MASTER THESIS

WORKFLOW MANAGEMENT SOLUTION FOR A SECURITY MANAGEMENT APPLICATION

Zhen Chen

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE
SOFTWARE ENGINEERING

EXAMINATION COMMITTEE
dr. Luís Ferreira Pires
dr. ir. M. J. van Sinderen
mr. W.Lindenhof

DOCUMENT NUMBER
EWI/SE - 2012-004

UNIVERSITY OF TWENTE.

NOVEMBER
2012

Abstract

Security management applications play important roles in the modern society due to the increasing demand for better security. Today's advanced security management systems tend to be bigger and more intelligent. The business processes involving these systems are far more than simple and passive monitoring done by human operators. Highly automated business processes are required. Moreover, these processes could be rather unstable due to the rapid changing customer's requirements and high flexibility of systems.

Workflow management (WFM) promises a promising solution to modeling, executing and managing business processes. WFM pushes the business process perspective out of the domain of software applications, which provides software sufficient flexibility to adapt to rapid changing business processes. It is very beneficial to adopt WFM in the domain of the security management applications.

This thesis proposes the WFM solutions for a concrete security management application, namely the FlinQ security management platform. The FlinQ platform selects Petri nets as workflow modelling language and contains a Petri nets-based workflow engine. Our work proposes the extensions for traditional Petri nets in order to improve the capabilities of representing and manipulating data. The extensions take full advantages of XML and XML data processing technology (i.e. XQuery). In addition, our workflow definitions can be represented in a XML format with a uniform syntax that is specified by XML Schema.

Moreover, we propose the design of the entire WFM framework for the FlinQ platform. Based on our Petri nets-based language with the data extensions, we integrate an XQuery processor and a native XML database into the existing workflow engine. Moreover, we created a workflow definition loader for loading the XML-based workflow definitions into our workflow engine.

Preface

This thesis describes the results of my master final project, carried out from February 2012 to September 2012 at FlexPosure, Netherlands. With this thesis I complete my Master program in Telematics at the University of Twente.

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. First of all, I would specially like to thank my supervisors, Mr. Wouter Lindenhof, Dr. Luís Ferreira Pires and Dr.Marten van Sinderen, who all provided me with guidance and feedback during my research. I would also like to convey thanks to Mr. Alef Schippers and Mr. Gaby Uljee who initiated my master project in FlexPosure and found a position for me.

My sincere appreciation goes to FlexPosure for providing a very pleasant work environment and all the great colleagues I have had the opportunity to meet and who where more than willing to help me, in particular Wouter, John, Gert-Jan, and Dieter.

I would like to acknowledge the support of Mr. Jan Schut who convinced me to come to UT and helped me finding a scholarship opportunity. Every time I had problems during the stay of two years, your door was always open to me.

Finally, I would like to express my gratitude to my parents who made it possible for me to study in the Netherlands and always stood behind me and my decisions.

Utrecht, 19th November, 2012

Zhen Chen

Table of Contents

1. Introduction

This chapter is further structured as follows: Section 1.1 briefly presents the motivation of this work, Section 1.2 states the objectives of this thesis, Section 1.3 presents the approach adopted in the development of this thesis and Section 1.4 outlines the structure of this thesis by presenting an overview of the chapters.

1.1 Motivation

The increasing demand for security by society leads to a growing need for surveillance activities in various environments. Based on this need, security management applications have emerged.

The evolution of security management applications started with video-based surveillance systems using analogue technology, namely Closed Circuit Television (CCTV). These systems, which are referred to as first generation surveillance systems, consist of a number of cameras located in multiple remote locations and connected to a set of monitors, usually placed in a single control room, via switches (a video matrix) [1]. Later on, most of the video cameras started to use a digital Charge Coupled Device (CCD) to capture images, and analogue techniques only to distribute and store data. With the increased performance of fully digital systems powered by computers, additional features could be realized, e.g. real time events detection and license plate recognition [2]. These technological improvements have led to the development of semi-automatic and more intelligent systems, known as second generation video-based surveillance systems.

Today's security management applications, also referred to as third generation surveillance systems, are designed to deal with a large number of cameras and sensors, a geographical spread of resources, many monitoring points, and to mirror the hierarchical and distributed nature of the human process of surveillance [1]. As a result, an automated system with high intelligence and scalability is required. Moreover, today's security management systems are required to offer an integrated solution that encompasses heterogeneous sensors and multiple separated traditional surveillance applications, including access control, intercom and barrier gates control, other than a single video-based surveillance system.

The business processes in today's security management systems are far more than simple and passive monitoring done by human operators. Automated and long running business processes that need to be predefined with a minimum of human interaction involved are highly required. In addition, due to the high

flexibility of security management systems, rapid changing customer's requirements and market conditions, business processes involving security management applications are also subject to frequent changes. Consequently, many applications have to be modified at implementation level in order to satisfy these changes, which is undesirable. Therefore, security management applications need to provide sufficient agility to unstable and changing business processes.

Workflow Management (WFM) promises a new solution to an old problem: modelling, executing, monitoring and optimizing business processes. The concept of WFM emerged in 1980's and enjoyed large popularity in the past three decades [3]. By identifying, modelling and managing business processes, companies get insight in what they are doing, which parts of the process can be automated or can be optimized. Moreover, WFM pushes the business process perspective out of the domain of software applications, instead of hard-coding business processes in the applications. This provides software sufficient flexibility to adapt to rapid changing business processes.

In the current IT industry, many WFM products are offered by various vendors. These products are used and integrated into different kinds of application domains. However, the adoption of WFM in the domain of the security management applications is rarely discussed in the literature. WFM provides a promising solution to handle increasingly complex business process issues for today's security management applications.

Process modelling is a dominant factor in WFM. Workflow modelling languages offer uniform syntax and structures for modelling and analyzing business processes. Popular workflow modelling languages include Petri nets, Web Service Business Process Execution Language (WS-BPEL) and Yet Another Workflow Language (YAWL). Some of these languages provide formal semantics and graphical notations. Some of them are also executable, which means that the processes defined with these languages can be deployed and directly executed by workflow engines.

Many workflow modelling languages are more concerned about the control flow perspective. However, the data perspective also plays a vital role in workflow modelling although it is usually ignored. Currently, many security management applications tend to be data-centric. The desired workflow modelling language should be capable of representing and manipulating data. In this work, we select Petri nets as our workflow modelling language. However, classic Petri nets is limited concerning data. Some new languages, like WS-BPEL and YAWL, wrap the data in XML format and completely rely on XML-based standards for data processing. XML offers high flexibility, high extendibility and strong data types support in terms of data representation. Moreover, the XML data processing standards, like XPath and XQuery are able to support high-level data manipulation. Considering

these benefits of using XML and XML data processing standards, it is worth investigating how Petri nets and XML technologies can be combined.

1.2 Objectives

The main goal of this thesis is to investigate and propose WFM solutions in the domain of the security management applications, including support for data representation and manipulation. By introducing WFM, security management systems are capable of modelling, automating and analyzing complicated business processes. Moreover, WFM pushes the business process perspective out of the domain of software applications. The software system is expected to enjoy more flexibility, and the cost of application development and maintenance should also be reduced.

The work has been inspired by an existing WFM framework that has been developed for a concrete security management platform (i.e. FlinQ) and contains a Petri nets-based workflow engine. However, this workflow engine has limited functionality, especially lacking of the capability of data handling. Our research thus mainly works towards the improvement on the ability of handling data in this WFM framework.

During the development and improvement of the WFM framework in the FlinQ security management platform, the following research questions are considered:

(1) Which extensions of Petri nets are available for improving the capability of data representation and manipulation?

(2) Which technology and standards can be used for improving the capabilities for data representation and manipulation in Petri nets?

(3) How to integrate data processing and management services in a Petri nets-based workflow execution environment?

1.3 Approach

The following steps have been taken in order to pursue the defined goals and answer the research questions listed above:

- We performed a literature study on WFM and workflow modelling languages. We studied three popular languages: Petri net, WS-BPEL and YAWL. In particular, we focused on the data perspective of workflow modelling languages. In our study, we investigated how the data behind

and related to WFM can be represented and manipulated in these languages.

- We investigated security management applications. We produced an overview of security management applications including their evolution, the main characteristics and challenges. This part of work has been conducted by means of literature study and also a case study of a concrete application (i.e. the FlinQ security management platform).
- We identified and determined the requirements of our WFM solutions for the FlinQ security management platform.
- We proposed an appropriate Petri nets variant as workflow modelling language to improve the capability of data handling by using XML technology.
- Based on this language, we designed our WFM framework for the FlinQ platform. In particular, we integrated the XQuery processor and a native XML database with the workflow engine core.
- We investigated possible tool environment and software components that could be used in the development of our WFM framework, and developed the proposed WFM framework by implementing some selected components.
- We performed a case study to test and evaluate our proposed WFM solutions.

1.4 Structure

The structure of this thesis reflects the steps of the approach taken in this work. This thesis is further structured as follows:

Chapter 2 gives an overview of security management applications and discusses the FlinQ products as an example application.

Chapter 3 gives an overview of workflow management. Firstly it explains the basic concepts of WFM and WFMS by means of a generic WFMS architecture, WFMS components and WFM phases. Secondly, it introduces three popular workflow modelling languages namely Petri net, WS-BPEL and YAWL. Thirdly, this chapter investigates the data perspective in workflow modelling languages. We investigate the capabilities of data representation and manipulation in these three languages.

Chapter 4 proposes and elaborates our Petri nets-based workflow modelling language with the extensions for data handling. In addition, it discusses our XML-based workflow definitions.

Chapter 5 presented the design of the entire WFM framework. We particularly discuss the integration of the XML data with an XQuery processor into the workflow engine.

Chapter 6 gives some examples in which business processes are modeled and executed in security management applications based on our WFM framework, in order to perform a case study in the FlinQ platform.

Chapter 7 presents the conclusions and future work of this research.

2. Security Management Applications

This Chapter gives an overview of security management applications and discusses the FlinQ security management platform as an example.

This chapter is structured as follows: Section 2.1 introduces the evolution of security management applications. Section 2.2 identifies and discusses the main characteristics of today's advanced security management systems. Section 2.3 gives an overview of the FlinQ security management platform and particularly identifies the issues in the domain of WFM.

2.1 Introduction

Nowadays, security management has been an inseparable part of our daily lives due to the demand for better security. Security management applications take the advantages of the modern ICT technology to perform the monitor and surveillance tasks.

The first security management solutions were based on an analogue technology called Closed-Circuit television (CCTV). In the most trivial case these were cameras connected with TV sets located in one room [2]. And the majority of these CCTV surveillance systems use analogue techniques for image distribution and storage.

Currently most of the video cameras use a digital Charge Coupled Device (CCD) to capture images. With high performance of fully digital systems powered by computers additional features can be expected, e.g. real time events detection and license plate recognition [2]. These technological improvements have led to the development of semi-automatic and more intelligent systems, known as second generation video-based surveillance systems [1].

The increasing demand for security by society leads to a growing need for surveillance activities in more various environments. Therefore, security management systems are getting bigger and more complicated. Today's security management systems are required to offer an integrated solution that encompasses heterogeneous sensors and multiple separated traditional surveillance applications, including access control, intercom and barrier gates control, other than a single video-based surveillance solution. In Chapter 2.2, we highlight and explain three key characteristics of today's advanced security management systems.

2.2 Today's Advanced Security Management Systems

- *Intelligent system*

Traditional video-based surveillance systems mainly focus on passive monitoring and video storage with human operators facing a large number of monitors, over a long period of time and trying to detect suspicious situations. In practice, this method is ineffective due to the decreasing level of attention of human beings after a short time. Additionally, in large security management systems the number of monitors is too large for an accurate observation. Today's security management systems are required to be more automated and intelligent.

A lot of work has been done to enable security applications to be more intelligent in both in commercial and academic environments. Some of research aims at improving *image processing* by generating more accurate and robust algorithms in object detection and recognition, tracking, human activity recognition, database and tracking performance evaluation tools [1]. Moreover, some intelligent security management systems tend to use specific-purpose hardware and digital intelligent cameras to perform intelligent tasks like intrusion and motion detection [1] and detection of packages.

- *Flexible framework for the large scale system*

Today's security management systems tend to have large scale and encompass heterogeneous software components, sensors and actuators. A flexible framework is thus required. Such a framework is supposed to provide decentralized architecture with high expandability and upgradability. Nowadays the Service-Oriented approach is usually chosen to build flexible and extendable software architecture. According to the Service-Oriented Architecture (SOA) paradigm, software should be delivered as loosely coupled and cooperating services which should be described, published and easily discovered. SOA is mature in the business world and has been adopted in the domains of security management like video conferencing [35] or the public security sector [36].

- *Handle with complex and changing business processes*

The more complicated system results in more complex business processes. The business processes in today's security management systems are far more than simple and passive monitoring done by human operators, which requires more automated business processes. In addition, high flexibility of system, rapid changing customer's requirements and market conditions lead to changing and unstable business processes. This needs sufficient flexibility and agility in the system. Workflow Management (WFM) provides a promising approach to assist security management applications in handling with complex and changing business processes. Network Enabled Surveillance and Tracking [34] is one of examples that WFM has been used

in security management applications. However, the implementation of WFMS in security management systems is rarely discussed in the literature and research.

Network Enabled Surveillance and Tracking (N.E.S.T.) [34] is an example of today's advanced security management solutions. N.E.S.T. is designed to manage and control a large number of objects (e. g. humans or sensors), tasks and events. It adopts a decentralized and SOA-based architecture that provides high expandability and upgradability. Furthermore, based on its SOA environment, N.E.S.T. consists of multiple intelligent services for data processing (e. g. motion detection and tracking in multi-sensor video, abandoned luggage detection) and information analysis (e.g. semantic description of complex situations).

In particular, N.E.S.T. uses a workflow modelling language, namely WS-BPEL, to handle and automate complex business processes. The surveillance processes in N.E.S.T are modelled in WS-BPEL that orchestrates the required intelligent services. The execution environment is a WS-BPEL engine which connects to the services through the so called *Service-Bus*.

The second bus (*Result-Bus*) is built for very frequent data with a singular semantic e.g. alarm events. A third infrastructure for streaming data is called *Streaming-Bus*. Static and dynamic data are stored in the *World Model* through the *Model Access Service*.

Figure 2.1 shows the overview of N.E.S.T system.

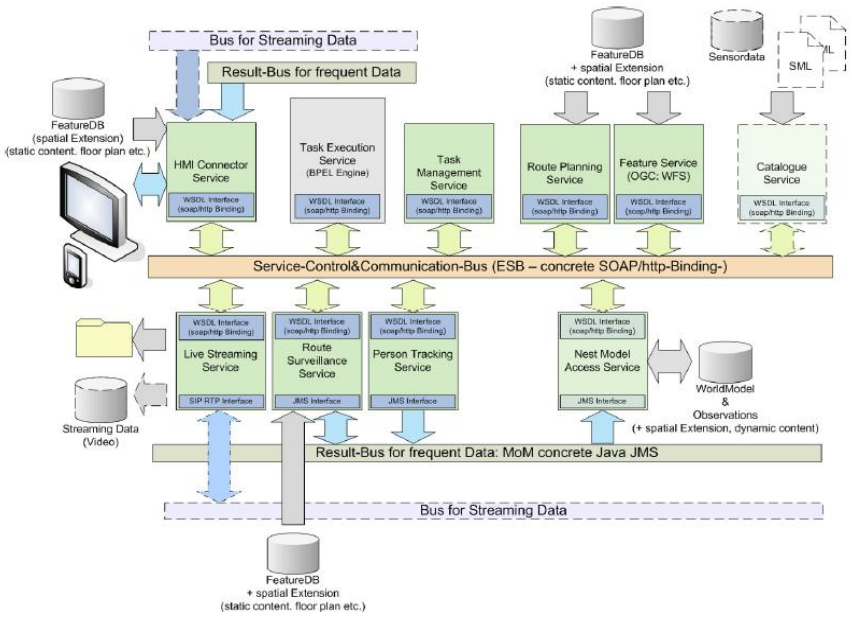

Figure 2.1 N.E.S.T system overview [34]

2.3 FlinQ Security Management Platform

FlinQ is an advanced security management application which is developed by FlexPosure B.V. It is designed as a platform that provides high flexibility to easily integrate with different traditional surveillance applications and heterogeneous sensors. Depending on different customers' requirements, these separate surveillance applications can be CCTV system, access control system, barrier gates control system and intercom system.

Figure 2.2 shows overview of FlinQ system. The entire platform is built on a Client/Server-based architecture. Client side applications are available for multiple platforms, including Windows, Android and iOS.

In particular, *Connectors* are built as additional layer between FlinQ server and external applications. A connector is basically designed as an adapter that enables server to be able to interact with heterogeneous software components, including cameras, actuators and sensors in the subsystems.

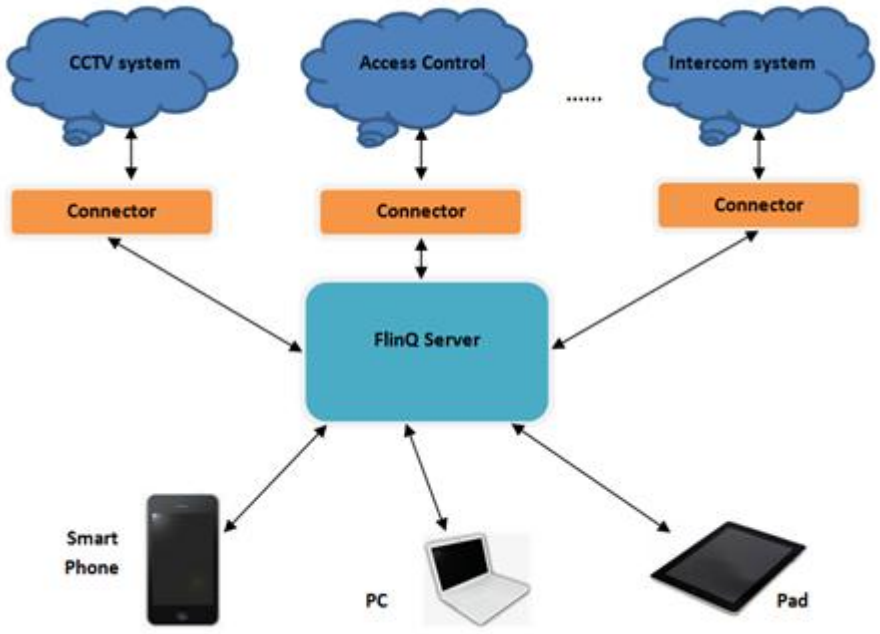

Figure 2.2 FlinQ security management platform overview

As an advanced security management application, the FlinQ platform takes full advantages of its integrated applications for achieving more intelligent and automated security management solutions. For instance, DIVA, a smart video-based surveillance system is one of applications which have been integrated into the FlinQ platform. It is able to perform several intelligent

tasks, like facial recognition, object recognition, license plate recognition and scene change detection.

Due to the centralized architecture, the majority of business processes are implemented and controlled at the server side. Server performs these business processes by interacting with both client applications and connectors. The interaction is quite event-based and conducted by sending the XML messages. Connectors and client applications send the *event* messages to the server when the events occur. While server sends the *command* messages to connectors and clients for performing tasks. Figure 2.3 depicts the interaction between server, connectors and clients by the XML messages.

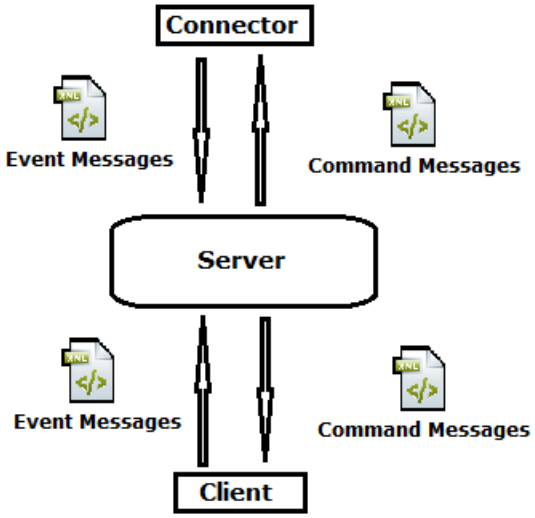

Figure 2.3 The interaction between server, connectors and clients

For example, when sensors detect smoke in the certain room, it will send an event message with relevant information to notify the server via connectors. Then the server has to decide which actions should be taken to handle this smoke event, which is actually the business process need to be handled. Server might firstly notify all active client applications this smoke event by enabling alarm and showing live video of relevant locations, which is done by sending the command messages. Then server stores the information related to this smoke event in database. In addition, based on the different event types (in this case it is a smoke event) and priorities, the client application might have different reactions, like only blinking button or enabling alarm or even automatically take actions without waiting for users' response. All of these things involve the business processes in the FlinQ platform.

Like most of advanced security management applications, FlinQ requires highly automated business processes with a minimum of human interaction involved. Moreover, the highly flexible architecture of the FlinQ platform

results in rapidly changing and unstable business processes. Therefore, there is a high demand for the WFM support in the FlinQ platform.

Currently, a WFM framework in the FlinQ platform is being developed. This framework selected a Petri nets variant as workflow modelling language and built a Petri nets-based workflow engine for workflow execution. However, the used modelling language and the corresponding workflow engine do not have the capabilities of representing and manipulating data. This becomes a fatal drawback for the FlinQ platform where the business processes tend to be quite data-centric. In addition to this, the entire WFM framework is still rather uncompleted as a typical workflow management system.

3. WFM Overview

This chapter reports on the conducted literature study, giving an overview of WFM. In Section 3.1, the emergence of WFM is discussed from a historical perspective. Section 3.2 investigates WFMS concepts and architecture based on the Workflow Reference Model. Section 3.3 distinguishes between BPM and WFM by considering the BPM lifecycles. In Section 3.4, we discuss three popular workflow modelling languages. Section 3.5 investigates the data perspective in these workflow modelling languages. Finally, Section 3.6 draws conclusion.

3.1 Background

Software systems in the past were designed to support the execution of individual tasks. However, due to the changing customer requirements and market conditions, today's software systems need to support more complex business processes. There is a high demand for controlling, monitoring and support the business processes and workflows. Based on this need the term workflow management has emerged [4].

However, there were no generic tools to support workflow management in the earlier periods. As a result, parts of the business process were hard-coded in the applications. For example, an application to support task *A* triggers another application to support task *B*. This implicitly means that one application knows about the existence of this other application. However, this is undesirable, because every time the underlying business process is changed, applications need to be modified. In addition, it is very common that similar workflows need to be implemented in the different applications and it is not possible to monitor and control the entire workflow. For the purpose of solving these issues, workflow management systems thus have been first introduced by several software vendors in 1980's.

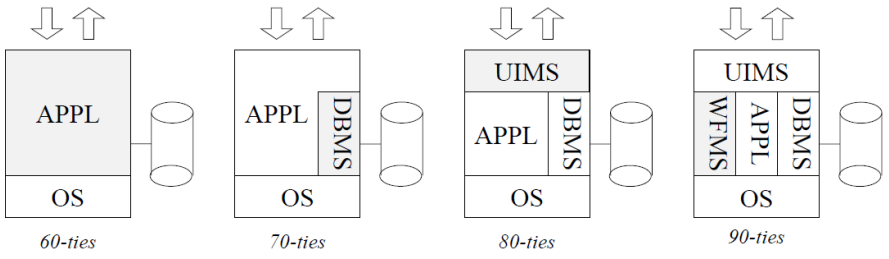

Figure 3.1 Historical evolution of software application architecture [6]

A workflow management system (WFMS) is a generic software tool which allows for the definition, execution, registration and control of workflows [5]. The impact of workflow management systems on the IT industry has been already widely acknowledged. Currently many vendors are offering a workflow management system. The benefits of a WFMS are comparable to the benefits of a user interface management systems (UIMS) or a database management systems (DBMS). Flexibility, integration of applications and a reduction in development costs are the incentives for using a WFMS. Just like DBMSs that push the data out of the applications, and UIMSs that push the user interaction out of the applications, the emergence of workflow management systems enables software developers to push the business processes out of the applications. Figure 3.1depicts the historical evolution of application architecture.

3.2 WFMS Concepts and Architecture

This section identifies WFMS concepts and architecture which are standardized by Workflow Management Coalition (WFMC) that was founded in 1993. WFMC is an international organization whose mission is to promote workflow and establish standards for WFMS [6].

The Workflow Management Coalition defines workflow as:

"*The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.*" [7]

In a typical WFMS, the workflows are *case-based*. An executing instance of a workflow model is called a ***workflow case*** or ***workflow instance***. Examples of workflow cases are an order, a request for information or an alarm event which needs to be handled in a surveillance system. Workflow cases are often generated by an external customer, devices or sensors.

A workflow is designed to handle similar cases. Cases are handled by executing tasks in a specific order described in a workflow definition. Since multiple cases can be handled simultaneously by following the same workflow definition, the same task can be executed for many cases.

WMFC also published the Workflow Reference Model that reveals the generic architecture of Workflow Management System, including relevant components and interfaces.

Figure 3.2 shows the main components and interfaces of WFMS.

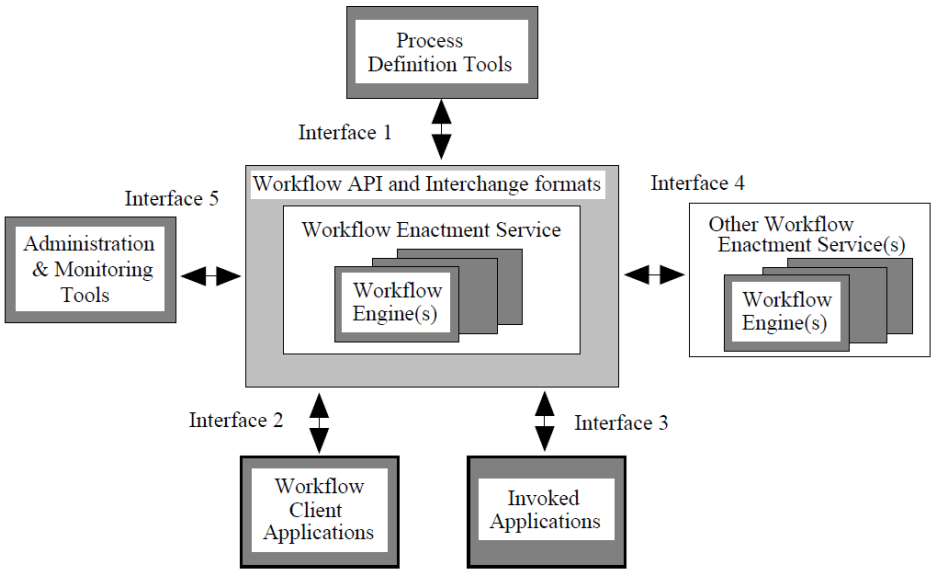

Figure 3.2 Workflow Reference Model [7]

The main components and interfaces of Figure 3.2 are discussed below:

Workflow Engine is a software service that provides the runtime execution environment for a workflow instance.

Workflow Enactment Service consists of one or more workflow engines and is responsible for creating, managing and executing workflow instances. The workflow engines that belong to a workflow enactment service may be deployed in a centralized or distributed manner.

Process Definition Tools are able to analyze, model, describe and document a business process. These tools may support various workflow modelling languages, like Petri nets and BPEL. The final output of the process definition tools is a process definition which can be interpreted at runtime by the workflow engine(s) within the enactment service.

Workflow Client Applications (also referred as *worklist handlers*) are software entities which interact with the end-user in those activities which require involve human resources.

Invoked Applications are any applications, programs or services which should be called and invoked in the workflows. The invoked application may be local to the workflow engine, co-resident on the same platform or located on a separate, network accessible platform. For instance, the web services that are involved in the WS-BPEL process are invoked applications.

Administration & Monitoring tools are created for management and control beyond the workflow engines. These tools are used to register the progress of workflow cases and to detect bottlenecks.

Interface 1 is responsible for the exchange of workflow definitions between process definition tools and workflow engines. A universal interchange format for the process definition is required.

Interface 2 was developed to facilitate Workflow Client Application integration with different workflow engines.

Interface 3 copes with the interaction between invoked applications and workflow engines. It is implemented according to the access mechanisms of the invoked applications.

Interface 4 supports workflow interoperability models and the corresponding standards for interworking between multiple workflow enactment services.

3.3 WFM Phases

Recently, Business Process Management (BPM) has been widely considered as the next step in the evolution of WFM. The WFMS, according to the definition given by the Workflow Management Coalition, emphasizes the focus on the process enactment, i.e., the use of software to support the execution of operational processes. However, it has been realized that the traditional focus on enactment is too restrictive [8].

The concept of BPM extends the traditional WFM. The differences between BPM and WFM can be identified by considering the BPM lifecycle.

The BPM lifecycle contains four phases: process design, system configuration, process enactment and diagnosis [9]. These four phases are overlapping and the whole lifecycle is iterative. Each phase is discussed as below:

1. Process design

This phase consists of identifying existing processes and capturing the business processes in process models.

2. System configuration

This phase is also referred to as Implementation phase. In the system configuration phase, designed processes are implemented by configuring a process-aware information system (e.g., a WFMS). Currently it is common that the configuration is done by deploying executable business process definitions in the BPMS.

3. Process enactment

The process enactment phase is the runtime phase of the lifecycle. In this phase, business processes are executed and monitored by a BPMS.

4. Diagnosis

This phase is also referred to as the Evaluation phase. In this phase, the executed processes are analyzed to identify problems and to find areas for improvement. The conclusions drawn in the diagnosis phase are input for the next iteration of the lifecycle.

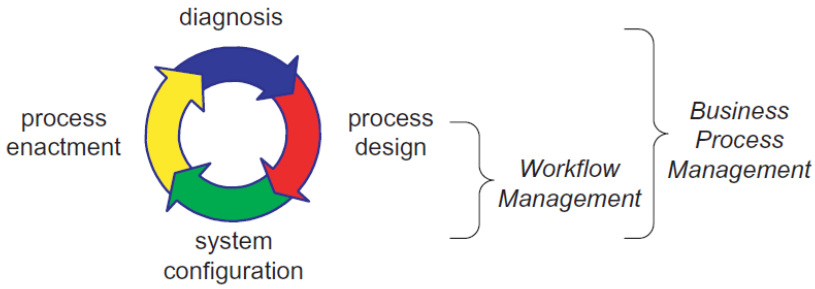

Figure 3.3 The BPM lifecycle to compare WFM and BPM [9]

Figure 3.3 depicts the relationship between workflow management and business process management using the BPM lifecycle. The focus of traditional workflow management (systems) is on the lower half of the BPM lifecycle. Specifically, there is little support in WFMS for the diagnosis phase, since only a few workflow management systems support simulation, verification, and validation of process designs. Moreover, the support offered by WFMS in design phase is also limited to the availability of an editor in most cases; however, analysis and real design support are often missing.

To sum up, BPM extends the capabilities of WFM by offering more sophisticated build-time and runtime diagnostic capabilities and more sophisticated capabilities in the process design phase.

3.4 Workflow Modelling Languages

Workflow modelling languages play important role in WFM. They offer a uniform form for abstracting, modelling and analyzing business processes. Moreover, some of them can also be executed after deployment in their corresponding workflow engines.

In this section, we discuss three popular workflow modelling languages, namely Petri net, Web Services Business Process Execution Language (WS-BPEL) and Yet Another Workflow Language (YAWL). For each language, we elaborate its languages syntax and constructs by giving some practical

examples. The benefits and drawbacks of each language are also summarized in this part.

3.4.1 Petri nets

Petri net is a mathematical modelling language which is widely used in workflow management systems. It is able to describe complex processes which are characterized as being concurrent, asynchronous, parallel and nondeterministic with both formal notion and graphical representation. The classical Petri net was invented by Carl Adam Petri in the 1960s [4].

Language constructs and rules

In brief, a Petri net is a directed, connected, and bipartite graph in which each node is either a place or a transition. Places and transitions are connected by directed arcs. Tokens occupy places. The elements and the rules of Petri nets are described as follows:

- *Place*

Places are graphically represented as circles. A place may have zero or more tokens. Places are passive components and model the system states.

- *Transition*

Transitions are graphically represented as squares. Transitions are active components modelling activities, processes or events.

- *Arc*

Arcs are graphically represented as arrows. Arcs link a place to a transition or vice versa, never places or transitions.

- *Token*

Tokens are graphically represented as black dots and occupy places. The distribution of tokens in the Petri nets is changed by the occurrence of transitions.

- *Marking*

Any distribution of tokens over the places that represents a configuration of the net is called a marking. A marking represents the current status of the Petri net.

- *Enabled Transition*

A transition is enabled if each of its input places contains at least one token.

- *Firing*

An enabled transition can fire (i.e., it occurs). When it fires it consumes a token from each input place and produces a token for each output place.

It is worth noting that the execution of Petri nets is nondeterministic. This means if a transition is enabled, it may fire, but it does not have to. Since firing is nondeterministic, and multiple tokens may be present anywhere in the net (even in the same place), Petri nets are well suited for modelling the concurrent behavior of distributed systems.

Mapping workflow management concepts onto Petri nets is rather straightforward: tasks are modelled by transitions, conditions are modelled by places, and cases are modelled by tokens.

Example

A simple example of using Petri net to model workflow in practice is discussed below. Figure 3.4 shows the Petri net diagram of the example.

The example shows the real business process of insurance claim for car damage. Four tasks check insurance, contact garage, pay damage and send letter are modelled directly by transitions in Petri net. Moreover, there are two additional transitions (fork and join) and five places (p1, p2, p3, p4 and p5) which are used to route a workflow case through the procedure in a proper manner. Finally, the place i and the place o represent the starting point and ending point of the entire workflow respectively.

When a token occupies the place i and the workflow start, the transition fork fires. Due to the additional transition fork, the tasks check insurance and contact garage are executed in parallel. Only if these two tasks are both finished, the transition join can be enabled. Place p5 has a condition that determines which tasks (pay damage or send letter) is executed. The decision is based on the validity of this insurance claim, which is known from the result of check insurance and contact garage. Then the token finally enters place o and the insurance claim process completes.

In particular, workflow cases (tokens) are processed independently, i.e. a task executed for some case cannot influence a task executed for another case. However, during the processing of a case there may be several tokens referring to the same case. For instance, if transition fork fires, then there are two tokens, one in p1 and one in p2, referring to the same claim.

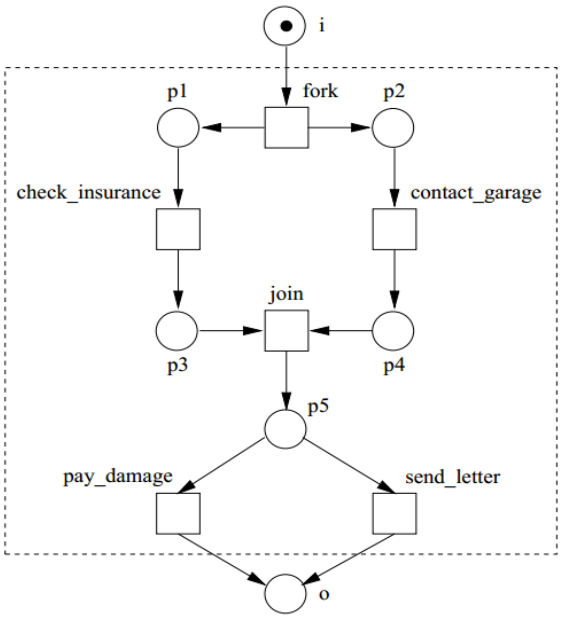

Figure 3.4 An insurance claim process modeled using Petri net

Extensions

In the last two decades the classic Petri net has been extended with colour, time and hierarchy [6]. These extensions enable the modelling of more complex processes especially where data and time are important factors.

- *Coloured Petri Nets (CPNs)*

Each token has attached a data value called the token colour. The token colours can be investigated and modified by the occurring transitions. With CPNs, it is possible to use data types and represent complex data manipulation.

- *Timed Petri Net (TPNs)*

A firing time to each transition is introduced. The firing rule is modified by considering transition's firing time. TPNs are normally used for performance evaluation.

- *Hierarchical Petri Net*

A subnet concept is added to handle the graphical and logical complexity of large workflow process definitions by modularization. The Petri net notation is simply extended by another special transition symbol that represents a particular subnet.

- *Workflow Nets (WF-Net)*

A WF-Net [4] is supposed to have a very regular structure: It must contain exactly one place with no incoming arcs and exactly one place with no outgoing arcs. Moreover, the net graph must be strongly connected, i.e. from each node there exists a directed path to any other node. The most important property of WF-Nets is soundness, which means that every maximal execution of the net which starts from the initial marking eventually leads to the final marking.

Table 3.1Advantages and Disadvantages of using Petri nets in WFM

Advantages	Disadvantages
Formal semantics	High complexity
Graphical representations	Not executable
High expressiveness	
Straightforward mapping onto WFM concepts	
Abundance of analysis techniques	

To sum up, Table 3.1 lists the main advantages and disadvantages of using Petri nets in the domain of WFM. As formalism, Petri nets provide formal notion but also graphical representation, which enables intuitive process design and abundance of analysis techniques. Petri nets also offer high expressiveness supporting all the primitives needed to model a workflow, including sequential, parallel, conditional and iterative structures. In addition, mapping Petri nets model onto WFM concepts is rather straightforward. The drawbacks of using Petri nets are also obvious. In the real world, Petri nets-based workflow definitions tend to become too large for design and analysis even for a modest-size system. Moreover, Petri nets are initially regarded as formalism rather than execution language. Due to the lack of available execution engines, it is hard to directly deploy and execute Petri net-based workflows.

3.4.2 WS-BPEL

In brief, the Web Services Business Process Execution Language (WS-BPEL) is a XML-based language for describing the behavior of business processes based on Web services. The first version, BPEL4WS 1.0, was originally submitted to OASIS WSBPEL Technical Committee by Microsoft and IBM in July 2002, which combined concepts from Microsoft's WSFL and IBM's XLANG. The latest version that was renamed as WS-BPEL 2.0 has been approved as an OASIS standard in 2007 [10].

The language gives its users the freedom to describe business processes in two ways: executable or abstract. An abstract process is a business protocol, specifying the message exchange behavior between different parties without

revealing the internal behavior for any one of them, while an executable process specifies the full implementation logic of the business process and is meant to be executed by an execution engine.

The WS-BPEL is also regarded as an Orchestration language other than a Choreography language. An Orchestration describes how services can interact with each other at the message level, including the business logic and execution order of the interactions from the perspective and under control of single endpoint, while choreography is typically associated with the public message exchanges, rules of interaction, and agreements that occur between multiple business process endpoints, rather than a specific business process that is executed by a single party. The choreographies can be described using the Web Services Choreography Description Language (WS-CDL).

Language structure

WS-BPEL language is based on XML format. The core elements of a WS-BPEL document are greatly influenced by web service concepts, and mainly include:

- roles of the process participants;
- port types required from the participants;
- orchestration, which is the actual process flow;
- correlation information, the definition of how messages can be routed to their corresponding instances.

The language constructs of BPEL are contained within a <process> construct that has a unique name and represents an executable process. The most important elements available in the language are:

1. Linking Partners

- *<partnerLinks>*

A BPEL process use <partnerLinks> to model conversational relationships with its partners. The relationship is established by specifying the roles of each party and the interfaces that each provides.

2. Process variables and data flow

- *<variables>*

It defines the data variables used by the process, providing their definitions in terms of WSDL message types, XML Schema types (simple or complex), or XML Schema elements. Variables allow processes to maintain state between message exchanges. In addition, using <assign> and <copy> message data

can be copied and manipulated between variables, which is discussed in Chapter 4.4.

3. Correlation

- *<correlationSets>*

Since there might be multiple business-process instances active at the same time in the BPEL Engine, all messages sent to the business process have to be transmitted to their corresponding business process instances. Correlation sets are used for this purpose.

4. Event, fault, and compensation handling

- *< eventHandlers >*

It specifies what to do when certain events happen within scope.

- *<faultHandlers>*

It provides fault handling functionality in WS-BPEL. It enables alternate execution path to be invoked for dealing with faulty conditions.

- *< compensationHandler >*

A business process often contains several nested transactions. The overall business transaction can fail or be cancelled after many enclosed transactions have already been processed. Then it may be necessary to reverse the effect obtained during process execution. WS-BPEL provides the capability to define compensation actions by defining compensation handlers.

5. Process main body

The process definition contains the process main body, which specifies activities and their relations. The process main body can contain multiple kinds of BPEL activities, like *<receive>*, *<reply>*, *<invoke>*, *<assign>*, *< sequence >*, *< scope >*, *< flow >*, etc.

Figure 3.5 shows the syntax structure of a WS-BPEL process.

```
<process name="NCName" targetNamespace="anyURI"
        queryLanguage="anyURI"?
        expressionLanguage ="anyURI"?
        suppressJoinFailure="yes|no"?
        exitOnStandardFault="yes|no"?
        xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">

      <extensions>?
            <extension namespace="anyURI" mustUnderstand="yes|no" />+
      </extensions>
```

```
<import namespace="anyURI"?
        location="anyURI"?
        importType="anyURI" />*

<partnerLinks>?
        <partnerLink name="NCName"
                partnerLinkType="QName"
                myRole="NCName"?
                partnerRole="NCName"?
                initializePartnerRole="yes|no"?>+
        </partnerLink>
</partnerLinks>

<messageExchanges>?
        <messageExchange name="NCName" />+
</messageExchanges>

<variables>?
        <variable name="BPELVariableName"
                messageType="QName"?
                type="QName"?
                element="QName"?>+
                from-spec?
        </variable>
</variables>

<correlationSets>?
        <correlationSet name="NCName" properties="QName-list" />+
       </correlationSets>

<faultHandlers>?
        <!-- faulthandler elements -->
</faultHandlers>

<eventHandlers>?
        <!--Eventhandler elements -->
</eventHandlers>
Activity
</process>
```

Figure 3.5 Syntax structure of a WS-BPEL process [10]

Table 3.2 summarizes the main advantages and disadvantages of WS-BPEL. WS-BPEL is the de-facto standard for implementing business processes based on web services and Service-Oriented Architecture (SOA). It properly supports both process modelling and execution. Many existing WFM products supporting WS-BPEL are ready to use. WS-BPEL's workflow-based structure is quite intuitively appealing for process designers. However, WS-BPEL is too tightly coupled with web services and the SOA environment. In addition, WS-BPEL has no formal notions and graphical representations.

Table 3.2Advantages and Disadvantages of using WS-BPEL

Advantages	Disadvantages
Easy deployment and execution	No graphical representation
Take advantages of Web Services and SOA	No formal notion
Block-structured	Tightly coupled with Web Services

3.4.3 Yet Another Workflow Language (YAWL)

Yet Another Workflow Language (YAWL) is a workflow modelling language based on so called Workflow Patterns. The language and its supporting system are developed and maintained by researchers at Eindhoven University of Technology and Queensland University of Technology [11].

The original goals of YAWL were to define a workflow language that would support all (or most) of the Workflow Patterns in [12] and that would have a formal semantics. The Workflow Patterns initiative [13] aims at establishing a more structured approach to the issue of the specification of control flow dependencies in workflow languages. Based on an analysis of existing workflow management systems and applications, this initiative identified a collection of patterns corresponding to typical control flow dependencies encountered in workflow specifications, and documented ways of capturing these dependencies in existing workflow languages. These patterns can be used as the benchmark to compare and evaluate various workflow languages. Figure 3.6 lists the main Workflow Patterns (control flow perspective) discussed in [12].

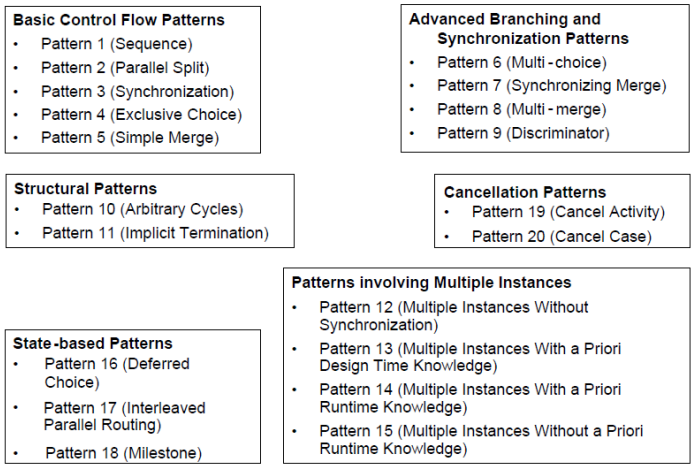

Figure 3.6 Overview of the 20 Workflow Patterns described in [12]

The related research revealed that Petri net support most of the Workflow Patterns [13]. Hence, YAWL language was initially developed using high-level Petri net as its starting point. Here the term high-level Petri nets refers to Petri nets extended with colour (i.e., data), time, and hierarchy. However, YAWL extends high-level Petri nets to overcome the limitation of its expressiveness in terms of control flow.

Just like Petri nets, YAWL has a formal foundation and also a graphical representation. Moreover, the designed YAWL workflow specifications can also be represented in XML format. YAWL has XML syntax and is specified in terms of an XML schema.

Last but not least, YAWL is not developed as a standalone language. It is supported by the YAWL system which provides the full implementation of a typical WFMS. Multiple components are contained in the YAWL architecture. The main components include a YAWL designer, a YAWL engine and a YAWL repository.

Language

YAWL is based on Petri nets. However, to overcome the limitations of Petri nets, YAWL has been extended with features to facilitate patterns involving multiple instances, advanced synchronization patterns, and cancellation patterns. Moreover, YAWL allows for hierarchical decomposition and handles arbitrarily complex data. The YAWL elements are shown in Figure 3.7.

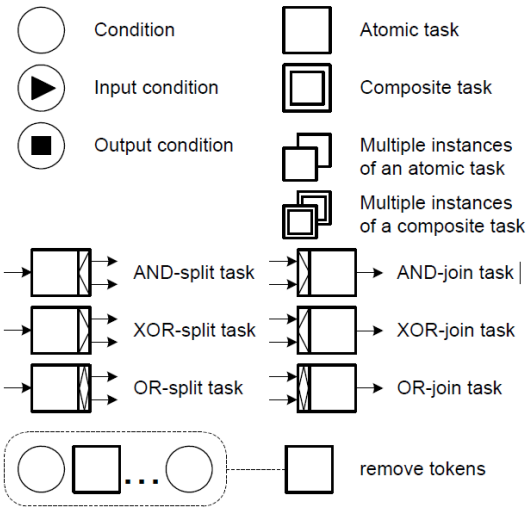

Figure 3.7 Symbols used in YAWL [13]

In brief, each YAWL process definition consists of tasks and conditions which can be interpreted as transitions and places in Petri nets, respectively.

Tasks are either atomic tasks or composite tasks. A composite task refers to a process definition at a lower level in the hierarchy, while an atomic task forms the leaves of the graph-like structure. This actually corresponds to the hierarchy extension to Petri nets.

Each process definition has one unique input condition and one unique output condition as the starting and end point, respectively.

Tasks and conditions are connected by directed arcs. However, unlike Petri nets, it is possible to connect 'transition-like objects' like composite and atomic tasks directly to each other without using a 'place-like object' (i.e., conditions) in-between. Compared to Petri nets, this can avoid unnecessary 'place-like objects' and decrease the size of the process definition.

Four special transition types are defined to express branching situations in a more compact way: AND-split, AND-join, XOR-split and XOR-join, each of them being associated with a graphical symbol. The former two types are used for modelling parallel behaviours (Pattern 2 and Pattern 3) while the latter two are used for modelling exclusive routing (Pattern 4 and Pattern 5). However, these special transitions are actually nothing more than shortcuts ("syntactic sugar") for an underlying equivalent in standard Petri nets.

OR-split and OR-join are introduced to be able to model Pattern 6 (Multi choice) and Pattern 7 (Synchronizing merge). In YAWL, both composite tasks and atomic tasks can have multiple instances (Patterns 12-15). In the case of multiple instances, it is possible to specify upper and lower bounds for the number of instances. And it is also possible to specify a threshold for completion that is lower than the actual number of instances. Finally, the symbol "remove tokens" shown in Figure 3.7 enables YAWL to effectively handle with cancellation patterns (Patterns 19 and 20).

Table 3.3Advantages and Disadvantages of YAWL

Advantages	Disadvantages
Inheriting the advantages of Petri nets	High learning curve for users
Highest expressiveness	Interaction with limited software components
Easy Deployment and Execution	
YAWL system behind	
Data manipulation support	

Table 3.3 reveals the main advantages and disadvantages of YAWL. YAWL extends high-level Petri nets and offers comprehensive support for the control-flow Workflow Patterns. It inherits the advantages of Petri net. In contrast with high-level Petri net, it is developed as an execution language

with support of the YAWL engine. Particularly, YAWL offers full support for the data perspective by using XML-based standards, i.e. XPath and XQuery. However, YAWL has limited capability to interact with heterogeneous software components.

3.5 Data Perspective in Workflow Modelling Languages

This section investigates the data perspective in workflow modelling languages. We explain how the data is represented and manipulated in different workflow modelling languages.

Most of the workflow modelling languages were initially designed focusing on the control flow perspective. However, the data perspective (data modelling and handling) of workflow modelling language is usually ignored. Current security management applications are data-centric and data plays a vital role in these applications. Therefore, workflow modelling language is required to provide capabilities of handling data. In the remainder of this chapter, we study and investigate how data is modeled, represented and manipulated in three different workflow modelling languages, namely Coloured Petri Net, WS-BPEL and YAWL.

3.5.1 Coloured Petri Nets

Classic Petri nets are limited concerning data, since tokens are indistinguishable. In other words, only one kind of token exists in Petri nets and it is not possible to attach any data values and data types into the tokens. Data manipulation in a Petri net-based WFMS is thus impossible. In order to solve this issue, Coloured Petri Nets (CPN) [14] have been introduced as an extension of classic Petri nets.

Compared with classic Petri nets, the most significant difference in CPN is that tokens are distinguishable. In other words, each token in a CPN has a data value called the token colour attached to it. These token colours can be inspected and modified by the occurring transitions. In addition to this, tokens are classified into different types according to their attached data, and each place has an associated type determining the kind of data that the place may contain.

Moreover, CPN also adds a collection of extensions to the arc element in Petri nets in order to take advantage of typed tokens attached data values. Firstly, the arcs going out of a transition can have expressions specifying how to compute the values of tokens being produced by the transition. When a

transition is successfully fired, the expressions on its outgoing edges are evaluated to produce new tokens to feed into the place at the end of the edge. Secondly, incoming arcs of a transition can have conditions. The transition is only enabled when some set of tokens from the source places satisfy the full set of conditions for its incoming arcs.

Figure 3.8 depicts a simple example of a CPN. This CPN contains 3 places, two of them have the pair type <Int,String>, and one has type <Int>. The transition takes one token of the pair type, and one of the integer type, and it produces one token of the pair type. The arcs coming into the transition declare names for the elements of the token values ((V,S) and (W)), and the arc leaving the transition describes the function that generates the values for outgoing tokens ((V*W,S)). This small net starts with two tokens, (4,"Foo"), and (2). After transition firing, a token is produced that contains a value corresponding to the pair (8, "Foo") in the bottom place.

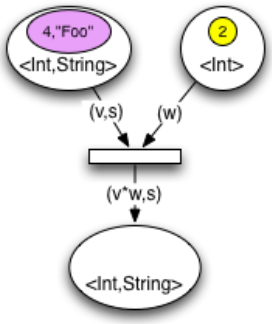

Figure 3.8 A simple example of Coloured Petri Net

In particular, CPN introduces a high-level programming language, namely Standard ML, to provide the primitives for the definition of data types and the manipulations of data values [14]. Standard ML is a functional programming language that supports multiple atomic data types (integers, reals, strings, booleans and enumerations) and structured data types (products, records, unions, lists and subsets). Arbitrary complex functions and operations can also be defined by using Standard ML. The computational power of Standard ML expressions is equivalent to lambda calculus (and hence to Turing machines) [15].

3.5.2 WS-BPEL

Due to the fact that WS-BPEL is completely XML-based, every piece of data in a WS-BPEL process is in XML forms. This includes the messages passed to and from the BPEL process, the messages exchanged with external services, and local variables used by the process.

WS-BPEL enables users to define variables. Variables provide the means for holding application data or messages that represent the state of a BPEL process. Variables are also used to hold the data that are needed to maintain the stateful behaviour or the process. A variable is associated with either an element defined in a schema, a simple XML Schemas type, or a message defined in a WSDL document. The syntax of the <variables> declaration is shown in Figure 3.9.

Data flow in WS-BPEL is not explicitly specified but can be realized using (globally) shared variables. In brief, <assign> activities are used to copy (parts of) variables to other variables or calculate the value of an expression and store it in a variable using XML data processing techniques, namely XPath and XSLT. Figure 3.9 shows the syntax of the <assign> declaration in WS-BPEL.

```
<assign standard-attributes>
    standard-elements
        <copy>+
            from-spec
            to-spec
        </copy>
</assign>
```

Figure 3.9 Syntax of the <assign> activity [10]

In general, the <assign> activity copies a type-compatible value from the source ("from-spec") to the destination ("to-spec"), using the <copy> element. By using XML query and expression languages (XPath and XSLT), it is also possible to construct and insert new data to the destination variables. Figure 3.10 shows a simple example. This <assign> activity copies part of the value of the variable “AutoLoanRequest” to update part of the value of the variable “InterstateCarLoanRequest”. In particular, the expressions “/creditRating/text()” and “/credit/text()” defined in the query attributes are XPath statements. These XPath expressions can select and retrieve the specific nodes in XML documents. For instance, the expression “/creditRating/text()” selects the text content from the root node “creditRating”.

```
<assign>
    <copy>
        <from variable="AutoLoanRequest"
              part="creditRating"
              query="/creditRating/text()"/>
         <to  variable="InterstateCarLoanRequest"
              part="credit"
              query="/credit/text()"/>
    </copy>
</assign>
```

Figure 3.10 Example of <assign> activity using XPath expressions

However, XPath is not only limited to offer the syntax for accessing parts of an XML document. With multiple defined operators, XPath enables users to implement basic mathematical and logical operations. Moreover, XPath provides users with a library of standard functions, including node set functions, string functions, boolean functions and number functions. These functions enables more complex data manipulations. WS-BPEL 2.0 supports XPath 1.0

XSLT can also be used for data manipulation in WS-BPEL. XSLT (Extensible Stylesheet Language Transformations) is a declarative, XML-based language used for the transformation of XML documents [16]. It is usually used to convert XML data into other data format, like HTML and XHTML. In addition to this, XSLT can also translate XML messages between different XML schemas, or make changes to documents within the scope of a single schema, for example, by removing the parts of a message that are not needed. Figure 3.11 shows how XSLT can be used in WS-BPEL.

```
<variables>
  <variable name="A" element="foo:AElement" />
  <variable name="B" element="bar:BElement" />
</variables>
...
<sequence>
  <invoke ... inputVariable="..." outputVariable="A" />
  <assign>
    <copy>
      <from>
        bpel:doXslTransform("urn:stylesheets:A2B.xsl", $A)
      </from>
      <to variable="B" />
    </copy>
  </assign>
  <invoke ... inputVariable="B" ... />
</sequence>
```

Figure 3.11 Complex document transformation in WS-BPEL using XSLT [10]

A common usage of XSLT in WS-BPEL processes involves receiving an XML document from one service, converting it to a different schema to form a new request message, and sending the new request to another service. Document conversion can be accomplished via the bpel:doXslTransform function. In the example of Figure 3.11, a service is invoked, and the result (foo:AElement) copied to variable A. The <assign> activity is used to transform the contents of variable A to bar:BElement, and copy the result of that transformation to variable B. Variable B is used to invoke another service. The style sheet A2B.xsl contains the XSL rules for converting documents of Schema foo:AElement to Schema bar:BElement.

3.5.3 YAWL

Although YAWL was initially designed with focus on the control flow, it has been extended to offer full support for the data perspective [13]. Compared to most of the existing workflow management systems which use a propriety language for dealing with data, YAWL completely relies on XML-based standards like XPath and XQuery. This is similar with data handling in WS-BPEL. XQuery is a query and functional programming language that is designed to query collections of XML data. It provides the means to extract, construct and manipulate XML data. In particular, XQuery uses the XPath expression syntax to address specific parts of an XML document. It supplements this with a SQL-like FLWOR expression. A FLWOR expression is constructed from the five clauses after which it is named: FOR, LET, WHERE, ORDER BY, RETURN. Figure 3.12 shows an example of XQuery expression.

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

Figure 3.12 XQuery expression

Like WS-BPEL, data are also represented as XML documents in YAWL. Two kinds of variables can be defined in YAWL, namely *net variables* and *task variables*. The former is used for storing data that need to be accessed and/or updated by tasks in a net, while the latter is used for storing data that needs to be accessed and/or updated only within the context of individual execution instances of a task. YAWL applies strong data typing. Data types are defined using XML Schema. Users can also write their own XML Schemas to define more complex data types. In YAWL, data usage is also part of variable definition. There are *input* and *output* variables, *input only* or *output only* variables, and *local* variables. In general, data are written to input variables and read from output variables. Local data usage is applicable to net variables only. The local (net) variables are used to store data that can be manipulated only internally within the scope of the corresponding net [17].

The example shown in Figure 3.13 defines the variable for storing the name of the customer. The type of this variable is string and an initial value is defined as “Tom”.

```
<rootNet id="make_trip">
   <localVariable
    name="customer">
    <type>xs:string</type>
    <initialValue>
       Tom
```

```
    </initialValue>
    </localVariable>
……
```

Figure 3.13 Variable definitions in YAWL

YAWL supports data passing between variables, which can be considered internal data transfer, and data interaction between a process and its operating environment (i.e. workflow engine users and web services), which can be considered external data transfer.

Internal data transfer is always conducted between nets and their tasks using XQuery. YAWL does not support direct data passing between tasks. Assume task A and task B in net N. To pass data from task A (e.g. variable Va) to task B (e.g. variable Vb), an appropriate net variable of N (e.g. Vn) must be available to convey data from Va to Vb. In YAWL, each task can be assigned an input parameter and/or an output parameter, which define internal data transfer associated with that task. Input Parameters use an XQuery to extract the required information from a net variable, and pass this information to the corresponding task variable, while output parameters define data passing in the opposite direction.

External data transfer does not apply to any local variable or any variable of a composite task. In YAWL, when data are required from the external environment at runtime, either a web form is generated requesting the data from the user or a web service is invoked that can provide the required data [17].

By using XQuery and XPath, YAWL supports data-based conditional routing. When tasks in YAWL have XOR or OR splits, the branch to choose is determined by conditions associated with branches. These conditions are boolean expressions that involve data within the process. The data may determine the evaluation results of the conditions and therefore influence the operation of the process.

In YAWL, the branching conditions are specified as XPath Boolean expressions in the flow detail for tasks with XOR or OR splits. The branches (flows) whose conditions (predicates) evaluate to true are executed by the YAWL engine. Figure 3.14 shows an example of data-based conditional routing in YAWL using XPath.

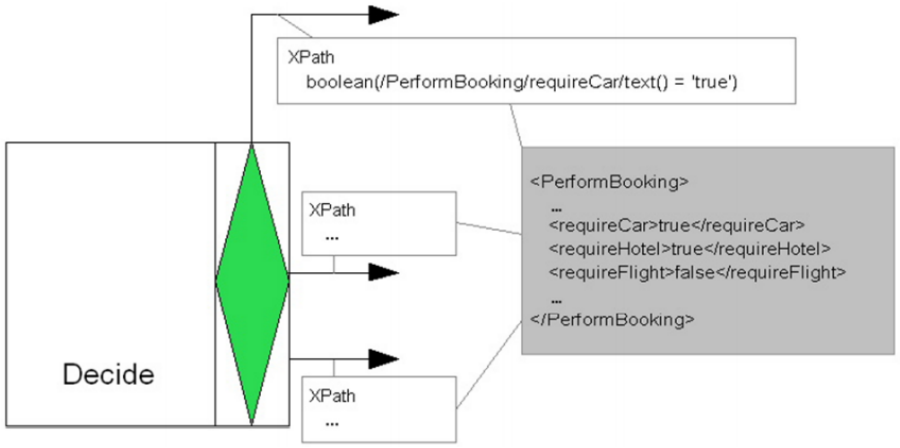

Figure 3.14 Example of data-based conditional routing [17]

As an example, Figure 3.14 shows the XPath expression "*/PerformBooking/requireCar/text() = 'true'*", which is specified at task "Decide", for choosing the branch of "Book Car" in the "PerformBooking" process. The corresponding branch is executed if the value of this XPath expression returns true.

3.6 Conclusion

In this chapter, we investigated the existing WFM theories and techniques. WFMS architecture was studied by identifying WFMS components and interfaces defined in the Workflow Reference Model. We also identified the basic WFM concepts. In addition, we distinguished between BPM and WFM by considering the BPM lifecycle.

Three popular workflow modelling languages (i.e. Petri net, WS-BPEL and YAWL) were discussed and evaluated. In particular, we investigated the data perspective in these languages.

Petri nets is widely used in the domain of WFM and it is also selected as our workflow modelling language in the FlinQ platform. Classic Petri net is limited concerning data. CPN extends classic Petri nets by attaching data values to the tokens and introducing a high-level programming language, namely Standard ML, to support data definition and manipulation. On the other hand, WS-BPEL and YAWL takes full advantage of XML techniques to represent and manipulate data. XML offers high flexibility, high extendibility and strong data types support in terms of data representation. Moreover, the XML data processing standards, like XPath and XQuery are able to support high-level data manipulation. Therefore, adopting XML technologies

provides a promising approach to improve the capabilities of data handling in Petri nets.

4. Petri nets-based Workflow Modelling Language

Petri nets has been selected as the workflow modelling language in the FlinQ platform. And a corresponding workflow engine that is based on a variant of classic Petri nets has also been built. However, as we discussed in Chapter 3, classic Petri nets is not capable of representing and manipulate data. In this chapter, we proposed our Petri nets-based workflow modelling language with the extensions for data handling. The language takes full advantage of XML techniques to represent and manipulate data.

This chapter is structured as follows: Section 4.1 presents the variant of Petri nets that is supported by the workflow engine in the FlinQ. Section 4.2 elaborates the proposed extensions for the data representation and manipulation. Finally, Section 4.3 defines and explains our XML-based workflow definitions.

4.1 Petri nets Elements Extensions

Before discussing our extensions for the data handling, we first briefly introduce the variant of traditional Petri nets that is supported by the existing workflow engine in the FlinQ platform. Our extensions for the data perspective of Petri nets are also based on this variant.

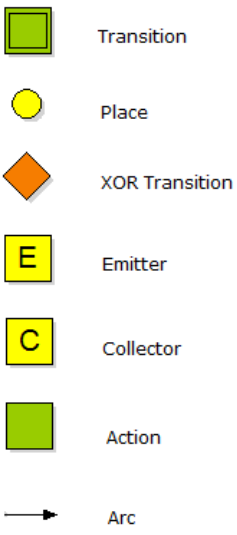

Figure 4.1 The basic elements of our Petri nets-based language

This variant reuses all the elements of Petri nets, including *transitions*, *places*, *arcs* and *tokens*. Moreover, this variant adds a collection of extensions to the elements in classic Petri nets, which is mainly inspired by WF-Nets [4] and YAWL [11]. These extensions focus on the improvement of Petri nets in the control-flow perspective. In the remainder of Section 4.1.1, we briefly discuss these extensions. Figure 4.1 shows the graphical representation of the elements in our modelling language.

(1) Transitions and Actions (Hierarchy structure support)

Transitions represent the tasks or activities that should be performed. Transitions must have zero or more *Actions*, which are atomic tasks (the smallest unit of work). This allows users to combine several smaller tasks as one transition to execute a more complex task. The benefit of using this hierarchical structure is that the size of Petri nets can be limited. *Actions* included in the same transition are executed sequentially. Figure 4.2 shows a simple example that illustrates the relationship between Transitions and Actions.

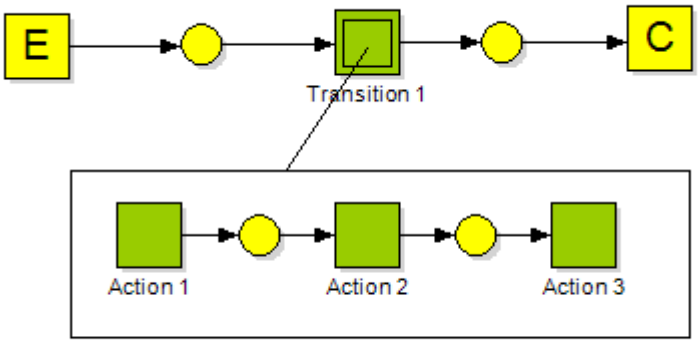

Figure 4.2 Relationships between Transitions and Actions

(2) Emitter and Collector transitions

Inspired by WF-Nets, we introduce the concepts of Emitter and Collector. The concepts of Emitter and Collector transitions are totally same with the input place and output place in WF-Nets, which explicitly indicates the start and end of the workflows.

An Emitter transition is the start of a workflow. It can be regarded as the source of tokens. When a workflow begins, its Emitter transition generates the tokens that represent workflow cases. Emitter transitions can not have input places.

A Collector transition is the end of a workflow. It is the sink for tokens and it consumes tokens like normal transitions. Collector transitions can not have output places.

(3) XOR transitions

XOR transitions are used to support conditional choices for branch structures. An XOR transition consumes a token from each input place, but generates a token for only one of its output places. Each XOR transition has two output places which are connected with one *Positive arc* and one *Negative arc*. XOR transitions can contains a condition, which explicitly defines an evaluation condition for choosing one of the workflow branches. In our language, these evaluation conditions are defined by XQuery statements that return boolean values. Figure 4.3 depicts a Petri net that contains an XOR transition.

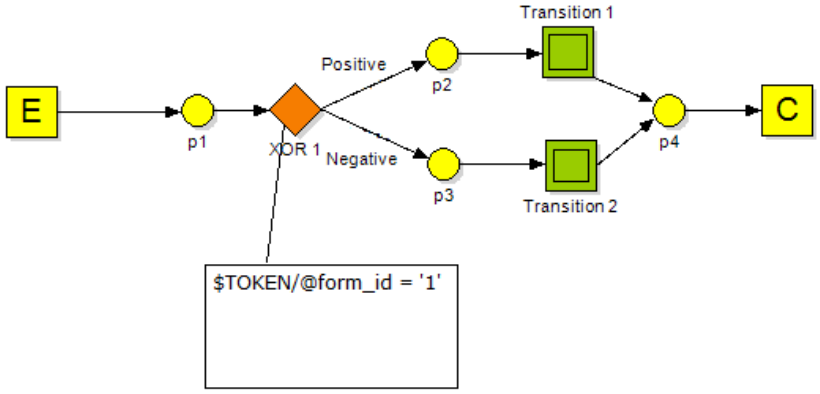

Figure 4.3 An example of XOR transition

(4) Criteria

This element is our extension for classic Petri nets. Each Petri net has *Criteria* to define the evaluation conditions to determinate if the token should be accepted and performed by this Petri nets. Like XOR transition, the evaluation conditions in Criteria are also specified by XQuery statements which return boolean values.

In practice, multiple Petri nets could be deployed and running in the workflow engine, representing different workflows. When a new token (i.e. workflow case) is generated, only some of workflows are responsible for handling it. By introducing the concept of Criteria, we ensure that each Petri net (workflow) only accept and handle the tokens (workflow cases) that it is responsible for.

(5) Reset and Block Arcs

These two existing extensions for Petri nets are also adopted in our language. Reset arcs enable Petri net to model cancellation. When the transition at the

end of a Reset arc fires, all tokens that occupy the place at the start of this Reset arc are removed. Reset arcs are represented by arrows with dashed lines. Block arcs are also referred to as inhibitor arcs in some literature [24]. A Block arc only connects a place to a transition. The transition that is connected with a Block arc only fire when the place is empty. This actually specifies a constraint on transitions' execution. Block arcs are represented by a bold point at the transition with a solid line to the place.

Like classic Petri nets, mapping WFM concepts onto our workflow modelling language is rather straightforward. Tasks are modelled by transitions or actions. Places explicitly model the workflows' states. Workflow cases (instances) are modelled by tokens.

4.2 Data Representation and Manipulation

Our extensions for Petri nets mainly focus on improving the capabilities of data representation and manipulation. The extensions are inspired by some ideas of Coloured Petri Nets (CPN) [14]. However, our language conducts data manipulation by totally relying on XML-based data processing standard instead of Standard ML, namely XQuery (and XPath). The extensions include:

(1) XML data representation

CPN allows tokens to contain different data (colours). In our extensions, these data are represented in XML forms. These XML data are attached to the corresponding workflow cases (tokens) in their entire life cycle (from Emitter to Collector).

(2) Explicit specifications for token data manipulation by XQuery

In CPN, the arcs going out of a transition can have expressions specifying how to compute the values of tokens being produced by the transition.

Our language also allows users to explicitly specify the actions for manipulating data within workflow definitions. Specifically, *actions* can contain XQuery statements that manipulate the XML data attached to tokens. In these defined XQuery statements, the symbol *$TOKEN* is used to represent the current XML data values of the tokens. When these actions fire, the data values of the tokens are changed. The results of defined XQuery statements are assigned to the tokens as the new data values.

Moreover, the data manipulations defined in *actions* are not limited to the token data. They can also involve other XML data in a known data source (e.g. an XML database). For example, the following XQuery statements can be defined in an action of our Petri nets. It retrieves the <*device*> elements of

which the *type* attribute equals the *device_type* defined in the attribute of the token (*$TOKEN/@ device_type*). Then we reconstruct the new token: add the <*newtoken*> as the root node and put the retrieved <*device*> elements inside. The resulting XML data of the entire XQuery expression are assigned to the token as its new data values.

```
<newtoken>
{doc("devices.xml")// device[@type= $TOKEN/@device_type]}
</newtoken>
```

Figure 4.4 XQuery statements for token data manipulation

(3) Branching conditions defined by XQuery

The evaluation conditions for the *criterias* and *XOR transitions* are also defined by XQuery statements. These XQuery statements are boolean expressions that return true or false. Again, consider the example shown in Figure 4.3. The XOR transition specifies the XQuery statements *$TOKEN/@form_id='1'* for its condition. This condition checks if the value of the token attribute *form_id* is *1* or not.

(4) Update and management of the persistent XML data

The data manipulations in our language are not limited to modifying the data values of the tokens. It is allowed to define the *actions* that are responsible for updating and managing the XML data in persistent data stores.

However, the XQuery standard [26] does not offer the facilities of creating and updating the persistent XML data. Some XQuery update extensions are thus established, including XQuery Update Facility 1.0 [33] and the XQuery update language proposed by [32]. These update extensions can be used in our language to define the actions for managing the persistent XML data. Figure 4.5 shows an example of the XQuery update language [32] statements defined within the Petri nets' action. The statement inserts a new <*state*> node into the selected elements of the *SiteData* XML document. And it does not modify the token data.

```
UPDATE INSERT
      <state name="{string($TOKEN/@event_type)}" />
INTO doc('SiteData')//Device[@id=string($TOKEN/@internal_id)]
```

Figure 4.5 XQuery update statements for updating persistent data

(5) XML Schema definitions and validation in places and criteria (optional)

In CPN, each place specifies *colour set* (the data type of tokens) which is allowed to reside on this place. In our language, each place can also specify the allowed XML data structures and types by XML Schema [25]. Places cannot accept the XML data that do not meet the defined XML Schemas. In the case that the XML data is not valid for the place, the corresponding

workflow cases are terminated. Figure 4.6 shows a simple example where each place (*P1* and *P2*) specify its XML Schema. In this workflow, Transition *T1* adds one more element <*time*> that stores the current time to the token. The schema in the place *P2* also adds the corresponding element to ensure that the new token produced by *T1* is valid. Similarly, the *criterias* in our Petri nets can also use XML Schemas for the XML data validation.

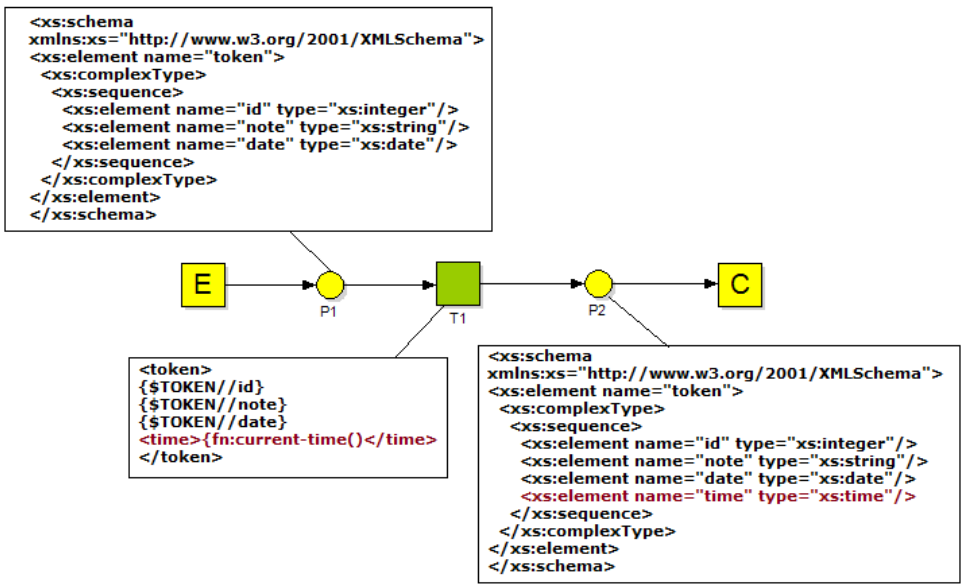

Figure 4.6 XML Schema specifications at places

The data validation at each place ensures the validity of the token data. Moreover, by specifying the XML data structure and types at each place, users can clearly aware how the XML data look like at each step of the workflows. Also, it is more convenient for the workflow designers to define the data manipulation actions based on these predefined XML Schemas.

However, for the cases where the XML data are in a uniform structure or the flexibility of data structures and types is required, the data validation at each place is not necessary. Therefore, our language proposes this feature as an optional extension. By default, places can accept any XML data.

In addition, our extensions also take advantages of XML Schema for strong data types support. The XQuery standard [26] already includes the primitive data types (e.g. string, integer, date) defined in XML Schema [25]. Moreover, the XQuery standard allows users to import the external XML Schema definitions for supporting the user-defined data types.

Figure 4.7 depicts an example of how data is represented and manipulated in our Petri nets. This example shows a simplified workflow for the login in the FlinQ platform.

When users log in FlinQ platform with their username and password, the client application will send a message to the server. In practice, this actually generates a new workflow case (i.e. token). And the data sent to the server will be wrapped in XML forms and attached to this workflow case. We assume the user logs in with the username "*admin*" and the password "*admin*". Therefore, the following XML data will be attached to this workflow case. And the entire XML data of the input token is shown in Figure 4.8.

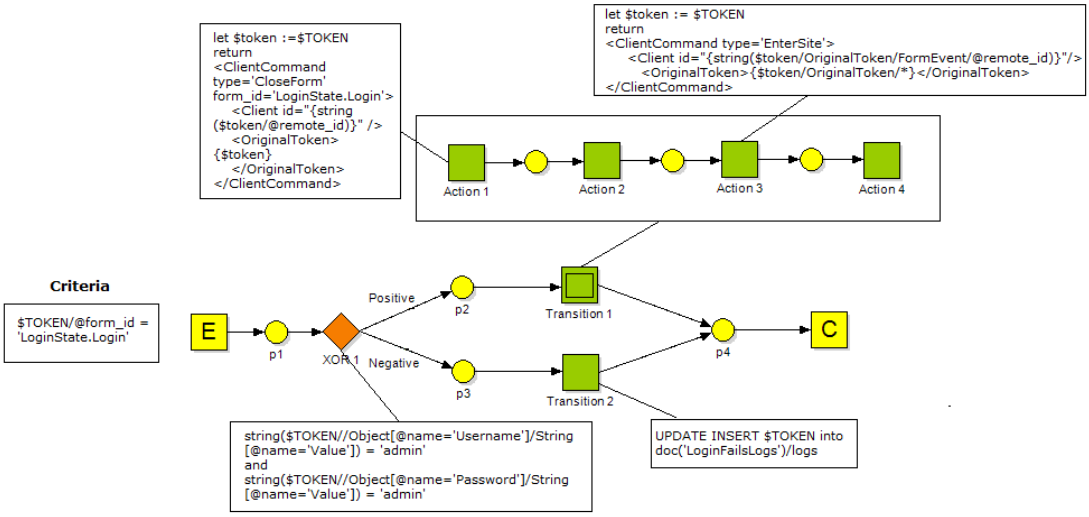

Figure 4.7 Data representation and manipulation in Petri nets

First of all, the *criteria* evaluates the input token data based the evaluation condition: *$TOKEN/@form_id = 'LoginState.Login'* . The XQuery expression checks if the value of the attribute *form_id* in XML data's root element is *LoginState.Login* or not. The evaluation condition returns true. Therefore, this new token is accepted by this Petri net, which means the corresponding workflow case is handled by this Petri net. Otherwise, this token will be ignored by this Petri net.

```
<FormEvent action_id="Login" form_id="LoginState.Login" remote_id="2">
  <FormData>
    <json>
      <String name="Action">Login</String>
      <String name="Form">LoginState.Login</String>
      <Object name="Properties">
        <Object name="Password">
          <String name="Name">Password</String>
          <String name="Type">Text</String>
          <String name="Value">admin</String>
        </Object>
        <Object name="Username">
          <String name="Name">Username</String>
          <String name="Type">Text</String>
```

```
            <String name="Value">admin</String>
          </Object>
        </Object>
      </json>
    </FormData>
</FormEvent>
```

Figure 4.8 The input token data for login

This token is then emitted by the Emitter transition and occupies the place *p1*. The XOR transition *XOR 1* is thus enabled. *XOR 1* contains an evaluation condition that is defined as Figure 4.9. The XQuery statements check the values of username and password. Only if the username is "admin" and the password is "admin", the condition returns true. Again, the evaluation condition is true for this token. The positive branch is selected and *Transition 1* will be performed.

```
string($TOKEN//Object[@name='Username']/String[@name='Value']) = 'admin'
and
string($TOKEN//Object[@name='Password']/String[@name='Value']) = 'admin'
```

Figure 4.9 XQuery statements in XOR 1

At this moment, the XML data in this token has not been modified yet. Data modifications are conducted by the *actions* that defined by XQuery statements. In the example, Transition 1 consists of four actions. *Action 1* and *Action 3* contain XQuery statements. When these two actions fires, their defined XQuery statements are also be executed to manipulate XML data in token. However, other actions with no XQuery statements do nothing with the XML data of the tokens. Figure 4.10 and 4.11 shows the XQuery statements defined in *Action 1* and *Action 3*, respectively.

```
let $token := $TOKEN
return
<ClientCommand type='CloseForm' form_id='LoginState.Login'>
    <Client id="{string($token/@remote_id)}" />
    <OriginalToken>{$token}</OriginalToken>
</ClientCommand>
```

Figure 4.10 XQuery statements in Action 1

```
let $token := $TOKEN
return
<ClientCommand type='EnterSite'>
    <Client id="{string($token/OriginalToken/FormEvent/@remote_id)}"/>
    <OriginalToken>{$token/OriginalToken/*}</OriginalToken>
</ClientCommand>
```

Figure 4.11 XQuery statements in Action 3

Action 1 actually reconstructs the XML data of this token for sending ClientCommand: *CloseForm*. *Action 2* then uses this reconstructed XML snippets to send the *CloseForm* command to the client application. However,

Action 2 does not change the XML data of the token. Next, *Action 3* reconstructs XML data again for sending ClientCommand: *EnterSite*. *Action 4* then uses this reconstructed XML snippets to send the *EnterSite* command to the client application. Again, *Action 4* does not change the token data.

When the login request fails (i.e. the negative path is selected), Transition 2 is performed. This transition contains only one *action* that specifies the operation for updating the persistent XML document (*doc('LoginFailsLogs')*). The action is defined by XQuery update language (Figure 4.12) and it simply stores the current token data into the log file for the failed login request. Notice that this action does not modify the token data values.

```
UPDATE INSERT $TOKEN into doc('LoginFailsLogs')/logs
```

Figure 4.12 XQuery update statements in Transition 2

Finally, the Collector transition consumes this token and the entire workflow is completed.

4.3 XML-based Workflow Definitions

The workflows modeled by our language can also be represented and saved in XML format. Firstly, like many executable language (e.g. WS-BPEL and YAWL), these XML-based workflow definitions with the uniform XML syntax can be easily deployed and executed by the workflow engine. Moreover, XML-based workflow definitions can be used as interchange documentations among the software developers, service officers and our customers.

Our XML-based workflow definitions have a uniform syntax that is specified in terms of an XML Schema. Appendix 1 shows this XML Schema definition.

Our XML-based workflow definitions include the structure of Petri nets, namely the control flow perspective of workflows. Moreover, they also contain the information about data definition and manipulation, namely XQuery statements.

Figure 4.13 depicts an example of XML-based workflow definitions that is the login workflow we discussed in Section 4.2. For each Petri net workflow, there is a root element <*PetriNet*> to start with. The elements <*Criteria*>, <*Transtion*>, <*Place*>, <*Arc*> can be defined as child elements with the scope of <*PetriNet*>.

The element <*Criteria*> with an attribute *type* is used to define the criteria of Petri nets. Its text contains XQuery statements.

The element <*Transition*> has an attribute *type* to declare the type of this transition that could be EMITTER, XOR, DEFAULT and COLLECTOR. The element <*Transition*> can contain child elements <*Action*>. In particular, The XOR <*Transition*> elements can contain child element <*Condition*>. The text of <*Condition*> elements is used to specify XQuery statements for the evolution condition.

The element <*Action*> has an attribute *type*. Only the <*Action*> elements with "*SednaQueryAction*" type can contain XQuery statements which are responsible for XML data manipulation. Other <*Action*> types, like "*SendCommandToClientAction*" and "*SendCommandToConnectorAction*", play no role for data manipulation and should not contain any XQuery statements.

The elements <*Transition*> and <*Place*> has an attribute *id*. Each Transition or Place should have a unique id number in the certain Petri net.

Finally, the <*Arc*> elements specify its source nodes and destination nodes. Its attribute *type* is used to declare the type of this arc that could be either POSITIVE or NEGATIVE.

```
<?xml version="1.0" encoding="utf-8"?>
<PetriNet  xsi:noNamespaceSchemaLocation="Petrinets.xsd" xmlns=""
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Criteria type="SednaCriteria">
    $TOKEN/@form_id = 'LoginState.Login'
 </Criteria>

 <Transition id="1" type="EMITTER" />
 <Transition id="2" type="XOR">
  <Condition type="SednaCondition">
   <![CDATA[
    string($TOKEN//Object[@name='Username']/String[@name='Value'])='admin'
    and
    string($TOKEN//Object[@name='Password']/String[@name='Value'])='admin'
    ]]>
  </Condition>
 </Transition>
 <Transition id="3" type="DEFAULT">
  <Action type="SednaQueryAction">
   <![CDATA[
   let $token := $TOKEN
    return
       <ClientCommand type='CloseForm' form_id='LoginState.Login'>
        <Client id="{string($token/@remote_id)}" />
        <OriginalToken>{$token}</OriginalToken>
       </ClientCommand>
     ]]>
  </Action>
  <Action type="SendCommandToClientAction" />
```

```
  <Action type="SednaQueryAction">
   <![CDATA[
    let $token := $TOKEN
    return
       <ClientCommand type='EnterSite'>
        <Client id="{string($token/OriginalToken/FormEvent/@remote_id)}"/>
        <OriginalToken>{$token/OriginalToken/*}</OriginalToken>
       </ClientCommand>
   ]]>
  </Action>
  <Action type="SendCommandToClientAction" />
 </Transition>
 <Transition id="4" type="DEFAULT">
  <Action type="SednaQueryAction">
   <![CDATA[
      UPDATE INSERT $TOKEN into doc('LoginFailsLogs')/logs
    ]]>
  </Action>
 </Transition>
 <Transition id="5" type="COLLECTOR" />

 <Place id="6" />
 <Place id="7"/>
 <Place id="8"/>
 <Place id="9"/>
 <Arc fromId="1" toId="6" type="POSITIVE"/>
 <Arc fromId="6" toId="2" type=" POSITIVE "/>
 <Arc fromId="2" toId="7" type=" POSITIVE "/>
 <Arc fromId="2" toId="8" type="NEGATIVE"/>
 <Arc fromId="7" toId="3" type=" POSITIVE "/>
 <Arc fromId="8" toId="4" type=" POSITIVE "/>
 <Arc fromId="3" toId="9" type=" POSITIVE "/>
 <Arc fromId="4" toId="9" type=" POSITIVE "/>
 <Arc fromId="9" toId="5" type=" POSITIVE "/>
</PetriNet>
```

Figure 4.13 Example of XML-based workflow definition

5. Design of the WFM Framework

This chapter reports on the design of the WFM framework in the FlinQ security management platform. Section 5.1 presents the requirements of our WFM framework. Section 5.2 proposes the overview of the WFM framework design. Section 5.3, 5.4 and 5.5 elaborate the design of three main components in the workflow engine.

5.1 Requirements

Our WFM framework should provide generic capabilities to address the issues and challenges of WFM in a concrete security management application, the FlinQ platform. The goal of this section is to identify and clarify the essential requirements for the WFM framework design to cover these WFM capabilities.

- *Workflow Modelling*

Workflow modelling is a dominant factor in WFM and also one of the most important functionalities our WFM solution should provide. Workflow modelling is usually supported by workflow modelling languages and conducted during the *process design* phase. Particularly, our workflow modelling language is required to be able to model both control and data flows.

In our WFM solution, workflow modelling is supported by a Petri nets-based language that is discussed in Chapter 4. Inheriting the high expressiveness of Petri nets, our language is capable of modelling most of the control flow patterns, including sequential, parallel, conditional and iterative structures. By using XML and XML data processing technology, our language is also able to model and manipulate data.

In addition to a modelling language, a process design tool (i.e. workflow editor) is required to provide users with an intuitive graphical design environment. This editor should support the Petri nets-based language we proposed. Moreover, the workflows defined by the workflow editor should be exported and saved as XML-based workflow definitions we discussed in Section 4.3.

- *Workflow Enactment*

This is the key functionality that our WFM framework should support. Based on the *Workflow Reference Model*, the core of any workflow system is the *workflow enactment service*. A workflow enactment service provides the run-time environment that takes care of the control and execution of the *case-based* workflows. With a workflow enactment service and the defined workflows, highly automated business processes can be deployed with a minimum of human interaction involved.

In our solution, the workflow enactment service should be supported by a Petri nets-based workflow engine. First of all, this workflow engine should be capable of executing the workflows defined in our Petri nets-based modelling language. In general, this workflow engine should have two basic capabilities: *workflow definitions interpretation* and *workflow execution*.

The desired workflow engine should be able to interpret and validate our XML-based workflow definitions. When these workflow definitions are deployed, the engine is expected to check the validity of workflows. If the workflows are valid, engine is supposed to parse these workflow definitions, which are represented in XML format, according to their corresponding XML Schema.

The core function of workflow enactment service is workflow execution. The desired workflow engine has to be capable of executing our workflow definitions in a *case-based* way. Specifically, it is responsible for initializing and managing the execution of those workflow cases.

- *Invoke applications of the FlinQ server*

Some tasks defined in workflows need to invoke external applications to accomplish business processes. According to the Workflow Reference Model, a workflow engine should provide sufficient logic to understand how to invoke potential external applications which might exist in a heterogeneous environment.

However, in the FlinQ platform, the desired workflow engine only needs to deal with restrictive range of applications. Workflow tasks are mainly used to interact with connectors and client applications from the FlinQ server side. FlinQ server interacts with connectors and client applications by sending XML-based *command* messages. Therefore, our workflow engine is required to invoke the FlinQ server applications that send *command* messages to connectors and client applications.

- *XQuery support*

Our Petri nets-based language is capable of manipulating XML data. This totally relies on a XML data processing standard, namely XQuery. Our WFM

framework should address the problem of integrating XML data processing technology in a workflow execution environment.

Specifically, our workflow engine needs an XQuery processor to perform the XQuery statements defined in our modelling language. The desired XQuery processor should support XQuery 1.0 [26]. Since XQuery 1.0 is an extension of XPath 2.0 [27], the XQuery processor should also support XPath 2.0.

In addition, standard XQuery does not provide any update facilities to change, add and delete the XML data stored in the persistent database. Our XQuery processor should be extended with update facilities.

XQuery (both XQuery 1.0 and XQuery 3.0) was developed by the XML Query working group of the W3C as a standardized language. Different tools support the implementation of XQuery. Some examples include Zorba XQuery processor [44] and XQilla [45].

- *XML Database support*

Database systems facilitate the realization of WFM in several ways. They can provide the necessary functionality to keep the workflow relevant data, as well as application data [28]. Our WFM framework is required to have databases support.

One potential purpose of the database in our WFM framework is to store and manage the workflow case data. Case data is referred to as the data elements that are specific to a process instance or case of a workflow [22]. In our Petri nets-based language, case data is attached to tokens in XML form. These case data could be stored in the database.

Besides the workflow case data, some information related to the workflow execution can be stored in the database as well. Again, consider the workflow for login that we discuss in Section 4.2. In the example showed in Figure 4.4, only the user who logs in with the username *admin* and the password *admin* can successfully log into the system. However, in practice there should be multiple valid registered users and this workflow is expected to check if there is one user information record that has the same username and password with the input ones. The database can be used to store information about the registered users.

Since the data handled in our WFM framework is mainly wrapped in XML format, the desired database should to allow data to be stored in XML format. A possible approach is to store XML documents as BLOBS (Binary Large Objects) in a classic relational database, and to use the management and query tools of that database. Another approach is to use native XML databases. The internal model of native XML databases totally depends on XML and uses

XML documents as the fundamental unit of storage. Native XML databases are expected to maintain and manage XML data more efficiently.

5.2 Framework Overview

Based on the requirements discussed in Section 5.1, we identify the necessary components of our WFM framework and the interactions between these components.

Figure 5.1 depicts a high level overview of the WFM framework in the FlinQ security management platform. Each component is discussed below.

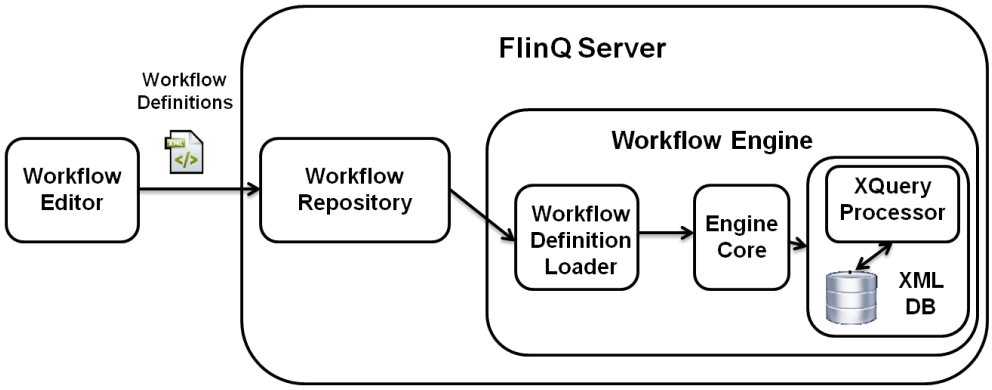

Figure 5.1 Overview of the WFM framework in FlinQ

5.2.1 Workflow editor

The workflow editor is the process definition tool in our WFM framework. It provides a graphical design environment for users to model workflows based on our Petri nets-based modelling language. The design includes both the structure of Petri nets (the control flow perspective) and XQuery statements (the data perspective).

In addition to workflow design, the editor provides basic workflow verification and simulation functionalities. The verification functionality is used to check the correctness of the workflows. This includes the validity of the Petri nets and the XQuery statements. The simulation functionality allows users to play the so called token game. Users can assign any XML data value to a token. The simulation executes these tokens based on defined Petri nets and manipulates XML data based on defined XQuery statements. The user can check the XML data value of future tokens in an each step.

An XQuery processor and native XML database in the editor are used for verification and simulation, which includes manipulating and managing XML data.

Currently, the workflow editor has not been implemented.

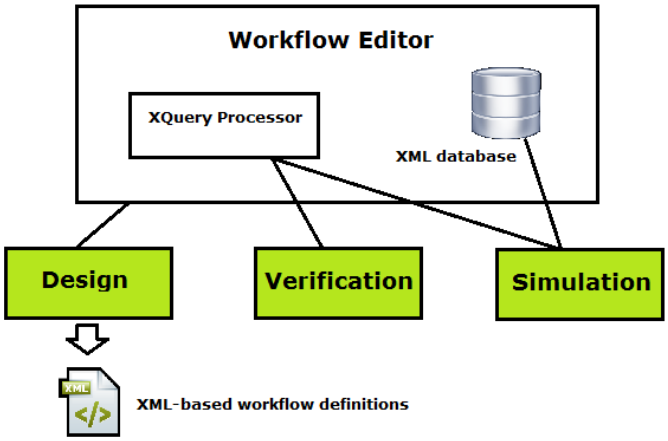

Figure 5.2 Workflow Editor

When a workflow definition is ready and verified, the workflow could be output to a workflow definition file, which is an XML representation conforming to a predefined XML Schema (shown in Appendix A). These XML-based workflow definitions can be deployed and executed by the workflow engine.

5.2.2 Workflow repository

XML-based workflow definitions need to be deployed in the workflow engine to be executed. This is done by storing them in our workflow repository. When FlinQ server starts up, the workflow engine loads all the workflow definitions stored in the workflow repository.

In addition, some configuration XML files are also stored in this repository. These files are mainly responsible for the initialization of the XML database, including creating XML documents and collections, and storing and loading the XQuery modules.

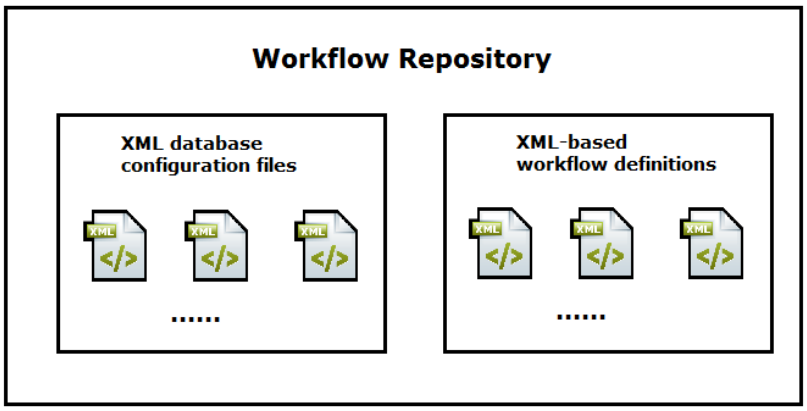

Figure 5.3 Workflow Repository

5.2.3 Workflow engine

This is the core component in our WFM framework. Our Workflow engine is tightly embedded into the FlinQ server. In general, a workflow engine consists of four components: workflow engine core, workflow definition loader, XQuery processor and native XML database. Figure 5.4 depicts an overview of workflow engine. Each component is discussed below.

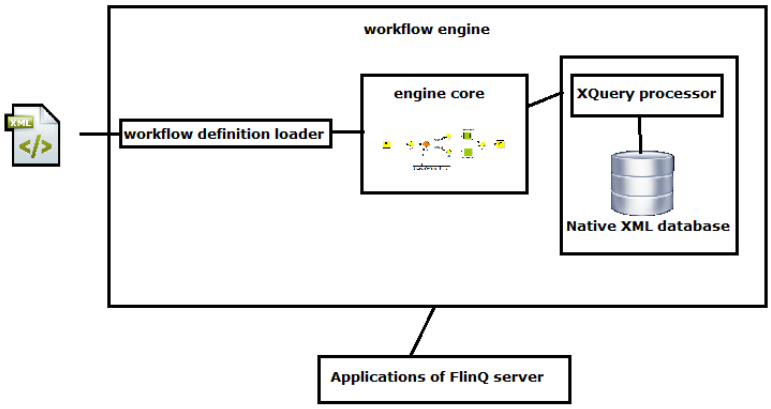

Figure 5.4 workflow enigine overview

(1) Workflow engine core

Workflow engine core provides typical workflow enactment service. It is responsible for executing workflows that are defined in our Petri nets-based modelling language.

After the XML-based workflow definitions are successfully loaded into the engine, the workflow engine core instantiates these definitions. Afterwards,

the workflow engine core manages the execution of the workflow cases. It initializes workflow cases, chooses the workflows (Petri nets) which is responsible for a certain cases according to Petri nets' *criteria*, executes each case according to its current state and control flow description (Petri nets' structure), and performs tasks defined in the workflow definitions.

(2) Workflow definitions loader

The workflow definitions loader plays the role of the *Interface 1*defined in the Workflow Reference Model shown in Figure 3.2. It parses XML-based workflow definitions stored in the workflow repository and instantiates workflows in the engine core.

(3) XQuery processor

An XQuery processor is embedded into our workflow engine to execute XQuery statements. It provides the capability of XML data processing within the workflow execution environment.

(4) Native XML database

A native XML database plays a role as a persistent XML data store in our WFM framework. Stored XML data could be workflow relevant data, like workflow case data and workflow execution logs. However, this database can also be used to store some application data for the FlinQ platform, like registered client users' information, connectors' information and event records.

In addition, our native XML database uses XQuery and its update extension as query language. The XML database works in cooperation with the XQuery processor to provide XML data management services for our WFM framework.

(5) FlinQ server applications

Our workflow engine provides the mechanisms to access applications of FlinQ server which should be involved during workflow execution. This works as the *Interface 3* defined in the Workflow Reference Model in Figure 3.2.

In the remainder of this chapter, we elaborate on the design of the components of our workflow engine.

5.3 Workflow Engine Core

As mentioned in Section 2.3, the FlinQ server interacts with all client applications and connectors by sending them *command* messages. When any events happen in the client applications (e.g. user clicks the button for login) or the connectors (e.g. sensors detect somebody enters the room), they send an *event* message to the FlinQ server.

In our WFM design, each received *event* message is regarded as a workflow case. The FlinQ server wraps these event messages as token objects and puts them in a so called *token queue*. The content of an event message is directly assigned to a token as its attached XML data. The workflow engine core takes the tokens from this token queue and determinates which Petri nets (workflows) are responsible for executing these tokens according to the defined *criteria* of the corresponding Petri net. Tokens can be accepted by zero or more Petri nets. Figure 5.5 shows the process of workflow cases initialization.

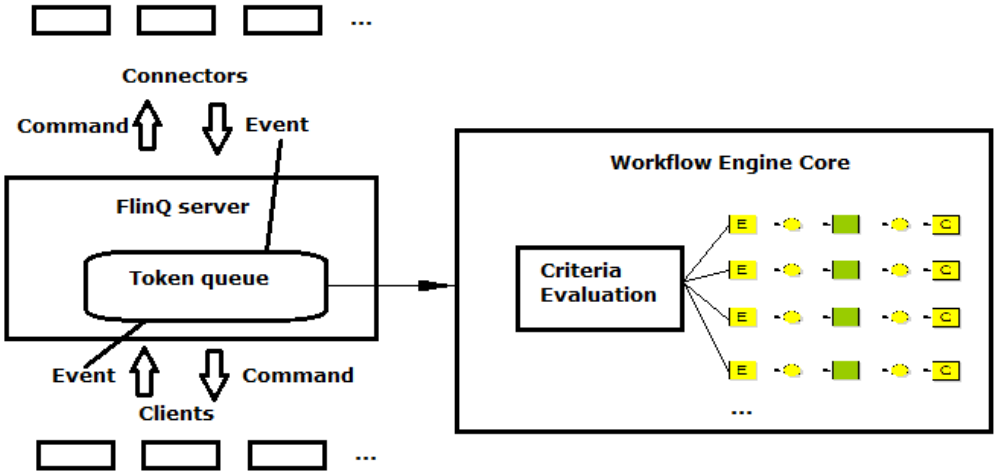

Figure 5.5 The creation of workflow cases

The workflow engine core has been designed totally based on our extended Petri nets model. Figure 5.6 shows simplified class diagram with a high level description of the workflow engine core. The class diagram reveals the high level overview of the workflow engine core design. Below we briefly explain each class in Figure 5.6.

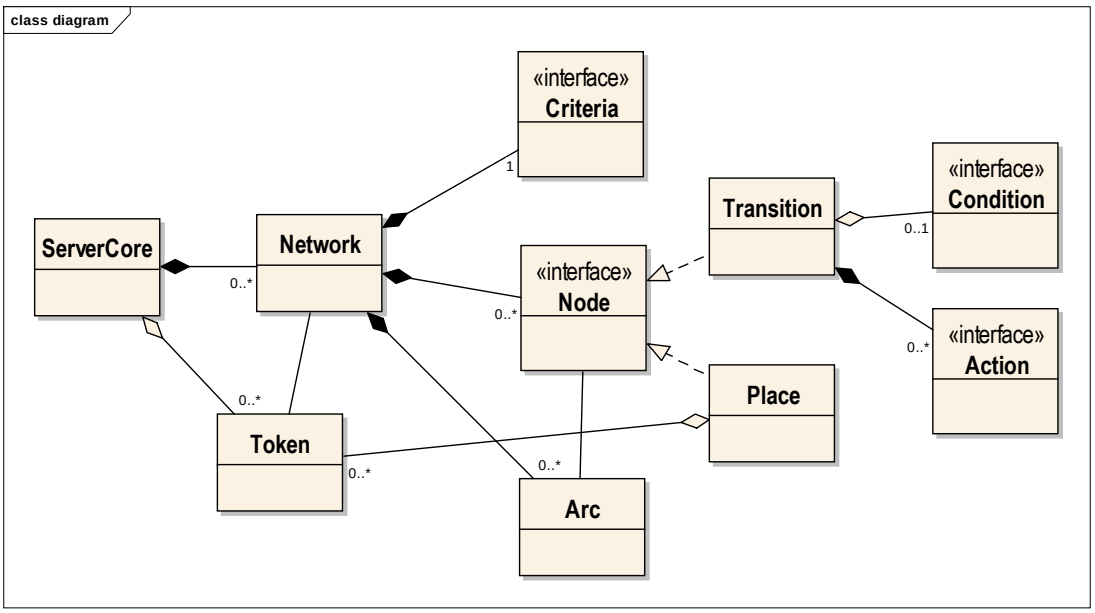

Figure 5.6 Simplified class diagram for the workflow engine core

- *ServerCore*

ServerCore is actually also part of the FlinQ server. By introducing this class, our workflow engine core is tightly integrated with the FlinQ server. At the top of the class hierarchy, ServerCore manages all deployed Petri nets and the tokens to be executed. Therefore, it has the attributes like *deployedPetrinets* and *tokensQueue*. And it also has operations to add and remove the deployed Petri nets and the tokens to the token queue.

- *Network*

Network represents to the concept of Petri net in our workflow modelling language. It consists of one Criteria, a set of nodes (transition and places) and arcs. Network can accept the tokens which meet its Criteria. The accepted tokens are consumed by invoking Network's operation, which implements the core algorithm of Petri nets and conducts the workflow execution.

- *Token*

Token corresponds to the workflow cases. Each Token object is firstly generated and stored in ServerCore's *tokensQueue*. A token can be accepted by zero to multiple Networks for execution. Token has a *data* attribute to store its attached data.

- *Criteria*

Each Network object can have exactly one object of class Criteria. Criteria has a member function *appliesTo(Token * token)* which returns boolean values. Criteria is declared as an interface.

- *Node*

Node is defined as an interface and it can be implemented as Transition or Place. A Node object has a *nodeID* attribute. Each node in a Petri net has a unique *nodeID*.

- *Arc*

An Arc object has references to its source and destination Node objects, represented by the attributes *fromNode* and *toNode*. Moreover, an Arc object specifies its type (Forward, Negative, Reset and Block) by the attribute *arcType*.

- *Transition*

A Transition object implements the Node interface. A Transition object specifies its type (Default, Emitter, Collector and XOR) through the *transitionType* attribute. A Transition object consists of zero or more Action objects, which are represented by the *actions* attribute. If the *transitionType* of Transition object is XOR, this Transition object should also have a Condition object.

- *Place*

A Place object implements the *Node* interface. A Place object has an attribute *occupiedTokens* to reference the tokens are currently in this place. It also provides the operations for adding or removing tokens.

- *Condition*

A Condition object is defined only for the XOR Transition. Similarly with the Criteria object, a Condition object has an operation *appliesTo(Token *)* which returns boolean values. Condition is declared here as an interface.

- *Action*

An Action object represents the actual tasks performed in the workflows, and is declared as an interface. An object that implements the Action interface has an operation *act()*. The implementation of this operation determines how the concrete task performed within an Action object. For example, it is possible to implement *act()* with the code for calling a certain application of the FlinQ server or invoking the XQuery processor to perform XQuery statements. This provides the mechanisms to invoke external applications within the workflow execution environment. In the FlinQ platform, the examples of the classes

defined to implements the Action interface include SednaQueryAction, SendCommandToClientAction and SendCommandToConnectorAction.

5.4 XQuery Processor and Native XML Database

The XQuery processor works in cooperation with a native XML database as one component in our workflow engine, which provides the capabilities of XML data processing and management. In practice, we select the Sedna native XML database [29] [30] for XML data processing and management.

5.4.1 Sedna native XML database

Sedna is a native XML database developed by the MODIS team at the Institute for System Programming of the Russian Academy of Sciences. It provides a full range of traditional core database services like persistent storage, ACID transactions, concurrency control, data query and update facilities, security, indices, hot backup [29]. In particular, the Sedna database takes the XQuery 1.0 and its data model [30] as a basis for the XML data query facility. In addition, it extends XQuery with update facility and establishes its own Data Definition Language (DDL) as a supplement for the XQuery 1.0 standard.

Moreover, the Sedna database has an XQuery processor that supports all the languages mentioned above.

The Sedna database is implemented by C++ and Scheme and its supported platform includes Windows and Linux [31]. The Sedna database provides APIs for multiple languages including C, Java and Scheme, which allows programmatic access to Sedna from client applications developed in these languages.

5.4.2 XML data processing languages

The XML data processing languages that are supported by the Sedna database are XQuery 1.0 (including XPath 2.0) [26], XQuery update language [] and Sedna DDL [31].

(1) XQuery 1.0 and XPath 2.0 support

The XQuery processor in Sedna supports XQuery 1.0 (including XPath 2.0) except a few features [30]. The features that are not supported include importing the external XML Schemas.

Moreover, the XQuery processor also has full support for two optional XQuery features:

- Full Axis. The following optional axes are supported: *ancestor*, *ancestor-or-self*, *following*, *following-sibling*, *preceding*, and *preceding-sibling*.
- XQuery Modules. This allows the XQuery prolog to import a module and allows library modules to be created.

(2) XQuery update language

The XQuery standard does not provide an update facility. The XQuery update language is used to update the persistent XML data stored in the XML database.

Table 5.1 Update statements

Update statement	Purpose	Syntax
INSERT	Inserts zero or more nodes into a designated position with respect to a target nodes	UPDATE insert SourceExpr (into|preceding|following) TargetExpr
DELETE	Removes target nodes from the database with their descendants	UPDATE delete Expr
DELETE_UNDEEP	Removes target nodes from the database preserving their content	UPDATE delete_undeep Expr
REPLACE	Replaces target nodes with a new sequence of zero or more node	UPDATE replace $var [as type] in SourceExpr with TargetExpr($var)
RENAME	Changes the name of the target nodes	UPDATE rename TargetExpr on QName

Table 5.1 lists the update statements of the XQuery update language. All update statements start with the keyword *UPDATE*. The result of each update statement should not break the well-formedness and validity of XML entities, stored in the database, otherwise, an error is raised.

- *Insert*

The insert statement inserts the result of the given expression in the position identified by the *into*, *preceding* or *following* clauses. For example, the following update statement inserts new *event* element into the *events* elements which have the *type* attribute equals *intercom*.

```
UPDATE
```

```
insert <event>intercom request</event>
into doc("EventsRecords")/events[@type=”intercom”]
```

Figure 5.7 Example of insert statement

- *Delete*

The delete statement removes persistent nodes from the database. It contains a subexpression, which returns the nodes to be deleted. The following example deletes all *event* nodes in which the *status* attribute equles *ignored*:

```
UPDATE
delete doc("EventsRecord")//event[@status=”ignored”]
```

Figure 5.8 Example of delete statement

- *Delete_undeep*

The delete_undeep statement removes nodes identified by *Expr*, but in contrast to the delete statement it leaves the descendants of the nodes in the database. The following example removes B nodes and also makes C and D nodes children of the A element.

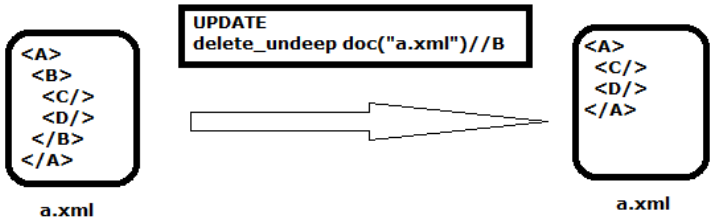

Figure 5.9 Example of delete_undeep statement

- *Replace*

The replace statement is used to replace nodes in an XML document. The following example replaces all the intercomEvent elements with *<event type=”intercom”/>*.

```
UPDATE
replace $x in doc("EventsRecord")//intercomEvent
with<event type=”intercom”/>
```

Figure 5.10 Example of replace statement

- *Rename*

The rename statement changes the name property of the all nodes returned by the *TargetExpr* expression with a new QName. The following expression changes the name of all the job elements with a new name *profession*.

```
UPDATE
```

```
rename doc("users.xml")//job on profession
```

Figure 5.11 Example of rename statement

Finally, it is worth noting that this update extension cannot be used for updating temporary XML data constructed within the XQuery statements.

(3) Sedna Data Definition Language

The Sedna DDL plays the same role as the DDL of SQL (support the CREATE, ALTER, RENAME, DROP and TRUNCATE statements) for rational databases. Most of parameters of the Sedna DDL are computable and specified as XQuery expressions. The tasks that can be conducted by the Sedna DDL include managing standalone XML documents, managing collections, managing XQuery modules and retrieving metadata of database.

- *Managing standalone XML documents and collections*

Table 5.2 lists all Sedna DDL statements for managing standalone XML documents and collections. In the Sedna database, a document can be either a *standalone document* (when it does not belong to any collection) or belong to some *collection*. Compared to standalone documents, all documents within a given collection have a common descriptive schema. The common descriptive schema allows addressing XQuery and update queries to all members of a collection.

These statements can be used to create, drop and rename standalone XML documents or collections in the XML database.

Table 5.2 DDL for managing standalone documents and collections

Sedna DDL statements	Purpose	Syntax
CREATE DOCUMENT	Creates a new standalone document with a name	CREATE DOCUMENT doc-name-expr
DROP DOCUMENT	Drops the standalone document	DROP DOCUMENT doc-name-expr
CREATE COLLECTION	Creates a new collection with a name	CREATE COLLECTION coll-name-expr
DROP COLLECTION	Drops the collection document	DROP COLLECTION coll-name-expr
RENAME COLLECTION	Renames collection with a new name	RENAME COLLECTION old-name-expr INTO new-name-expr
CREATE DOCUMENT IN COLLECTION	Creates a new document in a certain collection	CREATE DOCUMENT doc-name-expr IN COLLECTION coll-name-expr
DROP DOCUMENT IN COLLECTION	Drops the document in a certain collection	DROP DOCUMENT doc-name-expr IN COLLECTION coll-name-expr

- *Managing XQuery modules*

XQuery allows one to put functions in XQuery library modules, so that they can be shared and imported by any XQuery statements. This is beneficial for improving reusability of XQuery statements that are defined for similar purposes. An XQuery library module contains a module declaration followed by variable and/or function declarations. The module declaration specifies its target namespace URI which is used to identify the module in the database.

The defined modules are stored in .xqlib files. Before an XQuery library module could be imported from the Xquery statements, it is to be loaded into the database. Table 5.3 lists all Sedna DDL statements for managing XQuery modules in the database.

Table 5.3 DDL for managing standalone modules

Sedna DDL statements	Purpose	Syntax
LOAD MODULE	Loads the XQuery library modules	LOAD MODULE "path_to_file", ..., "path_to_file"
LOAD OR REPLACE MODULE	Replaces an already loaded module with new one	LOAD OR REPLACE MODULE "path_to_file", ..., "path_to_file"
DROP MODULE	Remove an XQuery library module from the database	DROP MODULE "target_namespace_URI"

One can obtain information about modules loaded into the database by querying the system collection named *$modules* as follows *collection ("$modules")*. This is actually the DDL statements for retrieving metadata that we discuss in the sequel.

- *Retrieving metadata*

The Sedna database stores various metadata about database objects , such as documents, collections, indexes, etc, in multiple *system documents* and *collections*. These metadata can be retrieved by querying these documents and collections. Names of the system documents and collections start with *$ symbol*. Users can query these documents or collections by using regular XQuery statements) but cannot update them.

Some important system documents and collections are:

- $documents document lists of all stand-alone documents, collections and in-collection documents (except system meta-documents and collection, like $documents document itself).

- $collections document lists of all collections.
- $modules document contains list of loaded modules with their names.

5.4.3 XML database design

In principle, the XML database can store any XML data that could be used in the FlinQ platform. However, it is still desirable to generally indentify and define what XML data should be persistently stored in this XML database.

- *Case data (optional)*

By default, the runtime case data (i.e. the XML data attached to the tokens during the execution) are not stored in the XML database. This can improve efficiency. If the runtime case data is stored in the database, the workflow engine core has to constantly interact with the XML database to retrieve the case data. However, by storing the runtime case data, the workflow engine can have a recovery mechanism in case of execution failures, which improves robustness.

- *Logs data*

One of important requirements of our WFM design is that the execution of workflows can be traced. In practice, this can be achieved by recording the XML data of each token at each step into the logs. These log data are stored in the XML database as well.

Our design provides a flexible way to record the log data during the execution of workflows. Specifically, actions can be defined in the workflows to write the data of the tokens to the log files.

Again, consider the example workflow for user login that is discussed in Section 4.2.2. In Transition 1 (labeled with id "3" in the XML definition), there are four actions defined. For the purpose of writing the data of the tokens to the logs, we added one action (shown in bold font) before each original action. The added actions specify the XQuery update statements to write the runtime data value of the token into the *logs* document stored in our XML database. In case it is not necessary to record the token data into logs, the users just do not specify such an action.

```
<Transition id="3" type="DEFAULT">
 <Action type="SednaQueryAction">
 <![CDATA[UPDATE INSERT $TOKEN into doc('logs')/logs]]>
  </Action>
 <Action type="SednaQueryAction">
  <![CDATA[
  let $token := $TOKEN
```

```
    return
       <ClientCommand type='CloseForm' form_id='LoginState.Login'>
        <Client id="{string($token/@remote_id)}" />
        <OriginalToken>{$token}</OriginalToken>
       </ClientCommand>
     ]]>
</Action>
<Action type="SednaQueryAction">
<![CDATA[UPDATE INSERT $TOKEN into doc('logs')/logs]]>
 </Action>
<Action type="SendCommandToClientAction" />
<Action type="SednaQueryAction">
<![CDATA[UPDATE INSERT $TOKEN into doc('logs')/logs]]>
 </Action>
<Action type="SednaQueryAction">
 <![CDATA[
  let $token := $TOKEN
  return
       <ClientCommand type='EnterSite'>
        <Client id="{string($token/OriginalToken/FormEvent/@remote_id)}"/>
        <OriginalToken>{$token/OriginalToken/*}</OriginalToken>
       </ClientCommand>
 ]]>
</Action>
<Action type="SednaQueryAction">
<![CDATA[UPDATE INSERT $TOKEN into doc('logs')/logs]]>
 </Action>
<Action type="SendCommandToClientAction" />
</Transition>
```

Figure 5.12 Example of updating logs data within workflows

- *Application data*

The XML database is designed to also be responsible for the storage of application data. However, this kind of data is closely related to the concrete applications and thus can be hardly predefined.

5.4.4 Integration with engine core

Based on our modelling language, the XQuery statements can be either defined in the Petri net actions that manipulate the XML data attached to the tokens, or in the criteria and XOR transitions for the evaluation conditions. In addition to these two cases, the XQuery update and Sedna DDL statements can be also defined in the Petri net actions that are responsible for updating and managing the XML data stored in the database.

Table 5.4 lists the cases in which XQuery, XQuery update extensions and DDL statements can be defined within our workflows. The XQuery processor executes these statements in different languages and returns information that

indicates whether the execution has been successful. In the case of XQuery statements, the resulting XML data are also returned.

Table 5.4 Use cases of XML data processing languages within our workflows

Use cases	Language type	Expected result
Actions for token data manipulations	XQuery 1.0 and XPath 2.0	success information with the resulting XML data OR error information when query fails
Actions for updating and managing the data in the XML database	XQuery update extension and Data Definition Language	success information OR error information
XOR transitions for conditional choices for branch structures	XQuery statements that returns Boolean values	success information with a Boolean value OR error information
Criteria in each Petri net for the evaluation conditions	XQuery statements that returns Boolean values	success information with a Boolean value OR error information

Figure 5.13 shows an example of a Petri nets-based workflow that includes all the use cases listed above. In this Petri net, Transition 1contains two actions. We assume that Action 1 specifies the XQuery statements for the token data manipulation and that Action 2 specifies the XQuery update statements for modifying the data stored in database. First, the XQuery statements defined in the criteria and the XOR transitions are executed by the XQuery processor. Action 1 also invokes the XQuery processor to manipulate token data. However, Action 2 interacts with the XML database and takes no action for the token.

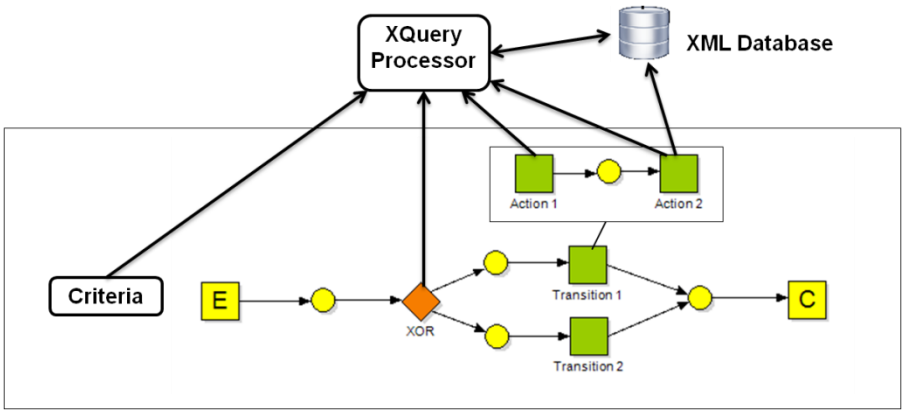

Figure 5.13 Interaction with the XQuery processor and XML database

Figure 5.14 depicts the simplified class diagram of the workflow engine core with the Sedna XML database.

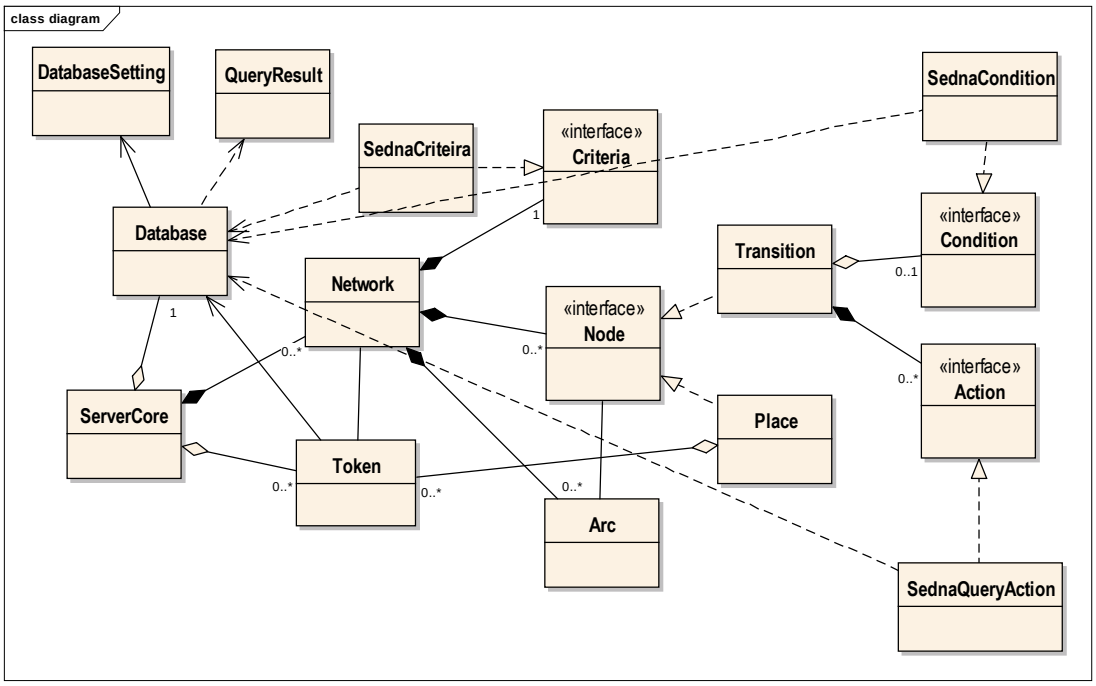

Figure 5.14 Simplified class diagram with the integration of XML database and XQuery processor

The main classes in Figure 5.14 are briefly discussed below:

- *Database*

A Database object references the XML database (with its XQuery processor) that the workflow engine core connects with. A ServerCore can contain a Database object. Examples of the Database object' operations are:

bool connectTo(DatabaseSetting & settings);

bool closeConnection();

QueryResult execute(string statements);

Each Database has a *settings* attribute (which is a DatabaseSetting object) to specify the basic settings for connecting to the database, like the URI of database server, the name of accessed database, username and password. The *connectTo()* operation is used to connect the database system according to the settings. The *execute()* operation invokes the XQuery processor to execute the XQuery statements, XQuery updates or DDL.

- *DatabaseSetting*

A DatabaseSetting object maintains the basic settings for database connection. Examples of the DatabaseSetting's attributes include:

string serverURI;

string databaseName;

string username;

string password;

- *QueryResult*

A QueryResult object contains the result of the execution of XQuery, XQuery update or DDL statements. It has a boolean attribute to indicate if the execution has been successful and a String attribute with the result of the XQuery statements.

- *Token*

The Token object is extended to have the attribute *xmldata* to refer to the attached XML data.

- *SednaCriteria*

SednaCriteria implements the Criteria interface. Its *appliesTo(Token * token)* operation invokes the operation *execute()* of the Database to perform the XQuery statements and returns a boolean value.

- *SednaCondition*

SednaCondition implements the Condition interface. Its *appliesTo(Token * token)* operation invokes the operation *execute()* of the Database to perform the XQuery statements and returns a boolean value.

- *SednaQueryAction*

SednaQueryAction implements the Action interface. Its *act(Token* token)* operation invokes the operation *execute()* of the Database to perform the XQuery, XQuery update and DDL statements. In particular, this act() operation replaces the $TOKEN symbols in the XQuery statements with the real data values of the tokens. In case the XML data of a token is manipulated (i.e., by executing regular XQuery statements), the result of these XQuery statements is assigned to the Token's *xmldata* attribute as the new data value. In case update and DDL statements are performed, the result only indicates whether the update or DDL statements have been successful, and the *xmldata* attribute of the Token object is not changed.

5.5 Workflow Definition Loader

The Workflow Definition Loader (referred to as loader) is responsible for interpreting the XML-based workflow definitions and instantiating the corresponding objects in the engine core. The loader parses the workflow definitions based on the XML Schema defined in Appendix A.

Figure 5.16 shows the simplified class diagram of the workflow engine with the loader. Three objects are created, namely Loader, ActionSerializer and ConditionSerializer.

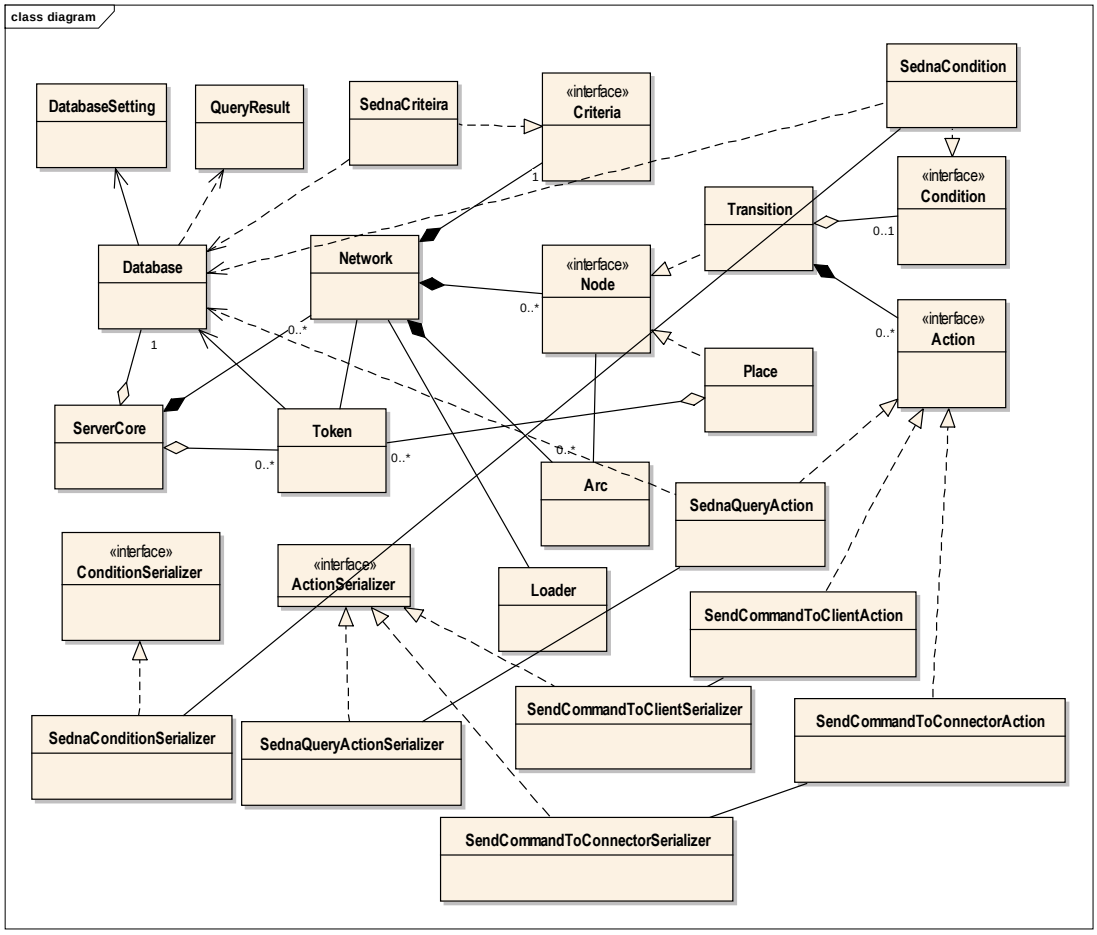

Figure 5.15 Simplified class diagram with the integration of the loader

- *Loader*

This is the main class for loading the XML-based workflow definitions. The class *Loader* class has an attribute *xmlPetri* that stores an XML workflow definition that is to be loaded. Four operations are defined for parsing the four basic elements (i.e. <*Criteria*>, <*Transition*>, <*Place*> and <*Arc*>) respectively:

*bool parseCriteria(Network *network, TiXmlElement* networkElement);*

*bool parseTransition(Network *network, TiXmlElement* networkElement);*

*bool parsePlace(Network *network, TiXmlElement* networkElement);*

*bool parseArc(Network *network, TiXmlElement* networkElement);*

The operation *createNetwork()* invokes all these four operations and finally instantiates the *Network* object.

In our XML-based workflow definitions, the <*Condition*> and <*Action*> elements are nested within the <*Transition*> elements. Moreover, these two elements can have different types and these types may be extended in future. Different types require different (de)serialization processes. For example, <Action> can be SednaQueryAction, SendCommandToClientAction or SendCommandToConnectorAction. The deserialization process for SednaQueryAction instantiates the SednaQueryAction objects, while the deserialization process for SendCommandToClientAction instantiates the SendCommandToClientAction objects. Therefore, we particularly create two interfaces for the (de)serialization of actions and conditions respectively.

- *ActionSerializer*

An ActionSerializer has two operations: *serialize()* and *deserialize()*. The latter one is used for the loader. *deserialize()* converts the XML elements into the corresponding objects. The concrete action types need to implement this interface by creating new classes (e.g. SednaQueryActionSerializer and SendCommandToClientActionSerializer).

- *ConditionSerializer*

Similarly, a ConditionSerializer also defines two operations: *Serialize()* and *Deserialize()*. The concrete condition types need to implement this interface by creating new classes (e.g. SednaConditionSerializer).

6. Case Study

This chapter demonstrates the applicability of the proposed WFM solutions discussed in the previous chapters, by performing a case study as a proof-of-concept. We give some real-life examples of modellingand executing business processes in FlinQ security management platform by using our WFM framework.

Section 6.1 introduces the stored XML data in the database. In Section 6.2 and 6.3, we elaborate several workflows which are designed for handling the business processes in the FlinQ platform. Section 6.4 draws the conclusion.

6.2 Data Initialization in XML database

Before discussing our designed workflows, we briefly introduce the XML data stored in our XML database. As mentioned in Chapter 5, some configuration files are stored in the workflow repository to initialize the XML data in the database. Specifically, when the FlinQ server starts up, some XML documents are created in the database based on the configuration files. Table 6.1lists some of created XML documents that are used in the workflows we discuss later.

Table 6.1 Some created XML documents in the database

XML documents	Purposes
preconnection_logs	Records all the preconnection requests from the client applications
users	Records all the registered users' personal information, including username, password, role and etc.
sessions	Keeps the records of all login and active users' information and their client applications' id.
SiteData	Maintains the information of all devices that connect with the FlinQ server. For example, it can contain the information of the intercom devices, including its internal id, status, name and device type.

6.3 Preconnection and Login of the Clients Applications

6.3.1 Proconnection workflow

In order to interact with the server side, the client applications should firstly *preconnect* with the FlinQ server. When the client application is started up, a *preconnection form* appears and users are required to specify the name, the IP address and the port number of the FlinQ server that they intend to connect with. This is done by sending an *event* message to the specified server. When the server receives this message, it will perform several tasks including closing the preconnection form and opening a new form for login. A workflow called *preconnection* is thus designed for executing these tasks.

The XML event message for the preconnection is shown in Figure 6.1, which is attached to a new token to be executed. In the FlinQ, only the event messages for the preconnection have the attribute *form_id* that equals "*LoginState.ServerSelect*". Therefore, this *form_id* can be used to identify the token for the preconnection workflow in the criteria. And Figure 6.2 shows the criteria of the preconnection workflow.

In addition, the attribute *remote_id* reveals which client applications send this message.

```
<FormEvent action_id="Connect" form_id="LoginState.ServerSelect"
remote_id="2">
  <FormData>
    ……
  </FormData>
</FormEvent>
```

Figure 6.1Example of input token data for preconnection workflow

```
<Criteria type="SednaCriteria">
  $TOKEN/@form_id = 'LoginState.ServerSelect'
</Criteria>
```

Figure 6.2 The criteria of preconnection workflow

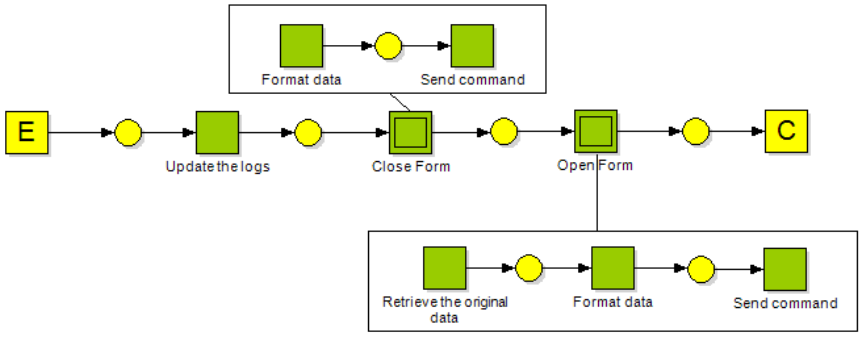

Figure 6.3 The pre-connection workflow

Figure 6.3 depicts the control flow of the preconnection workflow. The workflow mainly consists of three tasks (transitions).

- *Update logs*

This transition is created to update the workflow logs data and has only one action. As mentioned before, the update of logs data can be explicitly specified in our workflows. Figure 6.4 shows the definition of this transition. It writes the current token data to the *preconnection_logs* file stored in the XML database.

```
<Transition id="3" type="DEFAULT">
  <Action type="SednaQueryAction">
  <![CDATA[UPDATE INSERT $TOKEN into doc('preconnection_logs')/logs]]>
  </Action>
</Transition>
```

Figure 6.4 The transition for updating logs data

- *Close form*

The Close form transition (Figure 6.5) contains *two actions*. The first action reconstructs the token data for sending the CloseForm command to the client. The second action invokes the application of the server to send the CloseForm command by using the reconstructed token data. In our XML-based workflow definition, the actions with the *SendCommandToClientAction* type send the token data as the command message to the specified client applications (with *<Client id=”1”>*).

```
<Transition id="4" type="DEFAULT">
  <Action type="SednaQueryAction">
      <![CDATA[
          <ClientCommand type='CloseForm' form_id='LoginState.ServerSelect'>
           <Client id="{string($TOKEN/@remote_id)}" />
           <OriginalToken>$TOKEN</OriginalToken>
          </ClientCommand>
       ]]>
   </Action>
    <Action type="SendCommandToClientAction" />
  </Transition>
```

Figure 6.5 The CloseForm transition

- *Open form*

The Open form transition (Figure 6.6) contains *three actions*. The first action retrieves the original data value of the token from the element < *OriginalToken* >. Again, the second action reconstructs the token data for sending the OpenForm command. Finally, the third action sends the reconstructed token as the OpenForm command to the client. The entire

workflow is thus completed and a new form for login is presented in the client application.

```
<Transition id="5" type="DEFAULT">
    <Action type="SednaQueryAction">$TOKEN/OriginalToken/child::*</Action>
    <Action type="SednaQueryAction">
     <![CDATA[
          <ClientCommand type='OpenForm' filePath='server_login_form.json'>
       <Client id="{string($TOKEN/@remote_id)}" />
       <OriginalToken>$TOKEN</OriginalToken>
      </ClientCommand>
     ]]>
    </Action>
    <!--Send the command to the client-->
    <Action type="SendCommandToClientAction" />
</Transition>
```

Figure 6.6 The OpenForm transition

6.3.2 Login workflow

After the preconnection workflow is finished, users are able to log in with their username and password. When users press the login button in the client applications, an XML event message is thus sent to the server and then attached to a new token for the *login* workflow. Figure 6.7 shows an example of input token data for the login workflow. And Figure 6.8 shows the criteria of the workflow.

```
<FormEvent action_id="Login" form_id="LoginState.Login" remote_id="2">
  <FormData>
    <json>
      <String name="Action">Login</String>
      <String name="Form">LoginState.Login</String>
      <Object name="Properties">
        <Object name="Password">
          <String name="Name">Password</String>
          <String name="Type">Text</String>
          <String name="Value">12345</String>
        </Object>
        <Object name="Username">
          <String name="Name">Username</String>
          <String name="Type">Text</String>
          <String name="Value">zchen</String>
        </Object>
      </Object>
    </json>
  </FormData>
</FormEvent>
```

Figure 6.7 Example of input token data for login workflow

```
<Criteria type="SednaCriteria">
```

```
    $TOKEN/@form_id = 'LoginState.Login'
</Criteria>
```

Figure 6.8 The criteria of login workflow

Figure 6.9 depicts the login workflow. The login workflow mainly includes six transitions.

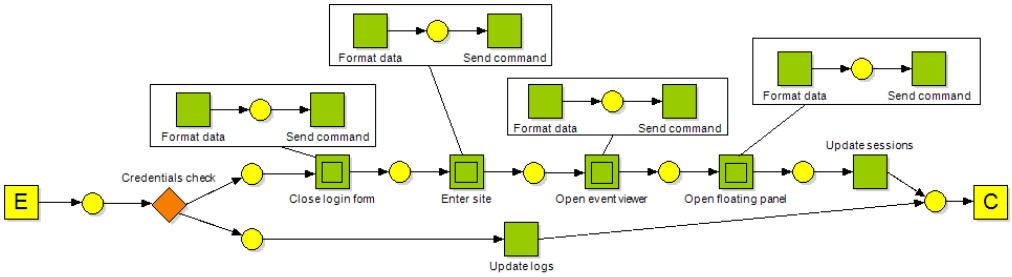

Figure 6.9 The login workflow

- *Credentials check*

The credentials check transition is an XOR transition that is responsible for validating the entered username and password according to the valid registered users' information stored in the XML database (i.e. the *users* documents). The following XQuery statements firstly retrieve the values of username and password from the input token. Afterwards, it checks if there is exactly *one* registered user whose username and password equal the entered ones for login.

```
<Transition id="2" type="XOR">
  <Condition type="SednaCondition">
   <![CDATA[
        let $token := $TOKEN
        let $login :=
        <login>
          <username>
            {string($token/FormData/json/Object[@name="Properties"]/Object[@
            name="Username"]/String[@name="Value"])}
          </username>
          <password>
            {string($token/FormData/json/Object[@name="Properties"]/Object[@
            name="Password"]/String[@name="Value"])}
          </password>
        </login>
        let $valid_accounts := for $user in doc('users')/users/user
        where $user/@name = $login/username
        and $user/@password =  $login/password
        return $user
        return count($valid_accounts) = 1
      ]]>
  </Condition>
```

```
  </Transition>
```

Figure 6.10 The transition for updating logs data

When the result of the check returns true, the following transitions are performed sequentially. Otherwise, the entire workflow ends.

- *Close login form*

This transition includes *two actions*, which reconstructs the token and sends the command to the client to close the login form.

```
<Transition id="3" type="DEFAULT">
  <Action type="SednaQueryAction">
      <![CDATA[
      let $token := $TOKEN
      return <ClientCommand type='CloseForm' form_id='LoginState.Login'>
        <Client id="{string($token/@remote_id)}" />
        <OriginalToken>{$token}</OriginalToken>
      </ClientCommand>
     ]]>
   </Action>
   <Action type="SendCommandToClientAction" />
</Transition>
```

Figure 6.11 The close login form transition

- *Enter site*

This transition includes *two actions*, which reconstructs the token and sends the command to the client for entering the site form.

```
<Transition id="4" type="DEFAULT">
  <Action type="SednaQueryAction">
      <![CDATA[
       let $token := $TOKEN/OriginalToken/*
       return <ClientCommand type='EnterSite'>
        <Client id="{string($token/@remote_id)}"/>
        <OriginalToken>{$token}</OriginalToken>
       </ClientCommand>
      ]]>
   </Action>
   <Action type="SendCommandToClientAction" />
</Transition>
```

Figure 6.12 The enter site form transition

- *Open event viewer*

This transition includes *two actions*, which reconstructs the token and sends the command to the client for opening the event viewer form.

```
<Transition id="5" type="DEFAULT">
  <Action type="SednaQueryAction">
```

```
      <![CDATA[
      let $token := $TOKEN/OriginalToken/*
      return <ClientCommand type='OpenForm' filePath='event_viewer_test.json'>
        <Client id="{string($token/@remote_id)}" />
        <OriginalToken>{$token}</OriginalToken>
      </ClientCommand>
      ]]>
    </Action>
    <Action type="SendCommandToClientAction" />
</Transition>
```

Figure 6.13 The open event viewer transition

- *Open floating panel*

This transition includes *two actions*, which reconstructs the token and sends the command to the client for opening the floating panel.

```
<Transition id="6" type="DEFAULT">
   <Action type="SednaQueryAction">
      <![CDATA[
      let $token := $TOKEN/OriginalToken/*
      return <ClientCommand type='OpenForm' filePath='floating_panel.json'>
        <Client id="{string($token/@remote_id)}" />
        <OriginalToken>{$token}</OriginalToken>
      </ClientCommand>
      ]]>
     </Action>
    <Action type="SendCommandToClientAction" />
</Transition>
```

Figure 6.14 The open floating panel transition

- *Update sessions*

This is last step of login. The action defined in this transition write the token data into the *sessions* documents stored in the XML database. The *sessions* documents is responsible to keep the records of the login users' information and their corresponding client applications' ids.

```
<Transition id="7" type="DEFAULT">
  <Action type="SednaQueryAction">
      <![CDATA[
      UPDATE INSERT $TOKEN/OriginalToken/* into doc('sessions')/sessions
      ]]>
  </Action>
</Transition>
```

Figure 6.15 The update sessions transition

- *Update logs for the failed login requests*

When the login request fails (i.e. the evaluation condition of *credentials check* transition returns false), the negative path is selected and Transition *update*

logs is performed. It stores the current token data into the log file (*doc('LoginFailsLogs')*) for recording the failed login requests.

```
<Transition id="7" type="DEFAULT">
  <Action type="SednaQueryAction">
     <![CDATA[
      UPDATE INSERT $TOKEN into doc('LoginFailsLogs')/logs
     ]]>
  </Action>
</Transition>
```

Figure 6.16 The update logs transition

6.4 Interaction with the Intercom Devices

The previous examples show the workflows where the FlinQ server only interacts with the client applications. In this section, we presented more examples of the workflows where the server interacts with both the connectors (devices) and the client applications.

Figure 6.17 Intercom devices animation in the client applications

The FlinQ platform integrates with an intercom subsystem by a so-called *Intercom connector*. This connector is able to connect multiple intercom devices. Users can request the calls, start the conversations and stop the conversations on these devices. Therefore, the intercom devices have three corresponding event types, namely *CallRequest*, *ConversationStarted* and *ConversationStopped*. When these events occur on the intercom devices, the connector sends the event messages to the server. The server records the latest statuses (*CallRequest*, *ConversationStarted* and *ConversationStopped*) of the intercom devices in the XML database. When a new event occurs, the server will update the status of these intercom devices and animate the devices in the client applications (e.g. blinking the devices) by the newest status. Figure 6.17 shows how the intercom devices are animated in the client applications when different types of intercom events happened. For example, when a *CallRequest* event happens on a certain intercom device, in the client application, this device is blinked as orange. And a new form pops up with the details of this *CallRequest* event and a StartConversation button.

Therefore, we created the *IntercomEvents* workflow to handle the business processes mentioned above. Figure 6.18 shows an example of intercom event message which is also the input token for the *IntercomEvents* workflow. The attribute *id* of the root element is the unique id number of the connectors. The text in the element *<String name="EventType">* specifies the event type. In particular, there are two numbers in the text of *<String name="Message">*. The first number is the code of the intercom device (here it's 1003) which sends this event message. The second number specifies if this message is used to cancel the event or not. For example, user can cancel the call request on the device that has already been requested a call. When this number is **1**, the message is used for canceling the event. While when the number is **0** (like the example in Figure), the message is for establishing the event.

```
<ConnectorData id="1">
  <json>
    <String name="Source">Commend_ICX_TCP</String>
    <Number name="Instance">0</Number>
    <String name="EventType"> CallRequest</String>
    <String name="Message">1003,0</String>
    <String name="Reason"></String>
    <String name="freetext"></String>
  </json>
</ConnectorData>
```

Figure 6.18 Example of input token data for the intercom events workflows

The criteria of *IntercomEvents* workflows is defined in Figure 6.19.

```
<Criteria type="SednaCriteria">
  <![CDATA[
```

```
        declare variable $token := $TOKEN;
        (string(node-name($token)) ="ConnectorData" and substring-
        after(string($token/json/String[@name="Message"]), ',')='0'
        or
        string(node-name($token)) ="ConnectorData" and substring-
        after(string($token/json/String[@name="Message"]), ',')='1')
        and
        (data($token/json/String[@name="EventType"])="CallRequest" or
          data($token/json/String[@name="EventType"])="ConversationStarted" or
          data($token/json/String[@name="EventType"])="ConversationStopped")
    ]]>
 </Criteria>
```

Figure 6.19 The criteria of the IntercomEvents workflow

The *IntercomEvents* workflow is mainly responsible for two tasks: updating the statuses of the intercom devices in the server and animate this device in the client applications according to the event types. Figure 6.20 depicts the intercomEvents workflow. At the start of the workflow, an XOR transition is defined to check if the token is for cancelling the events or not. The entire workflow is thus separated as two branches. The upper one handles the regular intercom events and the lower on handles the cancel events. Figure 6.21 shows the definition of the *event types check* transition.

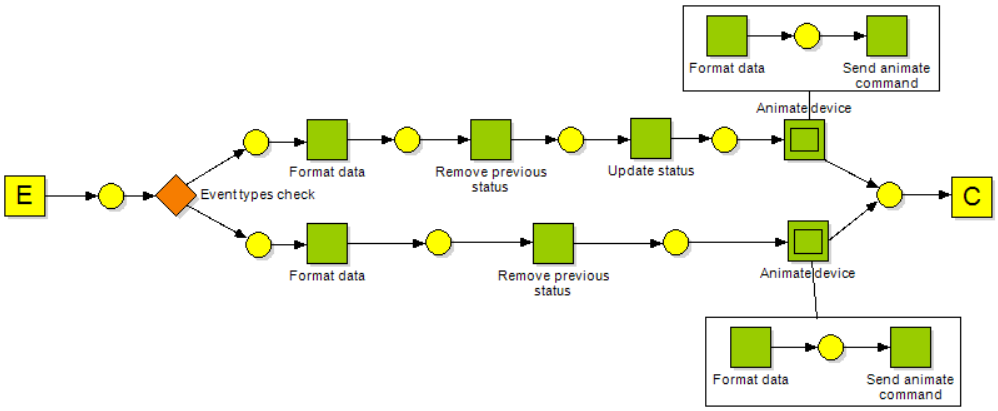

Figure 6.20 The IntercomEvents workflow

```
<Transition id="2" type="XOR">
  <Condition type="SednaCondition">
   <![CDATA[
     string(node-name($token)) ="ConnectorData" and substring-
     after(string($token/json/String[@name="Message"]), ',')='0'
     ]]>
  </Condition>
 </Transition>
```

Figure 6.21 The event types check transition

When the evaluation condition of the XOR transition returns true (i.e. the non-cancel events), the following transitions are performed.

- *Format data*

This transition contains one action that reconstructs the input XML event message. It simply retrieves the needed data values (like *rapi_id* and *event_type*) and formats a new token for the ease of use. As mentioned before, the first number in the text of *<String name="Message">* is the code of the intercom device. However, this code corresponds to an internal device id in the server (stored in the *SiteData* document). In this transition, we first declare *$ic_code* to retrieve the intercom device code. Afterwards, we retrieve the corresponding internal id and assign it to the *$device_id*. Finally, we put this internal id into the new token (assign it to the attribute *internal_id*).

```
<Transition id="3" type="DEFAULT">
  <Action type="SednaQueryAction">
   <![CDATA[
        declare variable $token := $TOKEN;
        declare variable $ic_code := substring-
          before(string(data($token/json/String[@name="Message"])), ',');
        declare variable $device_id :=
          doc('SiteData')//Device[Property[@Name="IC"]/@Value=lower-
          case(string($ic_code))]/@id;
        <token
           rapi_id="{ string($token/@id)}"
           event_type="{data($token/json/String[@name="EventType"])}"
           ic_code="{$ic_code}"
           internal_id='{string($device_id)}'
           is_on="{substring-
                  after(string(data($token/json/String[@name="Message"])), ',')}"
         />
    ]]>
  </Action>
</Transition>
```

Figure 6.22 The format data transition

- *Remove previous status*

This transition contains one action. Based on the internal id, it removes all the previous statuses of this intercom device in the *SiteData* document.

```
<Transition id="4" type="DEFAULT">
  <Action type="SednaQueryAction">
     declare variable $token := $TOKEN;
     UPDATE DELETE
              doc('SiteData')//Device[@id=string($token/@internal_id)]/state
  </Action> </Transition>
```

Figure 6.23 The remove previous status transition

- *Update status*

This transition contains one action. It updates the status of the intercom device in the *SiteData* document.

```
<Transition id="5" type="DEFAULT">
  <Action type="SednaQueryAction">
    <![CDATA[
     declare variable $token := $TOKEN;
     UPDATE INSERT
      <state name="{string($token/@event_type)}" />
      INTO doc('SiteData')//Device[@id=string($token/@internal_id)]
    ]]>
  </Action>
</Transition>
```

Figure 6.24 The update status transition

- *Animate the device*

The animate device transition has two actions. The first action formats the token as an *Animate* command. In particular, according to the different event types, the different *animation_file* (the json files) will be selected in the *<ClientCommand>*. In the client application, these json files are used to show the popped forms for the intercom devices (see Figure 6.17).

```
<Transition id="6" type="DEFAULT">
   <Action type="SednaQueryAction">
    <![CDATA[
      declare variable $token := $TOKEN;
      declare variable $Events :=
     <event json="CALLREQUEST.json" name="CallRequest" /> |
     <event json="CONVERSATIONSTARTED.json" name="ConversationStarted" /> |
     <event json="CONVERSATIONSTOPPED.json" name="ConversationStopped" />;
       declare variable $animation :=
          string($Events[@name=$token/@event_type]/@json);
       <ClientCommand type="Animate" device_id="{string($token/@internal_id)}"
         animation_file="{$animation}">
          <Client id="0" />
       </ClientCommand>
]]>
   </Action>
  <Action type="SendCommandToClientAction" />
</Transition>
```

Figure 6.25 The animate device transition

The workflow involving the lower branch that is responsible for handling the cancel events is actually very similar with the upper branch. The main difference is that the lower one only removes the cancelled the devices' statuses in the *SiteData* documents. And it does not update the statues. However, after removes the cancelled statuses, it still animates the intercom

devices with the current statuses. Here, the details of workflow definitions for the cancel event are not discussed any more.

6.5 Conclusion

This section presented several practical examples of the workflows designed for the FlinQ platform by using our workflow modelling language. Moreover, all these workflows have already been deployed in our workflow engine and successfully executed.

The case study demonstrates that our proposed workflow modelling language and WFM framework can successfully model and execute the data-centric business processes in the FlinQ security management platform. In addition, the workflow engine can successfully invoke the applications of the FlinQ server to interact with the connectors and the client applications.

However, the case study also reveals that the XQuery statements defined in the workflows could be very complicated and not so readable in some cases.

7 Conclusion

This chapter presents the main contributions of this thesis, draws some conclusions and identifies points where further investigation is necessary.

This chapter is further structured as follows: Section 8.1 presents our general conclusions and summarizes the main contributions of this thesis. Section 8.2 provides the answer of the research questions proposed in Chapter 1. Section 8.3 identifies some future work.

7.3 General Conclusions

In this work, we have proposed a WFM solution in the domain of security management applications.

We first investigated security management applications and identified the demand of adopting WFM technology in the security management domain.

We then performed a literature study on WFM and workflow modelling languages. In particular, we focused on the data perspective of workflow modelling languages.

In order to improve the capability of data handling in traditional Petri nets, we proposed our Petri nets-based workflow modelling language with extensions for data representation and manipulation. This language is a variant of classic Petri nets inspired by Coloured Petri Nets [14]. In particular, we introduced XQuery and its update extension in our language to explicitly specify the tasks for data manipulations within the workflows. In addition, our workflow definitions can be represented in a XML format with a uniform XML Schema syntax. This results in the easy deployment and execution of the workflows in our workflow engine.

We presented the design of the WFM framework in the FlinQ platform. Based on the existing Petri nets workflow engine core, we integrated an XQuery processor and a native XML database with this engine core. XQuery statements (with the update extension [32] and Sedna Data Definition Language [31]), defined in the workflow tasks for data processing and management, are executed by this XQuery processor. The XML database can be used for storing both workflow-relevant data and application data. In addition to this integration, we created a workflow definition loader to parse the XML-based workflow definitions and instantiate workflows for the engine core, which plays the role of *Interface 1* of the Workflow Reference Model [7].

We selected the Sedna XML native database [31] for XML data processing and management. The Sedna database has an XQuery processor to support

XQuery execution, the XQuery update language [32] and the Sedna DDL [31]. The implementation of the designed WFM framework mainly includes the integration the Sedna database with the workflow engine core and the workflow definition loader.

Finally, we tested our WFM solution by performing a case study in the FlinQ platform. We modelled some typical workflows in the FlinQ platform by using our modelling language and we deployed and executed these workflows in our workflow engine. The case study shows that our WFM framework can be successfully applied in real-life situations.

7.4 Answers to the Research Questions

(1) Which extensions of Petri nets are available for improving the capability of data representation and manipulation?

In Chapter 3.5, we discuss the data perspective in workflow languages. Coloured Petri Nets (CPN) [14] extends classic Petri nets by attaching data values to the tokens and introducing a high-level programming language, namely Standard ML [15] to support data definition and manipulation. Standard ML expressions can be defined in the arcs going out of a transition, to specify how to manipulate the data of tokens being produced by the transition. CPN greatly improves the capability of data representation and manipulation.

(2) Which technology and standards can be used for improving the capability of data representation and manipulation in Petri nets?

By investigating the data perspective in multiple popular workflow languages, our work identifies several technologies and standards that can improve the data processing ability in Petri nets. CPN adopts Standard ML as its high-level programming language. We also realize that more recent workflow systems take full advantage of XML and XML-related standards, like XPath, XQuery and XSLT. XML offers high flexibility, high extendibility and strong data types support in terms of data representation. Furthermore, these XML data processing standards are able to support high-level data manipulation. It is thus beneficial to combine XML technologies and the Petri nets model.

(3) How to integrate data processing and management services in a Petri nets-based workflow execution environment?

In Chapter 5, we elaborate the design of our WFM framework which integrates the data processing and management services with the Petri nets-based workflow engine. First, all the data attached to the workflow cases are

wrapped in XML format within the workflow engine. Afterwards, an XQuery processor and a native XML database are tightly integrated with the workflow engine core. By using our Petri nets-based modelling language, the tasks that perform the data processing and management can be explicitly specified in our workflow definitions. The data processing and management services within the workflow execution environment are thus realized.

7.5 Future Work

During the implementation of our work, we identified several research opportunities for future work:

- *XML Schema definitions and validation in places and criteria*

This feature has been proposed as one of our extensions for traditional Petri nets in Section 4.2. With this feature, XML Schemas are the typing system for the XML data of the tokens, so that the validity of the XML data can be ensured at each step of the workflows. However, this feature has not been implemented and tested in our work, so it is worth implementing this feature within our workflow engine.

- *Importing the XML Schemas in the XQuery*

Our extensions for traditional Petri nets take full advantage of XML Schemas for strong data types support. The XQuery standard [26] already includes the primitive data types (e.g. string, integer, date) defined in XML Schema [25]. Moreover, the XQuery standard allows users to import the external XML Schema definitions for supporting the user-defined data types. However, this feature is not supported by Sedna. This issue should be solved in the future.

- *Workflow verification and analysis*

The workflow definitions should be verified before the execution. The workflow editor is responsible for the workflow verification. The verification should not only include the workflow definitions but also include the XQuery statements. Moreover, it is preferable that the workflow engine itself can have the capability of checking the correctness of the deployed workflows.

One of the important benefits of using Petri nets is availability of many analysis techniques. These techniques can be used to prove properties and to calculate performance measures. These analysis techniques should be exploited in future work.

- *Improving the capability of invoking external applications*

Currently the workflow engine only invokes the FlinQ server applications. An appropriate workflow engine should be capable of interacting with heterogeneous software components [43]. In future the workflow engine capabilities of invoking external applications should be improved.

- *Implementation of the workflow editor*

A workflow editor that plays a role of a process definition tool has been proposed in our WFM framework. However, this editor has not been implemented during this Master project. Currently the workflow designers directly create the XML-based workflow definitions in textual form, which is not intuitive. Therefore, future work includes the implementation of the workflow editor with a graphical design environment.

References

[1] M. Valera and S.A. Velastin. *Intelligent distributed surveillance systems: a review vision*. In Image and Signal Processing, IEE Proceedings, volume 152, pages 192 – 204, April 2005.

[2] D. Mierzwinski, D. Walczak, M. Wolski, M. Wrzos, *Surveillance System in Service-Oriented Manner*, synasc, pp.427-433, 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2010.

[3] W.M.P. van der Aalst. *Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management*. In J. Desel, W. Reisig, and G. Rozenberg, editors, Lectures on Concurrency and Petri Nets, volume 3098 of Lecture Notes in Computer Science, pages 1–65. Springer-Verlag, Berlin, 2004

[4] W.M.P. van der Aalst. *The Application of Petri Nets to Workflow Management*. The Journal of Circuits, Systems and Computers, 8(1):21–66, 1998.

[5] P. Lawrence, editor. *Workflow Handbook 1997*, Workflow Management Coalition. John Wiley and Sons, New York, 1997.

[6] W.M.P. van der Aalst. *Three Good reasons for Using a Petri-net-based Workflow Management System*. In S. Navathe and T. Wakayama, editors, Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96), pages 179–201, Camebridge, Massachusetts, Nov 1996.

[7] D. Hollingsworth. *The Workflow Reference Model - Issue 1.1*. Technical Report Document Number TC00-1003, Workflow Management Coalition, 1995. Accessed from http://www.wfmc.org/standards/docs/tc003v11.pdf on 27 June 2012.

[8] W.M.P. van der Aalst, A.H.M. ter Hofstede, and M. Weske. *Business Process Management: A Survey*. Accessed from http://150.145.63.3/ruffolo/progetti/projects/23.Semantic%20BPM-%20in%20OntoDLP/Business%20Process%20Management%20A%20Survey--10.1.1.14.2433.pdf.

[9] Weske, M., van der Aalst, W., and Verbeek, H. *Advances in Business Process Management*. Data & Knowledge Engineering 50,1 (2004).

[10] Alves, A., A. Arkin, et al. (2007). *Web Services Business Process Execution Language Version 2.0*, April 2007. Available from: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, OASIS.

[11] W.M.P. van der Aalst and A.H.M. ter Hofstede. *YAWL: Yet Another Workflow Language*. QUT Technical report, FIT-TR-2002-06, Queensland University of Technology, Brisbane, 2002.

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

[13] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. *Design and Implementation of the YAWL System*. In A. Persson and J. Stirna, editors, Advanced Information Systems Engineering, Proceedings of the 16th International Conference on Advanced Information Systems Engineering, volume 3084 of Lecture Notes in Computer Science, pages 142–159. Springer-Verlag, Berlin, 2004.

[14] K. Jensen, L.M. Kristensen, and L. Wells. *Coloured Petri nets and CPN tools for modelling and validation of concurrent systems*. International Journal on Software Tools for Technology Transfer, 2007.

[15] *Standard ML of New Jersey*. www.smlnj.org.

[16] Michael Kay, et al. *XSL Transformations (XSLT) Version 2.0*, January 2007. Available from: http://www.w3.org/TR/xslt20/.

[17] M. Adams et al. *"YAWL User Manual". Version 2.0*. The YAWL Foundation. September 2009.

[18] Weske, M., Vossen, G.: *Workflow Languages*. Bernus, Mertins, Schmidt (Editors): Handbook on Architectures of Information Systems. (International Handbooks on Information Systems), pp 359–379. Berlin: Springer 1998.

[19] Chun Ouyang. *How to Manipulate Data in YAWL, Version 0.4*. The YAWL Foundation. November 2005. Available from: http://yawlfoundation.org/yawldocs/YAWLDataManual-beta7.pdf

[20] K. Grigorova, *Process Modelling Using Petri Nets*, Int. Conf. on Computer Systems and Technologies, CompSysTech'2003, Bulgarian Computer Science Conference, Sofia, Bulgaria, Jun 2001.

[21] Breugel, F.v., Koshkina, M.: *Models and Verification of BPEL*. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf (2006).

[22] Russell, N., ter Hofstede, A., Edmond, D., van der Aalst, W.: *Workflow data patterns*. Technical Report FIT-TR-2004-01, Queensland Univ. of Technology. (2004).

[23] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut fur instrumentelle Mathematik, Bonn, 1962.

[24] K. Reinhardt, *Reachability in Petri nets with inhibitor arcs*, Technical report WSI-96-30, Wilhelm Schickard Institut fur Informatik, Universitat T ¨ ubingen (1996).

[25] Shudi (Sandy) Gao, C. M. Sperberg-McQueen and Henry S. Thompson. *XML Schema Definition Language (XSD) 1.1 Part 1: Structures*, 5 April 2012. Available from: http://www.w3.org/TR/xmlschema11-1/

[26] Scott Boag, et al. *XQuery 1.0: An XML Query Language (Second Edition)*, 14 December 2010. Available from: http://www.w3.org/TR/xquery/

[27] Anders Berglund, et al. *XML Path Language (XPath) 2.0 (Second Edition)*, 14 December 2010. Available from: http://www.w3.org/TR/xpath20/

[28] J. Eder, H. Groiss, and W. Liebhart: *Workflow Management and Databases*. Proc. 2 ème Forum Int. d 'Informatique Appliqu ée, Tunis, 1996.

[29] FOMICHEV, A., GRINEV, M., AND KUZNETSOV, S. 2006. *Sedna: A native XML DBMS*. In 32nd Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2006.

[30] Ilya Taranov et al. *Sedna: native XML database management system (internals overview)*. SIGMOD Conference, pages 1037-1046, 2010.

[31] *Sedna Native XML Database system*. http://www.sedna.org/

[32] Patrick Lehti, "*Design and Implementation of a Data Manipulation Processor for an XML Query Processor*," Technical University of Darmstadt, Darmstadt,Germany, Diplomarbeit, August 2001.

[33] Jonathan Robie, et al. *XQuery Update Facility 1.0*, 17 March 2011. Available from: http://www.w3.org/TR/xquery-update-10

[34] A. Bauer, S. Eckel, T. Emter, A. Laubenheimer, E. Monari, J. Mosgraber, F. Reinert, *N.E.S.T. - Network Enabled Surveillance and Tracking*, Future security: 3rd Security Research Conference Karlsruhe; 10th-11th September 2008.

[35] J. Alcober, G. Cabrera, X. Calvo, E. Eliasson, K. Groth, P. Pawalowski, *High Definition Videoconferencing*: The Future of Collaboration in

Healthcare and Education, eChallenges e-2009 Conference, October 21-23 2009, Istanbul, Turkey.

[36] C. Mazurek, M. Stroi˜nski, D. Walczak, M. Wolski, *Supporting high-tech crime investigation through dynamic service integration*, Fifth International Conference on Networking and Services, 2009.

[37] H. Che, Y. Li, A. Oberweis and W. Stucky, *Web Service Composition Based on XML Nets*: Proceedings of the 42nd Hawaii International Conference on System Sciences, 2009.

[38] Lenz, K.; Oberweis, A.: *Interorganizational Business Process Management with XML Nets*, In H. Ehrig, W. Reisig, G. Rozenberg, H. Weber, Petri Net Technology for Commu-nication-Based Systems, Advances in Petri Nets vol. 2472 of LNCS, pp. 243-263. Springer-Verlag, 2003.

[39] P. Reimann, H. Schwarz and B. Mitschang: *Design, Implementation, and Evaluation of a Tight Integration of Database and Workflow Engines*. In: Journal of Information and Data Management. Vol. 2(3), SBC - Brazilian Computer Society, 2011.

[40] W.M.P. van der Aalst, Barros, A.P, Ter Hofstede, A.H.M., & Kiepuszewski, B.,. 2000. *Advanced Workflow Patterns*. In Proceedings of the 7th International Conference on Cooperative Information Systems (CooplS '02). Springer, London, UK, 18-29.

[41] W.M.P. van der Aalst. *XML-Based Schema Definition for Support of Interorganizational Workflow*, Information Systems Research (14:1), March 2003, pp. 23-46.

[42] S. Kepser. *A simple proof of the Turing-completeness of XSLT and XQuery*. In T. Usdin, editor, Extreme Markup Languages 2004. IDEAlliance, 2004. Available at http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2%004Kepser01.html

[43] Rossi, D., Turrini, E.: *What your next workflow language should look like*. 2nd International Workshop on Coordination and Organization (2006).

[44] *Zorba: The XQuery Processor*. http://www.zorba-xquery.com

[45] *XQilla*. http://xqilla.sourceforge.net.

Appendix A. XML Schema for the XML-based workflow definitions

This appendix depicts the XML Schema representing the syntax of our XML workflow definitions.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="PetriNet">
  <xs:complexType>
   <xs:sequence>
    <xs:choice maxOccurs="unbounded">
     <xs:element ref="Criteria" />
     <xs:element ref="Transition" />
     <xs:element ref="Place" />
     <xs:element ref="Arc" />
    </xs:choice>
   </xs:sequence>
  </xs:complexType>
  <xs:unique name="UniqueID">
   <xs:selector xpath="./*" />
   <xs:field xpath="@id" />
  </xs:unique>
 </xs:element>
 <xs:element name="Condition">
  <xs:complexType>
   <xs:simpleContent>
    <xs:extension base="xs:string">
     <xs:attribute name="type" type="xs:string" use="required" />
    </xs:extension>
   </xs:simpleContent>
  </xs:complexType>
 </xs:element>
 <xs:element name="Criteria">
  <xs:complexType>
   <xs:simpleContent>
    <xs:extension base="xs:string">
     <xs:attribute name="type" type="xs:string" use="required" />
    </xs:extension>
   </xs:simpleContent>
  </xs:complexType>
 </xs:element>
 <xs:element name="Place">
  <xs:complexType>
   <xs:attribute name="id" type="xs:unsignedByte" use="required" />
  </xs:complexType>
 </xs:element>
 <xs:element name="Arc">
  <xs:complexType>
```

```
    <xs:attribute name="type" use="required">
     <xs:simpleType>
      <xs:restriction base="xs:string">
       <xs:enumeration value="POSITIVE" />
       <xs:enumeration value="RESET" />
       <xs:enumeration value="BLOCK" />
       <xs:enumeration value="NEGATIVE" />
      </xs:restriction>
     </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="fromId" type="xs:unsignedByte" use="required" />
    <xs:attribute name="toId" type="xs:unsignedByte" use="required" />
   </xs:complexType>
   <xs:keyref name="fromKeyRef" refer="UniqueID">
    <xs:selector xpath="." />
    <xs:field xpath="@fromId" />
   </xs:keyref>
   <xs:keyref name="toKeyRef" refer="UniqueID">
    <xs:selector xpath="." />
    <xs:field xpath="@toId" />
   </xs:keyref>
  </xs:element>
  <xs:element name="Action">
   <xs:complexType>
    <xs:simpleContent>
     <xs:extension base="xs:string">
      <xs:attribute name="type" type="xs:string" use="required" />
     </xs:extension>
    </xs:simpleContent>
   </xs:complexType>
  </xs:element>
  <xs:element name="Transition">
   <xs:complexType>
    <xs:sequence minOccurs="0">
     <xs:element ref="Condition" minOccurs="0" maxOccurs="unbounded" />
     <xs:element ref="Action" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="id" type="xs:unsignedByte" use="required" />
    <xs:attribute name="type" use="required">
     <xs:simpleType>
      <xs:restriction base="xs:string">
       <xs:enumeration value="DEFAULT" />
       <xs:enumeration value="EMITTER" />
       <xs:enumeration value="COLLECTOR" />
       <xs:enumeration value="XOR" />
      </xs:restriction>
     </xs:simpleType>
    </xs:attribute>
   </xs:complexType>
  </xs:element>
</xs:schema>
```