs$ **do**
5: **if** $resExprFrac \Longrightarrow condExpr$ **then**
6: Number of Complete Trigger + 1
7: **break**
8: **else**
9: $condExprInCNF \leftarrow TransformToCNF(condExpr)$ ▷ Transform the condition expression to CNF
10: $condExprFracs \leftarrow SplitExpressionByAnd(condExprInCNF)$ ▷ Split the CNF expression into fractions by 'and' operators, each fraction is a disjunction of atoms
11: **for all** $condExprFrac \in condExprFracs$ **do**
12: **if** $resExprFrac \Longrightarrow condExprFrac$ **then**
13: Number of Partial Trigger + 1
14: **break**
15: **end if**
16: **end for**
17: **end if**
18: **end for**

▷ (algorithm continues on the next page)

Algorithm 1 Connection Classification Algorithm (continued)

```
19: if Number of Complete Trigger > 0 then
20:     return Complete Trigger
21: else if Number of Partial Trigger > 0 then
22:     return Partial Trigger
23: end if
```

continues the classification phase of partial trigger. To identify a partial trigger, the algorithm converts the condition expression of target storlet into a CNF, then break the CNF into *disjunction of atoms* by the operator "and". Next, the algorithm compares each disjunction fraction of result to each conjunction fraction of condition. If there is an implication exists, the algorithm announces that the current disjunction fraction of result connects to trigger condition with a partial trigger, and then enters the next loop. After all the conjunctions of result events are analyzed, if there is at least one conjunction forms a complete trigger, then the whole connection type is complete trigger. If there is no complete trigger but only partial trigger(s), the connection is determined as a partial trigger. Otherwise there is not connection, which is unrelated, between the result event and trigger condition.

4.3 Expression Matching

4.3.1 Introduction

Another challenging part of the Algorithm 1 is to determine whether there is a logical implication between two expressions (e.g. code line 5 & 12). As referred in section 4.1 the complete expressions are laying in the log entries of events for each activation and put of storlet, parsing the log and extracting information which will be the input for the algorithms. In this section, will be explained which log entries will be choosed to be compared, how to prove two statements are a match and how to match the constraint values of the expressions.

4.3.2 Underlying Methodologies

This subsection introduces the mathematical basis that applied in the expression matching related algorithms. Automated Theory Providing(ATP) [43] is used to solve the problem whether two expressions have logical implication. Moreover, the Deterministic Finite Automaton(DFA) [44] applied in algorithm to be able to match regular expressions. Lastly the partial ordering and causal relationship context of events used to achieve the result about the theoretical implication during time.

Automated Theorem Providing

Automated theorem proving (ATP), also known as automated deduction is a part of automated reasoning dealing with proving mathematical theorems by computer programs. The general idea of ATP is to prove that the conjecture of some statements is a logical consequence of a set of statements including axioms and hypothesis. For example, the disordered surfaces of a Rubic cube can be the conjuncture, all possible changes are treated as axioms, ATP system can prove that the cube can be rearranged into solution state. The language of conjuncture, axioms and hypotheses are formulated as logical expressions, not only in first-order logic, but also possibly a higher order logic. Logical expressions are produced based on the syntax declared by each ATP system, so that the system can recognize and manipulate the expressions. The ATP systems prove the conjuncture follows the axioms and hypothesis in a manner that can be agreed by the public. The proof is not only an argumentation of logical consequences but also describes a process to solve problems. For instance, the proof of Rubic cube example provides a solution to the rearrangement problem.

Among various of ATP systems and libraries, Orbital Library [45] is selected for the study due to the flexible portability and Java API supported. This library provides object-oriented representations for mathematical and logical expressions, it also provides algorithms for theorem proving, such as algorithms that convert regular logical expression forms to DNFs or CNFs, and logical implication proving algorithms. Additionally, it is well documented and simple to use.

Deterministic Finite Automata

According to Alo & Ullman [46], a deterministic Finite Automaton or DFA is defined by the quintuple:

$$M = (Q, \Sigma, \delta, q0, F) \tag{4.3}$$

where,

Q stands for a finite set of states,

Σ (Sigma) stands for the set of symbols, so called alphabet

$\delta : Qx\Sigma \rightarrow Q$ (Delta) stands for a total function called transition function

$q0$ stands for initial state synonymously start state, and $q0 \in Q$

F is a set of final states synonymously accepted states where each state of F is element of Q as well ($F \subseteq Q$ (F is subset of Q))

Strings or input strings are sequence of symbols chosen from alphabet Σ. For example **abb** is a string of alphabet $\Sigma = \{a, b\}$, also it can be $\Sigma = \{a, b, c\}$ but it just happens not to have any **c**'s.

Σ^* stands for the set of all string over the alphabet Σ and the special string called empty string represented by epsilon. The language of a DFA is a subset of Σ^* for some alphabet Σ.

With other words the language that DFA defines is the set of strings that may take the start state to a final state. A DFA operates in the following manner: when program starts the current state is assumed to be the initial state $q0$, on each input symbol or character the current state is supposed to move on another state (including itself). When the string reach the most right symbol (last one) the string is accepted only if the current state is one of the accepted states, otherwise the string is rejected.

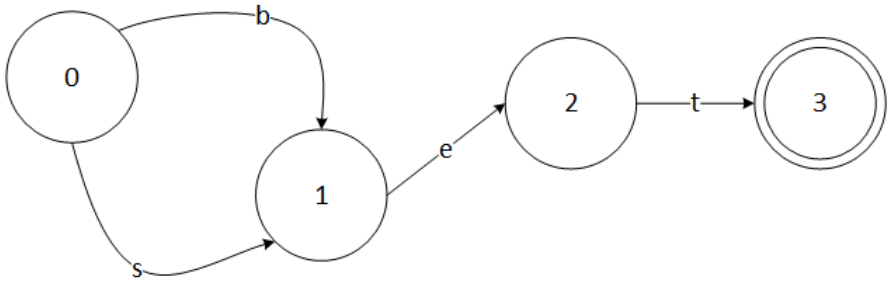


FIGURE 4.1: Regular expression modelled by DFA example.

DFA is used to model standard regular expressions in this study. States of DFAs represent possible intermediate or final strings of regular expressions, transitions stand for possible new characters appended to current strings. For example, the DFA in figure 4.1 models the regular expression [bs]et. From state 0, the automaton has two options, either chooses "b" or "s", and in the next two states there is only one path that is "et". As a result, the possible strings which match the given regular expression are "bet" and "set". In respect of the implementation, the java library dk.bricks.automaton [47] is selected for this study. It supports to model DFAs from regular expressions, and provides APIs of standard regular expression operations, such as concatenation, union, and intersection. More details will be explained in section 4.3.2.

Partial ordering & Causal relationship

The analysis of causality is closely related to temporal reasoning. As everyday experience tells us, every cause must precede its effect. Names such as "happened before"[48] and "is concurrent with" for relations which are causal rather than temporal reflect this fact. However, such a terminology — although quite suggestive — is somewhat misleading. In this section, we briefly discuss the relationship between time and causality. Let $t(e)$ denote the real-time instant at which event e of a given computation takes place. Obviously, idealized real time $(t, <)$ is *consistent with causality*;

it does not, however, *characterize causality*, because $t(e) < t(e')$ does not necessarily imply $e \rightarrow e'$. An additional problem is that a set of synchronized local real time clocks, i.e. a proper realization of an idealized "wall clock", is generally not available in distributed systems. Fortunately, it is possible to realize a system of *logical clocks* which guarantees that the timestamps derived are still consistent with causality. This was shown by Lamport in [48].

4.3.3 Value-based Matching

The value-based matching determines the logical relation between two constraint values. As mentioned in subsection 2.3.3, there are three different forms of values in constraints, which are constant form, alphabetical order form and regular expression form. Hence, there would be six pairs of compare, which are below explained.

Constant VS Constant

In this case if and only if two values are equal, they cover each other. Otherwise they do not. For example. = "1" covers = "1", but = "1" mutually exclusive with = "2".

Constant VS Alphabetical Order

To match a constant form with an alphabetical order form, the constant value is put on the left side of the alphabetical order value to composite an inequality. If the inequality is true, then the constant value is covered by the alphabetical order value, otherwise they are mutually exclusive. For example, = "1" is covered by < "2" because "1" < "2".

Constant VS Regular Expression

In the same manner as the above two cases, if the constant value string matches the regular expression, then the constant is covered. Otherwise they do not. For example, = "cloudcomputing" is covered by ~".*cloud.*" because "cloud computing" matches regular expression ".*cloud.*".

Alphabetical Order VS Alphabetical Order

This case is actually to compare two ranges. Obviously, if both two values have no upper bound or lower bound, the relation can be either covering or overlapping. If mnot, there must be one value has only an upper bound, and the other one has only a lower bound. In this case, if the upper bound is greater (or greater equal depends on the boundary type of values) than the lower bound, they overlap. Otherwise they are exclusive. For example, > "1" covers > "3", > "1" overlaps < "2", but < "1" and > "2" are exclusive.

Alphabetical Order VS Regular Expression

To check the relation type between an alphabetical order value and a regular expression value the technology of DFA is applied. Firstly, a DFA is constructed based on the given regular expression. Then the bound value is extracted from the order form. In the pseudo code, it is assumed that the order value has a lower bound. Next, since the order follows the alphabetical rule, comparison between strings are character by character. The first character of the lower bound value is chosen as a measure. If the possible first characters that produced by the DFA are all greater than the measure, then it is a covering relation. On the other hand, if some of the possible first characters are greater, then the two expression overlap. If the DFA can only produce an equal character as the measure, the algorithm goes into next loop. If all the characters in which the DFA can produce are less than the measure, values do not overlap. In the second loop (if exists), the algorithm follows the same manner as the first loop, the measure is compared with the greatest character can be produced by the DFA on the second position. Until all the characters of the lower bound value are compared, if there is no result yet, but the DFA can produce more characters, that means the DFA can produce a greater string, the regular expression covers the order value. Otherwise, the result depends on the boundary type. If it is $<$ then exclusive, if $<=$ then cover. Vice verses, the cases of $>$ and $>=$ compare the upper bound value with DFA, but the results contradict.

Regular Expression VS Regular Expression

The algorithm to match two regular expressions is also relying on the construction of DFA. Fortunately, the dk.brick.automaton library supports the operation of intersection. After the two regular expressions are converted to DFAs, the intersection method from library is invoked, if the two DFAs intersect, and the intersection is exactly equal to one regular expression, then this expression is covered by the other mone, otherwise these two expression overlap. If the intersection is empty, then the mtwo regular expressions are mutually exclusive.

4.3.4 Expression Analysis Algorithm

During runtime a large number of messages will be captured and stored in the logs. Making sense of a collection of captured messages requires parsing the data contained within them. Having both the activation and put events of storlets will be able to figure out the relationships between storlets. As mentioned from the start of the chapter is basically a need of ones storlet result expression to match anothers storlet trigger condition, so to end up with a chain of these two storlets. The expression analysis algorithm have some steps in its implementation that needs attention.

In algorithm 2, having the log parsed, there are two methods that categorize the parsed log entries to data structures of activation and put of a storlet(e.g. code line 1 & 2). At the beginning of the algorithm a data transfer object[49](DTO) for the storlet handler is presented. Looping the activation and put data structures and matching them with a UUID field, that is attached in every log entry to immediately distinguish in the log from which activation a put is derived (e.g. code lines 3-11), results in an actual storlet handler holding all the information that log entries have (Section 4.1).

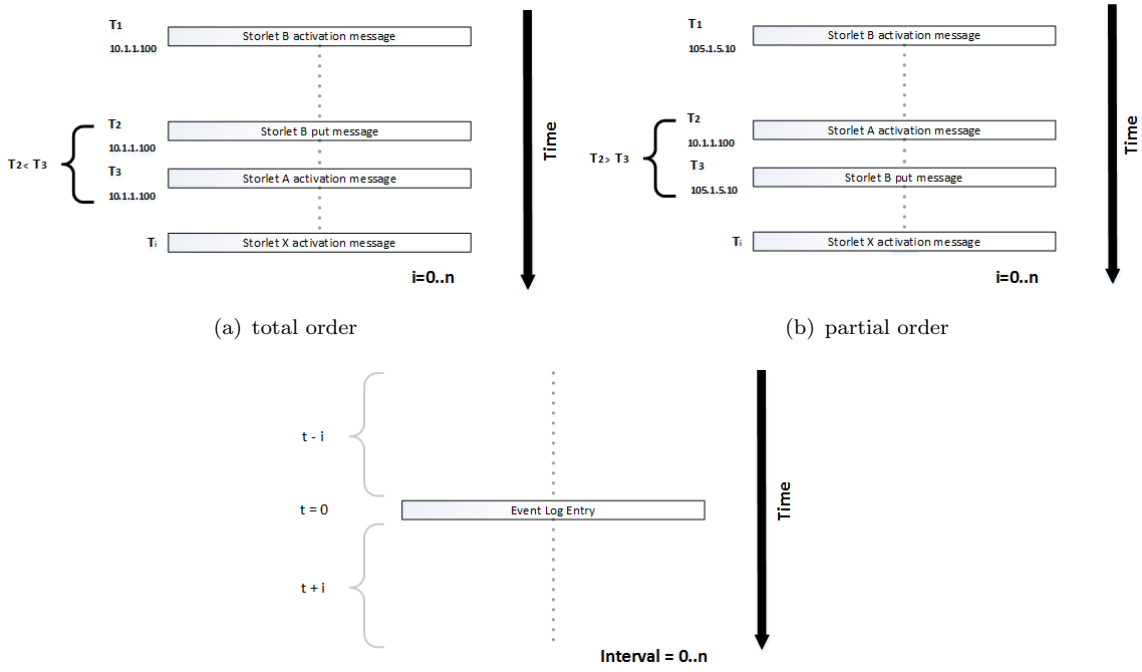
For every activation, there is a need to figure out the reason why that happened. The log entries are time ordered, if they produced from the same machine (same clock) there is a total order of events, but because the environment is distributed, the clocks of different machines might be out of sync. In figure 4.2 there is an example of previous statement. Subfigure 4.2(a), illustrates an example of log entries presenting the fact that storlet A might have triggered from the put of storlet B. Storlet A got activated at T_3 and storlet B wrote in the system a metadata tag at T_2, because the events produced from the same machine are supposed to be in total order, then certainly the T_2 precedes T_3. In contrary, in subfigure 4.2(b) the events are emmited from different machines, the put event of storlet B produced at T_3 from a different machine that storlet A got activated (T_2). If there is a logical implication between storlet B result expression and storlet A trigger condition expression they are assumed to be causal related. Therefore the T_2 might follows the T_3 and in this case there is a partial order of events. Hence, to confront programmatically this state there is a need to introduce a parameter to the system, that holds this maximum time distortion between event timestamps and defines a window as a continuous log slide(interval) over an event timestamp (Figure 4.2(c)).

Bear in mind the above definitions return to the expression analysis algorithm. Having in loop the trigger conditions from the DTO of storlet handlers (e.g. code line 12), searching for a proper put that match this expression. Besides the fact of time for causal relationships, the activation and put actions must be invoked on the same object to be considered legit (code lines 19 & 24). However, when the legit puts have derived for an activation, have to match and determine whether there is a logical implication between two expressions (e.g. code line 33). That was a difficult supplementary part to algorithm 1. To solve this problem the technology of automated theorem proving is used. As mentioned in subsection 4.3.2, ATP use a set of statements, normally axioms, to prove the conjuncture of other statements. In this algorithm, the implication between two expressions is the conjuncture of statements need to be proved. Therefore, the next step is to collect axioms. However, storlets information does not directly provide axioms but only complete expressions, axioms need to be extracted from these expressions. The format of axiom is expected to be the relations between atoms, and there are three possible types of relation, which are *cover, overlap and mutually exclusive.*

Algorithm 2 Expression Analysis Algorithm

Input: *timeframeParameter* ▷ An independent system parameter, the maximum time distortion between timestamps.

```
1: inputInfoList ← getParsedActivationsFromLog() ▷ Method categorize the log entries for activations of storlets.
2: outputInfoList ← getParsedPutsFromLog()▷ Method categorize the log entries for puts of storlets.
3: for all inputInfo ∈ inputInfoList do
4:     for all outputInfo ∈ outputInfoList do
5:         if inputInfo.getUUID().equals(output.getUUID()) then
6:             storletHandler ← newStorletHandler(inputInfo, outputInfo)
                ▷ A storlet handler has an input expression and an output expression.
7:             storletHandlersList.add(storletHandler)
8:             break
9:         end if
10:    end for
11: end for
12: for all storletHandlerTarget ∈ storletHandlersList do
13:    for all storletHandlerSource ∈ storletHandlersList do
14:        activatedOnObject ← storletHandlerTarget.getObjectInvokedOn()
15:        putOnObject ← storletHandlerSource.getObjectInvokedOn()
16:        activationTimestamp ← storletHandlerTarget.getInput().getTimestamp()
17:        putTimestamp ← storletHandlerSource.getOutput().getTimestamp()
18:        if timeframeParameter.equals(0) then
19:            if activatedOnObject.equals(putOnObject)
20:                && putTimestamp < activationTimestamp  then
21:                legitStorletHandlerSourceList.add(storletHandlerSource)
22:            end if
23:        else
24:            if activatedOnObject.equals(putOnObject) && (putTimestamp >=
25:                activationTimestamp - (timeframeParameter * 60000)
26:                && putTimestamp <= activationTimestamp
27:                + (timeframeParameter * 60000))  then
28:                legitStorletHandlerSourceList.add(storletHandlerSource)
29:            end if
30:        end if
31:    end for
32:    for all legitStorletHandlerSource ∈ legitStorletHandlerSourceList do
33:        result ← proveExpression(storletHandlerTarget.getInputExpression(),
34:        legitStorletHandlerSource.getOutputExpression())
35:        if result.contains(0) then
36:            finalResultMap.put(legitStorletHandlerSource.getStorletName,result)
37:            break
38:        else if result.contains(1) then
39:            finalResultMap.put(legitStorletHandlerSource.getStorletName,result)
40:        end if
41:    end for
42: end for
```



(a) total order

(b) partial order

(c) sliding window

FIGURE 4.2: Events with timestamps from a log file example

The *cover* type indicates that two atoms share the same key, the one value is a subset of the second one. For example, Atom1 : key,>"7", Atom2 : key,>"2", in this case Atom1 is a subset of Atom2, Atom1 is covered by thus the axioms are generated as $Atom1 \Longrightarrow Atom2$.

Overlap type means that two atoms have the same key, and their values have common cases. For example, Atom1 : key,>"4", Atom2 : key,<"8", in this case these two atoms are under overlap state. The solution of axioms generation is to break up Atom2 into two parts. One part Atom2a is covered by Atom1, while the other part Atom2b is mutually exclusive with Atom1. Besides, the matched atom in the trigger condition expression is also split into $Atom2a||Atom2b$

Mutually exclusive denotes that the two atoms share the same key but values can never match. This relation is declared as no implication. For example, Atom1 : key,="4", Atom2 : key,="6", they are mutually exclusive, then the axioms are obtained as $Atom1 ! \Longrightarrow Atom2$, $Atom2 ! \Longrightarrow Atom1$.

Lastly, at the end of the algorithm 2 there must be a result from algorithm 1, if and only expression values are matched, result in a complete trigger or a partial trigger. In the algorithm 2 (e.g. code lines 35-37) if there is a complete trigger for an activation it gets out of the loop and continues for the next activation, but if it is partially triggered (e.g. code lines 38-39) keeps looping the legit puts until the conjunction of statements fulfilled its trigger condition.

4.4 Graph Visualization

4.4.1 Motivation

In VISION Cloud there is a large number of storlets, which might mutually interact having dynamic interactions. Hence, the log entries are time-depended, as discussed in previous sections. The log files of captured events during runtime, are considered as data input for the analysis tool and the formulated workflows, which might discovered, are the system formulation at a given moment, represent the interaction graph corresponding to the state of the system made of storlets activity.

A dynamic graph [50] *G(t) = (N(t),E(t))* where t represents the time, is composed of:

- a set of nodes *N(t)*. In this case the nodes are the storlets. This set may change with t. Each node n owns a set of characteristics $C_n(t)$ that may vary with time.
- a set of edges *E(t)*. Edge is defined as the connection between two storlets. *E(t)* may change with t. Each edge $e_{i,j} = n_i, n_j$ is defined as a pair of nodes, and owns a set of characteristics $C_{e_{i,j}}(t)$ that may vary with time.

Therefore a dynamic graph *G(t)* may change with the time t. According to the underlying modeled system, a dynamic graph may be considered as a family of graphs or as a complete graph where, according to t some nodes or edges can be zero weighted. There exist however another way of defining a dynamic graph, based on a discrete-time representation, and thus very well suited for most simulations.

A dynamic graph can be defined as a finite or infinite ordered set of couples (date, {events}). Every set of graph events may modify the graph structure/ composition/ topology and/ or characteristics of some graph elements. The different snapshots of Fig. 4.3 correspond to the series of sets of graph events listed below:

(0, n_1 creation, n_2 creation, (n_1, n_2) creation)
(1, n_3 creation, n_4 creation, (n_1, n_3) creation)
(2, (n_3, n_4) creation)

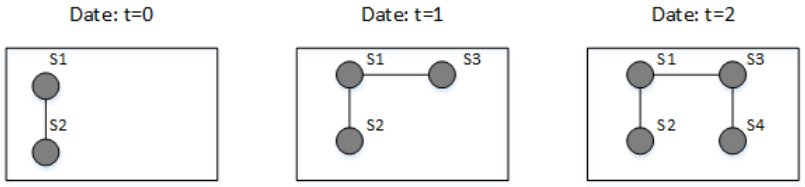


FIGURE 4.3:]
Snapshots of a Dynamic Graph on the time interval [0, 2].

This approach was chosen for the use of Java library GraphStream as explained in the forthcoming subsection.

4.4.2 Visualization tool: GraphStream

GraphStream [51] is a Java library whose purpose is to create and manipulate dynamic graphs in order to study them and to use them in simulations. The whole library has been devised with the idea that graph will evolve during time. It is also naturally well suited for the manipulation of "static" graphs.

The central notion in GraphStream is that graphs are not static objects, but streams of events, and these streams can flow from producers (simulations, graph generators, graph readers, the web, etc.) passing by filters (a graph structure, a converter, etc.) to outputs (graph writers, a network, a display, etc). There also exist a file format called DGS allowing the expression of graph changes as series of events that can be read by GraphStream. It is also supports most of the commonly used graph file formats (DOT, GML, GEXF, Pajek, GraphML, TLP). It can read files in these formats thus making the interface with other graph libraries easier.

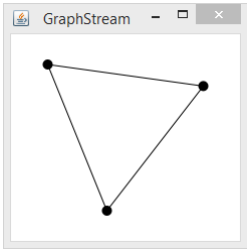


FIGURE 4.4: The GraphStream viewer.

Creating and visualizing a graph using GraphStream is more than easy. Indeed, the following piece of code creates a graph made of 3 nodes and 3 edges and open a window for visualizing it, the whole in only 8 lines of code. The result is shown in Fig. 4.4. The positions of nodes are automatically computed in order to enhance the display.

```
import org.graphstream.graph.Graph;
import org.graphstream.graph.implementations.DefaultGraph;

public class Easy {
    public static void main(String args[]) {
        Graph graph = new DefaultGraph("easy");
        graph.addNode("A");
        graph.addNode("B");
        graph.addNode("C");
        graph.addEdge("AB", "A", "B");
        graph.addEdge("BC", "B", "C");
        graph.addEdge("CA", "C", "A");
        graph.display();
    }
}
```

Chapter 5

Results

This chapter interprets the procedure of workflow extraction parsing a log file from the system. The fragment of log file was choosed in order to enhance the solution of use case requirements from a big research institute and a media company.

5.1 Usecase Examples

Research Institute Usecase

Scientific research articles provide a method to scientists where they are able to communicate with other scientists about the results of their research. With the exponentially increasing number of research areas, vast number of scientific reports are produced and there is a need of properly classification before they are stored.

A research institute uses VISION Cloud as an implemented infrastructure that classifies the research articles tagging them appropriately in order to be easily searched and found later. A researcher has finished his research and he needs to submit his report at institute's database. The administrator users have injected a set of storlets that help the users to classify their reports. Hence, researcher's final report is a plain text file and he uploads it to the container of VISION. A newly creation or insertion of text file is detected from the system and the TopicClassifier storlet is activated and made some text analysis to find the general topic of the text (e.g Science, Art etc). When it finishes, another specialized topic storlet got triggered, named ScienceClassifier, because the previous storlet "wrote" to the system a metadata tag where the topic of text is Science, which results in a more specialized topic of Computer Science. After it's execution, the ComputerScienceClassifier storlet automatically figures out a more specialised topic for the text that is about Cloud Computing.

Media Company Usecase

Nowadays, with the exponentially increasing number of smartphone users, it becomes a trend to watch high quality videos over the Internet on phones nowadays. However, as a light-weight portable device, smart phones only support a limited set of file formats, hence, there is an urgent demand to convert incompatible files to suitable formats for proper delivery.

In this usecase, the media company wants to deliver a high quality of service for its users and therefore it chooses to use VISION Cloud based on its needs. A video file in MP4 format and with H.264 codec can be supported from the all modern smartphones. Hence, one output video is able to cover all smartphone users. When a reporter finishes his story, the video footage from the news have to be accessible to the public for download. Therefore, he injects a set of storlets into the same container to help him automatically deal with the compatibility issues for mobile users. When a new video file is stored and not in MP4 video format, the VideoTranscoding storlet automatically converts it to MP4 format. Next, the VideoCompression storlet compress an MP4 format video file to video with H.264 codec. Finally, when the video file is ready mobile users can watch it from their smartphones.

5.1.1 Log file simulation for usecases

The fragment of log file that is depicted below, contains all the information about the events that happened in the system describing the above usecases actions. The log structure is presented in section 4.1.2, explaining the fields that a log entry is constructed of.

```
2013-12-06 16:15:30.205, ACTIVATION, TopicClassifier.handler1,
    ContainerA , 192.168.1.2, TENANT_A, Obj1, '('"FileType"="'txt
    '"')'
2013-12-06 16:16:15.355, ACTIVATION, VideoTranscoding.handler1,
    ContainerA , 192.168.1.101, TENANT_A, Obj5, '('"Format"="AVI"')
    '
2013-12-06 16:17:15.509, PUT, TopicClassifier.handler1, ContainerA
    , 192.168.1.2,  Obj1, '('"Topic"="'Science'"')'
2013-12-06 16:17:35.775, PUT, VideoTranscoding.handler1, ContainerA
     , 192.168.1.101, Obj5, '('"Format"="MP4"')'
2013-12-06 16:17:50.256, ACTIVATION, ScienceClassifier.handler1,
    ContainerA , 192.168.1.2, TENANT_A, Obj1, '('"Topic"="'Science
    '"')'
2013-12-06 16:25:40.653, PUT, ScienceClassifier.handler1,
    ContainerA , 192.168.1.2, Obj1, '('"SpecialTopic"="'
    ComputerScience'"')'
```

```
2013-12-06 16:25:50.205, ACTIVATION, VideoCompression.handler1,
    ContainerA , 192.168.1.101,  TENANT_A, Obj5, ’(’"Format"="MP4
    "’)’
2013-12-06 16:25:59.243, ACTIVATION, ComputerScienceClassifier.
    handler1, ContainerA , 192.168.1.2, TENANT_A, Obj1, ’(’"
    SpecialTopic"="’ComputerScience’"’)’
2013-12-06 16:26:10.447, PUT,  ComputerScienceClassifier.handler1,
    ContainerA , 192.168.1.2,  Obj1, ’(’"MoreSpecialTopic"="’
    CloudComputing’"’)’
2013-12-06 16:26:15.509, PUT, VideoCompression.handler1, ContainerA
     , 192.168.1.101, Obj5, ’(’"Codec"="H.264"’)’
```

The storlets are created and stored in the container and when their trigger conditions are fulfilled they are passing from the idle state to the activate state. Their activations and results are logged properly.

5.2 Workflow Visualization

To create a workflow the user should first have a clear design of the functionalities together with the initial trigger conditions, and then create the preliminary version of storlets. Next, those storlets are analysed in the service; a view that presents the connections between storlets is generated. However, based on the requirement, there might be some restrictions of the connections between those storlets. For example, some storlets are enforced to execute in parallel or sequential. In the next subsection an example of storlet workflow construction will be demonstrated.

5.2.1 Workflow construction and connection checking

In figure 5.1, the graph contains three storlets for analysis text workflow connected by two complete triggers and two storlets for media workflow connected by one complete trigger.

The 'Topic Classifier' handler is connected to 'Science Classifier' with complete trigger. It is exactly as expected, because the 'Topic Classifier' after the text analysis, for discovering the topic of the text document, it results in a metadata tag that fulfills the trigger condition of 'Science Classifier'. Following the workflow steps, a text analysis workflow is visualised. The user of the workflow service can interact with the graph by navigate in any node and a new window appears containing all the information about its activation.

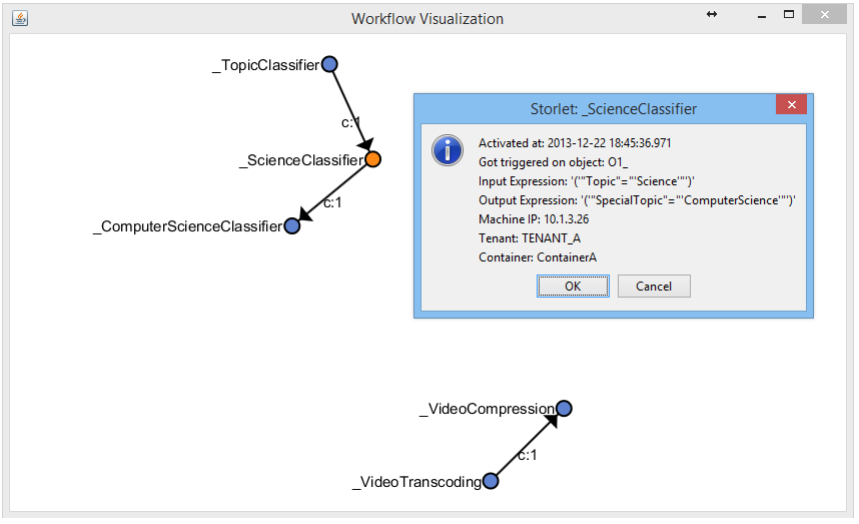


FIGURE 5.1: Graph visualization of text analysis chain and media chain.

Consequently, whenever new workflows are discovered they are completely separated from each other containing their information in their nodes. From the edges, information about the connection between two storlets is illustrated.

Chapter 6

Evaluation and Analysis

As noted in above chapters, VISION Cloud has an asynchronous nature and messages between nodes of a datacenter can be delayed or can be out of time order. An optimisation could be succeeded by an independent system parameter that introduced and discussed in section 4.3.4. Due to the limited size of log file, the evaluation will be done on this parameter by analysing the time cost and algorithm performance based on this value. The evaluation of analyzing the log file is done with a 4 cores, 8GB RAM machine. Also, possible design models that could be implemented in the system are being discussed.

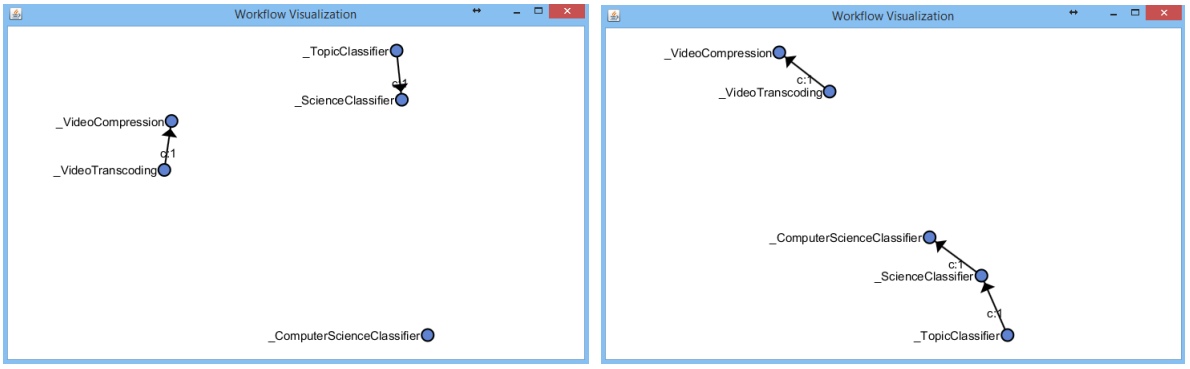
6.1 Performance Discussion

During the evaluation of this approach, was found that the most obvious drawback — false positives — actually causes relatively few problems in practice. The parameter that indroduced to this study, reflects the maximum time distortion between events timestamp. Thus, a put event was emitted from a machine at T_2 and received and logged at T_1 and because of that event a storlet in another machine was triggered at T_0. In time matter $T_0 > T_2$, but in fact it was actually the event that triggered the specific storlet. Hence, the system parameter has the capability to relate these two events in time (section 4.3.4). The user of the workflow service decides whether the results of using this parameter are “good enough”.

Using the log file from section 5.1.1 and proving the statement from the above paragraph, the T_2 is the timestamp from the result event of 'Science Classifier' handler and T_0 is the activation timestamp of 'Computer Science Classifier' handler. The events will originate from different machines and in different times.

```
2013-12-06 16:15:30.205, ACTIVATION, TopicClassifier.handler1,
    ContainerA , 192.168.1.2, TENANT_A, Obj1, '('"FileType"="'txt
    '"')'
2013-12-06 16:16:15.355, ACTIVATION, VideoTranscoding.handler1,
    ContainerA , 192.168.1.101, TENANT_A, Obj5, '('"Format"="AVI"')
    '
2013-12-06 16:17:15.509, PUT, TopicClassifier.handler1, ContainerA
    , 192.168.1.2,  Obj1, '('"Topic"="'Science'"')'
2013-12-06 16:17:35.775, PUT, VideoTranscoding.handler1, ContainerA
     , 192.168.1.101, Obj5, '('"Format"="MP4"')'
2013-12-06 16:17:50.256, ACTIVATION, ScienceClassifier.handler1,
    ContainerA , 192.168.1.55, TENANT_A, Obj1, '('"Topic"="'Science
    '"')'
2013-12-06 16:25:50.205, ACTIVATION, VideoCompression.handler1,
    ContainerA , 192.168.1.101,  TENANT_A, Obj5, '('"Format"="MP4
    "')'
2013-12-06 16:25:59.243, ACTIVATION, ComputerScienceClassifier.
    handler1, ContainerA , 192.168.1.2, TENANT_A, Obj1, '('"
    SpecialTopic"="'ComputerScience'"')'
2013-12-06 16:26:10.447, PUT,  ComputerScienceClassifier.handler1,
    ContainerA , 192.168.1.2,  Obj1, '('"MoreSpecialTopic"="'
    CloudComputing'"')'
2013-12-06 16:26:15.509, PUT, VideoCompression.handler1, ContainerA
     , 192.168.1.101, Obj5, '('"Codec"="H.264"')'
2013-12-06 16:40:40.653, PUT, ScienceClassifier.handler1,
    ContainerA , 192.168.1.55, Obj1, '('"SpecialTopic"="'
    ComputerScience'"')'
```

Observing the above log, the put event of 'Science Classifier' follows the activation event of 'Computer Science Classifier', but that was the reason why the 'Computer Science Classifier' storlet got activated, and for some reason the event has been delayed and logged at a later moment. Parsing the log file, by the use of the workflow engine, with no timeframe value will result in no connection between these two storlets but using a higher timeframe value there is a connection as it is illustrated in figure 6.1. After examing the nature of this parameter, it could be resulted that it helps to keep the erroneous results in low percentage. Additionally, this amount of time and algorithms performance are working inversely, because the engine uses it as a sliding window when parses the log file. In actual evaluation, the parameter was firstly set to 5 minutes and followed by 15 minutes and half an hour. In figure 6.2 that is illustrated below, the above assertion could be confirmed by the fact that by raising the value of the parameter the algorithm performance is getting relatively low by the increasing the execution times.



(a) timeframe=0 (b) timeframe=15

FIGURE 6.1: Graph visualization with and without timeframe use

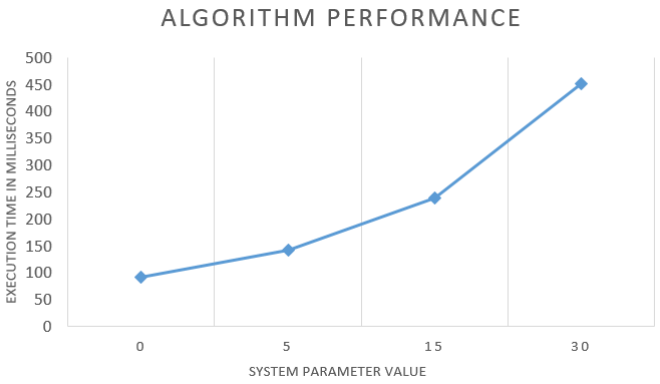


FIGURE 6.2: Algorithm execution times using a timeframe.

6.2 Workflow Service Design Models in VISION Cloud

The use of workflow service generally is expected to be an infrequent operation. Taking into account the lifecycle of a container, most of the requests will be at the setup phase of container when other operations are not invoked. On creation phase, when most changes are happening, an amount of storlets is added and the behavior of the container is being defined. At that point the model needs highly attention because it is not yet in the production mode, it is in the testing or set up phase. This model is used for debugging and understanding how things are connected. More particular, in the testing phase it observes the workflows, and decides if it proceeds in production. At the very beginning the amount of data which is collected is very small. Probably, there will be lots of operations invoking the model. The following design models are proposed and their advantages and disadvantages are being discussed.

6.2.1 Continuous design model

On this level of abstraction, small messages are sent frequently for continuous model, and considering SREs as the client that constantly keeps sending messages to workflow service existing as a server that loops building up the model.

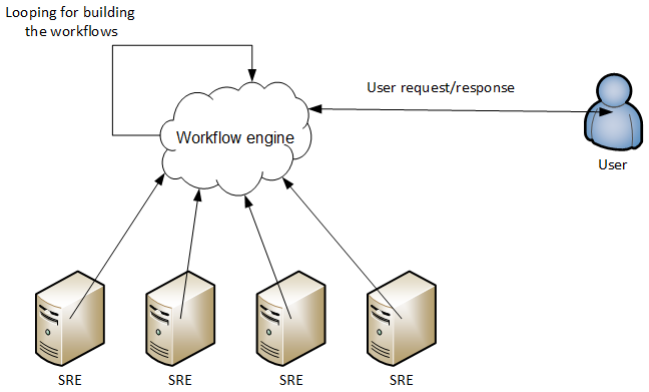


FIGURE 6.3: Continuous Model.

Due to the extra messages, the model receives and reforms things up that are already known and it can be discarded to save space and just keep the workflow model. The messages are short-lived, they are used immediately in order to update the model, they are written once and read once. Another aspect of the service, is the dynamically reaction to certain states of the system (e.g endless loops) and and its ability to eliminate these problems and revert to a healthy state of the system. Moreover, the information that is stored to the server will be useful for the user, as the model is already structured and a request from the user will have an immediate response and keep the latency in low levels. Hence, the process could be apprehended as an amortized job, because sending a message now and then, doesn't disturb the system and for that reason it's less bursty.

The disadvantages of this design are focused on the pointless fact of the existence of a large number of workflows, because in this continuous way will keep them all updated, even if the user asks for a single workflow. Also, the network traffic will be at high levels, because a message is constantly sent for every single event. According to the usage pattern this operation is more frequent at the beginning with a small amount of data and the burstiness is expected to be low. Whereas, later it will be very infrequent but the burstiness will be high.

6.2.2 On-demand design model

The indicators that are significantly important are the amount of data stored in the logs and what are those depending on. It is remarkable to mention that the basic requirement that has to be satisfied is that there will be two messages per request. The

bound cannot be more than twice the number of machines. The number of messages that will be sent, which is the load on the network, depends on two factors; the number of the machines and the size of the data that are going to be sent. Moreover, the processing power on the centralized machine, which builds up the workflow model and the impact on the network is two times the number of the machines involved and the amount of sent data. If the model server is a dedicated server then the processing part has not any impact on the system. Processing power can not be reduced for the user, the remainings are the messages that are sent as an impact.

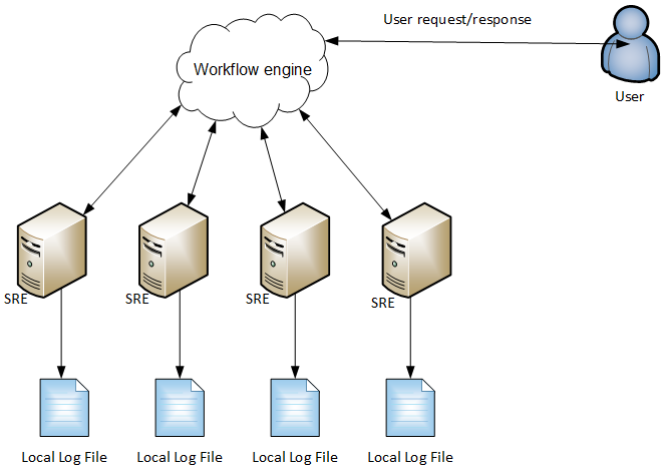


FIGURE 6.4: On-demand Model.

For the On-demand design model the user makes a request to the server and get a response. Because of the fact that big messages are sent on demand less frequently, things might get bursty. This burstiness will affect the user experience regardless the fact that the system will crash, that's another's care. When a user has a constant disturbance nothing is gained, but a high user disturbance at the very beginning and then never happen again is preferred. Generally speaking, the network traffic and the impact to the whole system has to be as low as possible. In order to build up the workflow model there is a need for every single event.

6.2.3 Optimized Continuous design model

For this design pattern there could be some logic at each SRE, either buffers and sends chunks to the workflow service or processes completely locally and then waits someone to ask it and might send a subset depending on the request. The user experience affection is the same because from the start it was choosed to have logic on each machine logging things and sending messages. In the continuous model there will be more network traffic, but the amount of storage will be low because log files will be stored locally. Also, a request will take longer because the workflow model needs first to built up. The user should not be upset, so it's acceptable to make an operation less

performant infrequently. It is more preferrable an operation to take place once a year and last half an hour to finish than having frequent 5-minutes operations.

This version is close enough to continuous. The SREs logs locally and for instance, once a day requests the log messages from the SREs and builds the workflows model out of it. The latency is almost the same with the continuous model and from the user's perspective it isn't necessary to build up the model first since the request can be served directly with a response. Since batch jobs are used, the model will be outdated. However, the user expects to have high latency to his response, because it is an infrequent operation.

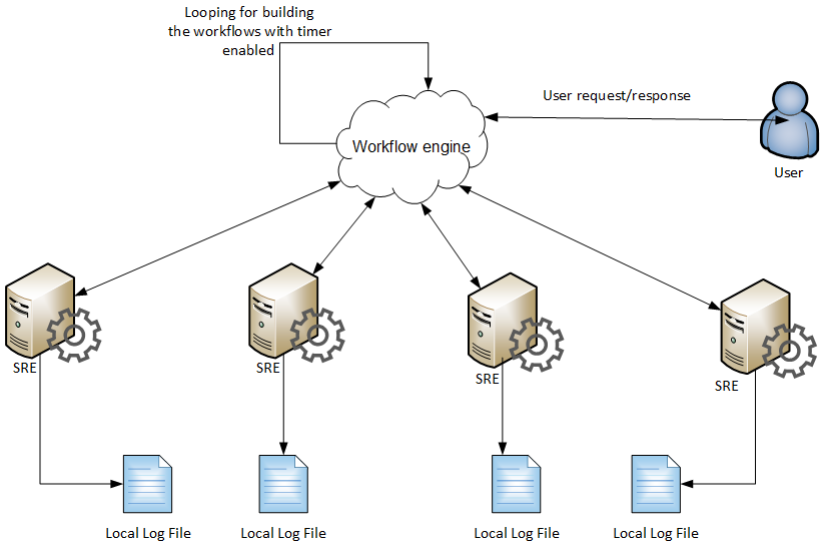


FIGURE 6.5: Optimised Continuous Model.

This model has almost all the advantages of the continuous. If no one asks for the workflow model, all the processing is wasted. What is gained by batching process is that there is a trade-off on how much batch together. The amount of messages that is sent can be dialed down by removing a bit message overhead of the system. The view from the network doesn't change that much, because the amount of data is the same but in a more bursty way. The size of these small peaks are depended on the size of the messages. The only cost is that the model is outdated. If is mandatory to have accurate worfkflow model the user should make GET operation to force update of the model. There is a dependency for the latency, which depends on the amount of data that are requested from the SREs; the more are requested, the more have to be processed and the more is the waiting time. The latency is getting increased when there are many requests.The messages travelling in the system (overhead) depends on the frequency of this update messages, therefore on the one hand, lower frequency leads to less message overhead and on the other hand when the messages are getting bigger means higher latency due to the larger process. In case the frequency is higher there will be more message overhead, whereas smaller messages signifies low latency. The whole amount of messages is pretty much the same as in the continuous model.

It is rather than complicated. If there is no care about the latency, that will result the lowest possible frequency of batching. There is trade with something that is not of high interest; the latency of the response.

6.2.4 Optimized On-demand design model

In general, there is a need to decrease the network traffic and to keep the impact of the workflow service on the system as low as possible. With previous design models, every single event will be sent to workflow service. The amount of data is always the same and likewise the network traffic but still it depends on the burstiness. The amount of data is sent in the first place has to be reduced and in order to achieved it the existance of some computation before send them back to workflow service is necessary. Every event is going to be processed and structure of the log that would be more efficient by store them in an intelligent way or/and make a partial model in the SREs.

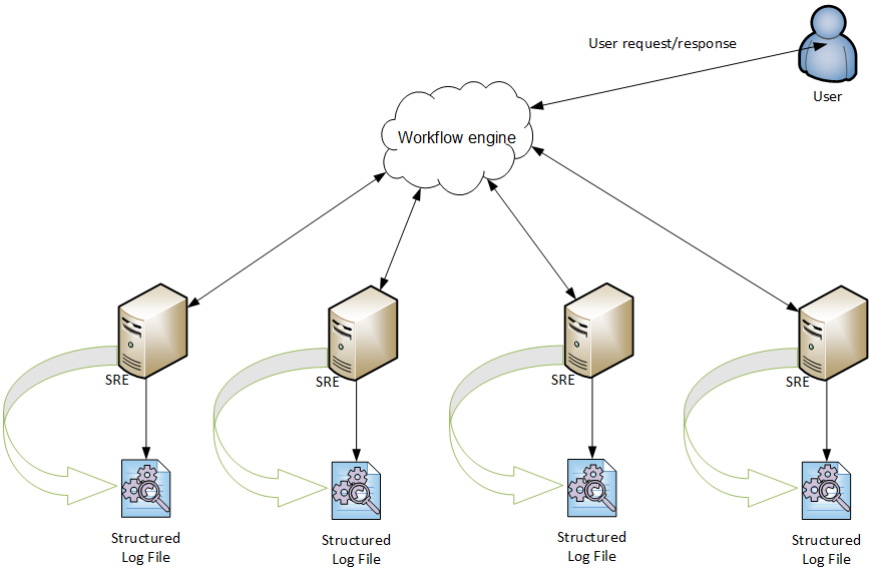


FIGURE 6.6: Optimised On-demand Model.

There are many containers running on the system but in case a user asks for the container 'A' there is no need the SRE to send information about other containers. Keep a log for every single container and keep them in a table to search for the them. The latency will be worst than the continuous model, since more work has to be completed, though it will be better than the complete on-demand because there will be already a pre-processing. Latency is acceptable and also the amount of messages as well, since there is no continuous message flow, but only when there is a need. A partial workflow is implemented, but an event could take place in any machine, hence the workflows have to be merged in order to have a global view, but it is not necessary to build them from the start. Both of them improve the latency and reduce the footprint on the system, especially the network traffic.

Chapter 7

Conclusion and Future Work

This paper contributes a complete approach that aids in understanding, debugging, and visualizing the behaviors of event-based applications. The approach consists of four parts: 1) capturing events of real, implemented systems, such as VISION Cloud, 2) classifying the captured events into semantically meaningful categories, 3) dynamically determining causal relationships between events using axioms generation and connection classification, and 4) visualizing events with respect to the system structure. The approach supports partial and incomplete behavior specification as well as black-box components.

This evaluation reveals that there are many benefits to using traces of real event-based systems. For example, if a component is not responding as expected, it is easy to verify whether it received a particular expected event, or if the event was emitted at all. The availability of causal chains makes it easier to solve a much wider range of problems. With chains, one can elide away irrelevant events and remain focused on a particular error. It also becomes easier to determine which components are actually using the services of other components, or how other components respond to a particular component's events.

Among the VISION users from different fields of industry, the requirement of workflow construction is non-trivial due to the tendency of facing more complex and general use case requirements. A well-designed storlet workflow can be easily created with the assistance of this service. Firstly, users create a preliminary version of storlets, and visualize the connections between them with this view. During runtime if one notices any connection types that are unexpected, the user can fix them easily. Apart from workflow construction, this service also supports workflow validation. When a workflow is newly created as mentioned above, it is necessary to check whether there will be conflicts or risks between the new workflow and existing storlets. The view demonstrates the total amount of storlets and the connections in the container among with the new workflow.

Finally, this heuristic approach allows a user to begin with little or no understanding of a system. This is especially useful when trying to understand a new system, or a different part of the system in which a component did not previously interact. The beneficial properties of event-based architectures combined with an increasing support from practitioners and researchers, could possibly result in the creation of more and more event-based systems. However, lack of end-to-end development and of maintenance support for this kind of architectures could hinder adoption and raise the costs of building event-based systems.

This approach contributes in a usable and viable way to understanding event-based systems, but it also exposes several important issues in event-based development that could be investigated. This includes the use of event tracing as a basis for testing and debugging, the role of heuristic techniques to find "good enough" answers to development problems and the finding of novel ways to deal with the deluge of events that occur in even moderate-sized event-based systems. Future research is needed in order to focus on applying this approach to those types of event-based systems and evaluate its feasibility and effectiveness.

Bibliography

[1] Yagil Engel. Towards proactive event-driven computing. *Proceedings of the 5th ACM international*, pages 125–136, 2011. doi: 10.1145/2002259.2002279. URL `http://portal.acm.org/citation.cfm?id=2002279`.

[2] Norman W. Paton and Oscar Díaz. Active database systems, 1999. ISSN 03600300.

[3] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system, 1989. ISSN 01635808.

[4] A. Goh, Y.-K. Koh, and D.S. Domazet. ECA rule-based support for workflows. *Artificial Intelligence in Engineering*, 15(1):37–46, January 2001. ISSN 09541810. doi: 10.1016/S0954-1810(00)00028-5. URL `http://www.sciencedirect.com/science/article/pii/S0954181000000285`.

[5] Coloured petri nets, 2014. URL `http://cs.au.dk/CPnets/`.

[6] W. M. P. van der Aalst. Formalization and verification of event-driven process chains. *Computing Science Reports 98/01*, 1998.

[7] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1083323.1083349`.

[8] Lustre. URL `http://wiki.lustre.org/index.php/Main_Page`.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945450. URL `http://doi.acm.org/10.1145/1165389.945450`.

[10] Welcome to Apache™ Hadoop®! URL `http://hadoop.apache.org/`.

[11] . Iii Ligon, W. B. and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 471–. IEEE Computer Society, 1996. ISBN 0-8186-7582-9. URL `http://dl.acm.org/citation.cfm?id=525592.823101`.

[12] AWS — Amazon Simple Storage Service (S3) - Online Cloud Storage for Data & Files. URL http://aws.amazon.com/s3/.

[13] Google Cloud Storage - Cloud Backup, Cloud Database & Online Storage — Google Cloud Platform. URL https://cloud.google.com/products/cloud-storage/.

[14] Azure: Microsoft's Cloud Platform — Cloud Hosting — Cloud Services. URL http://azure.microsoft.com/en-us/.

[15] Atmos - Cloud Storage, Big Data - EMC. URL http://www.emc.com/storage/atmos/atmos.htm.

[16] David J. DeWitt and Jim Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Rec.*, 19(4):104–112, December 1990. ISSN 0163-5808. doi: 10.1145/122058.122071. URL http://doi.acm.org/10.1145/122058.122071.

[17] Survey distributed databases - Toad for Cloud - Toad for Cloud Databases - Toad World. URL http://www.toadworld.com/products/toad-for-cloud-databases/w/wiki/308.survey-distributed-databases.aspx.

[18] NoSQL. URL http://en.wikipedia.org/wiki/NoSQL.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267308.1267323.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294281. URL http://doi.acm.org/10.1145/1294261.1294281.

[21] The Apache Cassandra Project. URL http://cassandra.apache.org/.

[22] Avinash Lakshman and Prashant Malik. Cassandra: Structured storage system on a p2p network. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-396-9. doi: 10.1145/1582716.1582722. URL http://doi.acm.org/10.1145/1582716.1582722.

[23] VISION Cloud - EU research project. URL http://www.visioncloud.eu/.

[24] Computational Storage in Vision Cloud. URL http://ercim-news.ercim.eu/en89/special/computational-storage-in-vision-cloud.

[25] S.M. Wheater F. Ranno, S.K. Shrivastava. A System for Specifying and Coordinating the Execution of Reliable Distributed Applications. URL http://www.cs.ncl.ac.uk/research/trs/papers/644.pdf.

[26] Gregor Joeris and Otthein Herzog. Towards flexible and high-level modeling and enacting of processes. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering*, CAiSE '99, pages 88–102, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66157-3. URL http://dl.acm.org/citation.cfm?id=646087.679895.

[27] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Tim Carver, Matthew R. Pocock, and Anil Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:2004, 2004.

[28] Ivona Brandic, Sabri Pllana, and Siegfried Benkner. Specification, planning, and execution of qos-aware grid workflows within the amadeus environment. *Concurrency and Computation: Practice and Experience*, 20(4):331–345, 2008. ISSN 1532-0634. doi: 10.1002/cpe.1215. URL http://dx.doi.org/10.1002/cpe.1215.

[29] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto, Jr., and Hong-Linh Truong. Askalon: A tool set for cluster and grid computing: Research articles. *Concurr. Comput. : Pract. Exper.*, pages 143–169. URL http://dx.doi.org/10.1002/cpe.v17:2/4.

[30] Vladimir Korkhov, Dmitry Vasyunin, Adianto Wibisono, Adam S. Z. Belloum, Márcia A. Inda, Marco Roos, Timo M. Breit, and L. O. Hertzberger. Vlam-g: Interactive data driven workflow engine for grid-enabled resources. pages 173–188, 2007. URL http://dl.acm.org/citation.cfm?id=1377543.1377547.

[31] Yang Zhang, Charles Koelbel, and Keith Cooper. Hybrid re-scheduling mechanisms for workflow applications on multi-cluster grid. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 116–123, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3622-4. URL http://dx.doi.org/10.1109/CCGRID.2009.60.

[32] David C Luckham. Rapide : A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events 1 Introduction 2 Interface Connection Architectures. 1996.

[33] David C Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent,

timed systems. *Journal of Systems and Software*, 21(3):253–265, June 1993. ISSN 01641212. doi: 10.1016/0164-1212(93)90027-U. URL http://dl.acm.org/citation.cfm?id=154327.154332.

[34] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule Representation, Interchange and Reasoning on the Web*, volume 5321 of *Lecture Notes in Computer Science*, pages 3–3. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-88807-9. doi: 10.1007/978-3-540-88808-6_2. URL http://dx.doi.org/10.1007/978-3-540-88808-6_2.

[35] Brian Frasca David C. Luckham. Complex event processing in distributed systems. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.876.

[36] CiteULike: Java PathFinder - Second Generation of a Java Model Checker. URL http://www.citeulike.org/user/bunge/article/1567341.

[37] et al. C. Demartini. Modeling and Validation of Java Multithreading Applications Using Spin. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.4703.

[38] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, and C.S. Pasareanu. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pages 439–448. ACM, 2000. ISBN 1-58113-206-9. doi: 10.1109/ICSE.2000.870434. URL http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=870434.

[39] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7: 149–174, 1994. ISSN 01782770. doi: 10.1007/BF02277859.

[40] ANTLR (ANother Tool for Language Recognition). URL http://www.antlr.org/index.html.

[41] *Encyclopedia of Mathematics*. Springer, 2001. ISBN 978-1-55608-010-4. URL http://www.encyclopediaofmath.org/index.php?title=p/c025090.

[42] *Encyclopedia of Mathematics*. Springer, 2001. ISBN 978-1-55608-010-4. URL http://www.encyclopediaofmath.org/index.php?title=p/d033300.

[43] An Overview of Automated Theorem Proving. URL http://www.cs.miami.edu/~tptp/OverviewOfATP.html.

[44] Deterministic finite automaton. URL http://en.wikipedia.org/wiki/Deterministic_finite_automaton.

[45] Orbital library. URL http://symbolaris.com/orbital/.

[46] A.V. Aho and J.D. Ullman. *Foundations of Computer Science: C Edition*. Principles of Computer Science Series. W. H. Freeman, 1995. ISBN 9780716782841. URL http://books.google.gr/books?id=q7-HQgAACAAJ.

[47] dk.brics.automaton - finite-state automata and regular expressions for Java. URL http://www.brics.dk/automaton/index.html.

[48] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. ISSN 00010782. doi: 10.1145/359545.359563. URL http://portal.acm.org/citation.cfm?doid=359545.359563.

[49] Data transfer object. URL http://en.wikipedia.org/wiki/Data_transfer_object.

[50] Antoine Dutot and F Guinand. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. *... Complex Systems. ...*, 2007. URL http://hal.archives-ouvertes.fr/hal-00264043/.

[51] GraphStream - A Dynamic Graph Library. URL http://graphstream-project.org/.