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

ZIWEI CHEN

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:187
Workflow Management Service based on an Event-driven Computational Cloud Storage

An Analysis and Prediction of the Process Activities

ZIWEI CHEN

Master’s Thesis at SICS
Supervisor: Per Brand
Examiner: Johan Montelius

TRITA-ICT-EX-2013:187
Abstract

The Event-driven computing paradigm, also known as trigger computing, is widely used in computer technology. Computer systems, such as database systems, introduce trigger mechanism to reduce repetitive human intervention. With the growing complexity of industrial use case requirements, independent and isolated triggers cannot fulfil the demands any more. Fortunately, an independent trigger can be triggered by the result produced by other triggered actions, and that enables the modelling of complex use cases, where 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.

As the developing platform of this study, VISION Cloud is 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. As a result, one storlet can also be triggered by the result produced by other storlets, and it is called connections between storlets. Due to the growing complexity of use case requirements, an urgent demand is to have storlet workflow management supported in VISION system. Furthermore, because of the existence of connections between storlets in VISION, problems as non-termination triggering and unexpected overwriting appear as side-effects.

This study develops a management service that consists of an analysis engine and a multi-level visualization interface. The analysis engine checks the connections between storlets by utilizing the technology of automatic theorem proving and deterministic finite automaton. The involved storlets and their connections are displayed in graphs via the multi-level visualization interface. Furthermore, the aforementioned connection problems are detected with graph theory algorithms.

Finally, experimental results with practical use case examples demonstrate the correctness and comprehensiveness of the service. Algorithm performance and possible optimization are also discussed. They lead the way for future work to create a portable framework of event-driven workflow management services.
Acknowledgement

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Per Brand and my co-supervisor Martin Neumann, for their patience, enthusiasm, and immense knowledge. Their guidance helped me in the whole process of researching and writing of this thesis. I would also like to thank my program coordinator and at the same time my examiner Johan Montelius for giving me the chance to participate in this program and guide me during the 2 years study. Finally, I would like to thank all my friends in this program for their support and encouragement.
3.2 Workflow Validation .................................................. 17  
3.2.1 Incorrect Connection Type ..................................... 18  
3.2.2 Non-termination ................................................... 18  
3.2.3 Unexpected Overwriting ........................................ 19  
3.3 Approaches ............................................................ 19  

4 Design and Implementation ............................................ 21  
4.1 Connection Classification .......................................... 21  
4.1.1 Motivation ......................................................... 21  
4.1.2 Complete Trigger ................................................ 21  
4.1.3 Partial Trigger .................................................... 22  
4.1.4 Unrelated .......................................................... 22  
4.1.5 Logical Consequences .......................................... 23  
4.2 Classification Methodology ......................................... 23  
4.2.1 Related Technologies ............................................ 23  
4.3 Classification and Analysis Algorithms .......................... 26  
4.3.1 Classification Algorithm ........................................ 26  
4.3.2 Axioms Generation Algorithm ................................. 28  
4.3.3 Value Matching Algorithms .................................... 29  
4.4 Visualization and Risk Detection ................................. 31  
4.4.1 Graph Modelling Language and Tools ........................ 32  
4.4.2 Multi-level View Design ....................................... 33  

5 Results and Analysis ................................................... 37  
5.1 Workflow Construction .............................................. 37  
5.1.1 Use Case Example ................................................ 37  
5.1.2 Abstract View for Connection Type Checking .............. 38  
5.1.3 Comprehensive View for Parameter Analysis ............... 39  
5.2 Advanced Workflow Validation ................................... 39  
5.2.1 Non-termination Risk ........................................... 41  
5.2.2 Overwriting Risk ................................................ 41  

6 Algorithm Performance Discussion ................................ 43  
6.1 Storlets Traversal .................................................... 43  
6.2 Classification ......................................................... 43  
6.3 Axiom Generation ..................................................... 44  

7 Conclusion and Future Work ......................................... 45  

Bibliography ............................................................... 47
Chapter 1

Introduction

VISION Cloud is a storage system that supports event-driven computing. Computations of VISION can be chained as workflows at runtime. This chapter first introduces the ECA rule as the essence of event-driven computing. Then it states the common problems of ECA-rule based applications like the trigger mechanism in databases, then analyses the specific trigger-related problems in VISION Cloud. Next, the related work is discussed and followed by a possible approach.

1.1 Event-Condition-Action Rule

1.1.1 Introduction

Event-driven computing [15] systems are typically based on Event-Condition-Action rules (ECA)[18].

ECA rules consist of events, conditions and actions[9]. The event part specifies the signal that triggers the invocation, the condition part which is a logical test to determine whether the action is to be carried out, and finally the action part 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.

To summarize, the actions of the rules are invoked by the occurrence of specific events under certain conditions.

1.1.2 ECA Rule-based Workflow Visualization Model

The result of actions can match other rules and so on trigger a graph of activities. As an important feature, ECA rule-based systems support workflow construction[19]. 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.
The two common visualization model of ECA rule-based workflows are coloured petri net (CPN) model[1] and event-driven process chain (EPC) model[31].

**Coloured Petri Net (CPN)**

Standard petri nets consist of elements such as places, transitions and arcs. Places in a petri net may contain a number of dots known as *tokens*, and they represent the current state of the petri net. Normally, a transition need a certain number of tokens to be triggered. Coloured Petri Net is an extended version of the ordinary petri net. CPNs got the name because they allow the tokens to be coloured and distinguished from each other. And today, coloured tokens are represented by typed tokens for convenience. The introduction of typed tokens exactly solve the problem of distinguishing different events.[22][20] In a CPN, places stand for the rule holders. Typed tokens represent the events together with conditions of ECA rules, while transitions denote actions. For example in figure 1.1, if a single process follows two different rules, disparate types of tokens A and B can represent the events lead the process to different transitions.

**Event-driven Process Chain (EPC)**

The three primary types of nodes in EPC are activities, events and connectors. Normally, they are connected by control flows. Control flows indicate how rules are triggered with a sensible process logic. They can be executed in sequential or alternatively in parallel.

In order to model ECA rules, event nodes are extended to represent both events and conditions. Function node is actually the action. Additionally, actions can produce result events that matches the event and condition of other rules. Figure 1.2 displays an example of ECA rule modelled by EPC.

### 1.1.3 Applications

**Active Database Management Service**

A well known application of ECA rule-based architecture is the trigger mechanism of Active Database Management Service (ADBMS)[30]. Triggers are usually used for maintaining the integrity of the information on the database. For instance, the trigger feature of MySQL database is used for intelligent data processing. Through the use of MySQL triggers, the system automatically performs checks of values or calculations when a particular event occurs on the specified table.

**Distributed NoSQL Data Store Management Service**

In the past few years, based on the ordinary trigger of traditional active databases, trigger mechanisms are also adapted to NoSQL data stores. Similar to the triggers of conventional databases, the Observers of HBase[12] are a procedure that users
can insert code by overriding upcall methods, the callback functions are executed from core HBase code when certain events occur.

1.2 Computational Storage in VISION Cloud

1.2.1 Introduction

VISION Cloud[8] is developed as a cloud storage system which joined together with an event-driven computational infrastructure. To meet the requirement of data intensive services, the storage and computational infrastructures are tightly coordinated to provide high scalability, tunable availability and means of massive data processing[2].
1.2.2 Storage Scope

Different from the global access in NoSQL data store, a container is the unit for placement in VISION Cloud. A user can store the same objects to multiple containers, but he/she cannot access to the container for which he/she is not authorized. It is noteworthy that containers not only separate the objects based on the owner (user or groups) but also form the boundary of trigger computation as because triggers can only be invoked by the events occurring in the same container.

1.2.3 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 unit 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, then it becomes an active process to execute the specified action on behalf of the user. When the execution finished, it becomes passive again.
1.3. PROBLEM STATEMENT

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, 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.

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 actually the result events produced by 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 the deletion of 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, usually rules are 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 the maintenance, triggers in a ordinary database is hardly ever changed unless new tables or views are created. Whereas new rules in VISION are added with the creation of new storlet more frequently. Moreover, when a storlet is running within one 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 storlet will be explained in section 2.3.3.

1.3 Problem Statement

1.3.1 General Problems of Trigger Mechanism

Since the appearance of trigger mechanism in active databases, problems such as inappropriate trigger design and non-terminate executions have been exposed to database engineers. Those problems are typical in ECA-rule based systems[10].

Although a DBMS gives exceptions for the occurrences of invalid execution, it is not easy for database engineers to detect an inappropriate trigger if it is not obvious. For example, a inappropriate trigger could accidentally update a wrong dataset or never get triggered without any exceptions by the system. On the other hand, because poor designed triggers can trigger each other indefinitely, if there is a cycle among the trigger interactions, it leads to the non-terminate execution[23].
1.3.2 Specific Problems for VISION Cloud

Based on the different event-driven mechanism, storlet, we have discussed in section 1.2.3. VISION users face more specific problems other than the general ones discussed above.

Firstly, VISION Cloud shares the same problems with common event-driven computing systems, but the problems become more serious in VISION. Because of the simplicity of storlets creation, it allows that the users who are empowered for storlet creation to be non-programmers. Although users only need to set a few parameters when creating storlets and storlet templates are assumed correct, it is not friendly to expect non-programmer users to be as good at detecting mistake parameter as database engineers.

Secondly, in traditional active database systems, triggers are managed by a centric DBMS. If something unusual happens, the engineer can debug the issue from trigger logs. Since VISION Cloud is a distributed architecture, logs are not guaranteed to be in total order. Hence it is not straightforward to trace the issue by logs.

Lastly, users tend to create more complex workflows in VISION. Nevertheless, it is difficult to predict the interactions between storlets, and users are not aware of the existing storlets created by other users. To design and check a collection of correlated storlets is obviously even more complicated.

1.4 Problem Scope

All problems mentioned above are significant to VISION Cloud. One can group the problems into static and dynamic problems. The static problems are basically related to the design and creation of a single or a set of new storlets for an existing environment completed with existing storlets. The possible new connections and problems such as cycles should be predicted before the new storlets are injected. The dynamic ones are about the problems of storlets that already happened at runtime, such as the existing non-termination storlet triggering, and unexpected activation of some storlets. The dynamic problems can be only detected after occurrence.

This study is focused on the static problems. It is always good to find out potential issues before execution like a compiler. Moreover, it is extremely difficult to build workflows without analyzing the interactions. Solving the static problems allows user to gather a number of storlets and check their relations, also it reveals the potential issues of existing storlets. Although the dynamic problems are also interesting, due to the time constraint, they are excluded from the study.

1.5 Related Work

Previous work has provided some solutions to the general problems of active database systems[16][10][21][25].

Adela[16] is a visualization tool suitable for the relational database model, it
1.6. POSSIBLE APPROACH

introduces multiple rule concept to capture different aspects of rule behaviours. An event/rule tree model is also provided to facilitate tracing the context in which rules are fired and the execution state. One restriction of this tool is the scalability of large rule sets or data models.

A visualization and explanation tool is developed in [10], it supports visualization of rules as post-execution analysis. Additionally, the tool also displays the context in which the rule is fired. Runtime visualization is not yet supported when the work is published.

PEARD[21] is a debugging environment for rules of active databases. The environment consists of a debugging tool and a visualization tool. The breakpoint debugging tool allows the state of variables to be changed anytime during the rule execution. The rule visualization tool displays the rule triggering process in graph form.

In [25], a coloured petri net based ECA rule analyser is designed to detect non-termination problems in active distributed databases. The analyser supports composite events and dynamic analysis at runtime.

Nevertheless, the previews solutions are not adequate to VISION Cloud, due to the triggering scope and use case requirements are different. Firstly, all the related solutions are designed for active database triggers. Data is stored and accessed globally in most relational and NoSQL databases, unlike data objects are scoped by containers. More importantly, none of the solutions support workflow management (e.g. augmentation, modification, validation), which is another primary motivation of this study.

### 1.6 Possible Approach

This study intends to build a service consists of a multi-level visualization interface and an connection analysis engine. The visualization interface not only gives a panoramic view of the graph formed with all the storlets and connections between them, but also supports a comprehensive view of conditions and connections for the purpose of issue tracing. The Analysis engine works by matching any two storlets and determines the relation between them. The result is used to create graphs or views in the visualization interface.
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 infrastructure for reliable and effective delivery of data-intensive services[29]. This goal is primarily achieved by two innovations: 1) A new data model of storage based on associating objects with metadata specified in a rich and flexible schema; 2) Event-driven computational storage. The rich metadata schema greatly facilitate data query and operation, intelligent computation mechanism automates the repetitive data operations on behalf of users. These two features enable efficient data access and reduce data transfer that greatly benefits the data-intensive services.

2.2 Data Model of Storage

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
A 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 and 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:

- Creating an object with associated metadata
- Replacing an existing object with new data and metadata
- Reading an object’s data
- Reading an object’s metadata
- Setting an object’s metadata
- Deleting an object

### 2.3 Computational Storage

#### 2.3.1 Storlet

**Introduction**

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.

**Lifecycle**

As shown in 2.1. Firstly, a storlet template will be selected from the library, and the user is required to set the parameters such as trigger condition and credentials. Then, the complete storlet will be injected to VISION. Initially, the storlet enters the passive state until it is triggered by the specified event under certain conditions. Next, the storlet becomes activated when got triggered and executes the pre-defined operations till finished, following by turning back to the passive state. An activated storlet can perform various computations, such as data object creation, modification and deletion. Also the result events produced by the computations may activate other storlets. When a storlet is not need any more, the user can delete it as a data object at any time.
2.3.2 Storlet Template

Normally, users create storlets through templates rather than start from scratch. Storlet templates are provided by tenants and third parties for different purposes, including compressing/decompressing, transcoding or classifying files. Those functionalities are predefined and hard-coded in the template. Based on the template, users are only required to set several parameters, such as trigger conditions as desired, to create a complete storlet. Filling storlet parameters is as simple as setting configurations with well known variables in the environment.

2.3.3 Programming Environment

As the event-driven computing mechanism in VISION Cloud, storlet programming model follows the ECA rule structure. In the programming model, a rule is held by a single trigger handler of storlet, and a storlet is capable to have multiple handlers. Similar to the typical ECA format, the rule of storlet contains an event part, condition part and action part. As a result, events and conditions cannot be clearly separated in storlets. They are discussed compositely as "trigger event" below.

The trigger event, also known as trigger evaluators, of a storlet is limited to the metadata change on a single data object within the same container. Typically, trigger events are presented by logical expressions, a unary expression presents one of the four statements, "appearance", "disappearance", "presence" and "absence", and contains a key-value string pair constraint, which will be explained in 2.3.3. 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'.

In terms of actions, they can be arbitrary computations on data objects, such as reading or writing on objects and updating metadata. Certainly, the operations are restricted by the authorizations on data objects and metadata. The target objects can be either the event object where the event comes from or different objects. Actions can be even executed on accessible objects in different containers. More importantly, there is a result event expression that makes a prediction of the metadata change when action finished. This expression follows the same format as trigger evaluator, with a flag (i.e. [same], [diff]) identifies whether the change happens on the event object or different objects. Normally the result is automatically generated by the programming environment, users can decide to expose or hide a part of the result. 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). But results that associated with "presence" and "absence" are optional to be displayed in the
result event. It is noteworthy that, the action result of a storlet can vary and highly depends on the data object(s) it operates on.

**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 Example of Storlet Creation

A researcher uses VISION Cloud to manage literatures, he/she would like to filter out those papers about cloud computing, then ensure all the text about cloud computing are converted to PDF format. Hence, he/she creates a storlet called PDF Converter. The trigger condition is designed as Appearance(filetype, ="text") && Appearance(title, ~".*Cloud.*") && Absence(format, ="pdf"), which means when a new text file appears in the container, and the title attribute stored in metadata contains the keyword "Cloud", also if the file is not in PDF format, the storlet will be triggered. Lastly, the action part is simply a PDF converting execution, and the action result will be generated as presence(filetype, ="text") && Presence(title, ".*Cloud.*") && Appearance(format, ="pdf")). The result expression denotes that the value of "filetype" attribute is still "text", "title" keeps containing the keyword "Cloud", but the "format" attribute becomes "pdf".
2.4 Execution Environment

Storlets are running in the environment as shown in figure 2.2 and mainly interact with three different components, which are Object Service, Notification Service and Storlet Runtime Environment. Conceptually, a single container has only one Notification Service and one Storlet Runtime Environment.

Object Service (OS)

The Object Service manages the data objects based on content. In the storlet execution environment, it collects and sends all the data object events that occurred in the container to the Notification Service.

Notification Service (NS)

When storlets are injected to VISION Cloud, they are immediately registered to the NS in the same container. Other than keeping the registration of storlets, a NS delivers event messages to the storlets who are interested in. There is only one NS per container that processes events spawned by data objects within the container.
If any registrations match an event, a trigger is sent to the registered storlets. Moreover, the service also sends notifications to users under specified conditions.

**Storlet Runtime Environment (SRE)**

SRE provides the runtime for storlet to execute and supports multiple storlets activated simultaneously. It receives the storlet objects from Object Service and execute the handler of the storlet when the trigger comes from the Notification Service. Additionally, all the interactions between VISION Cloud and storlets go through the SRE interfaces.
Chapter 3

Problems and Solution Approaches

In section 1.3, the general ECA-rule based workflow problems and specific storlet workflow problems of VISION Cloud are introduced abstractly. This chapter gives a comprehensive formulation to the intentions of storlet workflow construction and related problems. Later on, the solution approaches are discussed.

3.1 Workflow Construction Intentions

Generally, there are two main intentions when users create storlet workflows. Both intentions share the same demand to start with creating a set of storlets that connect to each other and form a workflow. But the difference shows up after the injection, one intention is willing to connect the new storlet workflow to existing workflows or isolated storlets, while the other intention would like to keep the new workflow independent from existing storlets. Particularly, injecting a single storlet is a special case of the two intentions.

3.2 Workflow Validation

It is never an easy work to design a storlet workflow based on the original use case requirement. According to the description of use case requirement, a user can intuitively get to know the essential functionalities, and then choose the appropriate storlet templates from the library. A tricky step is to properly set the trigger condition and result event of each storlet. Without an analysis tool, even if all the storlets seem to be fine that considered individually, users are unaware that whether the connection type between storlets are as expected. For example, if two storlets are designed to execute in sequential but accidentally executed in parallel, that could cause serious side effect. Moreover, cycles in the workflow graph might produce endless triggering loop, 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, which are incorrect connection type, non-termination and unexpected overwriting.

3.2.1 Incorrect Connection Type

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 the new storlet workflow connects to any existing storlets. Unfortunately, it does connects with some existing ones. Then this problem is regarded as an incorrect connection type problem.

3.2.2 Non-termination

Similar to the situation with database triggers, non-termination problem also exists among storlets. However, as mentioned in section 2.3.3, a storlet may contain multiple trigger handlers. Each handler carries a single ECA rule. A non-termination problem appears only if a cycle of handlers is formed. For example, in figure 3.1a, three handlers form a cycle that implies a non-termination problem. In contrast, in figure 3.1b, even though the storlets are connected as a cycle, there are two handlers from the same storlet are not connected so that leaves a gap in the cycle. Therefore, there is no non-termination problem in figure 3.1b. Additionally, the non-termination can happen on a single storlet when its result event triggers the storlet handler itself.
3.2.3 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, because of the last write wins policy, the operations which finishes early would always be overwritten by the latest one. As a result, the former ones become useless in this case.

3.3 Approaches

For the purpose of intuition, visualization is a common method applied in most rule debugging and management tools[11][10]. However, the most suitable content to display in the view varies due to the aims of the tool. For example, if a user need to extend an existing workflow, he/she would be more interested in the whole picture/graph of that workflow and possible entries be able to extend. On the other hand, if a user is about to trace a problem of storlet workflow, it would be the best to show the logical consequences between storlets connections more comprehensively.

In order to construct a graph or analysis the interactions between storlets, a storlet connection analyser is required. Connections are determined by the logical consequences between storlet trigger condition and result event of each handler. Because that the tool sometimes needs to analyse both the newly created storlet workflow together with existing storlets, it should accept custom input from the user interface and fetch storlet information from the system container.
Chapter 4

Design and Implementation

This chapter explains the solutions based on the approaches discussed in last chapter. As discussed before, storlets can be connected implicitly. Connections between storlets are classified into three different types: complete trigger, partial trigger and unrelated. Classification related algorithms are explained step by step afterwards. Moreover, visualization model and risk detection algorithms are discussed in the end. 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 Connection Classification

4.1.1 Motivation

If there is a connection between two storlets, the action result of one storlet should match the trigger condition of the other storlet. Since computation output of a storlet varies, and sometimes even no output due to execution failure, no event produced by result is guaranteed to meet the condition of other storlets. As a result, the connections between storlets are classified into three types mentioned above, which are distinguished by the strength of connections.

4.1.2 Complete Trigger

A complete trigger denotes that there is 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 is 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 is a result produced by correct execution of the source storlet, it always triggers 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 connects 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 overwrited, thereby the target storlet cannot be triggered anymore.

4.1.3 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. This type of connection is distinguished from unrelated (next subsection), because some of the partial triggers are occasionally created by inappropriate storlet design. A user can easily modify the storlet parameters to upgrade a partial trigger to complete trigger or eliminate the unexpected possible connections between storlets.

4.1.4 Unrelated

Unrelated means that there is no connection between two storlets. There are two cases that the result event of one storlet does not contribute to the trigger condition of another. The first case is that there are no key-value string format constraints in both result event and trigger condition that share the same key. For example, the result event of one storlet is "[same]Appearance(key1, = A)" , and the trigger condition of the other storlet is "Appearance(key2, = B)" . In the two expressions, the keys are different, thus the result event is unrelated to the trigger condition. In the second case, both the result event and the trigger condition share the same key, however the associated values are mutual exclusive. For example, the result expression "[same]Appearance(key, > 5)" is mutual exclusive to the trigger condition "Appearance(key, < 4)" .
4.1.5 Logical Consequences

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 or event unrelated. To formulate the relations between different connection types with logical consequences, it is presented as "complete trigger" implies "partial trigger", and "partial trigger" implies unrelated.

4.2 Classification Methodology

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, currently all registration information of storlets is kept in the Notification Service of VISION Cloud. Fortunately, NS provides APIs for directly fetching all the storlet information within the container.

After all the trigger condition expressions and result event expressions are collected, 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 4.3.1.

4.2.1 Related Technologies

This subsection introduces the technologies that applied in the classification related algorithms. 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. Automated Theorem Proving is used to figure out the implications between expressions. Lastly the Deterministic Finite Automaton is a technology for modelling regular expressions, which reveals the relations to regular expression form constraints.

Conjunctive and Disjunctive Normal Form

In boolean logic, a conjunctive normal form[26] 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)[27], 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

\[ \bigwedge_{i=1}^{n} \left( \bigvee_{j=1}^{m_i} C_{ij} \right) \]  

(4.1)
where each $C_{ij}, i = 1, ..., n; j = 1, ..., m_i$, 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 formulas, 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 normalization of a logical formula which is a disjunction of conjunctive clauses. The formula of DNF is displayed as (4.2).

$$\bigvee_{i=1}^{n} (\bigwedge_{j=1}^{m_i} C_{ij})$$  \hspace{1cm} (4.2)

**Automated Theorem Proving**

Automated theorem proving (ATP) [7], 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 Rubik cube can be the conjunction, all possible changes are treated as axioms, ATP system can prove that the cube can be rearranged into solution state. The language of conjunction, 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 conjunction 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 Rubik cube example provides a solution to the rearrangement problem.

Among various of ATP systems and libraries, Orbital Library[6] 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 Automaton**

A deterministic finite automaton (DFA) [3], also known as deterministic finite state machine, is a state machine that accepts or rejects finite strings of symbols and only produces a unique computation of the automaton for each single input.

Typically, a DFA is a tuple consist of 5 elements, they are

- a finite set of states $Q$;
4.2. CLASSIFICATION METHODOLOGY

25

(a) DFA Example

(b) NFA Example

Figure 4.1: Finite Automaton Examples

• a finite set of alphabet ($\Sigma$);
• a transition function ($\delta : Q \times \Sigma \rightarrow Q$);
• a start state ($q_0 \in Q$);
• a set of accept state ($F \subseteq Q$)

The example in figure 4.1a illustrates a DFA that accepts only binaries, and the DFA terminates at state 1 when "0" appears even number of times in the input. DFAs recognize exactly the set of regular language, and is a method for lexical analysis and pattern matching. The DFA in figure 4.1a can be given by regular expression $1^* (0(1^*)0(1^*))^*$. Other than DFA, if a given input produces a transition targets to multiple possible state, then it is a nondeterministic finite automaton (NFA) [5]. For example, 4.1b shown a NFA in which the state $S_2$ has alternative transition to both state $S_1$ and $S_{22}$ when receives "1" as input. However, a NFA can be always translated to equivalent DFA by specific algorithms.

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 4.2 models the regular expression $[ch]at$. From state 0, the automaton has two options, either chooses "c" or "h", and in the next two states there is only one path that is "at". As a result, the possible strings which match the given regular expression are "cat" and "hat".

In respect of the implementation, the java library dk.bricks.automaton [4] 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 4.3.3.
4.3 Classification and Analysis Algorithms

Since the scope of the storlet workflow management is the whole container, the connection types of any two storlets in the same container have to be analysed. Therefore, the trigger condition of one storlet has to compare with the result events produced by all the storlets in the same container, including the storlet itself. In the same manner, every result event has to match all the trigger conditions in the same container.

4.3.1 Classification 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 three different types: complete trigger, partial trigger or unrelated. 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. $\bigwedge_{i=1}^{n} \text{Atom}_i$), in order to find out the "strongest" connection types produced by fractions. And
Algorithm 1 Classification

1: \( resExpr \leftarrow \text{ResultExpression}(SourceStorlet) \) \{resExpr is the result expression of source storlet\}

2: \( condExpr \leftarrow \text{ConditionExpression}(TargetStorlet) \) \{condExpr is the condition expression of target storlet\}

3: \( resExprInDNF \leftarrow \text{TransformToDNF}(resExpr) \) \{Transform the result expression into DNF\}

4: \( resExprFracs \leftarrow \text{SplitExpressionByOr}(resExprInDNF) \) \{Split the DNF expression into factions by 'or' operators, each fraction is a pure conjunction of atoms\}

5: for all \( resExprFrac \in resExprFracs \) do

6: \( \text{if } resExprFrac \Rightarrow condExpr \text{ then} \)

7: \hspace{1em} Number of Complete Trigger + 1

8: \hspace{1em} break

9: \hspace{1em} else

10: \hspace{2em} condExprInCNF \leftarrow \text{TransformToCNF}(condExpr)

11: \hspace{2em} condExprFracs \leftarrow \text{SplitExpressionByAnd}(condExprInCNF)

12: for all \( condExprFrac \in condExprFracs \) do

13: \hspace{3em} if \( resExprFrac \Rightarrow condExprFrac \text{ then} \)

14: \hspace{4em} Number of Partial Trigger + 1

15: \hspace{4em} break

16: \hspace{3em} end if

17: \hspace{2em} end for

18: \hspace{1em} end if

19: end for

20: \( \text{if Number of Complete Trigger} > 0 \text{ then} \)

21: \hspace{1em} return Complete Trigger

22: \hspace{1em} else if Number of Partial Trigger > 0 \text{ then} \)

23: \hspace{2em} return Partial Trigger

24: \hspace{1em} else

25: \hspace{2em} return Unrelated

26: end if
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 implica-
tion exists, the algorithm continues the classification phase of partial trigger and 
unrelated. To identify a partial trigger, the algorithm converts the condition ex-
pression 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.

4.3.2 Axioms Generation Algorithm

A challenging part of algorithm 1 is to determine that whether there is a logical 
implication between two expressions (e.g. code line 6 & 13). The technology of 
automated theorem proving is used to solve this problem. As mentioned in section 
4.2.1, ATP is to 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 
and generated by another algorithm2. The format of axiom is expected to be the 
relations between atoms, and there are four possible types of relation, which are 
covering, overlapping, mutually exclusive and independent.

Covering indicates that two atoms share the same key, and the first value is a 
subset of the second one. For example, Atom1 : \{key, >'5'\}, Atom2 : key, >'3', in 
this case Atom1 is a subset of Atom2, Atom1 is covered by thus the axioms are 
generated as "Atom1 \(\Rightarrow\) Atom2".

Overlapping means the two atoms share the same key, and the values have 
common cases. For example, Atom1 : \{key, >'5'\}, Atom2 : key, ='7', in this case 
these two atoms are overlapping. The solution of axioms generation is to break 
up Atom2 into 2 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 items share the same key but values 
can never match. This relation is declared as no implication. For example, Atom1 
: \{key, ='5'\}, Atom2 : key, ='7', they are mutually exclusive, then the axioms are 
obtained as "Atom1 ! \(\Rightarrow\) Atom2", "Atom2 ! \(\Rightarrow\) Atom1".
Algorithm 2 Axiom Generation

\[ \text{resExpr} \leftarrow \text{ResultExpression(SourceStorlet)} \{ \text{resExpr is the result expression of source storlet} \} \]
\[ \text{condExpr} \leftarrow \text{ConditionExpression(TargetStorlet)} \{ \text{condExpr is the condition expression of target storlet} \} \]
\[ \text{sortedResAtoms} \ x \text{SortByKey(AtomExtractor(resExpr))} \]
\[ \text{sortedCondAtoms} \ x \text{SortByKey(AtomExtractor(condExpr))} \]
\[ \text{currentResAtom} := \text{first item of sortedResAtoms} \]
\[ \text{currentCondAtom} := \text{first item of sortedCondAtoms} \]
\[ \text{while not at the end of both sortedResAtoms & sortedCondAtoms lists do} \]
\[ \quad \text{if key of currentResAtom} < \text{key of currentCondAtom then} \]
\[ \quad \quad \text{currentResAtom} := \text{next item of sortedResAtoms till reach the end} \]
\[ \quad \text{else if key of currentResAtom} > \text{key of currentCondAtom then} \]
\[ \quad \quad \text{currentCondAtom} := \text{next item of sortedCondAtoms till reach the end} \]
\[ \quad \text{else} \]
\[ \quad \quad \text{match currentResAtom with currentCondAtom} \]
\[ \quad \quad \text{if covering then} \]
\[ \quad \quad \quad \text{return currentResAtom} \Rightarrow \text{currentCondAtom} \]
\[ \quad \quad \text{else if Overlapping then} \]
\[ \quad \quad \quad \text{return \ [currentResAtom} \Rightarrow \text{currentCondAtom1, currentCondAtom2!} \Rightarrow \text{currentResAtom} \]
\[ \quad \quad \text{else} \]
\[ \quad \quad \quad \text{return currentCondAtom!} \Rightarrow \text{currentResAtom} \]
\[ \quad \text{end if} \]
\[ \quad \text{end if} \]
\[ \text{end while} \]

The last relation type is independent which implies the two atoms holds different keys. As a result, no axioms are generated in this case.

4.3.3 Value Matching Algorithms

The value matching algorithm determines the logical relation between two constraint values. As mentioned in section 2.3.3, there are three different forms of values in constraints, which are constant form, alphabetical order form and regular expression form. All the six possible combinations are covered in the algorithm.

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 not, 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 in the algorithm 3. 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 hander, 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.

Algorithm 3 Alphabatical Order (‘>’, Greater Case) VS Regular Expression

Construct a DFA from the Regular Expression

if The order form start with symbol > or >= then
    LowerBoundValue := Order Value
    while LowerBoundValue has more characters do
        CurrentChar := next character of LowerBoundValue
        if All characters that produced are greater than CurrentChar at the same position of string then
            return Covering
        else if DFA can produce a character greater than CurrentChar at the same position of string then
            return Overlapping
        else if DFA can produce the same character as CurrentChar at the same position of string then
            Do nothing
        else
            return Mutually Exclusive
        end if
    end while
    if The symbol is > then
        return Mutually Exclusive
    else
        return Covering
    end if
else
end if

Regular Expression VS Regular Expression

The algorithm to match two regular expressions is also relying on the construction of DFA. Fortunately, the dk.bricks.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 one, otherwise these two expression overlap. If the intersection is empty, then the two regular expressions are mutually exclusive.

4.4 Visualization and Risk Detection

The storlets and connections form a complex network. From the above sections, enough information on storlets and their connections has been gathered. To visu-
alize them through the user interface, the first step is to design a well attributed graph. To facilitate the reviewing by users, the graph should not be very complex. However, all the key attributes has to be abstracted as annotations in the graph. For example, if an output is capable to produce two partial triggers and one complete trigger. As a solution to simplify the visualization, only the number of triggers produced by conjunction fractions of result events would be displayed above the edges in the visualization. The comprehensive view works as a guideline that points to the graph statistic details.

4.4.1 Graph Modelling Language and Tools

DOT Language

DOT is a simple and straight forward graph description language, and it is widely supported by various graph visualization and analyzing programs such as GraphViz, Tulip and Gephi. It also supports direct attributed graphs. A single vertex can be constructed with multiple sections and shapes, several different styles of edges help differentiate types of connections. Moreover, subgraph or cluster of vertices are also useful features of DOT language to group the handlers from the same storlet. Apart from attributed elements, DOT language also provides a property for customizing the default graph layout, which allows the developer fully utilize and allocate the space.

GraphViz

GraphViz is a package of open source graph visualization tools. It supports DOT language scripts and capable for all common image formats, SVG for web pages, PDF or Postscript for inclusion in other documents; and its Java library Grappa
even produce interactive graphs in the Java Component.

As a very simple example of the DOT syntax, the code below would produce a graph as figure 4.3,

```
graph g {
  a [label='Foo '];
  // Here, the node shape is changed.
  b [shape=box ];
  // These edges both have different line properties
  a — b — c [color=blue ];
  b — d [style=dotted ];
}
```

JGraphT

The GraphViz DOT provides features to plot an attributed graph with annotations. However, GraphViz package does not provide built-in algorithms for graph analyzing. Apart from building the graph, it is also necessary to detect risks as non-termination cycles and alternative paths to optimize the connections in the graph. JGraphT is introduced as a powerful library for graph manipulation and analysis. It figures out all the strongly connected subgraphs as potential cycles. However, this library is not designed for graph visualization, the built-in DOTExporter could not create a satisfying graph. As a solution, the graph instance of JGraphT is created only for graph analysis, and a specified translator is implemented to export the graph into DOT file when all the analysis is done. Finally the DOT file will be plotted with GraphViz tool.

JBPT

JBPT Java library is a compendium of process technologies. It provides a comprehensive collection of process model analysis, including the constructing, modifying and analysing EPC models. Models can be exported to DOT language files for easy graph plotting. Additionally, graph formats such as shapes, styles and annotations can be customized by implementing the element decorator interfaces.

4.4.2 Multi-level View Design

Two level of views are designed for the user interface. The abstract view provides a panoramic view of all the trigger handlers that are clustered by their host storlets, and their connections in the same container, including the risks such as non-terminations. While the comprehensive view elaborates the analysis results of connection types.
Abstract View

The abstract view is used to display all the storlets and their connection types in a single graph. From this view, users can inspect storlets in containers with a rough but complete perspective. In the storage system, a storlet may contain multiple trigger handlers, hence storlets and handlers are given unique names in the graph as well as in the system. Each handler is represented by a rectangle with three domains in the graph. The string in the left domain is the name of the handler, on the right side from top to bottom, trigger condition expression and result event are displayed. The graph aims to illustrate the possible trigger connections between the result events of one handler and the trigger condition of the other handler. Thus, the graph comes to be a directed graph, and so-called strict directed graph in the DOT language. As discussed in section 4.1, connections between handlers consist of three types, which are complete trigger, partial trigger and unrelated. Attributed edges are required to distinguish different connection types. Therefore, there are two types of arcs in the graph, the solid lines stand for complete triggers, and partial triggers are declared with dotted lines, if two handlers are unrelated, there is nothing between them. Above each single arc, there are two characters "c" and "p" followed by a number for each. The 'c' here means the number of complete trigger, and 'p' for the partial ones. Besides, all handlers are grouped by their host storlets. A large storlet rectangle clusters all handlers belong to it. If a handler or connection is in red, that means it participates in a potential cycle, which is probably a non-termination triggering risk.

Practically, the graph is initially constructed with Java library jGrapht. However, the library built-in DOT exporter does not support expected DOT format, a custom exporter is implemented to transfer the graph into DOT language. Lastly, the DOT file is plotted with GraphViz tool.

In figure 4.4, it shows a simple example of the abstract view. This graph contains 3 handlers. Handler1 has complete triggers target to both handler2 and handler3, because its result event ‘Appearance((Key1,='Value1'))’ exactly matches the trigger conditions of handler2 and handler3. Also handler3 has a complete trigger back to handler1. Therefore, handler1 and handler3 form a cycle, the connections and handlers are highlighted in red colour.
The comprehensive view is designed for expression analysis. When unexpected connection types appear in the abstract view, a user can specify a small set of storlets related to the unexpected connections, and "expand" those expressions for issue tracing. In this view, CPNs are used to model the connections between trigger handlers. Based on the procedure of ATP discussed in section 4.3.1, Trigger condition expressions are converted to DNF, and result event expressions are converted to CNF. In the model, trigger handlers are represented by rectangles with the handler names, while DNFs and CNFs are split into rectangle shaped conjunctions or disjunctions of atoms by "and", "or" connectors. Implications from conjunction atoms to disjunction atoms are represented by arcs. With the CPNs, it is much more convenient to find out the inappropriate parameters in trigger handlers. Technically, CPNs are modelled with jBPT library and visualized with GraphViz.

Figure 4.5 illustrates an example of the comprehensive view, in which there are two handlers connected by a partial trigger. As shown in the graph, the result event of handler1 is split into two parts, "Event Conj1" or "Event Conj2", which the trigger
condition of handler2 is split into: "EventDisjA" and "EventDisjB" and "EventDisjC". Due to none of the event conjunctions of "EventConj1" and "EventConj2" can fulfil all the three disjunctions of trigger condition, the connection types from both conjunctions are partial trigger. Therefore, the connection between these two handlers is a partial trigger.
Chapter 5

Results and Analysis

This chapter interprets the procedure of workflow construction and validation. Moreover, the validation is based on the risk detection features, such as the detection of non-termination triggering and unexpected overwriting. The solution of a concrete use case requirement of telecom industry is used as an example of this study.

5.1 Workflow Construction

To create a workflow, firstly the user should 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, an abstract view which presents the connections between storlets is generated. However, based on the requirement, there may be some restrictions of the connections between those storlets. For example, some storlets are enforced to execute in parallel or sequential. From the abstract view, when inappropriate connections are noticed, the comprehensive view is used to fix the storlets parameters. An example of storlet workflow construction will be given in the next subsection.

5.1.1 Use Case Example

With the exponentially increasing number of smartphone users, it becomes a fashion to read news and socializing over the Internet on phones nowadays. However, as a light-weight portable device, smartphones only support a limited set of file formats, hence, there is an urgent demand to convert incompatible files to suitable formats for proper delivery. A simple use case based on the requirement of a telecom company is illustrated in figure 5.1.

In this use case, like a publish/subscribe system, there are two publishers that serve only one subscriber. The first publisher likes writing, he modifies all the news as plain text and uploads them to the container of VISION. The second publisher works as a cameraman, all the source of news which he got are images in an arbitrary
size. After all the information is stored in the system container, the subscriber comes for downloading. However, the phone device only supports a limited set of formats. All the target files should be converted into a compatible format before being downloaded. Therefore, he injects a set of storlets into the same container to help him automatically deal with the compatibility issues. To obtain an English article in PDF, two different storlets are involved. When a new text file is recently stored, and if it is not a PDF file, the TextConverting storlet automatically converts it to PDF format. Following by the Translation stolet to translate a non-English PDF text into English. The last two storlets work for produce suitable images. If the newly stored image file is not in PNG, the ImageConverting storlet immediately converts it to PNG format. Moreover, if the size of image exceeds from the maximum acceptable size of the phone device (800*480), the Resizing storlet scales and resizes it to meet the requirement. Finally, when a suitable file appears in the container, the Notification Service will inform the subscriber to download.

5.1.2 Abstract View for Connection Type Checking

When all the storlets are created, the management service receives the user defined trigger conditions and result events of storlets, analyzes then displays a workflow graph as figure 5.2. In this graph, 4 storlets and a NS are connected by 1 complete trigger and 4 partial triggers. The "TextConverting" handler connects to the handler 'Translating' with a partial trigger. It is exactly as expected because the result of 'TextConverting' varies, those text files in English do not need to be translated. However, rather than a partial trigger between 'Resizing' handler and the NS, it is
expected that a complete trigger instead. The next subsection will explain how to analyze and fix the unexpected connection type with the comprehensive view.

5.1.3 Comprehensive View for Parameter Analysis

The comprehensive view is usually used to facilitate the connection type analysis. In the use case example above, the user wants to change the connection type between 'Resizing' handler and The NS from partial trigger to complete. Fortunately, the comprehensive view reveals the missing puzzle. Practically, the user only need to specify a set of storlets or handlers, the comprehensive view is generated automatically. In figure 5.3a, the result event of 'Resizing' and the trigger condition of Notification Service are split into fractions by logical connectors and connections between those fractions are represented by arcs. However, it is noticed that none of the two conjunctions from 'Resizing', 'Appearance((Width,<=480)&&(Height,<=800))' and 'Appearance((Width,=480)&&(Height,<=800))', can match all the three disjunctions from the NS. So that makes the connection partial trigger. To fix this problem, one can either modify the unmatched disjunction, 'Appearance((Format,=PNG)||(Format,=AAC)||(FileName ~en.*pdf$'))', or produce a new conjunction in the result event that matches all the disjunctions. As a solution, the trigger condition of the NS can be loosened to 'Appearance((Format,=PNG))||Presence((Format,=PNG))'. The 'Presence((Format,=PNG))' constraint is actually a hidden result event of 'Resizing'. The new comprehensive view looks like figure 5.3b. As a result, the connection type between 'Resizing' handler and the NS becomes complete trigger.

5.2 Advanced Workflow Validation

The work is not completely done after the workflow construction. There are more 'advanced' risks can exist in the workflow, such as non-termination risk, and Overwriting risk. A workflow validation phase is necessary in the workflow development procedure. The detection and issue tracing of these two risks are introduced in this chapter.
Figure 5.3: Comprehensive View for Analysis
5.2. ADVANCED WORKFLOW VALIDATION

5.2.1 Non-termination Risk

This problem is also revealed by the abstract view. As mentioned above, handlers and connections which participate in cycles are in red. However, it is not asserted that cycles would always produce non-termination issues. If a cycle contains partial triggers as components, it is unlikely to produce an endless trigger loop unless the partial triggers are complemented to complete triggers with other partial trigger. Even if the cycle is consist of complete triggers, it is not guaranteed that it will produce a non-termination trigger. Because of the intervention of user and other storlets, the metadata state can be changed at anytime unpredictably. In the same manner, users can choose to break an unexpected connection based on the analysis from comprehensive view.

5.2.2 Overwriting Risk

This issue is normally caused by simultaneous operations on the same object. On the perspective of storlet workflow analysis, parallel activation of multiple storlets which are operation on the same data object produces this problem. This risk cannot be directly detected by the visualization interface, but the detection is supported by the analysis engine. In the previous cases, the engine is used to determine the connection types between trigger conditions and event results. However, in order to find out the possible parallel activation, the engine is used to compare trigger conditions with trigger conditions. If trigger condition A 'completely triggers' trigger condition B, that means when A is activated, B must be activated. On the other hand, if it is a partial trigger between two conditions, then there is a low chance they will be activated simultaneously. For example, if handler1 and handler2 shares the same trigger condition Appearance(Key,=Value), and both of the computations operate on the event object, with flag "same", it is most probably a overwriting issue. On the other hand, if the condition of handler1 appears as Appearance(Key1,=Value1), condition of handler2 is Appearance(Key1,=Value1)&&(Key2,=Value2). With the analysis engine it is derived that Appearance(Key1,=Value1)&&(Key2,=Value2) \implies Appearance(Key1,=Value1), and both of them operate on the event object. In this case, if handler2 is activated, most probably handler1 will be activated simultaneously. In contrast, if handler 1 is activated, handler2 only have a relatively low chance to be activated.
Chapter 6

Algorithm Performance Discussion

Due to the limited numbers of storlets in one container (around 50 maximum), and the update frequency of storlets is only a small fraction comparing to the update frequency of data objects, algorithm performance is not considered in the highest priority. An evaluation of analyzing 50 storlets is done with a 4 cores, 6GB RAM machine, the time cost is less than 2 seconds. Since it is not common to update all storlets in one batch, the performance is acceptable. However, it is useful to discuss the performance with possible optimization approaches as a discussion of this study.

6.1 Storlets Traversal

In the current implementation, a double loop is used to compare each trigger condition with each result event, and that makes the algorithm complexity to $O(n^2)$, $n$ is the number of storlets. As a possible performance optimization, the storlets can be classified into different categories (e.g. Text related, image related, audio related, etc), one storlet can be placed into different categories. As a result, storlets from one category do not need to be compared with storlets from other categories. Assume the number of groups is $m$, and all groups can be analyzed in parallel by different processes or threads, the average complexity would reduce to $O((\frac{n}{m})^2)$. However, due to the small number of storlets per container, the classification overhead is relatively large compare to the computation time.

6.2 Classification

Ignoring the CNF(DNF) transformation cost, assuming the result event is split into $m$ different conjunctions while the trigger condition is split into $n$ disjunctions. If the connection type is complete trigger, the algorithm complexity is $O(m)$. However, if the connection is a partial trigger, the complexity in the best case is still $O(m)$, while the worst case reaches $O(m + n)$. A possible optimization is to properly order the disjunctions and make the partial trigger to be find out as early as possible. The
order of disjunctions can be shifted according to the keys of current conjunction. The shifting algorithm is still by design.

6.3 Axiom Generation

In the current axiom generation algorithm, atoms are extracted from expressions for every comparison of two storlets. A possible optimization is to create two big hash tables to store atoms by keys, one for atoms in trigger conditions and the other for result events. Because the axioms are produced by atoms share the same key, it is useful to keep all the axioms statically together with the hash tables, so that they can be reused for other iterations fast and easily.
Chapter 7

Conclusion and Future Work

This study has developed a service that solves the storlet workflow construction and validation problems of VISION Cloud. In the front-end, this service provides a multi-level view visualization. The abstract view provides a high level graph in the handler level that contains all storlets involved and the different types of connections between the handlers. On the other hand, the comprehensive view works as a complementary of the abstract one. When an unexpected connection type between two storlet handlers is noticed, the comprehensive view is used to trace the issue and guide the user to fix the connection type by modifying associated storlets parameters. In aspect of the back-end, the two level views share the same analysis engine. The engine takes response to fetch the handlers’ information of the involved storlets from NS automatically, and accept newly created storlet information externally from users. And then it determines the connection type between each two storlet handlers. Finally, the connection type results and storlets information are displayed as graphs in the multi-level view visualization.

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 the abstract view. Then if one notices any connection types are unexpected, the user can fix them easily with the analysis of the comprehensive view. Apart from workflow construction, this service also supports workflow validation. When a workflow is newly created as mentioned above, but just before injection. It is necessary to check whether there will be conflicts or risks between the new workflow and existing storlets. The abstract view displays all storlets and connection in the container together with the new workflow, and predicts the situation as it is already injected. As a result, the risks as incorrect connections between new storlet workflow and existing storlets, non-termination cycles and unexpected overwriting are revealed. In the same manner as workflow construction phase, those problems can be easily fixed with the assistance of the comprehensive view.
Future Work

Since VISION Cloud storage is a computational storage system based on ECA rules, it refers that this solution and model can be applied to similar ECA-rule based event-driven systems. Generally, the triggering workflow analysis procedure can be abstracted into three steps, they are axiom generation, connection type classification and visualization. A workflow management framework can be developed for this purpose. Intuitively in the framework, the definition of connection types and visualization model can be extended to meet more advanced requirements. Besides that the developers only need to implement the axiom generator based on the specified triggering scheme of the system. For example, this framework can be used to build a workflow management service for DBMS. Although DBMS does support interaction analysis between triggers, to have a panoramic view of all triggers and interactions would still be a great help. The rather that the analysis is done statically before the triggers actually start executing in the database systems.

Due to the flexibility of this service framework. It is great that algorithm implementations can be altered depends on the environment. For example, if the computation demands high performance, it is good to store axioms in a big hash table. However, if the environment is memory intensive and the performance is not strictly required, looping will be a better option in that case.
Bibliography


