Reliable coordination of data management services

Javier Alfonso Espinosa Oviedo

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité: **INFORMATIQUE**
Arrêté ministériel: 7 août 2006

Présentée par

Javier A. ESPINOSA-OVIEDO

Thèse dirigée par Christine COLLET, co-encadrée par Genoveva VARGAS-SOLAR et codirigée par José Luis ZECHINELLI-MARTINI

Préparée au sein du Laboratoire d'Informatique de Grenoble (LIG) dans l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (EDMSTII).

Coordination fiable de services de données à base de politiques actives

Thèse soutenue publiquement le **28/10/2013** devant le jury composé de:

Mme. Daniela GRIGORI
Prof. Université Paris Dauphine (*Rapporteur*)

M. Mohand-Saïd HACID
Prof. Université Claude Bernard Lyon 1 (*Rapporteur*)

M. Khalid BELHAJJAME
M.C. Université Paris Dauphine (*Examinateur*)

M. Samir TATA
Prof. TELECOM SudParis (*Examinateur*)

Mme. Christine COLLET
Prof. Grenoble INP (*Directrice de thèse*)

Mme. Genoveva VARGAS-SOLAR
C.R. CNRS (*Co-encadrante de thèse*)

M. José Luis ZECHINELLI-MARTINI
Prof. Universidad de las Américas Puebla (*Codirecteur de thèse*)

Université Joseph Fourier / Université Pierre Mendès France / Université Stendhal
Université de Savoie / Grenoble INP

Intentionally left blank

Acknowledgements

I express my deep gratitude to my research advisors Dr. Genoveva VARGAS-SOLAR, Prof. Christine COLLET and Prof. José Luis ZECHINELLI-MARTINI, for their patient guidance, enthusiastic encouragement and useful critiques of this research work.

I thank Prof. Mohan-Saïd HACID and Prof. Daniela GRIGORI for accepting reviewing this work, and Prof. Samir TATA and Dr. Khalid BELHAJJAME for accepting participating as members of the jury.

I would also like to extend my thanks to the members of the Heterogeneous Autonomous Distributed Data Services (HADAS) group and the Laboratory of Informatics of Grenoble (LIG) for their support during all these years.

Finally, I would like to thank the Mexican National Council for Science and Technology (CONACyT) for providing the funding that allowed me to undertake my research and the Franco-Mexican Laboratory of Informatics and Automatic Control (LAFMIA) for funding the last months of my research and for giving me the opportunity to travel and meet so many interesting people.

Intentionally left blank

Para una dulce **perla**

blanca **mexicana**

Poli
Zoe

Intentionally left blank

Table of Contents

Intentionally left blank

1 Introduction

1.1 Context and Motivation

Along with the emergence of services based computing and Web 2.0, the Web has become a development platform for building "*software as a service*". The notion of service abstracts an application that can be accessible on the network and which can be used as a building block for constructing more complex applications. The principle is based on the idea that *software* can be developed by coordinating calls to operations exposed by services without worrying about distribution problems (e.g., services location, message communication, execution environment).

Current standards implement services' coordinations by combining different languages and protocols. For instance, WSDL, SOAP and BPEL are example of languages used for describing (i) the operations exported by a service, (ii) the message exchange patterns (i.e., protocols) supported by a service in order to call one of its operations and (iii) the order among the services' operations calls (respectively). However, sometimes calling a service' operation requires to address complex non-functional properties (NFP) before accessing its functionality (e.g., *authentication*, *access control*, and *data encryption* protocols). In consequence, these properties must be weaved within a services' coordination logic for achieving an operation call.

The selection of the adequate strategies for adding a specific NFP to a services' coordination (e.g., security, transactional behavior and adaptability) is responsibility of a programmer. In consequence, the development of a services' coordination can be a complex and a time-consuming task, which is opposed to the philosophy of services that aims at facilitating the integration of distributed applications.

1.2 Problem Statement and Objectives

We believe that it is possible to provide solutions that can keep the simplification principle of services oriented programming by separating services calls from the strategies that can be used for interacting with services.

The problem is that the whole business logic underlying a services' coordination is expressed as a monolithic block, namely the *process specification*. Consider for instance that a programmer wants to develop a coordination called *Status Updater* that (i) estimates the *mood* of people based on the music they are listening and (ii) updates a user social net-

works status with the song information and the user estimated mood (e.g., if we consider that people listening to pop-music are happy and most songs listened by a person are pop-music, then the coordination estimates that the person is happy and updates the person status with the name of the song and the string "happy"). Let us suppose that, for developing this application, the programmer has to coordinate calls to the following services' operations:

- Music Services like **LastFM** that export methods for obtaining information about the music a given user is listening to. For instance, the operation:

  get-Last-Song (userID) : String

  gets the *string* representation of the last song that the user identified by *userID* has listen to.
- **Facebook** and **Twitter** services that export methods for getting and updating the status of a given user. For instance, the operations:

  get-Status (userID) : String

  update-Status (userID, new-status) : String

  get and update the string representation of the status of a user identified by *userID* in a social network.
- **Mood** Service that export methods for estimating the mood of a user. For instance, the operation:

  get-Mood (userID) : String

  abstracts a complex process that uses a log of the music listened by the user in LastFM during some period of time, and then according to the classification of each song, it will determine the string representing the user mood.

The corresponding services' coordination is depicted in **figure 1.1** using the Business Process Management Notation (BPMN). As shown in the figure, the process *Status Updater* is represented as a workflow composed of a set of activities, each activity in charge of calling a service' operation. The workflow works as follows: first it contacts the music service **LastFM** for retrieving the last song a user has listen to. Then this information is sent to the **Mood** service in order to calculate the mood of the user. Finally, the **Twitter** and **Facebook** services are contacted in parallel for updating the status of the user with the song and the user's mood.

Now consider that the developer wishes to ensure the following NFPs:

- **Authentication management**: only registered users can access the information provided by the services.
- **Exception management**: since the 3 services are accessible via an unreliable network, if the network is not available, the coordination must try to contact the service several times after a certain delay before stopping and notifying the failure.

Each NFP that must be enforced throughout the process has to be expressed in terms of activities that must be integrated to the process specification. Thus, the resulting modified process is complex and hard to maintain because of the *lack of modularity* (i.e., since all the business logic is defined in one unit, it is difficult to reuse parts of it). Another problem is

the *lack of flexibility* due to the fact that process-oriented languages assume that services' coordinations are predefined and do not evolve. Thus, the only way to accommodate changes is by modifying processes definitions (i.e., there is no support for dynamic changes for adding NFPs to a process).

We believe that the lack of flexibility is tightly related to the lack of modularity. In consequence, if we can break down the business logic underlying the composition into several parts or modules, the composition becomes more flexible since each of these parts can evolve independently of the rest.

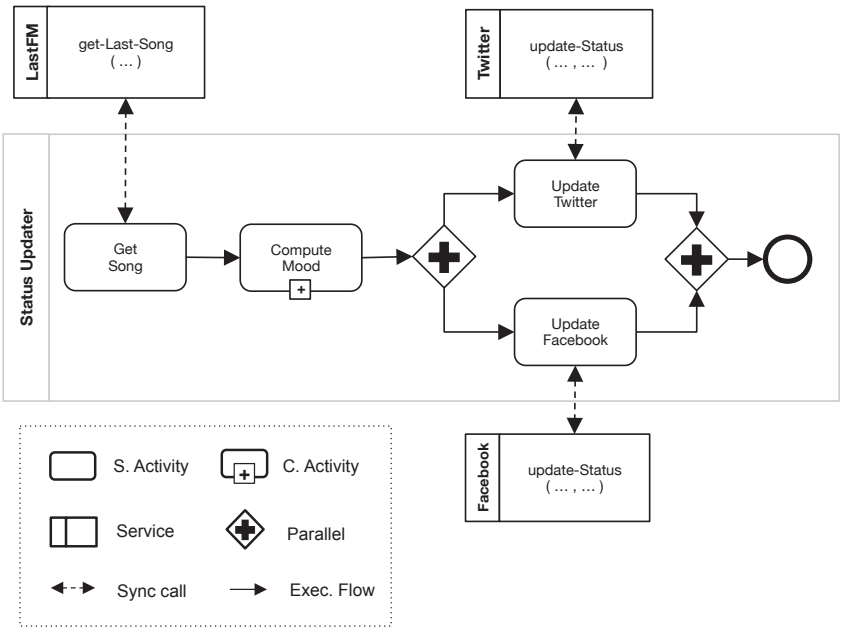

Figure 1.1 Status Updater process expressed as a workflow (BPMN diagram)

1.2.1 Problem Statement

The problem we address is providing NFPs to services' coordinations implemented as workflows. NFPs describe how to execute services' coordinations with respect to observable properties (e.g., reliability, security, adaptability). Adding NFPs to services' coordinations implies:

- Determining the elements concerned with these properties (e.g., activities, workflow' control flow, interaction of an activity with a service, data exchanged among workflow' activities and between an activity and a service).
- Modeling and expressing the strategies that implement the NFPs (e.g., security, exception handling, transactions, persistency, etc.).
- Associating such strategies to the service' coordination elements.
- Defining execution strategies that can specify how to synchronize the execution of a service' coordination with NFPs maintaining their separation.

1.2.2 Objectives

The objective of our work is to propose a model and associated execution strategies for adding NFPs to services' coordinations ensuring:

- The independence of the specification of NFPs and services' coordination in order to facilitate its maintenance and reuse.
- The dynamic reinforcement of NFP at execution time.

Besides, our work aims at providing a system (proof of concept) to show the feasibility of the model:

- Showing how to enact the services' coordination with NFPs.
- Specifying a general architecture of an engine able of executing a service' coordination with non-functional properties.

1.3 Approach and Main Contributions

Our work is inspired in the philosophy of *separation of concerns* adopted in the construction of middleware, where the NFPs and the business logic of an application are defined independently of each other. Thus, this thesis presents an approach for *observing* and *reacting* on the execution of *data services' coordinations* in order to ensure NFPs specified by the coordination designer. Instead of weaving different NFPs within the services' coordination logic, we propose to define *active policies* that contain the strategies implementing NFPs and then allow their use when executing a services' coordination. Such an approach (i) encourages a modular and flexible construction of applications and eases their maintenance if such strategies change and (ii) makes possible to associate a personalized behaviors to coordinations (e.g., using policies it is possible to associate an atomic behavior to the way data is retrieved and update from different social network services).

Our approach is based on the notion of *active policy-based workflows*. A policy-based workflow is a workflow composed of an ordered set of activities that implement a services' coordination and which has active policies associated to it. Activities represent programs implementing a call to a service' operation. The order among the activities specifies in which order services' operation has to be called. Active policies represent the conditions in which an activity or a set of activities must be executed and what to do if such conditions are not satisfied.

The following list summarizes the main contribution of this work:

- **Data Oriented Services Taxonomy** that classifies services according to the operations they export regarding the access to the data they "hide". The taxonomy allows to understand the differences between so called "*business services*" that are in fact applications exporting their functions as services, and *data services* that are centered in manipulation and retrieval of *data collections*.
- **Active Policy Model** (AP Model) and its associated **Active Policy Language** (AP Language) for developing active policy-based workflows with associated NFPs. The model provides a general pattern for defining different policy types that address security, persistence, exception handling, and atomic behavior. Using these specialized policy types, a programmer can specify concrete policies according to the coordination requirements and associate them to a target workflow.

- **AP Execution Model** and the execution strategies that we propose for executing active policy-based workflows. Based on the AP Model, the AP Language and the AP execution strategies we designed an *active policy-based workflow engine* for executing ensuring NFPs in the context of data services' coordination.

1.4 Document Organization

The reminder of the document is organized as follows:

- **Chapter 2** presents the background and state of the art related to our work. It first introduces the taxonomy of data oriented services that we propose. Then it details the models used for characterizing services coordinations an their execution. Finally the chapter classifies and analyzes the main approaches used for adding NFPs to services' coordinations.
- **Chapter 3** describes our AP Model and its associated language for expressing active policies and associating them to workflows implementing services' coordinations. The chapter introduces the main concepts of the model and the target workflow model to which policies can be associated.
- **Chapter 4** describes the execution of an active policy-based workflows that provides the dynamic aspect of our contribution. It introduces the execution strategies that we propose for synchronizing the execution of active policies with coordination workflows (i.e., workflows implementing services' coordinations).
- **Chapter 5** presents the definition of *reliable service coordination* using the concepts proposed in our AP Model. A reliable service coordination includes *exception management* and *state management policies* for making a services' coordination fault tolerant with atomicity and persistency properties. The chapter shows how to use the AP Model for defining such policies and then how to associate them to a target workflow.
- **Chapter 6** describes the general architecture of our policy engine that can execute active policy-based workflows according to the proposed strategies. The chapter also describes the experiment we conducted in order to validate our approach in the context of data services coordination. The use cases focus on the use of service coordinations for implementing queries that can have associated NFPs. Queries are executed without the need of an underlying full-fledged DBMS.
- **Chapter 7** concludes this document and discusses perspectives of our work, underlining some challenges that remain open.

Intentionally left blank

2 NFPs in Service's Coordinations

This chapter proposes a state of the art of approaches modeling and providing non-functional properties (NFP) for service's coordinations. Adding NFP to service's coordinations implies (i) characterizing the type of services building a coordination; (ii) modeling the coordination and the NFP; (iii) determining the type of NFP that can be associated to coordinations and how they can be reinforced.

Accordingly, the remainder of the chapter is organized as follows. **Section 2.1** introduces a taxonomy proposed in this work for classifying services. The taxonomy characterizes services in terms of the type of operations they export: those devoted for processing and managing data, and those providing functions ready to use. Services are building units that can be used for implementing application. This is done be coordinating calls to their operations. Thus, **Section 2.2** describes existing coordination models. Applications can have associated NFP for addressing aspects complementary to their application logic. **Section 2.3** synthetizes approaches for classifying and modeling NFP. And **Section 2.4** analyses strategies for adding NFP to service's coordinations. **Section 2.5** concludes the chapter.

2.1 Service Taxonomy

Services are software or hardware units that export operations over a network (e.g., Internet), and that play a growing part in business-to-business interactions. An *Application Programming Interface* (API) expressed in an *Interface Description Language* (IDL) describes the operations exported by a service. Current proposals assume that a service provider implements the service. Using a service implies calling a provider operation, via a protocol, and (eventually) waiting for execution results under a client-server interaction. Services are registered in a name service that provides look up functions so that applications can find services. For example in the Web, a typical IDL example is WSDL (Web Services Description Language) **[Wc00a]** an XML based standard **[Oasi04a]** for defining web services as a set of methods that can be bound using the SOAP protocol **[Wc00b]**. The services approach enables the construction of new applications out of existing services by merely ordering calls to their operations. These new application can be wrapped themselves as services.

Figure 2.1 shows the overview of the service taxonomy that we propose. The taxonomy identifies three families of services according to the role of the functions they export within an application: **business oriented**, **application oriented** and **data oriented**. The data oriented family is further classified into **data processing**, **data administration**, **data provision** and **data querying** ser-

vices. Finally, data provision services according to the mode in which they deliver data they can be **on-demand** and **stream** services.

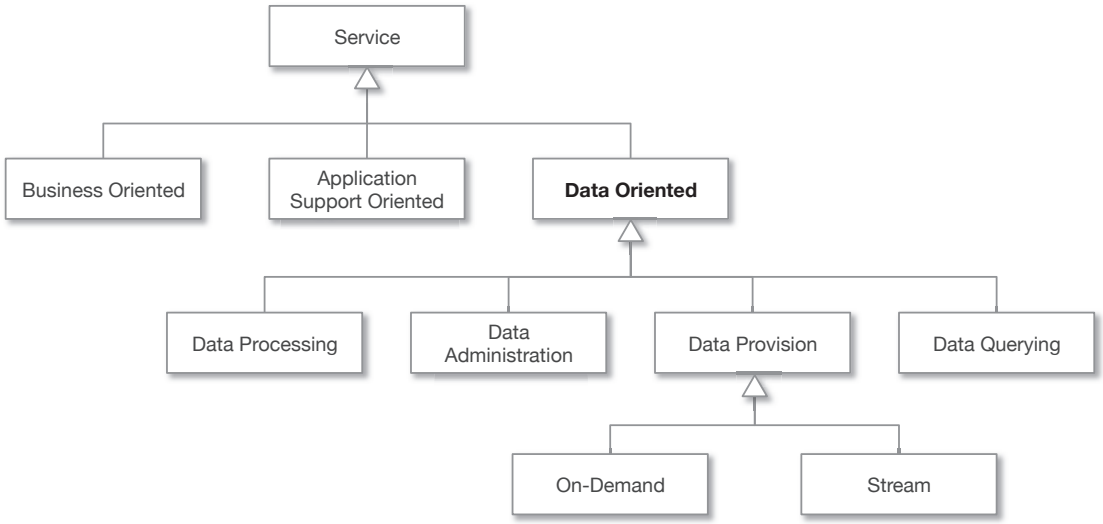

Figure 2.1 Service classification (UML class diagram)

The following sections describe these service families and give concrete examples.

2.1.1 Business Oriented

Business oriented services provide «*ready to use*» functions for final users. There is no standard API to be exported and, operations are related to the objective of the service. Specific purpose applications are wrapped as business services and provide ad-hoc operations, for instance, hotel reservation services, online stores like e-Bay or Amazon. They can correspond to the notion of Software as a Service (SaaS) introduced by service oriented cloud architectures.

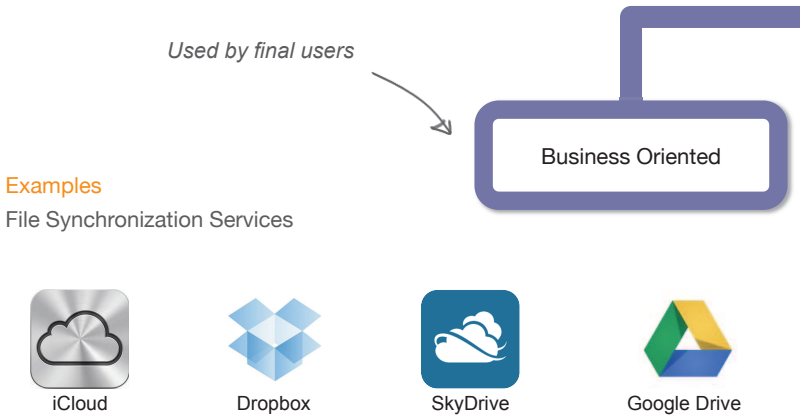

Figure 2.2 Business Oriented Services

Examples of business services oriented to data management are file synchronization services like Dropbox, SkyDrive, GoogleDrive as shown in **figure 2.2**. These services export

methods specialized for managing files within a storage space and eventually for managing the storage space size.

2.1.2 Application Support Oriented

Application support oriented services provide operations for managing platform systems. They are used by other applications or services as building blocks. There is no standard API to be exported and the exported API is related to the objective of the service. Examples of these services are Virtual Machines, message services like Message Queues, or full fledged DBMS exported as services (Data as Service), as shown in **figure 2.3**.

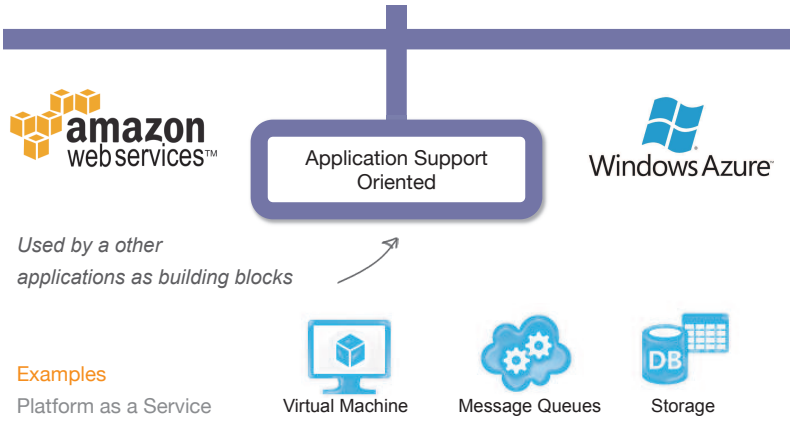

Figure 2.3 Application Support Oriented Services

This type of service can correspond to the "Platform as a Service" (PaaS) family identified in cloud architectures. Examples of providers of this type of services are Amazon Web services and Windows Azure.

2.1.3 Data Oriented Services

Data oriented services are used for exchanging (receiving and sending) data collections (see **figure 2.4**). This family of services is specialized into data processing, data administration and data provision services described in the following lines.

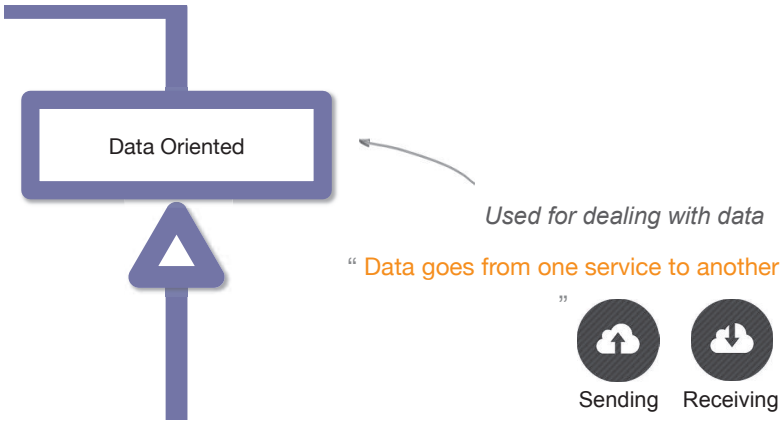

Figure 2.4 Data Oriented Services

Data Processing Service

This type of service exports a set of operations for processing data of a given collection (filtering, grouping by, counting elements of collections of elements see **figure 2.5**). The LINQ service of the .NET platform is an example of such type of service. It provides a language for expressing declarative/imperative programs for manipulating data. Coupled with a parallel execution service like Dryad, this data processing service is able to execute data processing tasks in parallel.

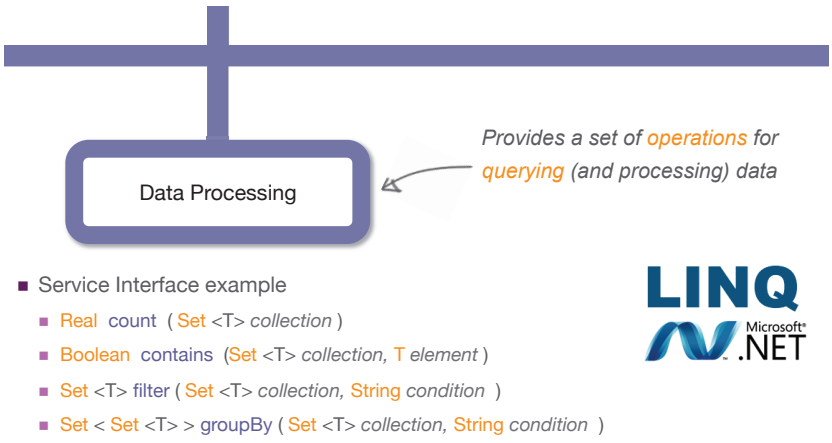

Figure 2.5 Data processing interface service example

Data Querying Service

As shown in **figure 2.6**, this type of service exports operations for expressing and processing queries on collections of data. A service of this type evaluates query expressions and returns the corresponding results expressed according to a specific data model (e.g., relational, semi-structured). For example, the service YQL in **figure 2.6**, accesses a data collection stored in a given address, (e.g. a collection of photos in Flickr). It receives YQL expressions, for example for selecting the photos in Flickr where the caption contains the string "*San Francisco*". The result is a collection of JSON elements.

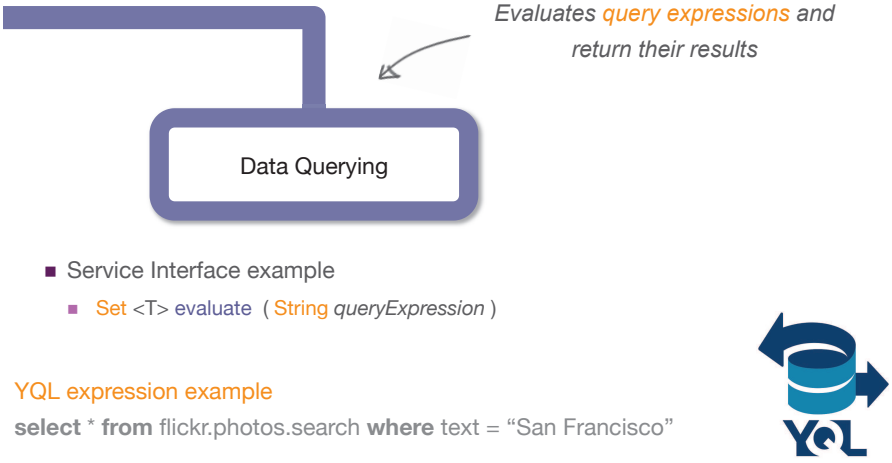

Figure 2.6 Data querying service interface example

Data Administration Service

This type of service exports operations for manipulating collections and their elements. As shown in **figure 2.7**, the interface provides methods for updating, inserting and deleting elements of a collection. For example, updating information about the contacts list of a user account in Facebook or inserting songs to the LastFM profile of a user.

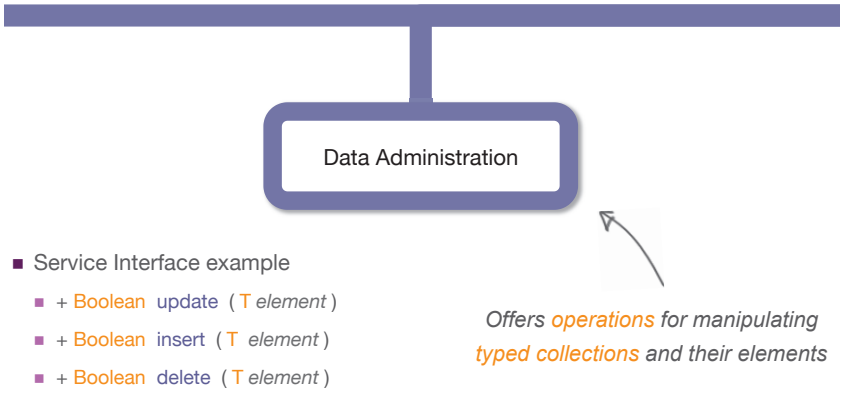

Figure 2.7 Data administration services interface example

Data Provision Service

A service of this type exports operations for giving read access to a data collection (for example a play list). As shown in **figure 2.8,** according to the rate in which data is provided, this type of service is specialized into on demand and continuous services.

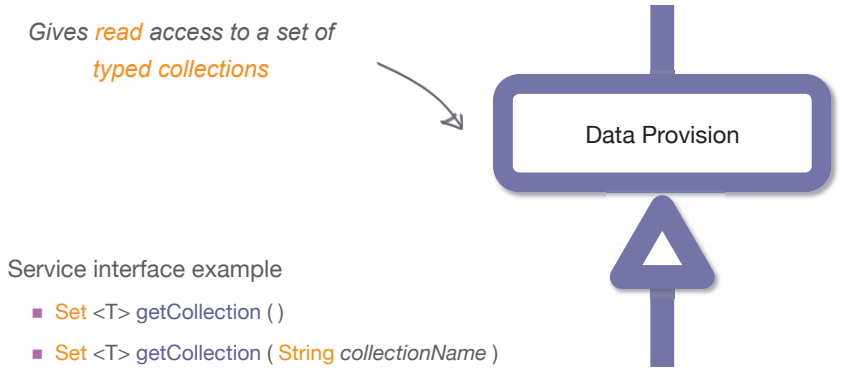

Figure 2.8 Data provision service interface example

- **On demand** services produce data in response to a given request that returns results of a given type, for example a tuple or a collection of tuples. An on demand service works under a Request - Response Communication Protocol. Thus, the client calling an operation for getting a collection will block until the complete collection is retrieved (see **figure 2.9**).

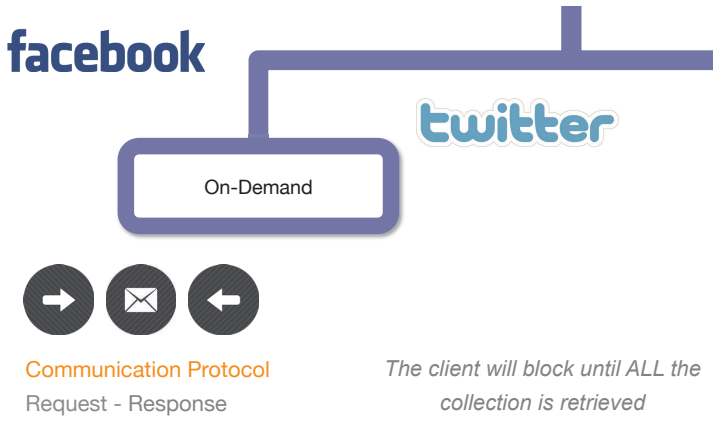

Figure 2.9 On demand data oriented services example

- **Continuous** service produces data streams[1] once a consumer has subscribed to it. The subscription is performed via a subscription operation exported by the service. The operation call includes parameters such as the destination address for the data stream and the time interval during which the subscription will be valid or until an event happens (an unsubscription explicit demand). A continuous data service produces tuples at a certain rate (e.g. 10 tuples/sec), which may vary in time. The client specifies a callback for receiving the data.

Figure 2.10 Continuous data oriented services example

Facebook and Twitter are examples of on-demand data services, while RSS notification services and music providers as Deezer are continuous data provision services.

[1] A stream is a continuous (and possibly "infinite") sequence of tuples. Each of these tuples has a temporal timestamp attribute denoting the time at which it was produced.

2.2 Service Composition

Service composition is a concept used to design and develop applications on the basis of existing services **[DuSc05]**. Thus, a service composition consists of a set of services, each of which realizes a process activity. In order to do this, composite applications invoke enterprise applications exported as services directly using standardized protocols (REST and WSDL). The order of these activities can be specified using a process model. Service composition is a recursive concept: each service composition can be specified as a service. Therefore, each service composition can participate as a building block in other, higher-level service compositions. Web services composition is a realization of these concepts **[Wesk07]**.

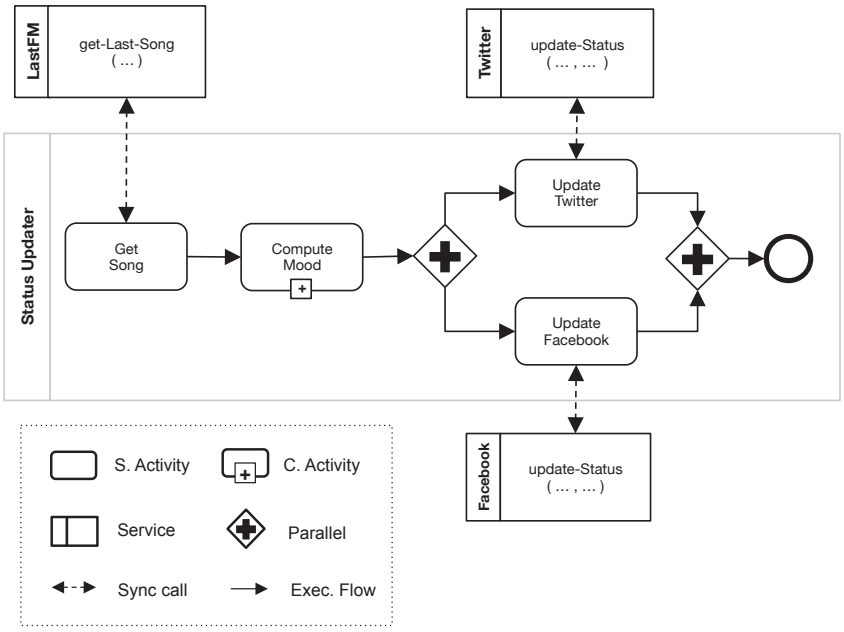

Figure 2.11 Workflow example

The diversity of systems **[BDSN02, CIJK00, LASS00, MGKS96]**, and languages **[LCRS01]** that have been recently proposed shows the potential and the relevance of service composition for building applications. Generally speaking, two complementary techniques are used for building and composing services: wrapping **[CIJK00]** and coordination **[BDSN02]**. Using the wrapping technique, pre-existing applications are wrapped to provide an entry point for activating the service. Wrapped services are then orchestrated by a workflow that is executed by an engine. For example, BMPL **[Bpmi04]** is a workflow based language that is used for specifying the orchestration of web services. Approaches proposed in **[GeHS95, Wmc96]** can be used for specifying and automating services orchestration[2]. Coordination **[ADHW02]** describes the automated arrangement (i.e., coordination) and management of complex computer systems, consisting of services. Service coordination can be modeled using

[2] We define an orchestrator as the entity that manages complex cross-domain (system, enterprise, firewall) processes and handle exceptions.

business process models, coordination models and workflow models. **Figure 2.11** already shown in the Introduction illustrates how a workflow is used for coordinating services. Activities are used to control services calls, data flow specifies data exchange, and control flow describes dependencies among them. The control flow is expressed using ordering operators such as *sequence*, *and-split*, and *and-join* **[AHKB03]**.

Certain parts of a business process can be enacted by workflow technology. A workflow management system (see **Appendix A**) can make sure that the activities of a business process are performed in the order specified, and that the information systems are invoked to realize the business functionality. A workflow realizes a part of a business process, with regard to the types of activity; system activities are associated with workflows.

Models can be classified, from our point of view, into those adopting the notion of composition based on the notion of activity; and those focusing on the control flow, that is, the way in which services operations are called for defining an application logic. The following describes both approaches.

2.2.1 Activity Oriented Workflow Model

A business process model consists of a set of activity models and execution constraints among them. A business process model defines the ways in which operations are carried out to accomplish the intended objectives of an organization. Such a model remains an abstraction and depends on the intended use of the model. It can describe the workflow or the integration between business processes. It can be constructed in multiple levels. **Figure 2.12** introduces a model of the concepts of business process expressed in the Unified Modeling Language (UML).

According to the UML business process model, a business process consists of activities whose coordinated execution realizes some business goal. These activities can be system activities, user interaction activities, or manual activities. Manual activities are not supported by information systems. User interaction activities go a step further: these are activities that knowledge workers perform, using information systems. There is no physical activity involved. Since humans use information systems to perform these activities, applications with appropriate user interfaces need to be in place to allow effective work. These applications need to be connected to back-end application systems that store the entered data and make it available for future use. Some activities that are conducted during the enactment of a business process are of manual nature, but state changes are entered in a business.

System activities are executed by information systems assuming that the actual parameters required for the invocation are available. If a human user provides this information, then it is a user interaction activity. Both types of activities require access to the respective software systems.

Business process modeling tools are used for expressing implementing and executing business processes. Some business process modeling tools are: Business Process Model and Notation (BPMN); Cognition enhanced Natural language Information Analysis Method (CogNIAM); Extended Business Modeling Language (xBML); Event-driven process chain (EPC); ICAM DEFinition (IDEF0); Unified Modeling Language (UML), extensions for business process such as Eriksson-Penker's.

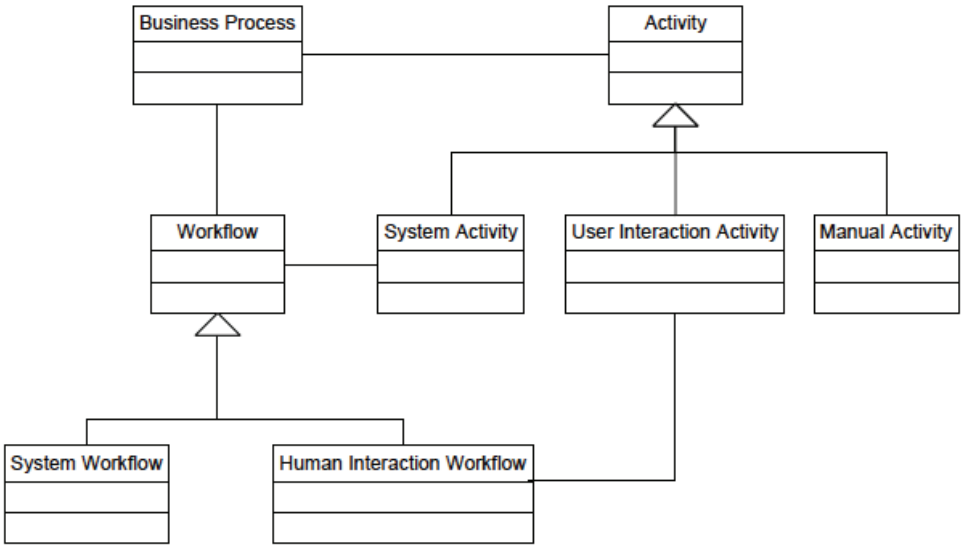

Figure 2.12 Business process conceptual model [Wesk07]

2.2.2 Orchestration

Orchestration is the process that controls the interactions among services. Provided a set of services their interactions are ordered with respect to a given application specification, defining a control flow. The following lines describe a general overview of patterns that can be used for defining a control flow from two points of view: a centralized approach through orchestration and a "distributed" approach through choreography.

Control Flow Patterns

Control flow patterns express process orchestrations (workflows). Basic control flow patterns include *sequence*, *and-split*, and *and-join*, as well as exclusive *or-split*, and exclusive *or-join* **[AHKB03]**.

- **Sequence**: An activity in a workflow process is enabled after the completion of a preceding activity in the same process.
- **And-split** (parallel split): The divergence of a branch into two or more parallel branches each of which execute concurrently. This pattern allows a single thread of execution to be split into two or more branches that can execute activities concurrently. These branches may or may not be re-synchronized at some future time.
- **And-join**: The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. And-join provides a means of re-converging the execution threads of two or more parallel branches. In general, these branches are created from an And-split construct earlier in the process model. The thread of control is passed to the activity immediately following the synchronizer once all of the incoming branches have completed.
- **Xor-split** (exclusive choice): The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the outcome of a logical ex-

pression associated with the branch. This pattern allows the thread of control to be directed to a specific activity depending on the outcome of a preceding activity, the values of elements of specific data elements in the workflow or the results of a user decision. The routing decision is made dynamically allowing it to be deferred to the latest possible moment at runtime.

- **Xor-join** (simple merge): The convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.

Any process meta-model supports these control flow patterns that are referred to as simple control flow patterns in **[RHAM06]**. In this work in fact authors refer to twenty basic control flow patterns expressed using Coloured Petri Nets. More recently, authors have identified a number of additional scenarios. As a consequence twenty three new patterns, some of them new and some based on specializations of existing patterns have been proposed: advanced branching and synchronization, structural, multiple instance, state based, cancellation, loop and recursive variations with blocking and non-blocking activities. In this thesis we assumed workflow models addressing the simple patterns, so we will not further describe these advanced patterns.

Choreography

The interactions of a set of business processes can be also specified in a choreography. The term choreography indicates the absence of a central agent that controls the activities in the business processes involved. Sending and receiving messages achieves interaction. Service choreography is a form of service composition in which the interaction protocol between several partner services is defined from a global perspective. At run-time each participant in a service choreography executes its part of it (i.e. its role) according to the behavior of the other participants **[Pelt03]**. A role specifies the expected messaging behavior of the participants that will play it in terms of the sequencing and timing of the messages that they can consume and produce **[SBFZ08]**. A set of service choreography formalisms have been proposed: Interaction Petri Nets **[DeWe07]** and Open Workflow Nets **[Schm05]**, Finite State Machines **[BGGL06]**, Guarded Automata **[BuSF06]**, Timed Automata **[MCHP08]**, Pi calculus **[BaWR09]**, Process calculi **[KaPi06, QZCY07]**.

2.2.3 Service Composition Languages

Business Process Administration languages:

- BPML (Business Process Modeling Language) is a meta-language for describing business processes. A business process is described as a set of simple activities representing the invocation of a service operation, the sending and reception of the parameters of the invocation and optional, sequential and parallel activities.
- BPEL4WS (Business Process Execution Language for Web Services) specifies the control-flow and the logic of a services' coordination. BPEL4WS specifies a service as an interface described in WSDL.
- YAWL (Yet Another Workflow Language) is a workflow language based on workflow patterns. The formal semantics of YAWL is defined as a system of labeled transactions. YAWL considers the patterns or-join, cancellation sets and multi-

instance activities. YAWL adds syntactic elements to reflect patterns such as xor-split, xor-join and or-split.

Web Services Flow Language can be considered an XML serialization of Flow Definition Language. This script language was used in IBM's workflow product, enhanced with concepts to access Web services. It is based on a graph-based process language, where activities are ordered in an acyclic form by control flow links. Data dependencies are specified by a data flow among activities. Process behavior is specified by transition conditions attached to control flow links. This means that there is no explicit split and join behavior defined. However, by attaching conditions, any splitting behavior can be realized.

[That01] is a block structured language that was used in BizTalk, Microsoft's enterprise application integration software focusing the integration of heterogeneous back-end systems using processes. In block-structured languages, a strict nesting of control flow blocks is used to structure business processes. As a result, for instance, the paths following an "**and-split"** can never be combined by a join other than an "**and-join"**.

The standard in Web services composition is the Business Process Language for Web Services, WS-BPEL, or BPEL **[AAAB07]**. It is the outcome of a combination of the Web Services Flow Language by IBM and XLANG **[That01]** by Microsoft. BPEL uses a block structuring to organize service compositions, and links can be defined to express graph-like structures **[Wesk07]**. Using WS-BPEL it is possible to specify an execution context (scopes) for encapsulating the application logic with logical variables, exception managers, compensation mechanisms and event detection. An execution context can be serialized for managing the concurrent access to its variables.

W3C has proposed specifications for expressing choreography. WS-CDL (WS Choreography Definition Language) is a language based on XML describing the collaboration and the behavior of peer-to-peer services inspired by Pi calculus. The collaboration is expressed as a set of messages exchanged for achieving a common goal. Web Service Choreography Interface (WSCI) is an XML-based specification proposed by Intalio, Sun Microsystems, BEA Systems and SAP AG. It serves as input to the Web Service Choreography Description Language (WS-CDL). Academic proposals for service choreography languages include Let's Dance **[ZBDH06]**, BPEL4Chor **[DKLW07]**.

Coordination models and languages are used for expressing and implementing application logics that user services. Yet, software systems, aside from implementing all the desired functionality, must also cope with non-functional properties (NFP) such as: reliability, security, accuracy, safety, and performance. The following section defines these properties and presents strategies for specifying them, when they are intended to be associated to services' coordinations.

2.3 Non-Functional Properties

A non-functional property (NFP) specifies criteria about the behavior of a system. These criteria are related to the conditions in which they are executed and to its performance. Non-functional properties are also referred as "constraints", "quality attributes", "quality goals", "quality of service requirements" and "non-behavioral requirements" **[StGr05]**. In the case of service-based applications, non-functional requirements concern the application itself as well as its component services.

The majority of the work on NFPs uses a product-oriented approach, which is concerned with measuring how much a software system is in accordance with the set of NFPs that it should satisfy **[BBKL78, FePf96, KiDa96, Lyu96, MuIO90]**. We are interested also in approaches that provide means for implementing NFP and reinforcing them at execution time.

2.3.1 Classifying NFPs

[Souz12] proposes a classification of NFP as a result of a study concerning software methodologies for the construction of service-oriented systems. The classification is shown in **Figure 2.13** and it is organized in three layers representing: *application modeling*, *services composition* and *services*. The service composition layer serves as an integration layer between the services layer that exports methods and has associated constraints and characteristics; and the application layer that expresses requirements (NFP in the Figure).

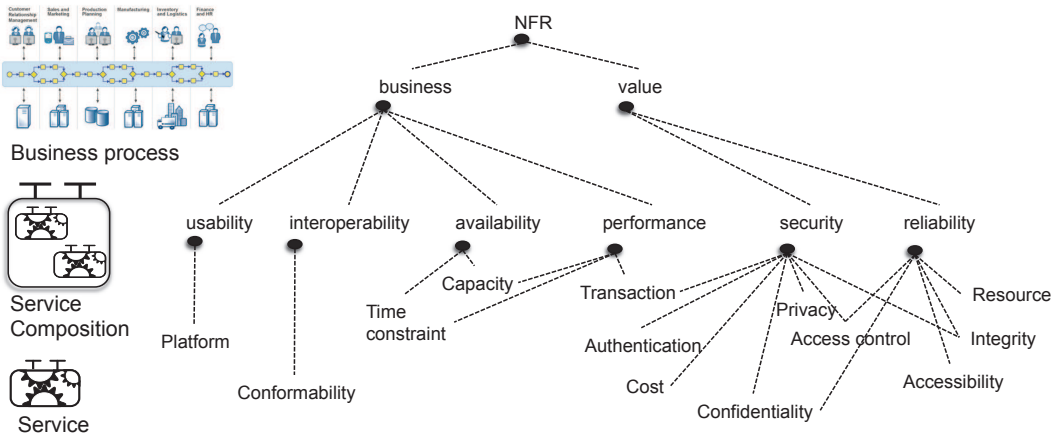

Figure 2.13 Classification of NFP

At the application layer NFP can refer to business rules (e.g., only the user can publish data on her/his wall) and values for example, the email address is a string containing an "@" and a ".". A value NFR expresses constraints about the way data and functions can be accessed and executed. For example accessing methods under security protocols.

Business NFP at the service layer concerns properties that are associated to services and defines how to call their exported operations (business properties). For example, response time, storage capacity (e.g., Dropbox service provides 5Giga free storage). Value constraints concern more on the conditions in which services can be used. For example, accessing to a function within an authentication protocol.

Finally, at the service composition layer gives an abstract view of the kind of properties exported by services that can be combined for providing NFP for a composition. For example, confidentiality, authentication, privacy and access control can provide security at the service composition layer.

[ScCM10] proposes concepts and requirements for characterizing and analyzing existing approaches addressing NFP for service coordinations. Therefore they propose:

- A meta-model that introduces for characterizing NFP according to the entity to which they are associated: attribute, concern, action, and activity.

- Six requirements for studying NFP definition approaches for service coordinations: NFP specifications, NFP actions specification, Web service subjects specification, non functional attributes execution order specification, composite Web service subjects specification, stateful non functional constraint specification.
- Seven requirements for studying NFP enforcement approaches: separation of concerns, transparent integration of functional and NFP, quantification, superimposition, integration of NFP with distributed Web service, programming language independence, Web service composition support.

NFP concepts are close to those defined by **[Souz12]** but they are not organized into layers. They correspond essentially to the service and service composition layers. The enforcement requirements describe the way NFP are weaved to service coordinations. Exiting works respect all or a subset of these requirements. The chosen requirements have an impact in the way NFP are enforced at execution time.

2.3.2 Specifying NFPs

Extensions to UML are proposed to express NFPs. NFPs can be integrated into the Class, Sequence, and Collaboration Diagrams **[CyPr04]**. The performance engineering **[DiSD02]** annotates quantitative performance constraints to UML diagrams such as actor-use case communication, message in the sequence diagram, and state in the state machine diagram. The UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms **[Omg04]** describes a set of QoS characteristics such as throughput, latency, integrity, dependability, among others. The characteristics can also be parameterized with for example the type of keys or types of encryption algorithms in case of the security characteristics. For all UML profiles based approaches it is generally possible to apply the stereotypes and its tagged values to arbitrary UML meta-classes, e.g. Class or Operation. The ordering of stereotypes cannot be expressed but composition- related requirements can theoretically be realized through behavioral diagrams like activity diagrams or state diagrams. **[OrHe07]** incorporate extra-functional properties into UML models by the use of UML stereotypes. For instance, there are stereotypes for encryption, decryption and logging which can be added to classes or operations that are representing services.

Design patterns help in documenting and communicating proven design solutions to recurring problems. Design patterns not only describe how to solve design problems but also document the designers' intentions - why the solution is chosen over others and what tradeoffs are made. **[GrYu01]** observe that non-functional properties (NFPs) are pervasive in the descriptions of design patterns and propose their systematic treatment in descriptions of patterns and when applying patterns during design.

Business rules provide means for expressing, managing and updating pieces of business domain knowledge independent of the rest of the application. According to the Business Rules Group **[Hay00]**, a business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control the behavior of the business **[Hay00]**. Business rules are usually expressed either as constraints or in the form if conditions then action. The conditions are also called rule premises. The business rule approach encompasses a collection of terms (definitions), facts (connection between terms) and rules (computation, constraints and conditional logic) **[Hall01]**. Terms and Facts are statements that contain sensible business relevant observations, whereas rules are statements used to discover new information or guide decision-making. A business rule system is a system in which the rules are separated logically and perhaps physically from

the other parts **[ChMe04]**. **[ChMe04]** investigate a hybrid approach, which combines business processes as known e.g., from BPEL with business rules **[Hay00]**.

[ChLa09] proposed a generative framework to address NFPs for web service compositions. As part of the framework they separate the orchestration meta-model from other non-functional meta-models. They propose a non-functional meta-model for security where concepts exist for requirements and how these requirements are ensured. The NFPs are then added to the composition model by adding annotations.

KAOS **[DaLF93, Lams01]** is a goal-oriented framework for addressing functional requirements with a formal temporal logic that requirements engineer can use to precisely specify NFPs, such as performance, as part of attribute specification of functional requirements.

The AO4BPEL **[CSHM06]** deployment descriptor (DD) is used for specifying non-functional requirements for BPEL processes. First the service (one of logging, security, transactions and reliable messaging) is selected. Then the class (e.g. authentication or confidentiality for security) can be specified. The type (e.g. encryption or decryption for confidentiality) is then the concrete action that must be executed in order to enforce the requirement. Then a selector can be defined such as for a certain BPEL activity. Each selector can be mapped to a requirement.

The W3C proposes a set of languages and protocols for specifying the NFP of a services' coordination. **Figure 2.14** shows a classification of NFP protocols, provided by the WS-* family protocols proposed by the World Wide Web Consortium (W3C). These protocols suppose an underlying communication protocol. The following sections describe representative examples of the WS-* protocol-based approaches.

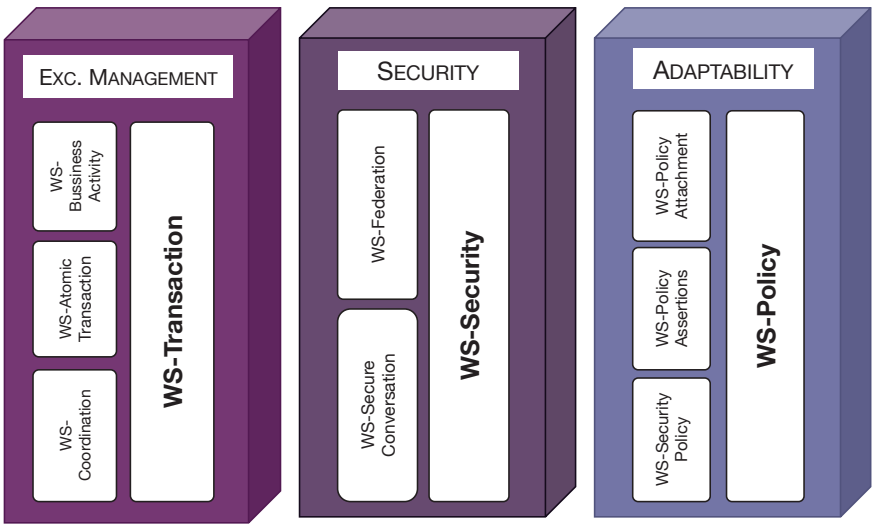

Figure 2.14 Classifications of NFP protocols

Communication Protocols

Communication gives basic support for communicating a given number of services. The principle is based in messages exchange, where a message contains the information necessary for invoking a method exported by a service. Therefore, a set of protocols for specifying and sending messages must be specified. For example, a communication protocol

defines a naming model for locating a service and eventually, as shown later the protocol can be extended with security elements for ensuring data integrity exchanged among services. In the context of W3C the communication protocols SOAP, WS-Addressing and WS-Reliability have been proposed.

- SOAP (Simple Object Access Protocol) is a communication protocol based on XML for invoking a method of a remote service. The important concepts in SOAP are (i) the envelope containing the message information and (ii) a data model defining the format of the message to be transmitted.
- **WS-Addressing** defines a mechanism for adding information about the services location into a SOAP message. **WS-Addressing** defines two constructors for integrating the information of transport protocols and, an application based on messages. The constructor also homogenizes the information by using a common format that can be processed independently of the transport protocol or the application.
- **WS-Reliability** combines the HTTP protocol with some security protocols for adding reliability when exchanging messages in a services' coordination. **WS- Reliability** also guarantees the reception of messages using acknowledgments.

Transaction Management

The Web services transaction protocol (WS-Tx) **[Oasi09a]** provides strict and semantic atomicity to services coordination. It adds a transactional behavior to a services' coordination using **WS-AtomicTransaction** and **WS-BussinessActivity.** WS-Tx is layered over the Web services coordination specification (WS-C[3]) **[LiWe03, Oasi09b]**. WS-Transaction describes the types of services' coordination that can be used with WS-Coordination e.g., coordinations having an atomic behavior are described using the concept of **AtomicTransaction** and coordinations having a long duration are described using the concept of **BussinessActivity**.

Ws-Tx extends WS-C to create a transactional coordination context offering two types of atomicity:

- Atomic transaction (AT). It is used to coordinate activities with strict atomicity-like behavior. **WS-AtomicTransaction** defines 3 types of protocols: completion, volatile two-phase commit and durable two-phase commit. These protocols are used for assuring the property "all or nothing" when executing the activities of a services' coordination.
- Business activity (BA). It is used to coordinate activities with a semantic atomicity-like behavior. In this protocol business activities can be subdivided into small ones called scopes. A scope is a collection of operations that can be nested to arbitrary degrees.

[3] WS-C describes an extensible framework that provides protocols for coordinating actions in distributed applications that require to ensure certain policies. The WS-Coordination framework enables a service to create a context to propagate an activity to other services and, to register a service as participant in a services' coordination.

WS-Tx has defined five protocols for committing based on classic 2PC: **Completion**, **CompletionWithAck**, **Phase zero**, **2PC**, and **Outcomenotification**.

The business transaction protocol (BTP) **[Oasi02]** is a protocol for coordinating processes with strict and semi-atomicity behavior. A BTP coordination protocol is a set of well-defined messages that are exchanged between participants to address a transactional behavior. BTP defines two coordination protocols to provide transactional behavior:

- Atom protocol implements a strict atomicity behavior.
- Cohesion protocol enables the definition of semi atomicity, where some participants commit and others cancel based on some pre-defined business rules. A cohesion protocol commits using the rules that users define.

Atom and cohesion protocols use a modified two phase commit protocol (2PC) that cannot be adapted or extended to new requirements. During the second phase of 2PC protocol, the coordinator uses the set of rules for deciding whether to commit or abort.

[BoCR05] extends BTP by means of an ontology expressed in OWL-S. The ontology categorizes services according to three aspects:

- The functionality of the service.
- How it is accessible the service.
- How the service works.

Regarding atomicity a service is classified as unprotected, semi-protected, protected, negotiable, and real. This classification is used for implementing the atom, cohesion, and atomic transactions of BTP on top of the transactional properties that a service can provide. In such a way, it is possible to provide strict and semi-atomicity behavior to a given services coordination.

[TMWD04] proposes a policy based transactional model implemented on top of **WS-Coordination**, **WS-Transaction**, and **WS-ReliableMessaging** protocols. A policy is used to advertise and to match three types of atomicity:

- Direct transaction processing provides atomicity based on the 2PC protocol.
- Queued transaction processing provides atomicity using a non-blocking 2PC protocol.
- Compensation-based transaction processing provides atomicity using the notion of compensation that relaxes the notion of atomicity.

Using such transaction models it is possible to provide strict and semi-atomicity behavior.

[DFDB05] introduces a protocol for dynamic composition of services based on the tentative hold protocol that enables the definition of an atomic behavior for activities. The tentative hold protocol adds a phase to 2PC protocol where participants can request tentative reservations on the resources that they want to use in following phases. This new phase can be seen as an exchanging of messages prior to the transaction for minimizing compensation actions.

In this kind of approaches, atomicity is implemented within the coordination by using pre-defined protocols (e.g., extensions of the two phase commit protocol[4] and advanced transactional models[5], see **[Port06a]**). When using a protocol at least two assumptions are considered:

- The service operations participating in a transaction have a homogeneous behavior with respect to transactional requirements (e.g., all operations accept reservation for committing). Therefore the protocol can define the way in which a transaction will be committed or undone.
- Services and the execution infrastructure export operations for supporting the execution of atomicity protocols (e.g., the compensation capability).

Security

WS-Security **[Oasi04b]** (Web Services Security, short WSS) is an OASIS extension to SOAP to apply security to web services. The protocol specifies how integrity and confidentiality can be enforced on messages and allows the communication of various security token formats, such as SAML, Kerberos, and X.509. Its main focus is the use of XML Signature and XML Encryption to provide end-to-end security. WS-Security describes three mechanisms for:

- Signing SOAP messages to assure integrity. Signed messages also provide non-repudiation.
- Encrypting SOAP messages to assure confidentiality.
- Attaching security tokens to ascertain the sender's identity.

In general, WSS by itself does not provide any guarantee of security. It is up to the programmer to ensure that the result is not vulnerable. Key management, trust bootstrapping, federation and agreement on the technical details (ciphers, formats, algorithms) are outside the scope of WS-Security. The following specifications are associated with **WS-Security: WS-Policy, WS-SecureConversation, WS-Trust and ID-WSF**.

WS-Trust is a language based on WS-Security for defining rules for sending, exchanging and diffusing security tokens **[Oasi12]**. **WS-Trust** also defines the rules for granting and disseminating credentials within domains of trust. For example, for assuring the communications between two services, both services have to send their security credentials to determinate if they can trust each other.

WS-Federation is based on **WS-Trust** and defines mechanisms for federating (sharing) attributes and information about services within domains of trust. WS- Federation characterizes services as passive and active services that can interact with service providers.

---

[4] The two phase commit protocol (2PC) consist of two phases: i) during the first phase (preparation) every participant in transaction extern its response to commit or abort, and ii) in the second phase (commitment), a coordinator makes a global decision which is communicated and executed by all participants of the transaction.

[5] Most representative examples of advanced transactional models are: saga **[GaSa87, GaUW00, GGKK91]**, flexible transactions **[ELLR90, ZNBB94]**, and contracts **[ReWä92]**.

Adaptability

Web Services Policy[6] (**WS-Policy**) is a machine-readable language for representing these Web service capabilities and requirements as policies in a machine-readable form. Service providers use **WS-Policy** to represent combinations of behaviors (capabilities and requirements). Web service developers use policy-aware clients that understand policy expressions and engage the behaviors represented by providers automatically. These behaviors may include security, reliability, transaction, message optimization, etc. **WS-Policy** allows web services to use XML to advertise their policies (on security, quality of service, etc.) and for web service consumers to specify their policy requirements.

It provides a set of specifications that describe the capabilities and constraints of security (and other business) policies on intermediaries and end points (for example, required security tokens, supported encryption algorithms, and privacy rules) and how to associate policies with services and end points.

The general principle for adding NFPs to a services' coordination is to define protocols for controlling its execution. Associating non-functional properties to services composition can help to ensure that the resulting application is compliant to the user requirements and also with the characteristics of the services it uses. As services are independent components, ensuring non-functional properties is a challenge. Programming non-functional properties is not an easy task and different studies **[AgLS09, BaKS10, ChLa09, GuRF09, JeCL09, MaZN10, TsNA12, XCZB08]** associate non-functional requirements and services using different approaches. In the following section we describe methods that can be used for weaving NFP and services' coordinations.

2.4 Weaving NFPs to a Services' Coordination

A strategy for weaving NFP to service coordinations is to define them explicitly within the application logic. Adding sequences of activities that implement well-known protocols, for example advanced transactional models. This technique has been used particularly in works adding transactional properties (atomicity) to service coordinations. The challenge is to determine the composition pattern that will include NFP. Another strategy for defining NFP is to separate their specification and the services coordination. This captures the interactions that are necessary to provide a given behavior by enforcing NFP. The following sections describe works adopting these approaches.

2.4.1 Modularizing NFP

Aspect-oriented programming (AOP) is an approach for modularizing the implementation of business rules and thereby allowing the separation of crosscutting concerns. The principle is breaking down program logic into distinct parts (so-called concerns, cohesive areas of functionality). The implementation of the business rules tends to cut across several activities of a process definition; AOP provides means to modularize crosscutting concerns and has already been found valuable for modularizing business rules of object-oriented software **[Dhon04]**. The integration of aspect behavior into the functional behavior can be done without changing the functional code by means of pointcuts. Pointcuts can be under-

[6] http://msdn.microsoft.com/en-us/library/ms996497.aspx

stood as a facility to query over join points. Join points on the other hand are well-defined points during the execution of a program (e.g. an execution of an operation). In a single query, multiple join points can be selected and multiple aspects can point to the same join point.

[HeBW06] use AOP technology, namely AspectJ **[Ladd03]** to implement QoS monitoring without changing the existing service implementation. They have written a performance aspect that measures the execution time by defining two join points, one at the beginning and one at the end of a method call. The advice captures the time when the request arrives at the service and when the response is returned by the service. Further, they implemented a cost aspect that counts the types of service invocations and cumulates them to calculate the monetary total cost of usage. The reliability aspect logs every invocation that ends with an exception.

[MSMR08] introduce the Web Service Aspect Language (WSAL) providing aspects that can be freely specified, implemented, deployed and executed as loosely coupled web services. Weaving is realized at network level by Web intermediary technology. Supported join point types are message part, service name, service operation, service location, client location and combinations of them. Advice elements are implemented by Web service operations and the following types are allowed: before request, upon request, after response, upon response, after exception, upon exception, and around. Additional context information like the intercepted message or the service location can be passed to the advice implementation operation. The context information can be used to restrict the information that is transmitted to the advice Web service. Some advice types, e.g., before and after can be executed in parallel with the join point. Aspects are specified by a XML syntax and contain a set of join point and advice elements. The aspect weaver is implemented by using IBM WBI **[Ibm00a]** intermediary technology.

[VeVJ06] propose the Web Services Management Layer (WSML) for the dynamic integration, selection, composition, and client-side management of web services in Service-Oriented Architectures (SOA). There are different aspect types. Service Redirection Aspects specify the communication logic to swap from a web service to an alternative one or to a web service composition implementing the same logic by calling multiple services. Service Selection Aspects enforce a certain selection policy which can be of one of the types Client-initiated selection, Non-functional service property based selection, Service behavior-based selection and Service-initiated selection. Service Management Aspects modularize concerns like encryption, billing, reliable messaging, transactions, etc. which can be enforced per service type, per composition or per web service. The aspects are implemented by JAsCo **[VSVC05]**. JAsCo is an adaptive programming approach that supports Superimposition by Combination Strategies. Combination Strategies is an imperative approach that allows for employing the full expressiveness of Java.

AO4BPEL **[ChMe07]** is an aspect-oriented extension to WS-BPEL for defining aspects that enforce NFPs such as security, reliable messaging and transactions. An aspect in AO4BPEL is a container for multiple point-cuts and advice constructs and can be activated at runtime. The point-cut language is XPath **[Wc00c]** for selecting arbitrary BPEL activities. When a point-cut matches a certain join-point, its corresponding advice is executed. An advice is defined in BPEL and can be used to call dedicated middleware services in order to enforce NFPs. A security, reliable messaging and transaction middleware services are provided. Besides, predefined aspects for integrating NFP into BPEL processes exist and they are generated from a deployment descriptor.

The second alternative for modularizing the implementation of business rules expressing NFP is to combine the process-based specification of service composition with dedicated approaches to declarative specification of business rules, such as Java Expert System Shell Jess **[Frie12]** and JRules **[Ibm00b]**.

2.4.2 Contract Driven Programming

Contracts are behavioral descriptions of Web services Code Contracts provide a language-agnostic way to express coding assumptions in .NET programs **[FaLo10]**. The contracts take the form of preconditions, post-conditions, and object invariants. Contracts act as checked documentation of your external and internal APIs. The contracts are used to improve testing via runtime checking, enable static contract verification, and documentation generation. Code Contracts bring the advantages of design-by-contract programming to all .NET programming languages. Contracts are expressed using static method calls at method entries. Tools take care to interpret these declarative contracts in the right places. A precondition must be true on entry to the method. It is the caller's responsibility to make sure the pre-condition is met. A post-condition must be true at all normal exit points of the method. It is the implementation's responsibility that the post-condition is met.

[CaGP09] proposes a theory of contracts that formalizes the compatibility of a client with a service, and the safe replacement of a service with another service. Contracts are used to ensure that interactions between clients and services will always succeed. Intuitively, this happens if whenever a service offers some set of actions, the client either synchronizes with one of them (i.e., it performs the corresponding coaction) or it terminates. The service contract allows to determine the set of clients that comply with it, that is to say that will successfully terminate any session of interaction with the service **[CaGP09]**.

The use of contracts statically ensures the successful completion of every possible interaction between compatible clients and services. The technical device that underlies the theory is the filter, which is an explicit coercion preventing some possible behaviors of services and, in doing so, make services compatible with different usage scenarios.

2.4.3 Policy Driven Programming

The main objective of a policy driven system is to enable a flexible and adaptable management by defining strategies that can be defined and modified without having to recompile the system. Policies are used for defining constraints on the behavior of the systems for ensuring that the system complies with its objectives **[Slom94]**.

Policy driven programming is a computer-programming paradigm based on an idiom for C++ known as policies. The main idea is to use commonality-variability analysis to divide the type into the fixed implementation and interface, the policy-based class, and the different policies. The central idiom in policy-based design is a class template (called the host class), taking several type parameters as input, which are instantiated with types selected by the user (called policy classes), each implementing a particular implicit interface (called a policy), and encapsulating some orthogonal (or mostly orthogonal) aspect of the behavior of the instantiated host class.

KaOS **[UBJS03]** is a collection of agent services based on components for executing mobile agents. One of the components of the KaOS platform is based on policies for the specification, management, conflicts resolution, and policies execution. Policies are expressed using DAML **[Darp00]**. Policies are of two types authorization and obligation and

they can be specialized for addressing application requirements. KaOS has been used for managing Web services deployed on Grid architectures.

Security Policy Language (SPL) **[RiGu99]** is a security language for expressing policies that can help to decide on the validity of events according to their properties (e.g., author, destination, action, etc.), the conditions in which they were produced and also the properties of previous events. A policy consists of targets and rules. Targets are the entities using policies and rules (simple and compound), implement decision making about the acceptance of events. Compound rules implement the behavior of the policy it is associated to. Policies can be compound using an algebra.

2.5 Conclusions

This chapter proposed a classification of services that organizes them into families according to their exported operations. It mainly identifies services that provide operations that do not imply the management of data collections (i.e., business services) and those that provide different types of data management and processing operations. Since services are used for implementing applications by coordinating calls to their operations, the chapter described coordination models.

A service's coordination defines the business logic of an application. Yet, NFPs must be also defined and associated to the application logic in order to address some application requirements. The chapter analyzed the principles adopted by different approaches for expressing NFPs intended to be associated to services' coordinations. It showed that since NFPs are associated to the coordination, it is important to consider the way they are modeled to determine to which entities of the services' coordination they can be added.

Finally, the chapter described techniques that can be used for adding NFPs to services' coordinations. We showed that the strategies for adding NFPs can imply expressing them as protocols and weaving them within the coordination by adding activities. The second strategy is defining NFPs separately together with reinforcement actions and synchronization points for associating them to service coordinations. This approach enables loosely coupled approaches for adding NFP, which can modularize NFP and ease the maintenance of the coordination. Our work adheres to this strategy because it does not «pollute» the coordination specification, it considers different types of services and it can be handled separately thereby facilitating service coordination maintenance and evolution.

Chapter 3 introduces our AP model devoted for adding NFP to service coordinations based on an activity oriented model. It also provides concepts for specifying different NFP protocols in a reactive way.

Intentionally left blank

3 Active Policy Model

This chapter describes the **Active Policy Model** (AP Model), for representing *services' coordinations* with *non-functional properties* as a collection of *types*. In our model, a services' coordination is represented as a *workflow* composed of an *ordered* set of activities, each *activity* in charge of implementing a call to a *service' operation*. We use the type **Activity** for representing a workflow and its components (i.e., the workflow' activities and the order among them). A *non-functional property* is represented as one or several **Active Policy** types, each policy composed of a set of *event-condition-action rules* in charge of implementing an aspect of the property. Instances of *active policy* and *activity* types are considered in the model as *entities* that can be executed. We use the **Execution Unit** type for representing them as entities that go through a series of *states* at *runtime*. When an active policy is associated to one or several *execution units*, its rules verify whether each unit respects the implemented non-functional property by evaluating their conditions over their *execution unit state*, and when the property is not verified, the rules execute their actions for enforcing the property at *runtime*.

The chapter is organized as follows. **Section 3.1** introduces the types composing the AP Model and a scenario used for describing the types. **Section 3.2** defines the types in which the model is built on: *data type* and *service type*. **Sections 3.3, 3.4, 3.5** describe respectively the **Activity** type, the **Execution Unit** type and, the **Active Policy** type. **Section 3.6** introduces the AP language and illustrates how to use it for defining AP types and thereby specifying non-functional properties. **Section 3.7** concludes the chapter.

3.1 Introduction

The AP Model is based on the notions of *type* and *instance*. A **type** represents a collection of *objects* that have the same *properties*, *behavior* and *semantics*. An **instance** represents a *specific* object that can get *concrete values* at runtime. We use the UML class diagrams for representing types. For instance, the diagram of **figure 3.1** shows the *types* composing the AP Model as *classes*. They are grouped according to the concept they describe:

- The classes used for representing a *services' coordination* (i.e., *Activity*, *Atomic Activity*, *Workflow Activity* and *Control Flow Activity*). The main class in this group is the *Activity* class. It represents the **Activity type**.

- The classes representing a *non-functional property* (i.e., *Active Policy*, *Rule* and *Event*). The main class in this group is the *Active Policy* class. It represents the **Active Policy** type.
- The classes used for associating non-functional properties to services' coordinations (i.e., *Execution Unit* and *Execution State*). The main class in this group is the *Execution Unit* class. It represents the **Execution Unit** type.

All possible objects considered in the model can be represented as an *instance* of one of these types. For example, **figure 3.2** illustrates some examples of *active policy instances*. Instances $\mathbf{ap_1}$ and $\mathbf{ap_2}$ are examples of objects belonging to the *Basic* policy type (i.e., they conform to the properties, behavior and semantics described by the type). Instance $\mathbf{ap_3}$ is an example of an object belonging to the type *OAuth* policy. Instance $\mathbf{ap_4}$ is an example of an object of type *Exception Management* policy.

Figure 3.2 also shows that the AP Model types can be specialized in *subtypes*. The type *Active Policy* type represents all policy instances that can exist in the universe, no matter what non-functional property they deal with. In contrast, the type *Authentication* policy type represents policy instances that deal with authentication aspects (i.e. a specific non-functional property). When a type T_1 is a subtype of T_2 all the instances of T_1 are also instances of T_2 (i.e., a instance can belong to more than one type). For instance, since the *Basic* and *OAuth* policy types are subtypes of the *Authentication* type (see **figure 3.2**), instances $\mathbf{ap_1}$, $\mathbf{ap_2}$ and $\mathbf{ap_3}$ are also instances of the *authentication* policy type. In the same way, even if $\mathbf{ap_3}$ and $\mathbf{ap_4}$ belong to different types, they are examples of instances of the Active Policy type.

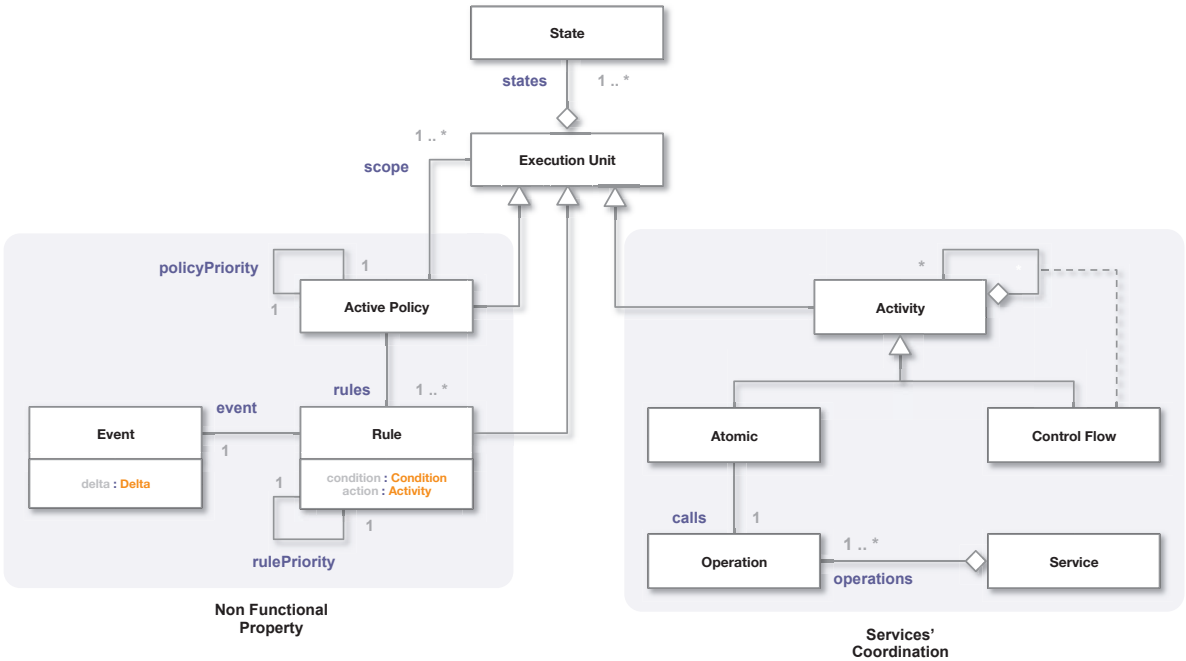

Figure 3.1 AP Model types UML class diagram

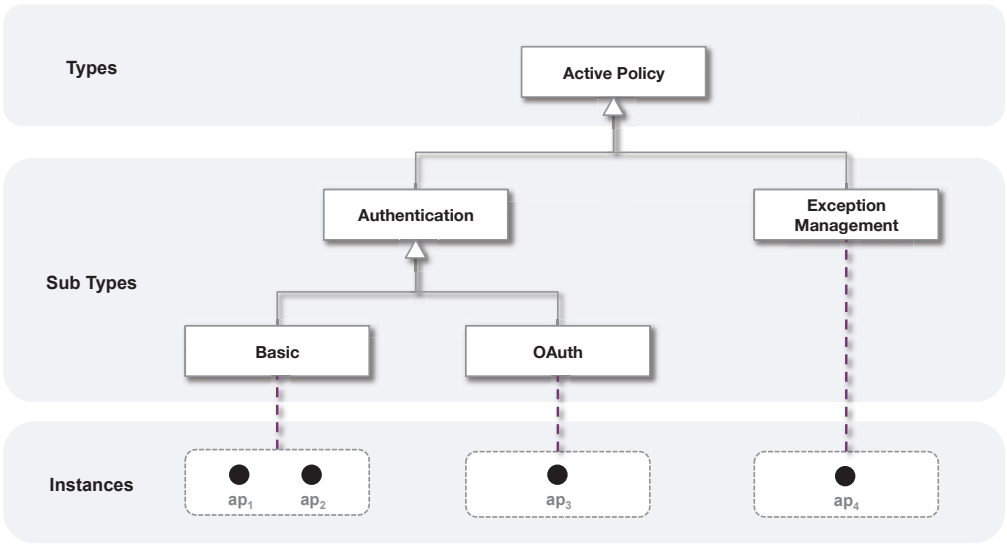

Figure 3.2 AP Model levels of abstraction

In order to describe the types composing the AP Model consider the following scenario. A developer wants to implement a services' coordination called *Status Updater* that (see **figure 3.3**):

- **Reads** information about the *song* a user is currently listening to (e.g., song title and artist name) using the LastFM[1] service.
- **Computes** the *mood* of the user using the history of played songs and a custom algorithm.
- **Posts** song information and user mood into the user's Facebook and Twitter *accounts* (e.g., the string "*Sting – Fields of Gold* ☺") using the Facebook[2] and Twitter[3] services.

Figure 3.4 illustrates a *workflow* implementing the Status Updater coordination by ordering calls to the operations exported by the LastFM, Facebook and Twitter services. In the diagram, *services* are represented as rectangles containing the name of the *operation* used by the workflow. *Activities* are represented as rounded rectangles connected to a service' operation. O*rder* among activities (i.e., execution flow) is represented as solid arrows connecting activities.

The *Status Updater* workflow is composed of 4 *activities*:

- **Get Song** calling the operation *getLastSong* from the LastFM service. This activity receives a *user id* as input and outputs the *song information*.

[1] http://www.lastfm.fr/api

[2] http://developers.facebook.com

[3] https://dev.twitter.com

- **Compute Mood** represents the algorithm that computes the *user mood* based on the music listened by a user during some period of time. This information is saved in a *log*.
- **Update Twitter** and **Update Facebook** calling the operations *updateStatus* of the Facebook and Twitter services. Both activities receive as input a *user id* and some *text* (in this case the song information) and produce no result as output.

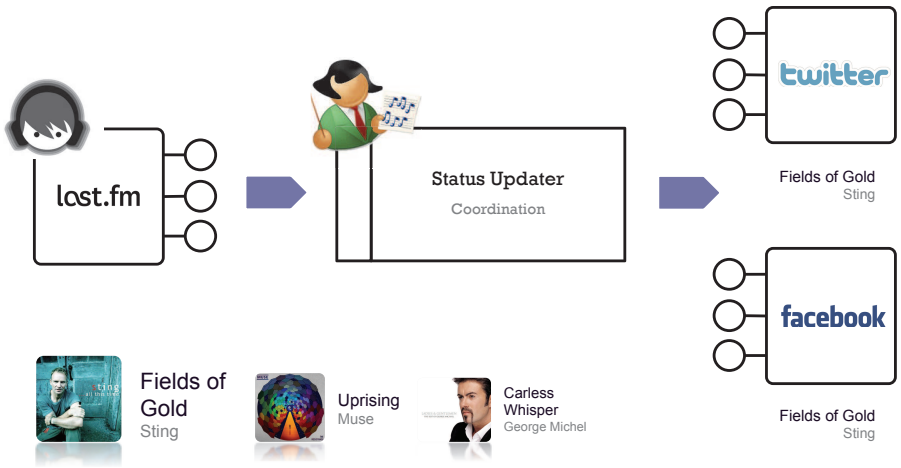

Figure 3.3 Status Updater services' coordination

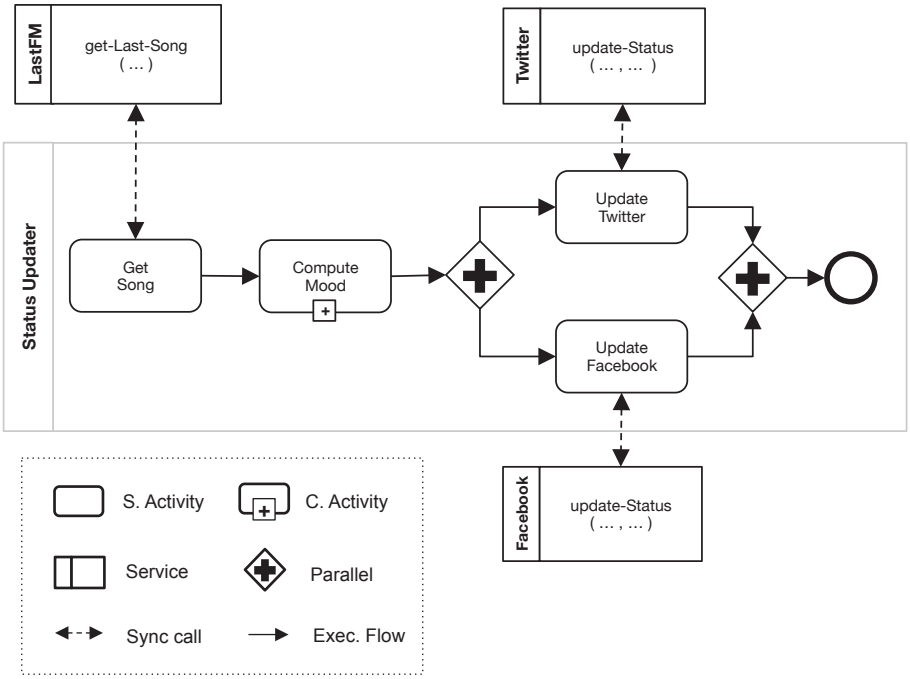

Figure 3.4 Status Updater workflow BPMN diagram

These activities are *ordered* according to the following criteria:

- Since the output of **Get Song** is needed by activities **Compute Mood**, **Update Twitter** and **Update Facebook** the activity *Get Song* has to be executed before the others (i.e., in sequence).
- Since the output of **Compute Mood** is needed by activities **Update Twitter** and **Update Facebook**, the activity *Compute Mood* has to be executed before the other two activities.
- Since there is non-dependency among **Update Twitter** and **Update Facebook** the activities can be executed in *parallel*.
- Since the workflow has to run **while** the user listens music, the activities have to be executed *iteratively*.

Now let us assume that the developer wants to extend the *Status Updater* workflow for addressing the following non-functional requirements (see **figure 3.5**):

- **Authentication**. Due to privacy protection Twitter and Facebook implement *authentication protocols* that control access to user' information (e.g., only authorized applications can read and/or update users information). For instance, Twitter uses a *basic authentication protocol* based on *username* and *password* for preventing applications of updating the user status. In contrast, Facebook uses the *OAuth*[4] *protocol*, a protocol based on *third-party authentication* for the same purpose.
- **Network unavailability**. Due to network unreliability services can be unavailable at certain periods of time. This can cause exceptions that must be handled by the workflow (e.g., by retrying the calls to the services or by compensating the activities).

In the AP approach these non-functional requirements are addressed by defining A-Policy *types* and by associating these types to a target workflow. For example, a developer can define an *Authentication A-Policy* type for addressing the authentication requirements of workflows. Then, as shown in **figure 3.6** the developer can specialize this type for defining policy types defining Basic and OAuth protocols. These new types can be associated to the *Update Twitter* and *Update Facebook* activities of the *Status Updater* workflow. These policy types will be used for adding the basic and OAuth protocols (respectively) required by each service associated to the workflow activities (see **figure 3.6**).

At execution time, the policies can execute some code *before* and *after* the execution of an activity in order to implement an authentication protocol. As shown in **figure 3.7**, this can be equivalent of adding activities to the *Status Updater* workflow before and after the activities *Update Twitter* and *Update Facebook* of the workflow. Yet, thanks to policies the authentication issues are maintained separated to the workflow. Thus, in the AP approach A-Policy types are defined *independently* of concrete workflows (i.e., in orthogonal way). Indeed, it is possible to keep separated the logic implementing a services' coordination from the implementation of its non-functional requirements, simplifying the *maintenance* process. For example, if one of the services changes its authentication protocol, the

[4] http://oauth.net

change remains transparent to the coordination because the A-Policy can be modified without touching the coordination implementation.

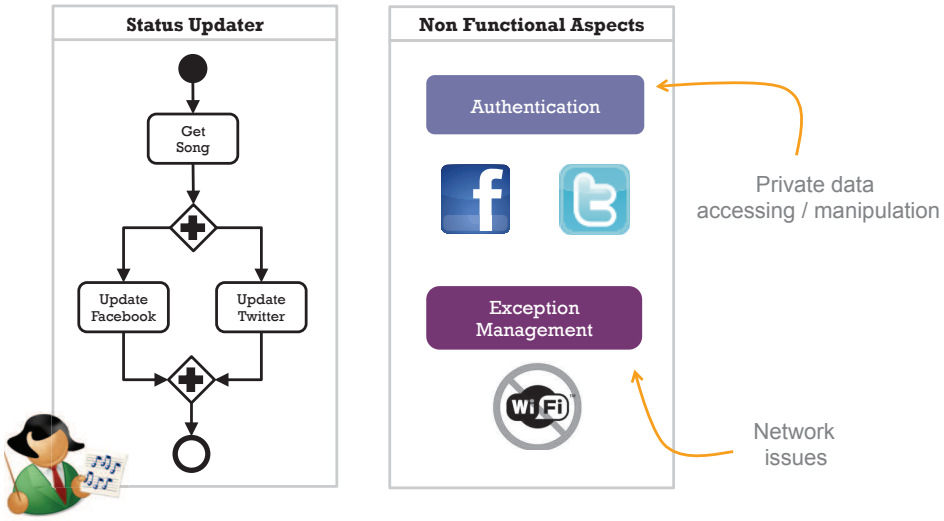

Figure 3.5 Non-functional properties of the Status Updater

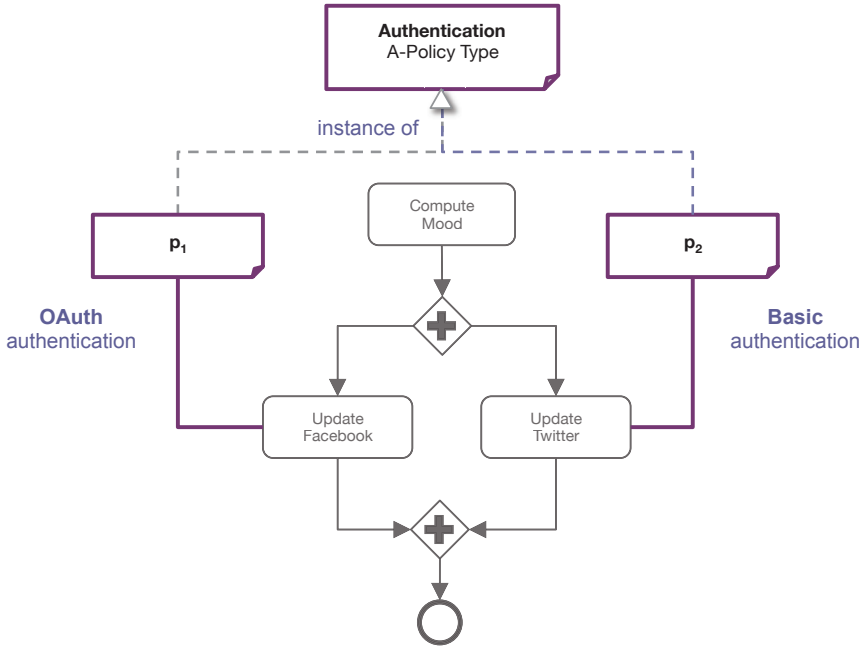

Figure 3.6 Status Updater workflow with associated authentication a-policies

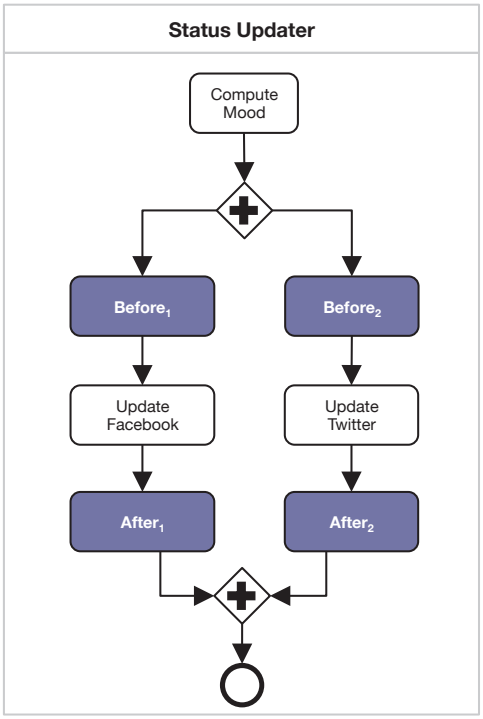

Figure 3.7 Status Updater after the execution of authentication a-policies

3.2 Data and Service Types

3.2.1 Data Types

AP Model types are built on top of the *data types* shown in **figure 3.8**. As shown in the figure, any type in the model can be classified as an **Atomic** or **Composite** type.

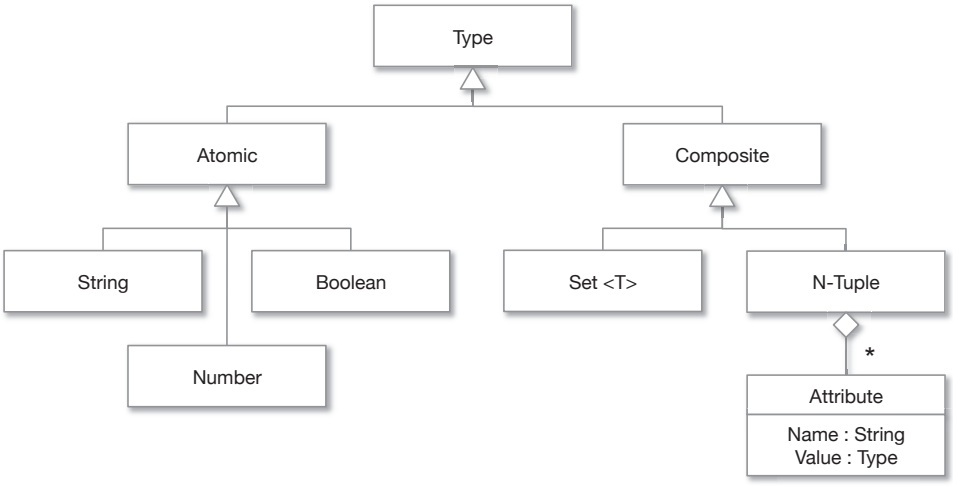

Figure 3.8 AP Model data types UML class diagram

Atomic types represent instances that cannot be further divided in other values. The *atomic types* considered in the model are **String**, **Number** and **Boolean** (see **figure 3.8**). The type

String represents any *chain of characters* (e.g., "www", "James Bond", "pwd123"). The type *Number* represents any *real* or *natural* numbers (e.g., -1, 0, 1, 3.1416). The type *Boolean* represents values *true* and *false* (i.e., *truth* values).

Composite types represent instances that are composed of other values (i.e. instances of other types). The *composite types* considered in the model are the types **Set**, **N-Tuple** and **Attribute**. These types conform to the following rules:

- If $\mathbf{A_1 \ldots A_n}$ are different *attribute names* (i.e., $\mathbf{A_i \neq A_j}$, $\forall_{i,j} \varepsilon [1 \ldots n]$), and $\mathbf{T_1 \ldots T_n}$ are *type names*, the expression $\mathbf{<A_1 : T_1, \ldots, A_n : T_n>}$ represents a type **N-Tuple**. For example, the expression:

  < *name*: String, *value*: Type >

  represents the type **Variable** as a binary tuple composed of (i) a *name* attribute of type *String* representing the *variable' name* and (ii) a *value* attribute (of any type) representing the *variable value*.
- If **T** is a *type name* then the expression **Set <T>** represents a *typed collection* where all the elements in the collections are of type **T**. For example, *Set <String>* represents a collection containing only elements of type *String*.

3.2.2 Service Type

The AP Model is also based on the notion of *service*. As shown in **figure 3.9**, we use the type **Service** for representing a service as an autonomous entity that exports a collection of operations via a network.

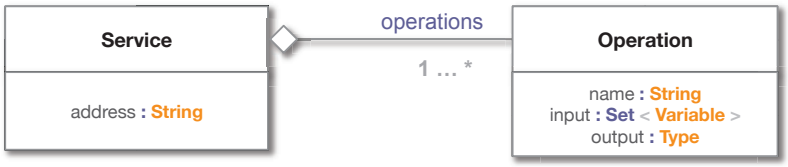

Figure 3.9 Service type UML class diagram

The type **Service** is defined as follows:

< *address*: String, *operations*: Set <Operation> >

Its attributes are interpreted as follows:

- *address* represents the URL where the service can be located.
- *operations* represents the set of operations exported by a service, each operation represented as an **Operation** type.

The type **Operation** has the following form:

< *name*: String, *input*: Set<Variable>, *output*: Type>

Its attributes are interpreted as follows:

- *name* represents the name of the operation.
- *input* represents the set of variables that an operation receives.
- *output* represents the *type* of the *value* produced by the execution of the operation.

Consider for instance the *LastFM service* of the *Status Updater* scenario. This service exports a single operation: the *GetLastSong* operation. **Figure 3.10** shows how we represent this service using the type **LastFM**, a *subtype* of the type **Service**.

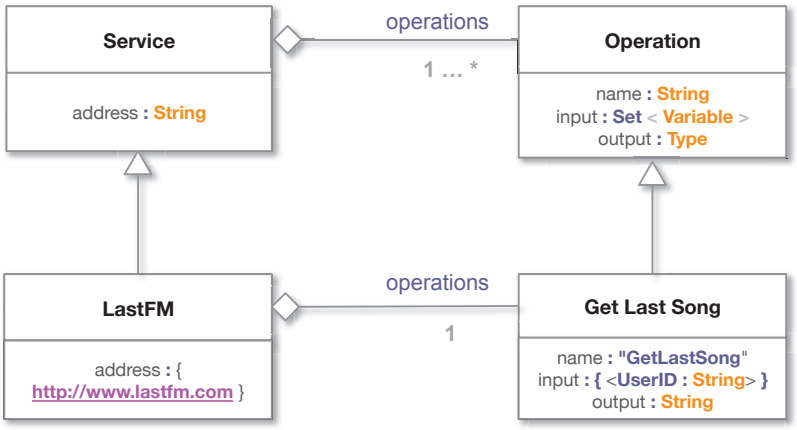

Figure 3.10 UML class diagram illustrating the representation of a specific service as a Service subtype

The figure also shows how we add *constraints* to the attributes of a type in order to represent a concept. For instance, the **LastFM** type defines constraints for all the attributes of the **Service** type in order to represent the specific LastFM service:

1. Since we know in advance the location of the service, we constraint the *address* attribute to a single possible value (i.e., the value "http://www.lastfm.com/").
2. Since the service only exports one operation, we constraint the cardinality of the *operations* attributes to 1. Besides, we specify the *operation type* contained by the attribute *operations* (i.e., the **Get Last Song type**).
3. Since we know in advance the signature of the GetLastSong operation, we constraint the values of the **Get Last Song type** attributes as follows: the *name* attribute has only one possible value: the string "*GetLastSong*". The *input* attribute contains a single variable (of type UserID) that represents the *user id*. The *output* attribute is constraint to any valid String.

For the sake of simplicity we will not explicitly describe the *constraints* added by subtypes in the rest of the document. The reader should infer them from the UML class diagrams describing the types.

3.3 Activity Type

We use the **Activity**[5] type for representing the concepts abstracting a services' coordination (i.e., *workflow*, workflow' *activities* and the *order relationship* among the workflow' activities). As shown in the **figure 3.11**, the **Activity** type has the following structure:

< *name*: String, *input*: Set<Variable>, *local*: Set<Variable>, *output*: Type>

Its attributes are interpreted as follows:

- *name* uniquely identifies an activity type participating in a workflow.
- *input* contains the variables received as input of the activity.
- *output* represents the value produced by the execution of the activity.
- *local* contains the set of local variables used for computing the activity output.

Figure 3.11 also shows the *activity types* used for representing a services' coordination:

- **Workflow** activity type that represents a workflow.
- **Atomic** activity type that represents a workflow' activity.
- **Control Flow** activity type that represents the order relationship among the workflow' activities.

Since these types are *subtypes* of the **Activity** type, they all share the same structure i.e., *every possible activity* considered in the model (i) receives a set of variables as input (possible empty), (ii) produces a single variable as output and (iii) have a set of local variables used internally for computing its output.

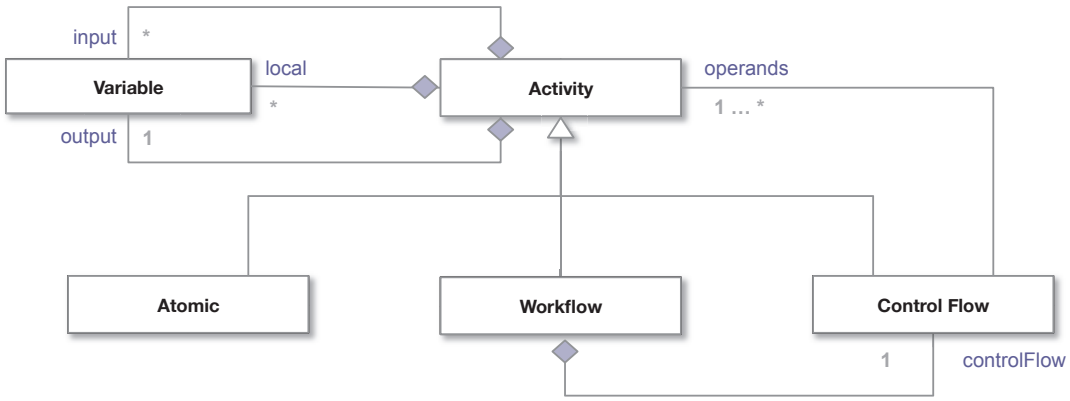

Figure 3.11 Activity type UML class diagram

[5] The use of *activity types* for representing a services' coordination is not proper to our model. It was first used in the Business Process Execution Language (BPEL) as a *building block* for expressing executable workflows **[AAAB07]**.

3.3.1 Atomic Activity

The **Atomic Activity** type represents the logic handling a call to a *service' operation*. As shown in **figure 3.12**, the **Atomic Activity** type has the following structure[6]:

< *calls*: Operation, *operationResult*: Type >

Its attributes are interpreted as follows:

- *calls* represents a *reference* to a service' operation. As shown in **figure 3.12**, an atomic activity can be associated (i.e., calls) to maximum one service' operation.
- *operationResult* contains the *result* of a service' operation call. Note that attribute can contain a value of any type.

When executed, an instance of type **Atomic Activity** uses the *service' address*, the *operation' name* and the activity *input* variables for calling the service' operation. Then, once the operation completes, the activity stores the operation result in the *operationResult* attribute. Finally, the activity uses the *operationResult* value for computing the activity *output*.

As an example, consider the *Get Song* activity of the *Status Updater* scenario. Recall that this activity calls the *Get Last Song* operation of the *LastFM* service. **Figure 3.13** shows the representation of this activity as a subtype of the **Atomic Activity** type. Note that this representation uses the types defined in **section 3.2.2**.

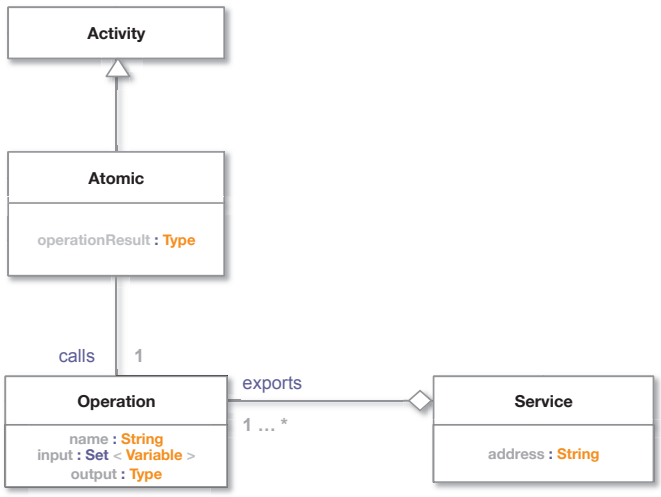

Figure 3.12 Atomic Activity type UML class diagram

As shown in the figure, the **Get Song Atomic Activity** type:

1. Is *identified* in the Status Updater workflow by the name "Get Song".
2. Is associated to the **Get Last Song** operation of the **LastFM** service type.

[6] Recall that, as any other activity subtype, the **Atomic Activity** type has *input*, *output* and *local* variables inherited from the **Activity** type.

3. Receives as *input* a single variable called *UserID* that is mapped to the *UserID variable* of the **Get Last Song** operation at runtime.
4. Maps the string produced by the execution of the operation to its operationResult attribute
5. *Outputs* the string containing the information about the song the user is currently listening to.

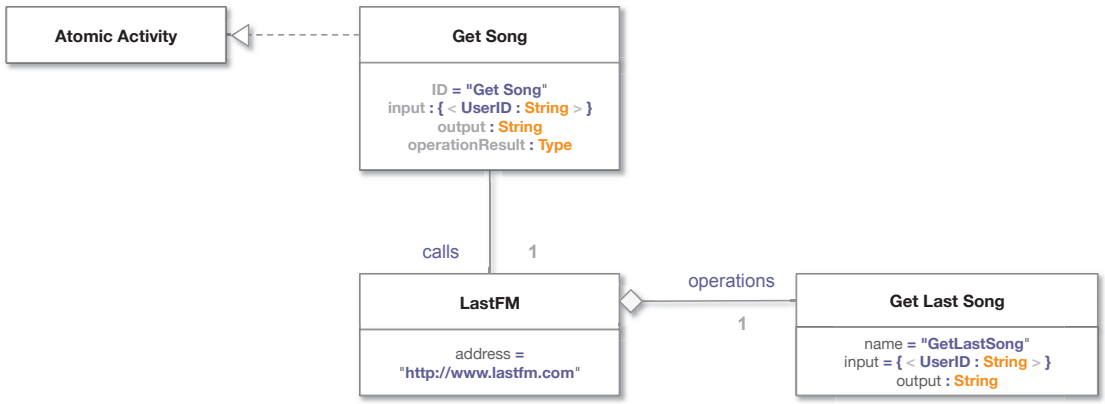

Figure 3.13 UML class diagram illustrating a specific Atomic Activity subtype.

3.3.2 Control Flow Activity

The **Control Flow Activity** type represents the logic implementing an *order relationship* among a set of activities called *operands* (see **figure 3.11**). The AP Model considers 3 types of control flow activities (see **figure 3.14**): **Sequence**, **IF** and **While**.

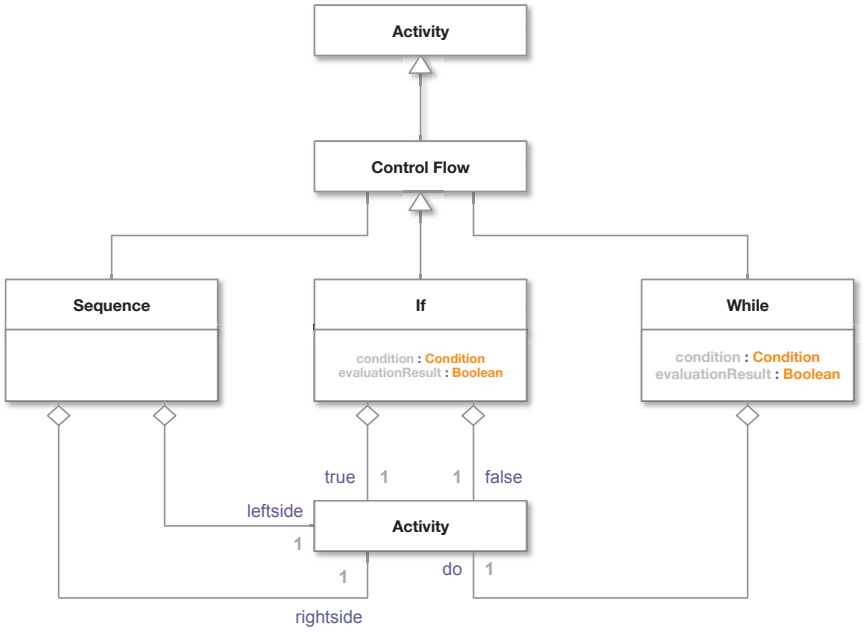

Figure 3.14 Control Flow activity type UML class diagram

Sequence Activity

The **Sequence Activity** type represents the logic implementing a *sequential execution*. For instance, let us assume that **sequence (A, B)** represents a *sequence activity* that relates activity types **A** and **B**. Now assume that **seq (a_1, b_1)** represents an *instance* of **sequence (A, B)** where a_1 and b_1 are instances of **A** and **B** (respectively). This relationship implies that, when executed, instance **seq** will execute a_1 until *completion* before executing b_1. More formally, if a_1**.ends** is an integer representing the *time* when a_1 *ends* its execution, and b_1**.starts** is an integer representing the time when b_1 *starts* its execution, instance **seq** will respect the relationship $a_1 < b_1$ at runtime.

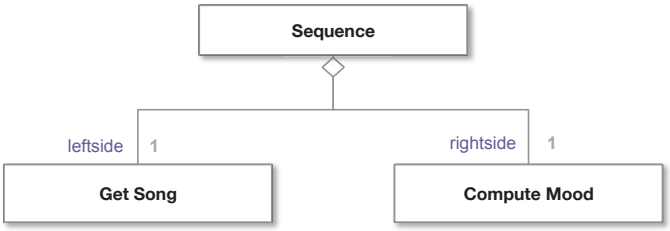

Figure 3.15 UML class diagram illustrating the use of the Sequence Activity type

As shown in **figure 3.14**, the **Sequence Activity** type has the following structure:

< *leftside*: Activity, *rigthside*: Activity >

Its attributes are described as follows:

- *leftside* represents the activity operand that is *first* executed.
- *rightside* represents the activity operand that is executed next.

As an example consider the *Get Song* and *Compute Mood* activities of the *Status Updater* scenario. According to the scenario these activities have to be executed in sequence. **Figure 3.15** illustrates the use of the **Sequence Activity** type for implementing this order relationship. In the figure, **Get Song** and **Compute Mood** activities are represented as **Atomic Activity** types. When executed, an instance of the sequence type will execute *first* an instance of the **Get Song** type (i.e., the *lefside* operand) and *then* an instance of the **Compute Mood** type (i.e., the *rightside* operand).

IF Activity

The **If Activity** type represents the logic implementing a *conditional execution*. For instance, let us assume that **IF (Θ, A, B)** represents an *if activity* that relates condition expression Θ and activity types **A** and **B**. Now assume that **if (Θ_1, a_1, b_1)** represents an *instance* of **IF (Θ, A, B)** where Θ_1, a_1 and b_1 are instances of **String**, **A** and **B** (respectively). This relationship implies that, when executed, instance **if** will execute a_1 *if and only if* Θ_1 is evaluated to *true*. Otherwise instance **if** will execute b_1.

As shown in **figure 3.14**, the **If Activity** type has the following structure:

< *condition*: String, *evaluationResult*: Boolean, *true*: Activity, *false*: Activity >

Its attributes are described as follows:

- *condition* contains a string representing a boolean expression.
- *evaluationResult* contains the *result* of the evaluation of the activity condition.
- *true* represents the activity operand that is executed when the activity condition is evaluated to *true*.
- *false* represents the activity operand that is executed when the activity condition is evaluated to *false*.

As an example consider an extended version of the *Status Updater* scenario where the songs listened by a user can be retrieved from two different services: the *LastFM* and the *Deezer*[7] services. Also consider that the Status Updater coordination decides which service to call by evaluation the *condition* "*useLastFM == true*". **Figure 3.16** illustrates the use of the **If Activity** type for implementing this decision by relating the activity types **Get Song From LastFM** and **Get Song From Deezer**. When executed, an instance of the *if* type will evaluate whether the value associated to *useLastFM* variable is true. In that case it will execute an instance of the **Get Song From LastFM** activity (i.e., the *true* operand). Otherwise it will execute an instance of the **Get Song From Deezer** activity (i.e., the *false* operand).

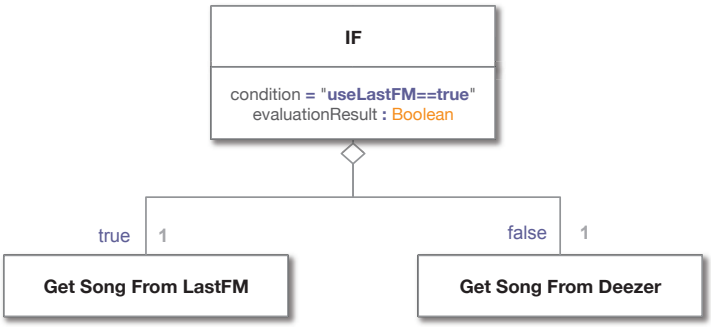

Figure 3.16 UML class diagram illustrating the use of the IF Activity type

While Activity

The **While Activity** type represents the logic implementing an *iterative execution*. For instance, let us assume that **While (Θ, A)** represents a *while activity* that relates condition expression Θ and activity type **A**. Now assume that **while (Θ_1, a_1)** represents an *instance* of **While (Θ, A)** where Θ_1 and a_1 are instances of **String** and **A** (respectively). The relationship **while (Θ_1, a_1)** implies that, when executed, instance **while** will execute a_1 *if and only if* Θ_1 is evaluated to *true*. Otherwise it will execute nothing. In case *if* Θ_1 is evaluated to *true*, instance **while** will re-instantiate activity type **A** for preparing a new iteration e.g., obtaining the instance represented by the expression **while (Θ_1, a_2)**, and then it will re-evaluate Θ_1 for deciding whether a_2 has to be executed or not. Note that an instance of the **While Activity** type may interact with multiple instances of other activity type at runtime. For instance, in the **While (Θ, A)** example, instance **while** can interact with a_1, a_2 … a_n where a_i represents the i_{th} iteration.

[7] http://www.deezer.com

As shown in **figure 3.14**, the **While Activity** type has the following structure:

< *condition*: String, *evaluationResult*: Boolean, *do*: Activity >

Its attributes are described as follows:

- *condition* contains a string representing a boolean expression.
- *evaluationResult* contains the *result* of the evaluation of the activity condition (i.e. either *true* or *false*).
- *do* represents the activity operand that is executed when the activity condition is evaluated to *true*.

As an example consider the *Get* Song activity of the Status Updater scenario. As you may recall this activity has to be executed several times in order to obtain the last song played by a user. Also recall that we represent this activity using the **Get Song Atomic Activity** type (cf. **section 3.3.1**). **Figure 3.17** illustrates the use of the **While Activity** type for implementing the required iterative behavior over the **Get Song** activity. As shown in the figure, the expression "*isUserListeningMusic()==true*" is used for deciding whether a new iteration has to be executed (i.e., the condition). If the user is listening music, an instance of the while activity will evaluate this expression to *true* and then it will execute an instance of the activity type **Get Song** (i.e. the *do* operand). Otherwise the while activity will complete its execution.

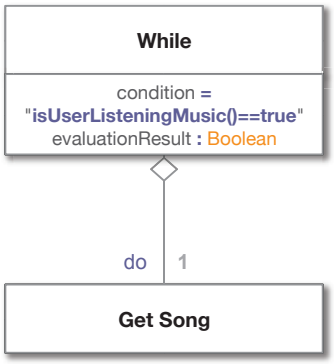

Figure 3.17 UML class diagram illustrating the use of a While Activity type

3.3.3 Workflow Activity

The **Workflow Activity** type represents the logic implementing a services' coordination as an *ordered* set of activities. As shown in **figure 3.11**, a **Workflow Activity** type has the following structure:

< *controlFlow*: ControlFlowActivity >

Besides having a *name*, an *input*, an *output* and *local* variables (inherited from the **Activity** type), a **Workflow Activity** type has a *controlFlow* that represents the order in which the activities composing a workflow have to be executed. For instance, **figure 3.18** shows the **Workflow Activity** type representing the *Status Updater* services' coordination.

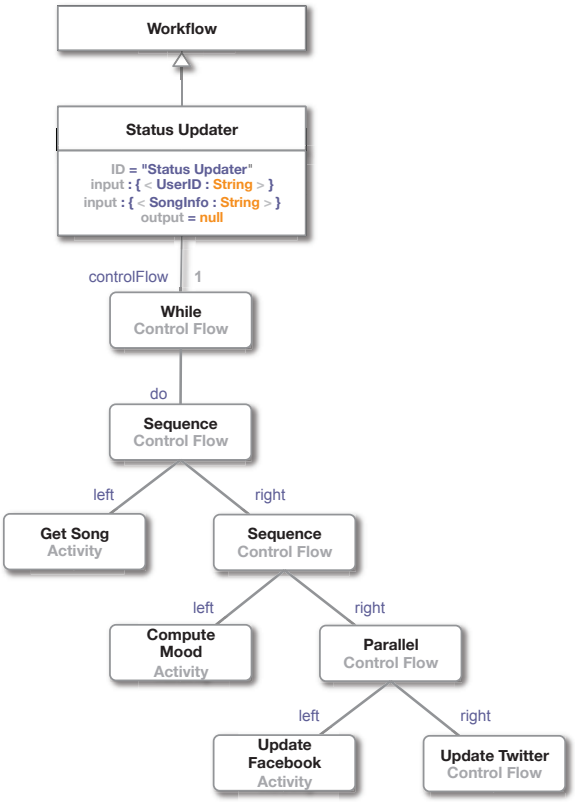

Figure 3.18 UML class diagram illustrating a specific workflow as a Workflow Activity type

As shown in the figure, the **Status Updater Activity Workflow** type receives a *userID* as input, produces *no output* and defines a local variable called *songInfo*. The figure also shows the **Control Flow Activity** type representing the workflow' *control flow*. Note that we represent it as an *activity tree* where:

- The *leaf nodes* represent the **Atomic Activity** types composing the workflow (i.e., activities *Get Song*, *Compute Mood*, Update Facebook and Update Twitter).
- The *internal nodes* represent the **Control Flow Activity** types representing the order among activities (e.g., the fact that the *Get Song* activity must be executed before the *Compute Mood* activity).
- The *root node* represents the **Control Flow Activity** type containing a workflow implementation (e.g., the While activity containing the logic that specifies what to do for each song listen by a user).

Since the AP Model is based on the same concepts as the Object Oriented Model (i.e., type, instance of a type, inheritance), we can use an *object-oriented language* for defining instances of **Workflow Activity** types (e.g., C++, Java, C#). For instance, **listing 3.1** shows the C# expression used for creating an instance of the **Status Updater Workflow Activity** type[8]. The expression is interpreted as follows:

[8] The example assumes that all the AP Model types were previously defined as C# classes.

- **Line 1** creates an instance of the **Status Updater Workflow Activity** type called *myWorkflow*.
- **Line 2** assigns the *name* of the workflow. Recall that this name uniquely identifies an activity type participating in a workflow.
- **Lines 3 and 6** create instances of the type **Variable** and assign them to the *input* and *local* attributes of the workflow (respectively).
- **Line 9** creates the instance of the **While Activity** type that implements the workflow' *control flow*. Recall that this instance represents the *root* of the workflow' control flow (see **figure 3.18**).
- **Lines 10, 15 and 20** create the activity instances composing the control flow. Recall that these instances represent the *internal nodes* of the control flow (see **figure 3.18**).
- **Lines 11, 16, 21 and 22** create the instances of the **Atomic Activity** types participating in the workflow. Recall that these instances represent the *leaf nodes* of the control flow (see **figure 3.18**).

```
1  var myWorkflow = new StatusUpdater( ) {
2      name = "Status Updater",
3      input = new Set<Variable>( ) {
4        new Variable<UserID, String>( )
5      },
6      local = new Set<Variable>( ) {
7        new Variable<SongInfo, String>( )
8      },
9      controlFlow = new WhileActivity( ) {
10       do = new Sequence() {
11           lefside = new GetSong() {
12             name = "Get Song",
13             input = this.input[0]  // User ID
14           },
15           rigthside = new Sequence( ) {
16             lefside = new ComputeMood( ) {
17                 name = "Compute Mood",
18                 input = this.local[0]  // Song Info
19             },
20             rigthside = new Parallel( ) {
21                 lefside = new UpdateFacebook( ) { ... },
22                 rigthside = new UpdateTwitter( ) { ... }
23             }
24           }
25       }
26     }
27 }
```

Listing 3.1 Example of the definition of a Workflow Activity instance

Listing 3.1 also shows how *data flow* among activity instances: if **a** and **b** are activities of the same workflow, **a** can access the attributes of **b** and **b** can access the attributes of **a** (i.e., activities share the same *memory space*). For instance, the expression of **line 13**:

input = this.input[0]

copies the value of the workflow *UserID* **input** variable (i.e., referenced by *this.input[0]*) to the *Get Song* input variable. In the same way the expression **line 18**:

input = this.local[0]

copies the value of the workflow *songInfo* **local** variable (i.e., referenced by *this.local[0]*) to the *Get Song* input variable.

3.4 Execution Unit Type

We use the **Execution Unit** type for representing an *entity* that goes through a series of *states* at runtime and *notifies* its progression to the *execution environment* by producing *events*. As shown in **figure 3.19**, the **Execution Unit** type has the following structure:

< *states*: Set<State>, *notifies*: Event >

Its attributes are described as follows:

- *states* represents the set of *ordered states* through which goes an execution unit when executed (e.g., an activity instance goes first to the *initial* state and sometime later it arrives to the *final* state). We use the type **State** for representing execution unit' states.
- *notifies* represents the *type* of events produced by an execution unit (e.g., the execution of an activity produces instances of type **ActivityEvent**). We use the type **Event** for representing *events*.

As shown in **figure 3.1**, the AP Model classifies execution units into *activities*, *active policies* and *policy' rules*. This implies that the behavior of instances of **Activity**, **Active Policy** and **Rule** types can be described as a *series of states* and *transitions* among the states. This will be further described in **chapter 4**.

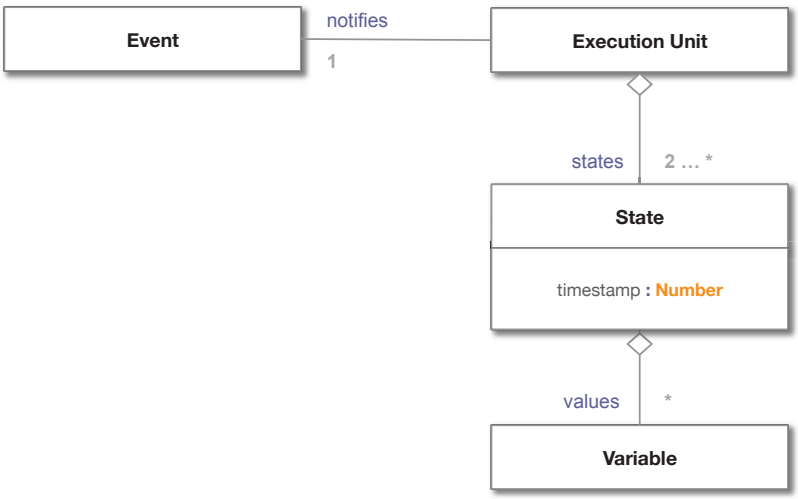

Figure 3.19 Execution Unity type UML class diagram

The type **State** represents a *momentum* during the execution of an *execution unit* (see **figure 3.19**). This type has the following form:

< *timestamp*: Number, *values*: Set<Variable> >

Its attributes are described as follows:

- *timestamp* represents the *time* associated to a state.[9]
- *values* represents the set of values that contained in the variables of an *execution unit* at time *timestamp*.

Consider for example the instance $\mathbf{a_1}$ of type **Get Song Atomic Activity** (cf. section 3.3.1). Figure 3.20 shows the representation of $\mathbf{a_1}$ as an *execution unit* that passes through two states: (i) the *initial state* containing the values of $\mathbf{a_1}$ *before* the service' operation call and (ii) the *final state* containing the values of $\mathbf{a_1}$ *after* the operation call. Before the call the *operationResult* and *output* attributes have no value (i.e., they are *null*). Then, after the call, these attributes have specific values (i.e., the string "*Sting – Fields of Gold*").

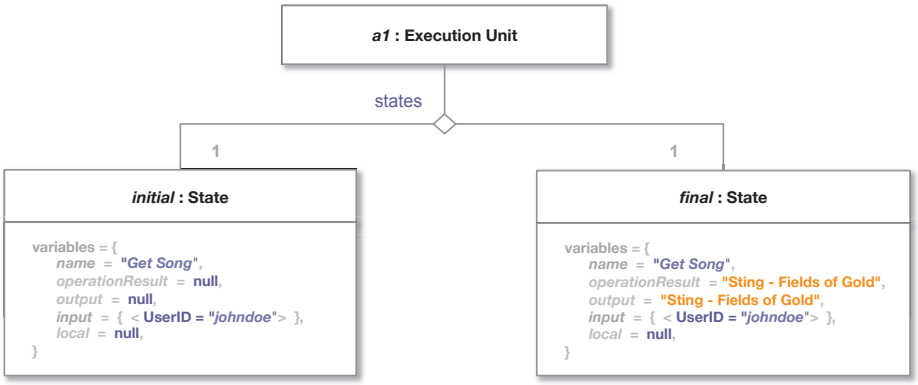

Figure 3.20 UML object diagram illustrating the states of an execution unit

The type **Event** represents an interest *happening* during the execution of an execution unit (see figure 3.19). For instance, an activity has *started* or *ended* its execution. As shown in figure 3.21, the event type has the following structure:

< *timeStamp*: Number, *producerID*: Number, *delta*: Set<Variable> >

Its attributes are described as follows:

- *timestamp* represents the *time* at which an event was produced.
- *producerID* represents a string *uniquely* identifying the *execution unit* producing an event. We assume that this identifier is assigned by the system at the moment of instantiating a *type*.

[9] The representation of time can be of different granularities (e.g., hour, day, minute, etc.). We consider that this granularity is determined by the system executing the workflows. We also consider that, independently of its representation, time can be transformed into a positive integer (i.e., a number $i \in [0, \infty]$).

- *delta* contains *information* about the conditions in which an event is produced (i.e., the state of the execution unit producing the event). We represent the delta as a set of pairs of the form *variable = value*.

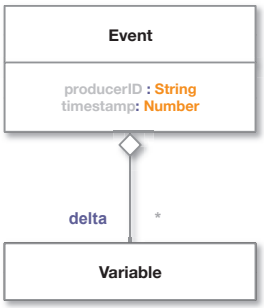

Figure 3.21 Event Type UML class diagram

As an example consider the instance a_1 of the **Get Song** activity type (see **figure 3.20**). When this activity is executed, it produces instances of the **ActivityEvent** type (a subtype of the **Event** type) each time a_1 reaches a new state. For instance, **figure 3.22** illustrates the event e_1 produced by a_1 when it reaches the *initial* state. As you can see, e_1 was produced at time **5** by the execution entity identified as "a_1" (see *timestamp* and *producerID* respectively) and contains in its *delta* the state of a_1 at the time of production of the event (cf. **figure 3.20**).

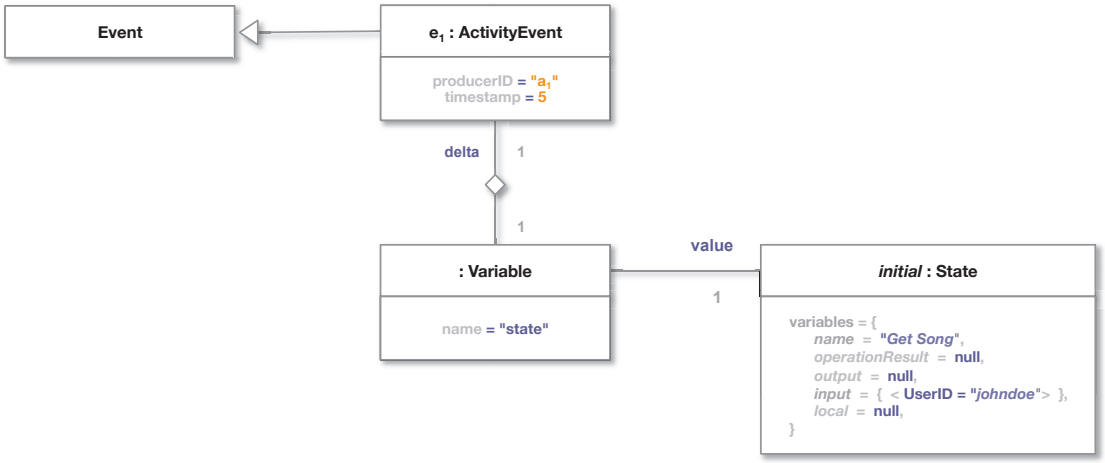

Figure 3.22 UML object diagram illustrating an Activity Event instance

In the AP Model we assume that an *execution unit* produces an *event* every time it enters a state (e.g., when an activity is initialized, it produces an event instance of type **Activity-Initialized**). Thus, in order to describe the behavior of all the AP Model *execution units*, we have specialized the **Event** type into several subtypes (see **figure 3.23**):

- **Activity Life Cycle Event** types. They represent the events describing the *lifecycle* of every activity instance. For instance, the activity has been *initialized*, *completed* or *failed*.

- **Atomic Activity Event** types. They represent the events produced by an *atomic activity instance* during a service' operation call. For instance, the activity is *prepared* for invoking the operation or the operation has been *invoked*.
- **Control Flow Activity Event** types. They represent the events produced during the execution of *control flow activity* instances. For instance, an *if* activity has *evaluated* its associated condition or a *sequence* activity has executed one of its operands.
- **Active Policy Event** types. They represent the events describing the *lifecycle* of an active policy instance. For instance, a policy has been (de) *activated* or *completed*.
- **Rule Event** types. They represent the events describing the *lifecycle* of a *rule instance*. For instance, a rule has been *triggered* or its action has been *executed*.

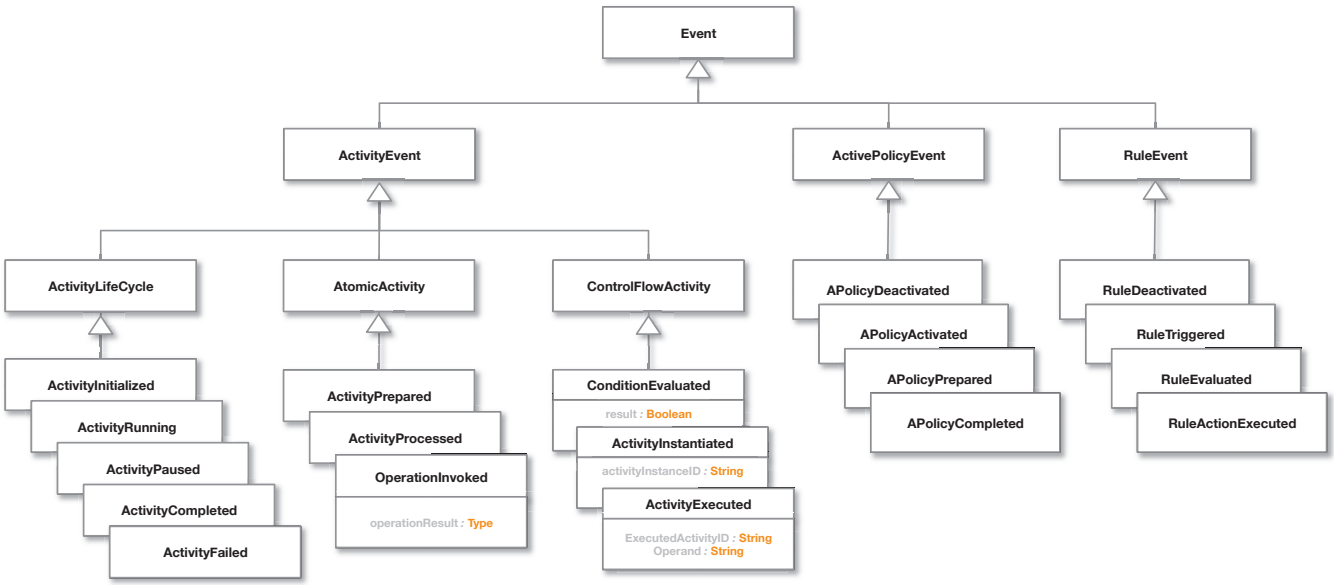

Figure 3.23 Event types UML class diagram

3.5 Active Policy Type

We use the **Active Policy** type for representing the *business logic* implementing a *non-functional property*. As shown in **figure 3.24**, the **Active Policy** type has the following structure:

< *rules*: Set<Rule>, *scope*: Set<ExecutionUnit >, *variables:* Set<Variable> >

Its attributes are described as follows:

- *rules* represents a non-empty set of *event-condition-action* (ECA) rules that *verifies* and *enforces* a non-functional property at *runtime*. We use the type **Rule** for representing ECA rules.
- *scope* represents the association of an active policy with one or several *workflow' execution units*. When an execution unit **eu** is associated to a policy **ap** it is said that "**eu** *belongs to the scope of* **ap**" or that "**ap** *applies to* **eu**".

- *variables* represents the set of *local variables* composing an active policy and which are accessible to the policy' rules (i.e., rules can read and modify variables' values).

The **Rule** type represents an ECA rule with the classical semantics: « on the notification of an event of type *E*, if a condition *C* is verified, execute an action *A* ». As shown in **figure 3.24**, the **Rule** type has the following structure:

< *event*: Event, *condition*: Condition, *action:* Activity >

Its attributes are described as follows:

- *event* represents a significant *happening* occurring during the execution of an *execution unit* at time *t*.
- *condition* represents a *boolean predicate* that is evaluated over the *state of a policy* (i.e., the values of the *policy variables* and the state of the *scope* execution units).
- *action* represents an activity that can (i) act on the execution of a workflow (e.g., stop, resume, update), (ii) signal an event and (iii) activate (deactivate) the rules of a policy.

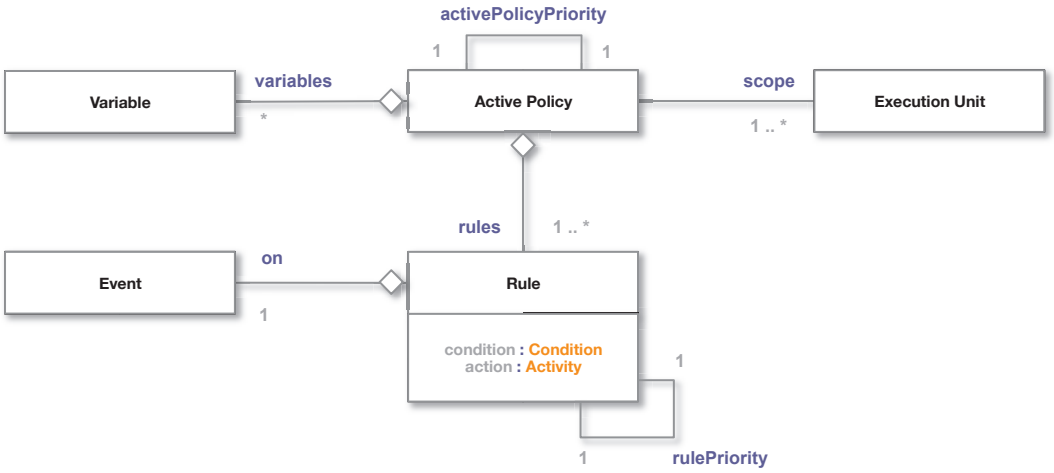

Figure 3.24 Active policy type UML class diagram

Consider for instance the **Log Active Policy** type shown in **figure 3.25**, which represents policy instances that *track* the execution of other policies and that *save* into a *log* the event produced by these policies. As shown in the figure, the **Log** policy type (i) can be associated to any number of execution units of type **Active Policy** (see *scope* relationship), (ii) has a single variable called *log* that contains the *string* representation of the events produced by the execution of units (i.e., the policies belonging to the policy' scope) and (iii) is composed of a single ECA rule named **Save Into Log** that has the following semantics: *on* the notification of any event (e.g., instances of **ActivityEvent** or **ActivePolicyEvent**), if the event was produced by any of the policies belonging to the policy' *scope*, execute an instance of the activity **EventToString**, which transforms the event into a *string* and save it into the log.

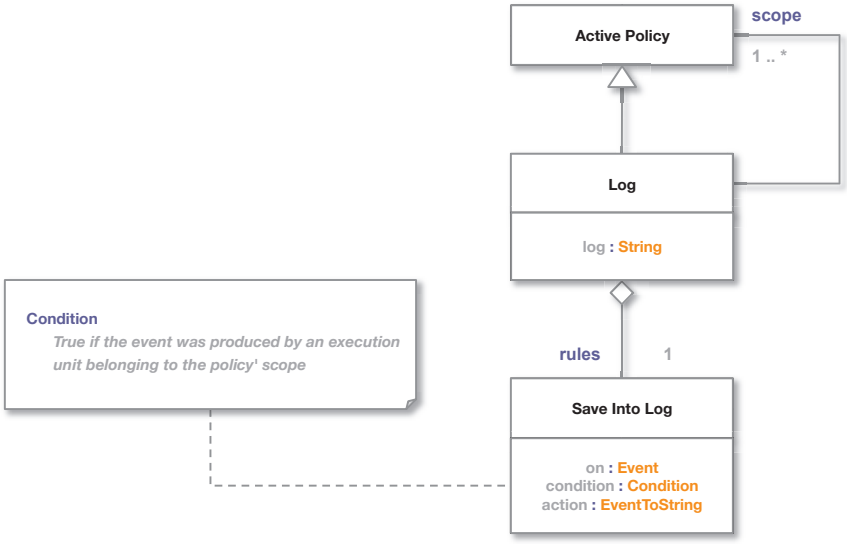

Figure 3.25 Log Active Policy type UML class diagram

Figure 3.25 also shows that active policies can be *applied to* any number of *execution units* as long as they belong to the same type (e.g., if **ap** is a policy instance that can be applied to activities and **a**, **seq** and **wf** are instances of *atomic activity*, *control flow* and *workflow* activity types, then applying **ap** to the set {**a**, **seq**, **wf**} is a valid association). However, we have identified two special kinds of associations (see **figure 3.26**):

- Applying a policy to a *single activity*. We call this kind of policies *activity policies*. They are represented using the **A-Policy** type.
- Applying a policy to a *set of policies* of exactly the same type **T**. We call this kind of policies *policy of policies*. They are represented using the **P-Policy** type.

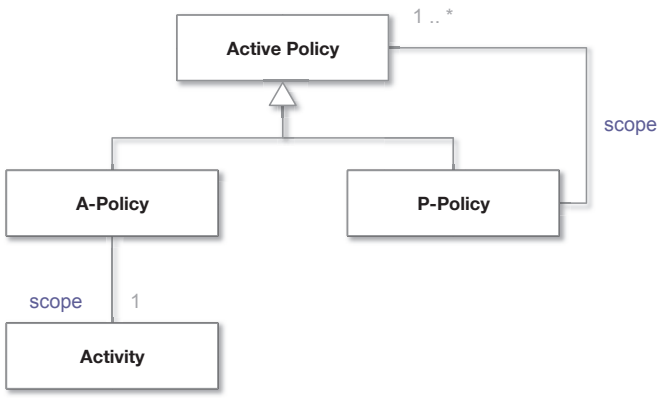

Figure 3.26 UML class diagram illustrating the identified Active Policy subtypes

For instance, consider that a programmer wants to keep track of the execution of the *authentication policies* associated to the Status Updater workflow (see **figure 3.6**). **Figure 3.27** illustrates the use of a **Log** policy instance called $\mathbf{p_3}$ for tracking the execution of the authentication policies $\mathbf{p_1}$ and $\mathbf{p_2}$ that are already associated to the Status Updater workflow. In this example, $\mathbf{p_1}$ and $\mathbf{p_2}$ are examples of **A-Policy** instances (i.e., *activity policies*) since they are

applied to activities *Update Facebook* and *Update Twitter*. In the same way, p_3 is an example of a **P-Policy** instance since it applies to policies p_1 and p_2 (i.e., p_3 is a policy of policies). Finally remark that the association of p_3, p_1 and p_2 can be represented as a tree (see **figure 3.27**), where *leaf nodes* represent activities and *internal nodes* represent active policies. We call this tree an *active policy tree*.

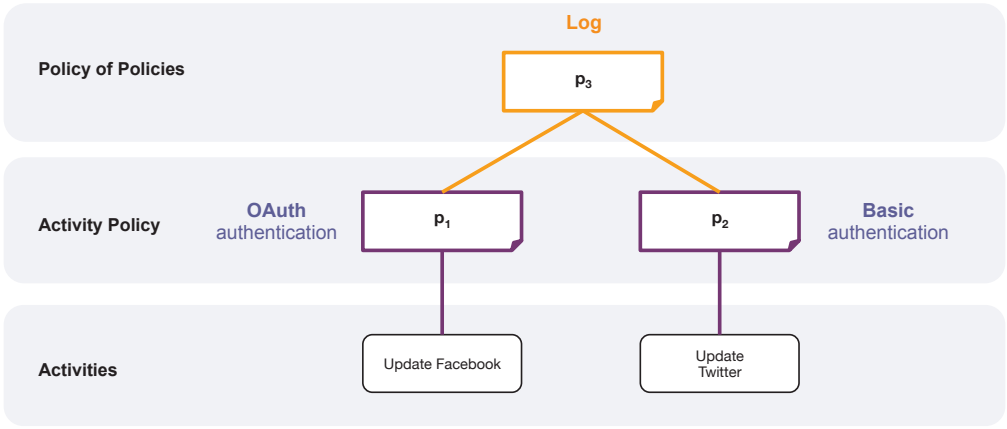

Figure 3.27 Example of A-Policy and P-Policy instances

Active policy rules are related to each other using the notion of *priority*. At execution time, priorities are used for *scheduling the execution* of rules that are triggered *at the same time* (i.e., triggered by the same event instance). We consider two kinds of priorities (see **figure 3.24**):

- *Priorities among rules*. They specify the *execution order* among the rules of a same policy. For instance, if r_1 and r_2 are rules belonging to policy **ap**, and r_1 has *lower priority* than r_2, when both rules are triggered r_1 has to be executed *before* r_2.
- *Priorities among policies*. They specify the *execution order* among the rules belonging to different policies. For instance, if r_x and r_{x+1} are rules belonging to policy ap_x, r_y is a rule belonging to policy ap_y and ap_x has *lower priority* than ap_y, then rules r_x and r_{x+1} have to be executed *before* r_y when triggered at the same time.

3.6 AP Language

We propose the *AP Language,* for defining *active policy types.* **Figure 3.28** illustrates the informal *grammar* of the language focusing *on* the expression used for defining an active policy type. The *AP Language*, is an extension to the C# language with constructors for (i) defining **Active Policy** types and (ii) associating these types to a **Workflow**.

Since the AP Language is based on the C# language, defining a type implies defining a *class* (i.e., use of the **class** keyword and definition of the *variables* and *methods* composing the class) and: (i) use the **policy** keyword for declaring a policy type (a class), (ii) specify the *rules* composing the policy and (iii) specify the scope of the policy.

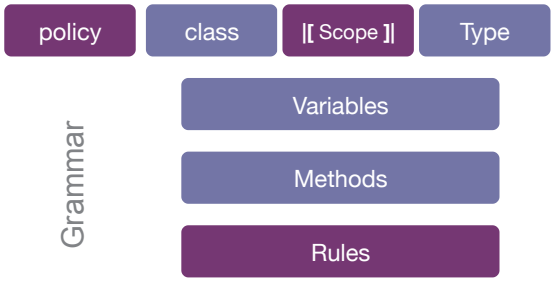

Figure 3.28 Structure of the AP Language expression defining an Active Policy type

An active policy type defined using the AP Language specializes the types of the AP Model. A new active policy type can represent concrete *non-functional properties* (e.g., *security*, *exception handling*, *atomicity*, *persistence*, etc.). The following sections illustrate how to define new **Authentication Active Policy** types using the *AP language* for dealing with the *authentication* requirements of the Status Updater coordination (i.e., the implementation of the *OAuth* and *Basic* authentication protocols).

3.6.1 Defining Active Policies

Listing 3.2 illustrates the definition of the **Log P-Policy** type (cf. **figure 3.27**). The expression is interpreted as follows:

- **Line 1** defines **Log** as an **Active Policy** subtype that *applies to* **Authentication** policies. For example instances of types **OAuth** and **Basic** (see **figure 3.27**).
- **Lines 2 and 3** define the *variables* composing a log policy: (i) a string called *log* used for recording the events produced during the execution of an authentication policy, and a *workflow* used for transforming an event instance into a string.
- **Line 4** defines the method **producedBy** that verifies whether an event **e** was produced by an *execution unit* belonging to the scope of a log policy. For instance, if (i) **log** and **auth** are instances of types **Log** and **Authentication**, (ii) **log** *applies to* **auth** and (iii) **e** was produced by **auth**, then **log.*producedBy*(e)** will return *true*.
- **Lines 7 to 10** define the *single* rule composing a log policy. The rule is interpreted as follows: if rule *SaveIntoLog* receives the notification of an *active policy event* and this event was *produced by* the policy scope, the rule transforms the event into a string and stores it into the *log*.

The AP Language supports the concept of *inheritance*. Thus, when an active policy type $\mathbf{AP_1}$ is defined as a subtype of the policy type $\mathbf{AP_2}$ we consider that "$\mathbf{AP_1}$ inherits all the properties of $\mathbf{AP_2}$" (i.e., its *variables*, *methods* and *rules*). For instance, the expression in **listing 3.3** defines the type **Authentication Log** as a special type of **Log** policy (see **listing 3.2**) that can be associated to *authentication policies* only. Since **Authentication Log** is a subtype of the **Log** type, it inherits all the properties defined in **Log** (i.e., the *log* variable and the rule *SaveIntoLog*) and thus it can use them. For instance, the authentication log policy can initialize the *log* with a default value.

```
1   policy class Log |[ Active Policy * ]| : Active Policy
2       String log ;
3       Activity EventToString ;
4       Boolean producedBy ( Event e ) {
5           // Code comparing scope and event producer
6       }
7       rule Save Into Log
8           ON Event event
9           IF producedBy ( event )
10          DO this.log += EventToString ( event )
```

Listing 3.2 Example of an active policy type definition

```
policy class AuthenticationLog |[ Authentication * ]| : Log {
    AuthenticationLog ( ) {
        this.log = new String ( );
    }
}
```

Listing 3.3 Example of the use of inheritance in the AP Language

3.6.2 BasicAuth Active Policy

Listing 3.4 shows the definition of the **BasicAuth** policy type. This type represents a policy that implements a basic authentication protocol based on a user' *username* and *password* (e.g., the one used by HTTP protocol[10]). As shown in the figure, the policy type (i) is composed of one rule and two variables, and (ii) can be associated to an activity of any type. The variables denote the values for **username** and **password**. The rule implements the authentication protocol by using the variables. Assuming that the policy is associated to an activity type, the policy works as follows:

- When the policy is instantiated the *values* for **username** and **username** are passed to the policy instance. Then the policy waits for the notification of triggering events (i.e., events of type **ActivityPrepared**).
- During its execution the activity will get *prepared* before calling its associated service operation. This causes the notification of an event of type **ActivityPrepared** that *triggers* rule $\mathbf{r_1}$.
- When $\mathbf{r_1}$ is triggered, the rule uses the values in variables **username** and **password** for configuring the activity **request** variable. Since rule $\mathbf{r_1}$ is marked as **sync**, the rule is executed *synchronously* with respect to the activity i.e., $\mathbf{r_1}$ first pauses the execution of the activity, then $\mathbf{r_1}$ configures the activity **request** variable and finally $\mathbf{r_1}$ resumes the activity execution.

[10] http://en.wikipedia.org/wiki/Basic_access_authentication

- During execution the activity uses (internally) this **request** variable for calling the service operation. Since the variable now contains the user' username and password, the operation call can be identified by the service as an *authorized call*.

```
policy class BasicAuth |[ Activity activity ]| : Authentication {
    String username, password;
    rule R1
      on ActivityPrepared
      if event.producedBy ( activity )
      do {
          activity.Request.Username = username;
          activity.Request.Password = password;
      }
}
```

Listing 3.4 Definition of the BasicAuth Policy type

3.6.3 OAuth Active Policy

Open authentication (OAuth) is a protocol that allows a user to grant a third-party application access to her information, without necessarily revealing her credentials (i.e., her username and password). **Figure 3.29** illustrates the steps involved in the OAuth protocol:

- The developer registers his/her application at the Service Provider (e.g., Facebook or Twitter) for obtaining a *request token* (**step 1**).
- When the application is used for the first time, the application redirects the user to the Service Provider for granting the application to access his information (**step 2**). During this step the application identifies it self by using the *request token* obtained. When the user grants the application, the application obtains an *access token* that represents the user authorization (**step 3**).
- When the application calls a service operation the application includes the *access token* as a parameter, which is used by the service provider for determining whether the application is authorized to access/update data (**step 4**).
- Once the *access token* no longer valid the application contacts the service provider for renewing the access token.

Listing 3.5 shows an implementation of the policy type **OAuth** using the AP Language. As you can see, the type **OAuth** policy is a kind of **Authentication** policy type that defines:

- An activity **getToken** that implements the logic for (i) retrieving the *access token* (i.e., steps 2, 3 and 4 of the OAuth protocol) and (ii) setting this value into the execution unit (i.e., the activity).
- Activity **renewToken** that implements the logic for renewing the *access token* (i.e., step 5).
- Variable **token** used for containing the *access token*.
- Rule **tokenretrieval** stating that, if the activity in the policy scope is **prepared** for execution, and the **token** variable does not contain a value (i.e. the variable is null), the variable **token** will take the value retrieved by the activity **getToken**.

- Rule **tokenrenewal** stating that, if the activity in the policy scope is **prepared** for execution, and the **token** variable has expired, the variable **token** will take the value retrieved by the activity **renewtoken**.

Note that the policy type *imports* the types defined in the namespace OAuth. The type **Token** belongs to this namespace.

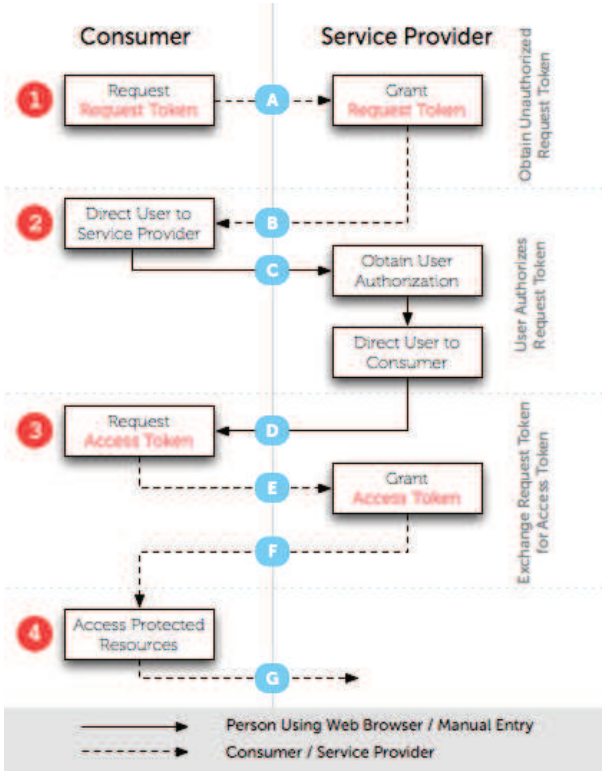

Figure 3.29 Steps of the OAuth Protocol

```
policy class OAuth |[ Activity activity ]| : Authentication {

  Token  token ;

  rule TokenRetrieval
    on ActivityPrepared
    if  event.producedBy ( activity ) && token == null
    do token = GetToken ( )

  rule TokenRenewal
    on ActivityPrepared
    if  event.producedBy ( activity ) && token != null && token.isExpired
    do token = RenewToken ( )

}
```

Listing 3.5 Definition of the OAuth Active Policy type

3.6.4 Applying Active Policies to Activities

Figure 3.30 illustrates the use of the AP Language for associating the **HttpAuth** policy type to the *Status Updater* workflow. In the example the scope of the policy type is associated to activity $\mathbf{N_5}$, which represents the **Update Twitter** activity. Remark that the values *"jane.doe"* and *"abc.123"* are passed to the *username* and *password* policy variables, respectively.

```
Map BasicAuth |[ Root.N1.N2.N4.N5 ]| as P1 with
    username = "jane.doe"
    password = "abc.123"
```

Figure 3.30 Association of a BasicAuth policy type to the Status Updater workflow

In order to illustrate the use of the authentication policy types in the *Status Updater* scenario, **figure 3.31** shows once again the expression tree of the *Status Updater* workflow. However, this time the nodes of the tree are annotated with unique identifiers.

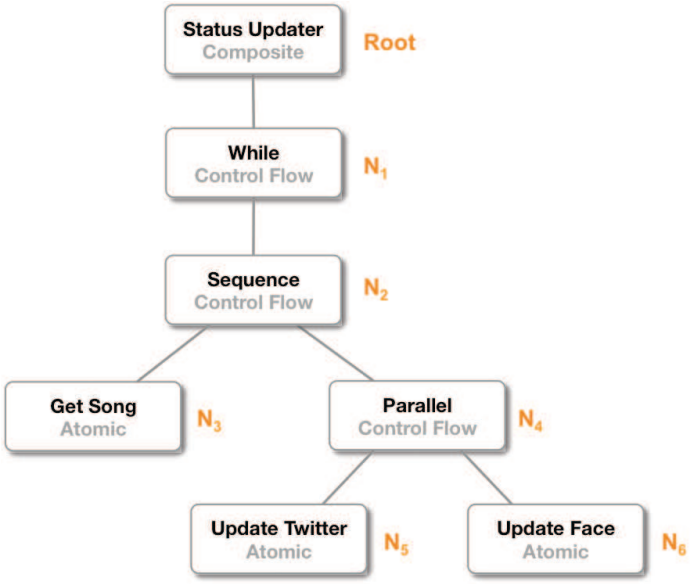

Figure 3.31 Annotated version of the status updater activity tree

3.7 Conclusions

This chapter introduced the AP Model, providing types for defining *active policy based workflows* that ensure non-functional properties at runtime. Our approach adheres to the hypothesis that modular programming has the usual benefits associated with *structured programming*, *information hiding* and *reusability*. Through the notion of active policy (AP) we provide means for modeling AP oriented services' coordination as workflows in a modular way. Using our model the workflow and the AP can be defined independently. According to the AP model an AP can be associated with an entity of a workflow (simple and composite activity).

In order to couple a workflow and its policies at runtime, it is necessary to synchronize them. This synchronization is modeled using event types and ECA rules. Event types are used for representing some of the execution states of the activities of a workflow. ECA rules represent NFPs enforcement actions modeled by AP types. Finally, this chapter showed through a simple example how the AP model and the AP Language can be used for defining AP types and associating them to a workflow. The specification of specific AP types and their association to a workflow is addressed in **chapter 5**. These definitions can be then executed according to strategies that are presented in the following chapter.

4 Executing Active Policy Based Workflows

This chapter describes how *active policy-based workflows* behave at runtime (i.e., how instances of the **Active Policy**, **Rule** and **Action** types interact among them for adding non-functional properties to a services' coordination implemented as a **Workflow**). In particular this chapter describes (i) how *events* produced by *execution units* (i.e., active policies and activities) trigger active policy' rules and (ii) how *action workflows* (i.e., the action part of policy rules) interact with *coordination workflows* for enforcing non-functional property at runtime.

The chapter is organized as follows. **Section 4.1** describes the behavior of **Activity** instances (i.e., *atomic activity*, *control flow* and *workflow* instances). **Section 4.2** describes the behavior of **Active Policy** and **Rule** instances. In particular this section describes the execution relationship among an active policy, the policy rules and the *execution entities* belonging to the policy scope. **Section 4.3** enumerates the interaction patterns among action workflows and coordination workflows that we identify for enforcing non-functional properties at runtime. Finally **section 4.4** concludes the chapter.

4.1 Executing an Activity

Recall from **chapter 2** that the AP Model considers 3 types of activities: **Atomic**, **Control Flow** and **Workflow** activities. Thus this section first describes the *general behavior* of an activity instance independently of its type. Then it details the behavior of instances of each activity type.

4.1.1 Activity General Behavior

The diagram of **figure 4.1** describes the *general behavior* of an **Activity** instance. As shown in the diagram, when an activity instance is created, it enters the **initialized** state. At this point the activity has already received its *input variables* and prepared its *local variables*. Then, when the activity receives the *run* signal, it moves to the **running** state and executes its associated *logic*. For example, an **Atomic** activity executes the logic implementing a service operation call during its *running* state, whereas a **Control Flow** activity coordinates the execution of the activities composing the control flow activity during this state. If an activity terminates

the execution of its logic *without error*, the activity moves to the **completed** state and sets the values for its *output variables*. Otherwise it moves to the **failed** state and stops its execution.

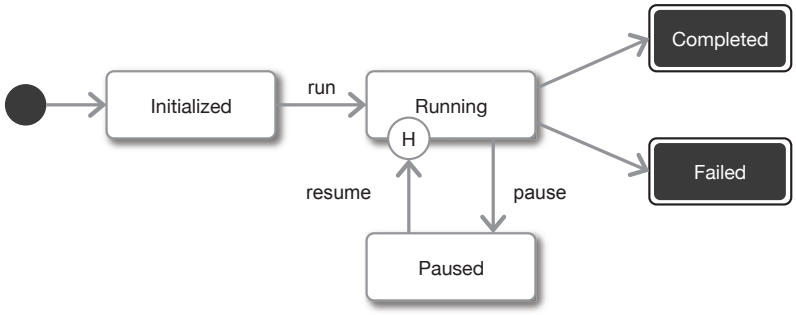

Figure 4.1 General behavior of an Activity instance (UML state machine diagram)

The diagram of **figure 4.1** also shows that activities can be *paused* and *resumed* while **running**. When an activity is an **Atomic** activity, entering the **paused** state causes the activity to immediately suspend its execution. In contrast, when the activity is a *composite activity* (i.e., a **Control Flow** or **Workflow** activity), entering the **paused** state causes the activity to send a *pause* signal to all its *sub-activities*, and once all the *sub-activities* are **paused**, the activity finally suspends its execution. For instance, **figure 4.2** illustrates the behavior of the *Status Updater* workflow when *paused*. As shown in the figure, when the *Status Updater* workflow receives the *pause* signal, this causes a cascade of *pause* signals through the workflow' activity tree. And once all the workflow activities are **paused**, the *Status Updater* workflow can move from the **running** state to the **paused** state.

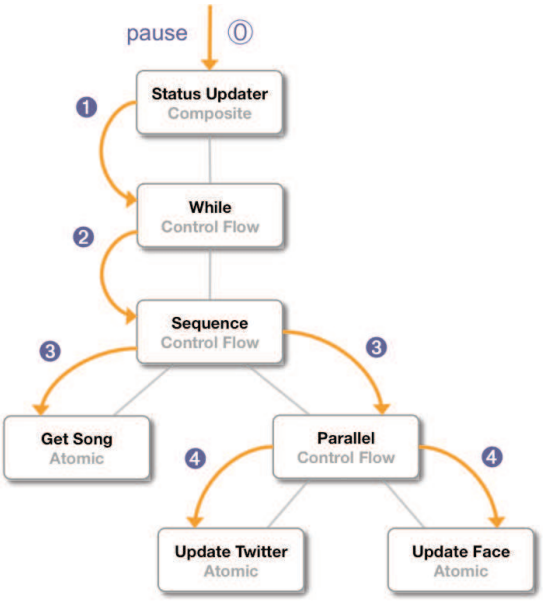

Figure 4.2 Behavior of a composite activity when "paused"

An analogous behavior is obtained when *resuming* a **paused** activity. If the activity is an atomic activity, the reception of the *resume* signal causes the activity to re-enter the **running**

state (exactly to the point it was before receiving the *pause* signal). In contrast, when the activity is a *composite activity*, the reception of the *resume* signal causes a cascade of resume signals through the activity tree (similar to the one shown in **figure 4.2**). And once all the activities participating in the activity tree are **running** again, the composite activity finally moves to the **running** state and continues its execution.

As shown in **section 3.4**, an activity instance is also an **Execution Unit** instance and thus it produces a series of *events* during its execution. An activity produces an *event* every time it enters a state. The event contains information (i.e., the execution environment) about the progression of its *life cycle* (cf. **ActivityLifeCycle** event type in **figure 3.23**).

For example, **figure 4.3** illustrates the events produced by the execution of 3 activity instances named $\mathbf{a_1}$, $\mathbf{a_2}$ and $\mathbf{a_3}$. As shown in the figure, the events produced by instance $\mathbf{a_1}$ describe a normal execution (i.e., $\mathbf{a_1}$ entered the **initialized** state, then it moved to the **running** state and finally it moved to the **completed** state). In contrast, the events produced by $\mathbf{a_2}$ describe an abnormal execution (i.e., when $\mathbf{a_2}$ was running an *error* was produced and thus it moved to the **failed** state). Finally note that the events produced by instance $\mathbf{a_3}$ also describe a normal execution but in this case the instance was **paused** and **resumed** before completing its execution.

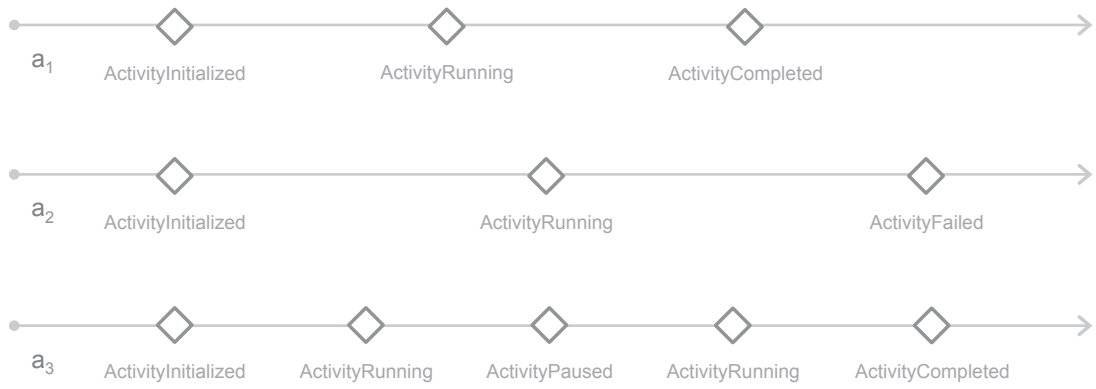

Figure 4.3 Example of events produced by the execution of an activity

Events are messages with information about the conditions in which it was produced. Event produced by activities, notify data bout the conditions in which the activity is executed (the variables' values of the activity execution state). As an example consider (i) the activity *Update Facebook* and (ii) the instance $\mathbf{a_1}$ of the *Update Facebook* activity in **figure 4.4**. **Table 4.1** illustrates the information notified by the events produced by $\mathbf{a_1}$ during its life cycle. The table is interpreted as follows:

- When $\mathbf{a_1}$ was *initialized*, it set the variable *status* to "Fragile - Sting" (i.e., the input) and initialized its local variables *user* and *successfulCall* to their default values (i.e., null and false respectively). Then $\mathbf{a_1}$ produces the event $\mathbf{e_1}$ (of type **ActivityInitialized**) at time $\mathbf{t_x}$ containing as *delta* all the variables of $\mathbf{a_1}$ (i.e., variables *status*, *user* and *successfulCall* and their values).
- When $\mathbf{a_1}$ started *running*, it produces the event $\mathbf{e_2}$ (of type **ActivityRunning**) at time $\mathbf{t_{x+1}}$. Note that at this point the variable *user* was already set and thus $\mathbf{a_1}$ had all the necessary values for updating a user Facebook status.

- When $\mathbf{a_1}$ was *completed*, it produced the event $\mathbf{e_3}$ (of type **ActivityCompleted**) at time $\mathbf{t_{x+2}}$. Note that at this point $\mathbf{a_1}$ had already set the value for its variable *successfulCall* to *true*, representing the fact that there was no error during the service' operation call.

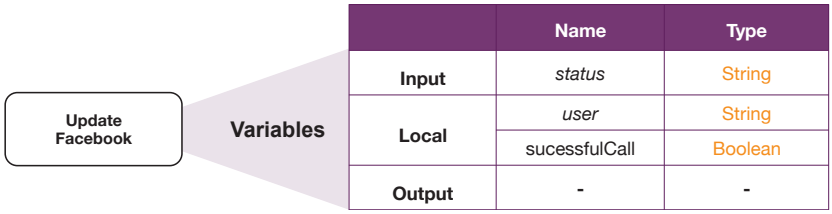

Figure 4.4 Update Facebook activity variables

	ActivityInitialized	ActivityRunning	ActivityCompleted
eventID	e_1	e_2	e_3
timestamp	t_x	t_{x+1}	t_{x+2}
producerID	a_1	a_1	a_1
delta	status = "Fragile Sting" user = null succesfulCall = false	status = "Fragile Sting" user = "john.doe" succesfulCall = false	status = "Fragile Sting" user = "john.doe" succesfulCall = true

Table 4.1 Example of the information contained in the events produced by an activity

4.1.2 Behavior of an Atomic Activity

As shown in the diagram of **figure 4.5**, an instance of an **Atomic** activity type passes through 3 sub-states when **running**:

- **Preparing**. During this state the activity prepares the values for the service' operation input parameters. Once the parameters are prepared, the activity moves to the **invoking** state.
- **Invoking**. During this state the activity *calls* the service' operation and *waits* for the result. If an error occurs during the operation call (e.g., the network is unavailable or the service responds with an *exception*), the activity moves to the **failed** state (see **figure 4.1**). Otherwise it moves to the **r-processing** state.
- **R-Processing**. During this state the activity processes the service' operation result (e.g., if a service sends its result using the XML format, the activity transforms the XML data types into AP Model *data types* during this state). Once the activity ends processing the result it moves to the **completed** state (see **figure 4.1**).

As an example consider the atomic activity *Get Song*. Recall that this activity is associated to a **Sequence** activity in the *Status Updater* workflow (see **figure 4.2**). Let us suppose that **act** and **seq** are instances of the *Get Song* and *Sequence* activities, respectively. Also suppose that **act** is one of the operands of **seq**. The diagram of **figure 4.6** illustrates the be-

havior of **act** when executed by **seq**. As shown in the diagram, when **act** receives the *run* signal, it starts **running** and thus it moves to the **preparing** state. This causes the preparation of the *getLastSong* input parameters (i.e., the *username* used for retrieving the song information). When **act** ends preparing the parameters, it moves to the **invoking** state, calls the operation in the LastFM service and waits for the result. Then, once **act** receives the operation result (i.e., the song information), it moves to the **r-processing** state and processes the returned data (e.g., by transforming the song information into an AP Model string). Finally, when **act** ends processing the result, it moves to the **completed** state and notifies the environment about its completion (e.g., **act** notifies **seq** about its completion and **seq** can continue its execution).

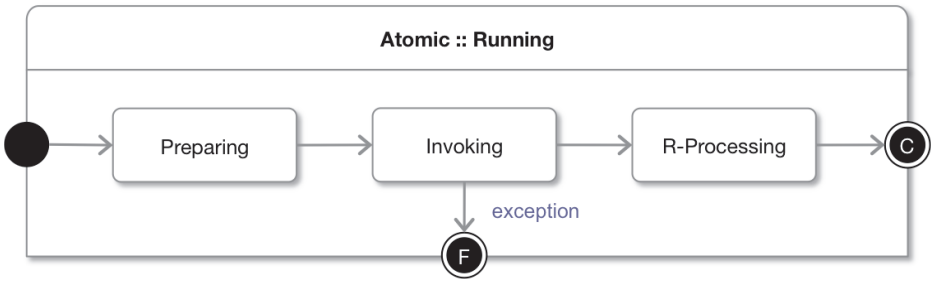

Figure 4.5 Behavior of an Atomic activity instance when running (UML state machine diagram)

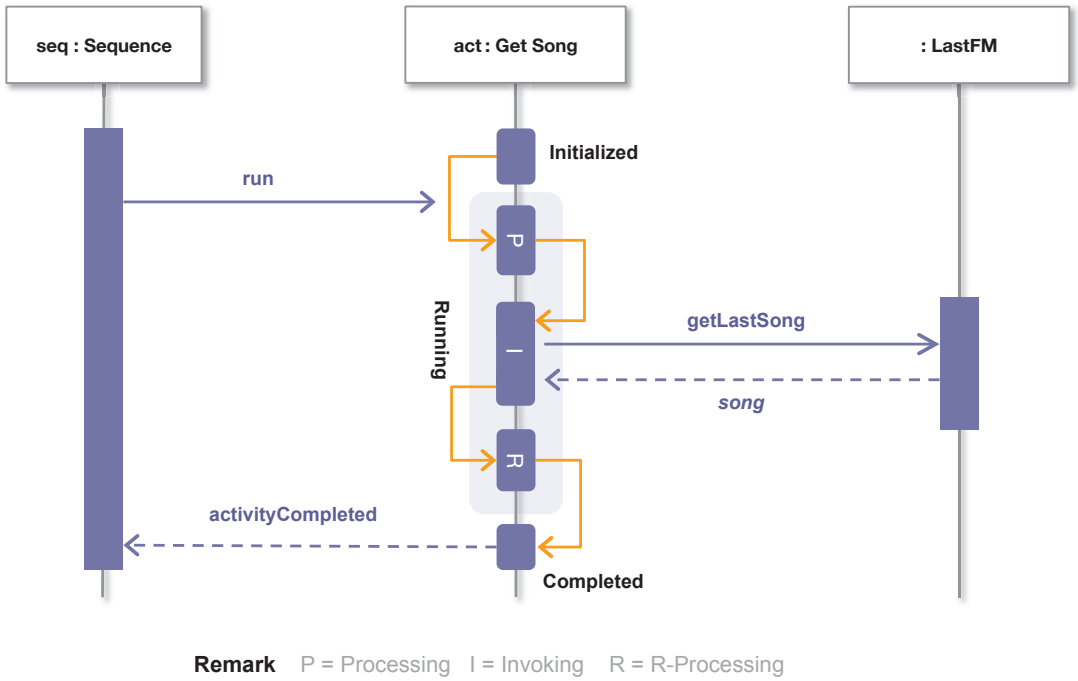

Figure 4.6 Example illustrating the execution of an Atomic activity instance when running (UML sequence diagram)

Figure 4.7 completes the previous example by showing the event trace produced by the execution of **act**. As shown in the figure, an *atomic* activity produces 3 events when **running**:

- An event of type **ActivityPrepared** representing the fact that the activity ended preparing the service' operation parameters (see event $\mathbf{e_5}$). The event delta[1] contains the list of parameters and their values.
- An event of type **ActivityInvoked** representing the fact that the activity received the service' operation result (see event $\mathbf{e_6}$). This event delta contains the operation' output in its delta.
- An event of type **ActivityProcessed** representing the fact that the activity finished processing the operation' result (see event $\mathbf{e_7}$). The event delta contains the value resulted from the processing in its delta.

Finally note that the events shown in **figure 4.7** illustrate the order relationship among **seq** and **act**. For example, since $\mathbf{e_1}$ and $\mathbf{e_2}$ are located at the same point in the timeline, this implies that both activities were **initialized** at the same time. Also note that $\mathbf{e_3}$ was produced before $\mathbf{e_4}$. This implies that **seq** was **running** before **act**. Besides, since we know that **act** and **seq** belong to the same activity tree, this also implies that **act** is a descendent of **seq**.

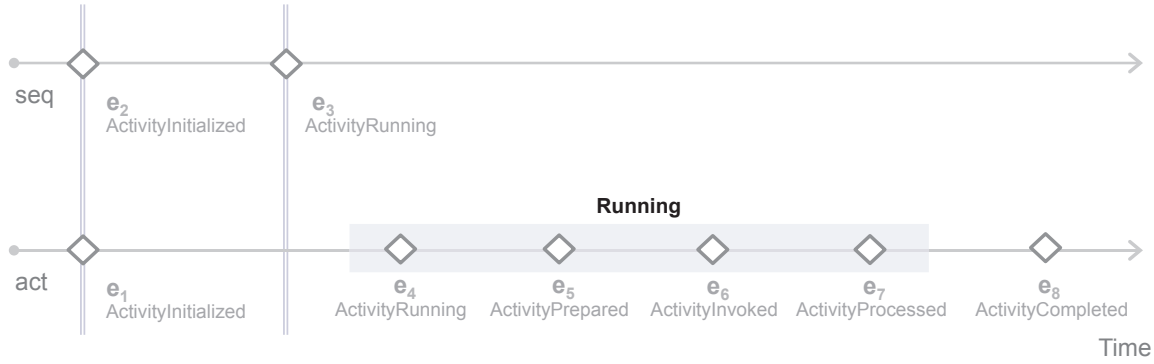

Figure 4.7 Events produced by an Atomic activity normal execution

4.1.3 Behavior of Control Flow Activities

Sequence (A, B)

As shown in the diagram of **figure 4.8**, an instance of a **Sequence** activity type passes through 2 sub-states at **run-time**:

- **Executing Left Side**. During this state the activity sends a *run* signal to its LEFT-SIDE operand for starting its execution and waits for its completion. If the LEFT-SIDE operand *fails*, the activity moves to the **failed** state (cf. **figure 4.1**). Otherwise it moves to the **executing right side** state.
- **Executing Right Side**. During this state the activity sends a *run* signal to its RIGHT-SIDE operand for starting its execution and waits for its completion. If the RIGHT-SIDE

[1] See the event type definition in Chapter 2.

operand *fails*, the activity moves to the **failed** state. Otherwise it moves to the **completed** state.

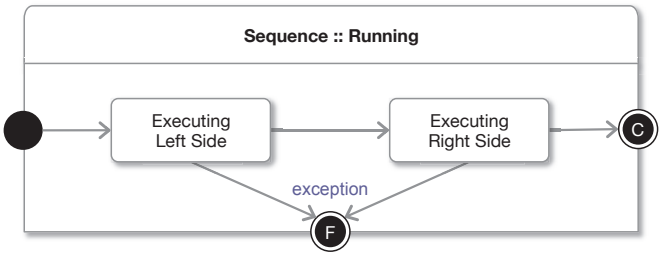

Figure 4.8 Behavior of a Sequence activity instance when running (UML state machine diagram)

As an example let us assume that **seq** (a_1, a_2) is an instance of the **Sequence (GetSong, Parallel)** activity type (i.e., **seq** has as left-side and right-side operators instances a_1 and a_2 of types **Get Song** and **Parallel**, respectively). The sequence diagram of **figure 4.9** illustrates the behavior of **seq** when executed. As shown in the diagram, when **seq** receives the *run* signal, it starts **running** and thus it moves to the **execute left side** state. Then **seq** sends a *run* signal to a_1 and waits a_1 to complete. Since a_1 completes without error, **seq** moves to the **execute right side** and starts the execution of a_2. Finally, since a_2 also completes without error, **seq** moves to the **completed** state and notifies the environment about its completion.

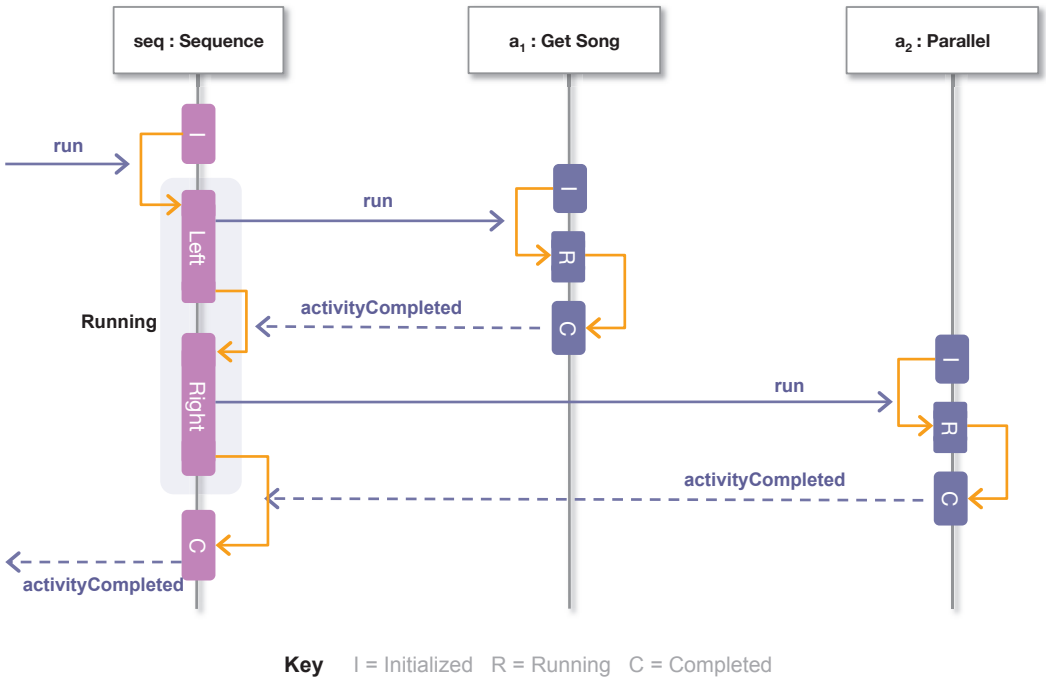

Figure 4.9 Example illustrating the execution of a Sequence activity instance (UML sequence diagram)

Figure 4.10 completes the previous example by showing the event trace produced by the execution of **seq**. As shown in the figure, a *sequence* activity produces 2 events when **running**:

- An event of type **ActivityExecuted** representing the fact that the LEFT-SIDE operand was executed (see event $\mathbf{e_5}$). This event contains the identifier of the executed activity in its delta.
- An event of type **ActivityExecuted** representing the fact that the RIGHT-SIDE operand was executed (see event $\mathbf{e_9}$).

Figure 4.10 Event trace of Sequence activity normal execution

If Θ do A else B

As shown in the diagram of **figure 4.11**, an instance of an **IF** activity type passes through 3 sub-states at **execution time**:

- **Evaluating**. During this state the activity evaluates whether its associated condition Θ is *true* or *false*. If Θ is *true*, the activity moves to the **executing A** state. Otherwise it moves to the **executing B** state.
- **Executing A**. During this state the activity executes its TRUE operand (i.e., it executes an instance of type **A**). If the TRUE part completes without error, the activity moves to the **completed** state. Otherwise it moves to the **failed** state (cf. **figure 4.1**).
- **Executing B**. During this state the activity executes its FALSE operand (i.e., it executes an instance of type **B**). If the FALSE part completes without error, the activity moves to the **completed** state. Otherwise it moves to the **failed** state.

As an example let us assume that **f** is an instance of the activity type **IF (Θ, A, B)**. The diagram of **figure 4.12** illustrates the behavior of **f** assuming that Θ evaluates to *true*. As shown in the diagram, when **f** receives the *run* signal, it starts **running** and thus it moves to the **evaluating** state. Since we assume that Θ evaluates to *true*, **f** moves to the **executing A** state and sends a *run* signal to **a** (i.e., the instance associated to its TRUE part). Finally, when **a** completes its execution, **f** moves to the **completed** state and notifies an event with its delta containing the execution environment specifying the completion of the execution of **a**.

Figure 4.13 completes the previous example by showing the event trace produced by the execution of **f**. As shown in the figure, an *if* activity produces 2 events at **execution-time**:

- An event of type **ConditionEvaluated** representing the fact that the activity terminated the condition evaluation (see event $\mathbf{e_2}$). This event delta contains the result of the condition evaluation (i.e., whether Θ evaluated to *true* or false).
- An event of type **ActivityExecuted** representing the fact that the TRUE / FALSE operand of the activity was executed (see event $\mathbf{e_6}$). This event delta contains the identifier of the executed activity.

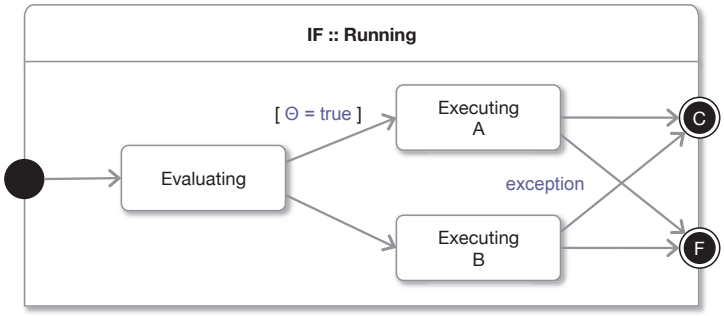

Figure 4.11 Behavior of an IF activity instance when running (UML state machine diagram)

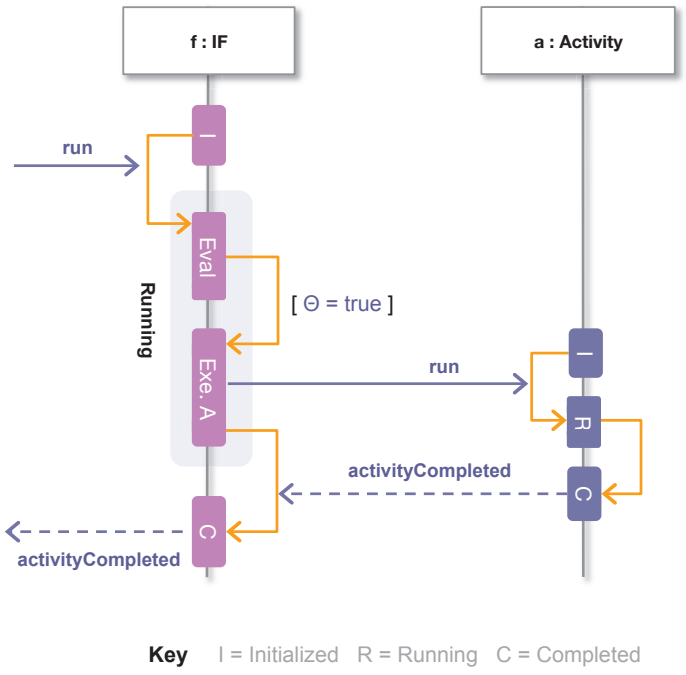

Figure 4.12 Example illustrating the execution of an IF activity instance (UML sequence diagram)

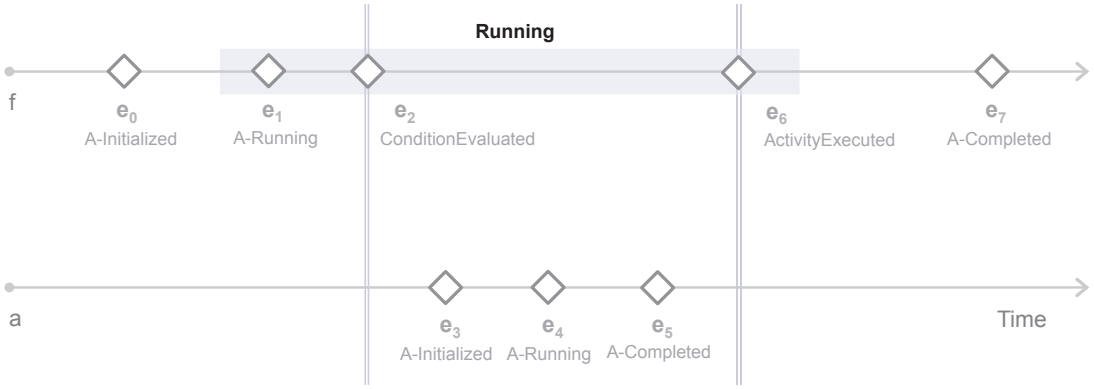

Figure 4.13 Event trace produced by the normal execution of an IF activity instance

While Θ Do A

As shown in the diagram of **figure 4.14**, an instance of a **While** activity type passes through 3 sub-states at **execution time**:

- **Evaluating**. During this state the activity evaluates whether its associated condition Θ is *true* or *false*. If Θ is *true*, the activity begins a new iteration by moving to the **instantiating** state. Otherwise it moves to the **completed** state (cf. **figure 4.1**).
- **Instantiating**. During this state the activity creates an instance $\mathbf{a_i}$ of its DO operand (i.e., $\mathbf{a_i}$ is an instance of **A** where **i** represents the number of the iteration). Once the activity ends the instantiation process it moves to the **execute instance** state.
- **Execute Instance**. During this state the activity executes $\mathbf{a_i}$ (i.e., it sends a *run* signal to the activity created in the previous state). If $\mathbf{a_i}$ terminates without failure, the activity moves to the **evaluating** state for determining whether a new iteration is necessary. Otherwise it moves to the **failed** state (cf. **figure 4.1**).

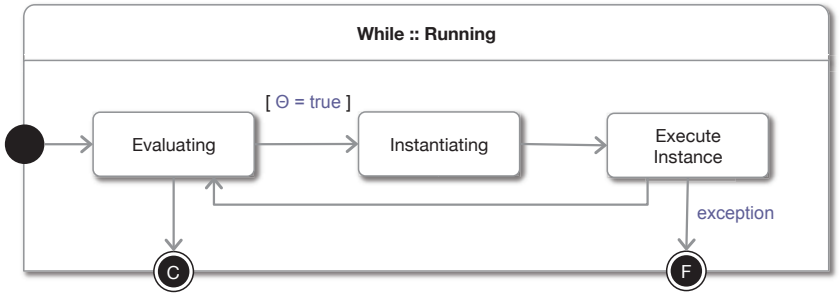

Figure 4.14 Behavior of a While activity instance when running (UML state machine diagram)

As an example let us assume that **w** is an instance of the activity type **While (Θ, A)**. The diagram of **figure 4.15** illustrates the behavior of **w** assuming that **w** iterates once before completing its execution. As shown in the diagram, when **w** receives the *run* signal, it starts **running** and thus it moves to the **evaluating** state. Since Θ evaluates to *true*, **w** moves to the **instantiating** state and creates a new instance of the activity type **A** called $\mathbf{a_1}$. Once $\mathbf{a_1}$ is created, **w** moves to the **execute instance** state, sends a *run* signal to $\mathbf{a_1}$ and waits $\mathbf{a_1}$ to complete. Since $\mathbf{a_1}$

completes without error, **w** moves to the **evaluating** state for determining whether a new iteration is necessary. Since we assume that **w** iterates only once, Θ evaluates to *false* and then **w** moves to the **completed** state.

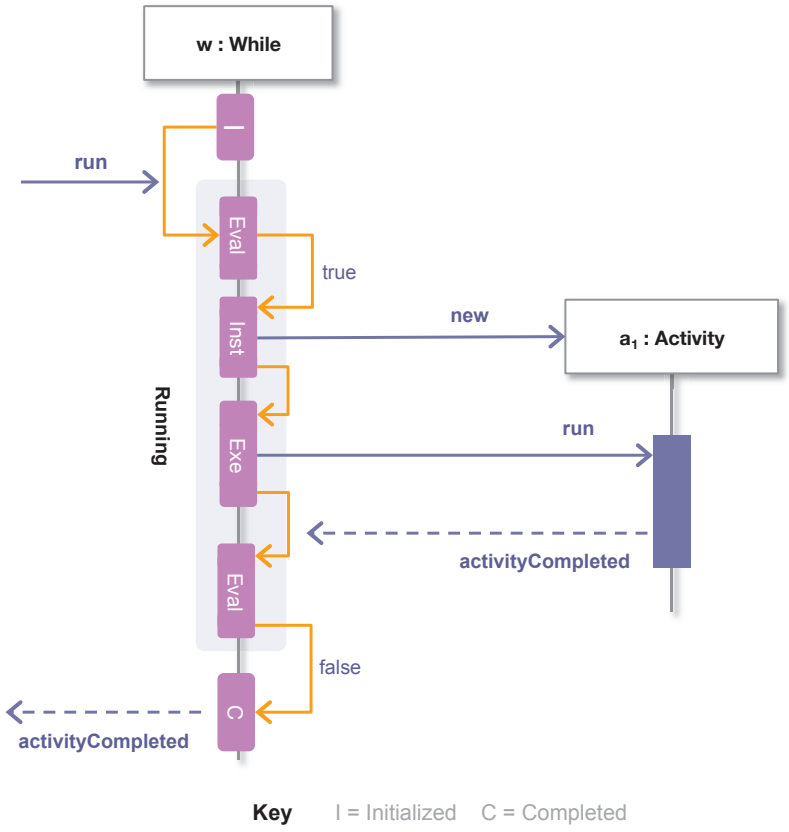

Figure 4.15 Example of an iteration of a While activity instance (UML sequence diagram)

Figure 4.16 completes the previous example by showing the event trace produced by the execution of **w**. As shown in the figure, a *while* activity produces 3 events at **execution-time**:

- An event of type **ConditionEvaluated** representing the fact that the activity ended the condition evaluation (see event $\mathbf{e_1}$). This event delta contains the result of the condition evaluation (i.e., whether Θ evaluated to *true* or false).
- An event of type **ActivityInstantiated** representing the fact that the activity created an instance of its DO part (see event $\mathbf{e_2}$). This event delta contains the identifier of the new activity instance (i.e., the name of instance $\mathbf{a_i}$).
- An event of type **ActivityExecuted** representing the fact that the activity created during the **instantiation** state was executed (see event $\mathbf{e_6}$). This event delta contains the identifier of the executed activity.

Note that each time a **While** activity begins an iteration, a new instance of its DO part is created. In consequence a *while* activity may have multiple instances associated to it at runtime. For example, let us assume that **w** iterates 3 times. **Figure 4.17** shows the activity tree representing **w** during the 3rd iteration. As shown in the figure, an iteration is represented

as a child (or branch) of a *while* activity in an activity tree. For example, since $\mathbf{a_1}$, $\mathbf{a_2}$ and $\mathbf{a_3}$ are children of **w**, they represent the 1st, 2nd and 3rd iteration of **w**. The figure also shows that no matter how many instances **w** creates, only one of them is in the **running** state at a time (e.g., since **w** executes the 3rd iteration, $\mathbf{a_3}$ is the only child activity that is in the **running** state).

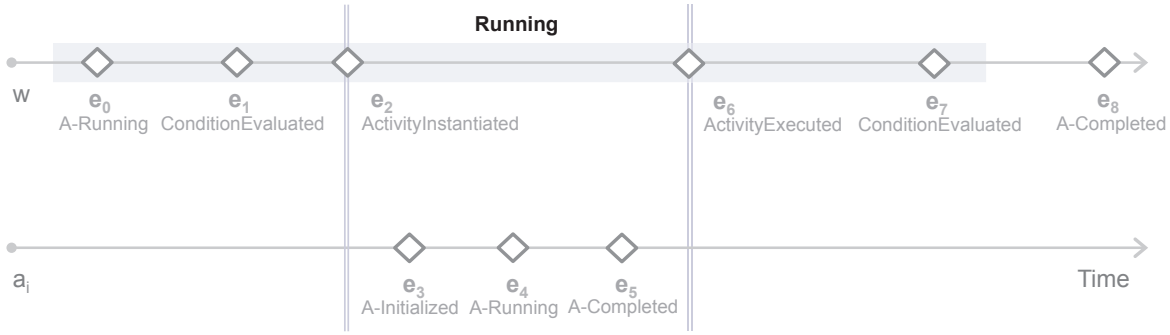

Figure 4.16 Event trace produced by a normal execution of a While activity instance

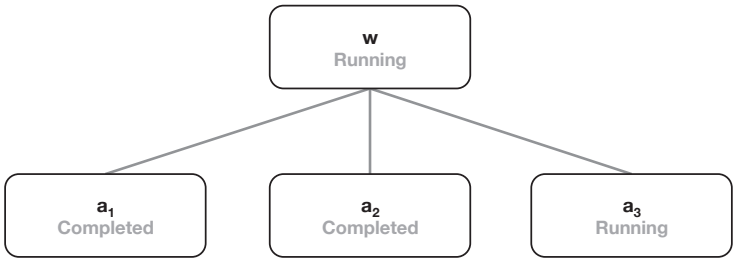

Figure 4.17 Activity tree representing a While activity instance iterating for the 3rd time

Parallel (A, B)

As shown in the diagram of **figure 4.18**, an instance of a **Parallel** activity type passes through 2 sub-states when **running**:

- **Executing A**. During this state the activity executes its LEFT-SIDE operand (i.e., it executes an instance of type **A**).
- **Executing B**. During this state the activity executes its RIGHT-SIDE operand (i.e., it executes an instance of type **B**).

Note that a *parallel* activity enters the states **Executing A** and **Executing B** simultaneously once it starts **running**. In this work we assume that a *parallel* activity ends running either when (i) both of its operands complete or (ii) one of its operands fails. As shown in **figure 4.18**, in the first case the activity moves to the **completed** state. In the second case the activity moves to the **failed** state.

As an example let us assume that **par (a, b)** is an instance of the activity type **Parallel (A, B)**. The diagram of **figure 4.19** illustrates the behavior of **par** assuming that both of its operands (i.e., activities **a** and **b**) completes without error. As shown in the diagram, when **par** receives the *run* signal, **par** moves to the **running** state and begins the execution of **a** and **b** at the same time. Then **par** waits for the completion of **a** and **b**. Since **b** completes before **a**, **par**

ignores the completion of **b** and remains in the **running** state. When **a** completes, since **b** already completes, **par** moves to the **completed** state and notifies an event notifying the conditions in which its execution was completed. **Figure 4.16** completes the example by showing the event trace produced by the execution of **par**. As shown in the figure, a *parallel* activity produces 2 events of type **ActivityExecuted** when **running**. Each event represents the fact that an activity' operand has completed its execution.

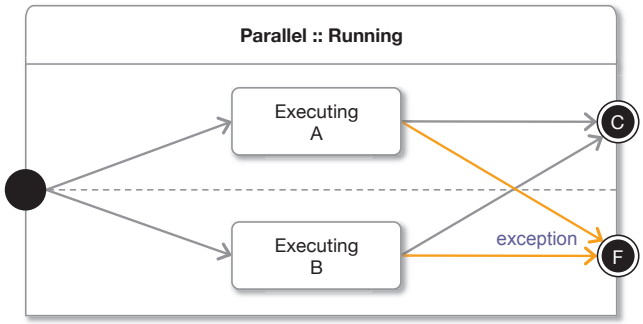

Figure 4.18 Behavior of a Parallel activity instance when running (UML state machine diagram)

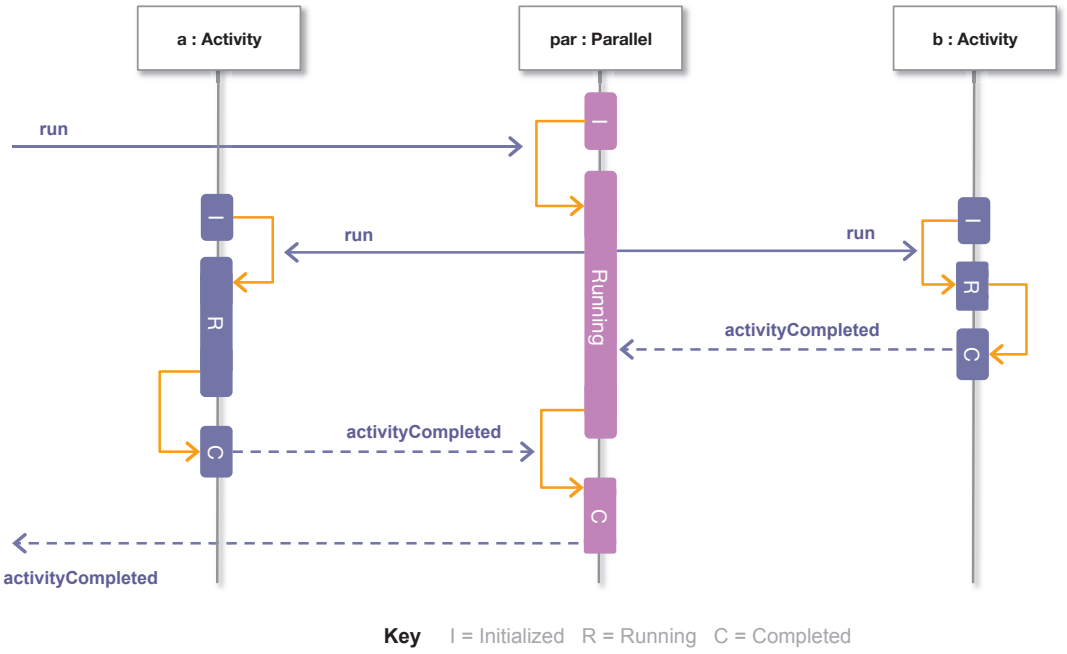

Figure 4.19 Example illustrating the execution of a Parallel activity instance (UML sequence diagram)

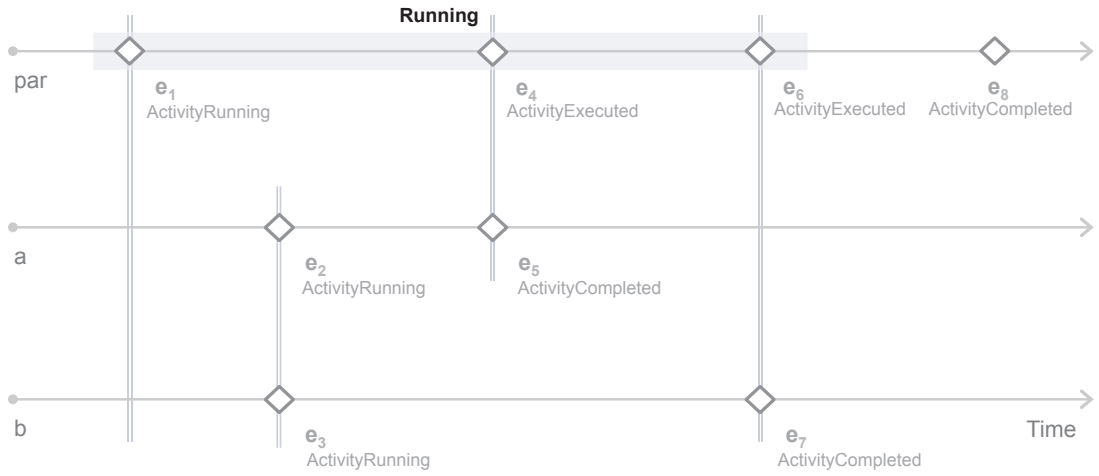

Figure 4.20 Event trace produced by a normal execution of a Parallel activity instance

4.1.4 Behavior of a Workflow

As shown in the diagram of **figure 4.21**, an instance of a **Workflow** activity type has a simple behavior at **execution-time**: once a workflow activity receives the *run* signal, it moves to the **executing control flow** sub-state and starts the execution of its associated control flow. Then, if the control-flow activity completes its execution without error, the workflow moves to the **completed** state and produces an event of type **ActivityExecuted** representing the completion of the control-flow. Otherwise the workflow moves to the **failed** state.

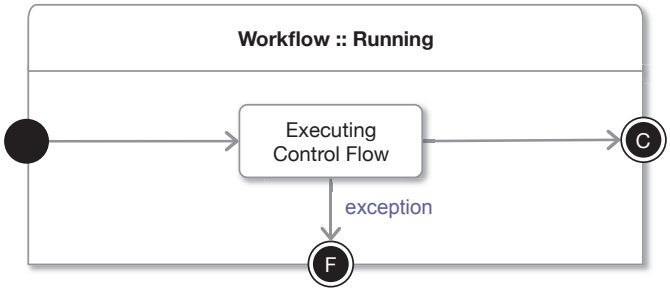

Figure 4.21 Behavior of a Workflow activity instance when running
(UML state machine diagram)

4.2 Executing an Active Policy

As you may recall from **chapter 2**, an *active policy* is an entity composed of a set of *rules* that can be associated to a set of *execution units* through its *scope* relationship. In this section we describe how instances of the **Active Policy** and **Rule** types *behave* at runtime. In particular we describe the relationship between the execution of an **Active Policy** instance and the set of **Execution Unit** instances composing its scope.

4.2.1 Active Policy Behavior

As shown in the diagram of **figure 4.22**, an instance of the **Active Policy** type passes through 4 states during its life cycle:

- **Deactivated**. During this state the policy rules are "dormant" and they cannot be triggered. A policy enters this state twice during its life cycle: (i) after the creation of the policy and (ii) once *all* the execution units composing the policy' scope are in one of these states: **completed**, **failed** or **paused**. When a policy enters this state, it sends a *deactivate* signal to each of its rules for deactivating them.
- **Activated**. During this state the policy rules are "waked up" and thus they can be triggered. A policy enters this state when *any* of the execution units composing its scope are **running** and/or **activated**. When a policy enters this state, it sends a *activate* signal to each of its rules for activating them.
- **Waiting**. A policy enters this state once *all* the execution units in its scope are in a final state (i.e., in the **completed** or **failed** state). Even if no more rules can be triggered at this point, some of the rules may still be running at this point. Thus the policy *waits* for their completion during this state.
- **Completed**. This state is entered once all the policy rules **completed** their execution. At this point we consider that the policy finishes its execution.

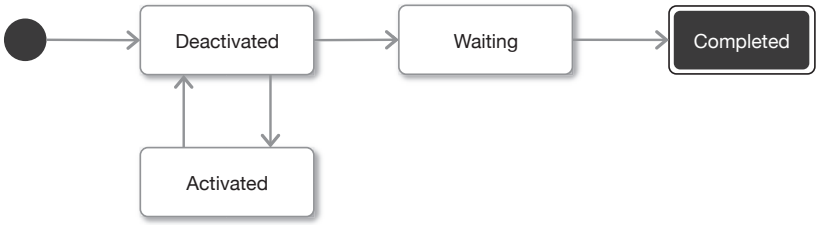

Figure 4.22 Behavior of an Active Policy instance when executed (UML state machine diagram)

The main responsibility of an active policy is to *activate* (*deactivate*) its associated *rules* based on the life cycle of the execution units in its scope. This is illustrated in the diagram of **figure 4.23**, where **ap** is an **Active Policy** instance that *applies to* activity **act**. As shown in the diagram, when **act** starts *running* it produces an event of type **ActivityRunning** that causes **ap** to move to the **activated** state. This causes **ap** to send an *activate* signal to each of its associated rules for activating them. In a similar way, when **act** *completes* its execution it produces an **ActivityCompleted** event that causes **ap** to move to the **deactivated** state and thus it causes **ap** to send a *deactivate* signal to all its rules.

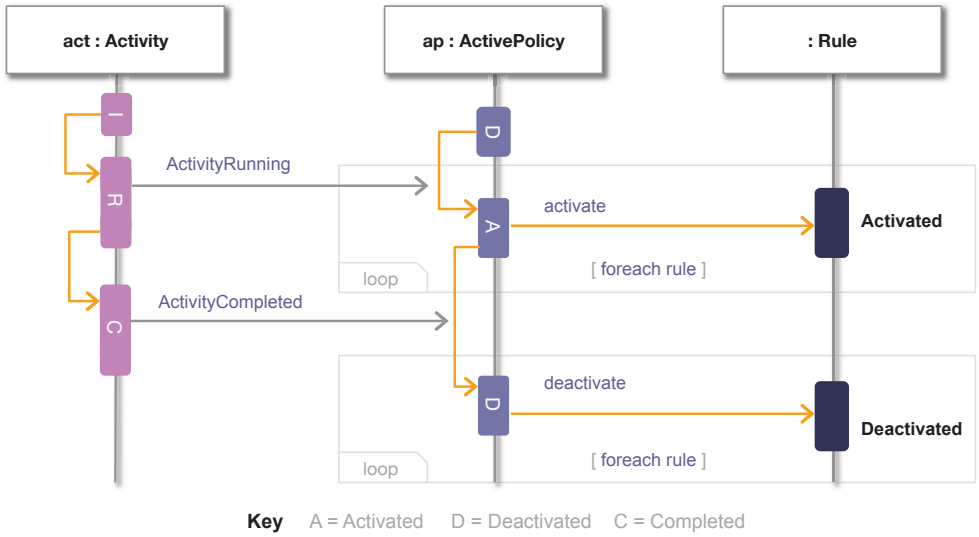

Figure 4.23 Activation and deactivation of an Active Policy instance (UML sequence diagram)

Figure 4.24 completes the previous example by showing the event trace produced by the execution of **ap**. As shown in the figure, an *active policy* produces an event instance every time it enters a state: when the policy enters the **activated** state it produces an event of type **APolicyActivated** (see event $\mathbf{e_4}$).

- When the policy enters the **deactivated** state it produces an event of type **APolicyDeactivated** (see event $\mathbf{e_4}$ and $\mathbf{e_6}$).
- When the policy enters the **waiting** state it produces an event of type **APolicyWaiting** (see event $\mathbf{e_7}$).
- When a policy enters the **completed** state it produces an event of type **APolicyCompleted** (see event $\mathbf{e_8}$).

Figure 4.24 shows the *trigger-ability interval* of the rules of **ap** that represents the interval during which the rules can be executed. This interval corresponds exactly to the *activation interval* of **ap**. In consequence, the trigger-ability interval of the rules of a policy *depends* on the events produced by the policy' scope (i.e., the events produced by the execution units belonging to the policy scope). For example, in the case of **ap**, the *trigger-ability interval* of $\mathbf{rule_i}$ is defined by 2 of the events produced by **act** (i.e., the scope of **ap**): $\mathbf{e_3}$ representing the beginning of the interval and $\mathbf{e_5}$ representing the end of the interval.

In general a *trigger-ability interval* associated to a policy is defined as follows:

- If a policy' scope is composed of a *single activity* **act**, the trigger-ability interval is delimited by the pairs of events **[ActivityRunning, ActivityFailed]** or **[ActivityRunning, ActivityCompleted]** produced by **act**.
- If a policy' scope is composed of a *single active policy* **ap**, the trigger-ability interval is delimited by the pair of events **[APolicyActivated, APolicyCompleted]** produced by **ap**.
- If a policy' scope is composed of several execution units, the trigger-ability interval is delimited by the pair of events [$\mathbf{e_x}$, $\mathbf{e_y}$], where $\mathbf{e_x}$ represents the first **ActivityRunning** /

APolicyActivated event produced by the policy' scope, and e_y represents the last **ActivityFailed** / **ActivityCompleted** / **APolicyCompleted** event produced by the scope.

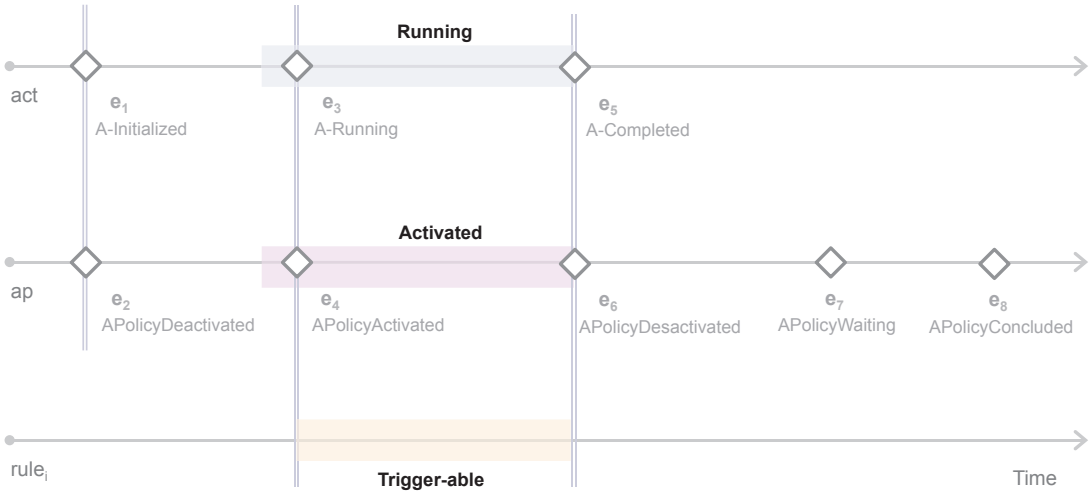

Figure 4.24 Event trace produced by an Active Policy instance

4.2.2 Rule Behavior

As shown in the diagram of figure 4.25, an instance of the **Rule** type passes through 4 states during its life cycle:

- **Deactivated**. During this state a rule does not consume triggering events (e.g., if a *triggering event* is notified during this state the rule simple "ignores" the event). A rule enters this state twice during its life cycle: (i) after the creation of the rule and (ii) once the policy where the rule belongs to become **deactivated** (cf. figure 4.22).
- **Trigger-able**. A rule enters this state when the policy where the rule *belongs is* **activated** (cf. figure 4.22). During this state a rule *waits* for the notification of a triggering event.
- **C-Evaluation**. This state is entered when a rule receives the notification of a triggering event. During this state a rule evaluates its condition Θ. If the condition is *false* the rule moves back to the **trigger-able** state. Otherwise the rule moves to the **a-execution** state once it receives the *execute* signal.
- **A-Execution**. During this state the rule executes its associated *action* (i.e., the workflow implementing the action). Once the action *completes* its execution, the rule returns to its **trigger-able** state.

There are criteria to be taken into consideration for executing a rule that are not described by the state machine. For instance:

- How does the execution of a rule is related to the notification of a triggering event?
- How is the execution of its action synchronized with the execution of an activity?

We adopt the classical behavior of rule considered in rule execution models as the one proposed in **[Coup96]** (see table 4.2). Some criteria can be parameterized like those concerning the execution of the action. Criteria concerning events are constant, meaning that trig-

gering events are handled as stated by these criteria. The following lines define criteria and values.

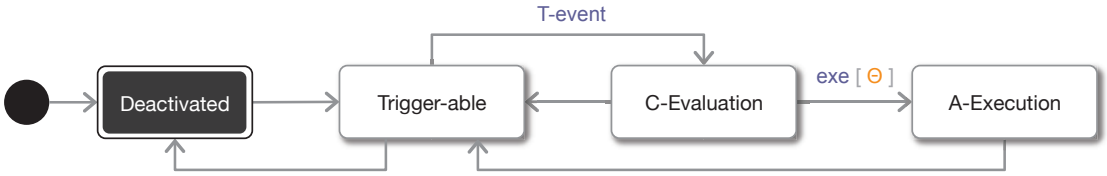

Figure 4.25 Behavior of a Rule instance when executed (UML state machine diagram)

	Criterion	Values
Event	**Processing Mode**	{ Instance-Oriented }
	Consumption Mode	{ Consumption }
	Scope of Consumption	{ Local }
Action	**Execution Mode**	{ Synchronous, Asynchronous }
	Execution Instant	{ Immediate, Deferred }
	Coupling Mode	{ Same WF Instance, New WF Instance }

Table 4.2 Parameters describing the behavior of a Rule

Event Processing Mode

The **event-processing mode** specifies the relationship between the notification of a t-event and the execution of a rule: **instance-oriented**. When a rule consumes events under the *instance-oriented* strategy, the rule is triggered every time a t-event is notified. **Figure 4.26** shows an example of a rule with *instance-oriented event consumption*. In the example the execution of the activity **Sequence (A, B, C)** produces the events e_1, e_2 and e_3, which triggered the rule. When the rule receives e_1, the rule becomes triggered for the first time. Since the rule is still processing e_1 at the reception of e_2 and e_3, the rule puts e_2 and e_3 into a queue called Q. Once the rule ends processing e_1, the rule gets triggered a second time when getting e_2 from Q. Finally, when the rule ends processing e_2, the rule gets triggered a third time when getting e_3 from Q.

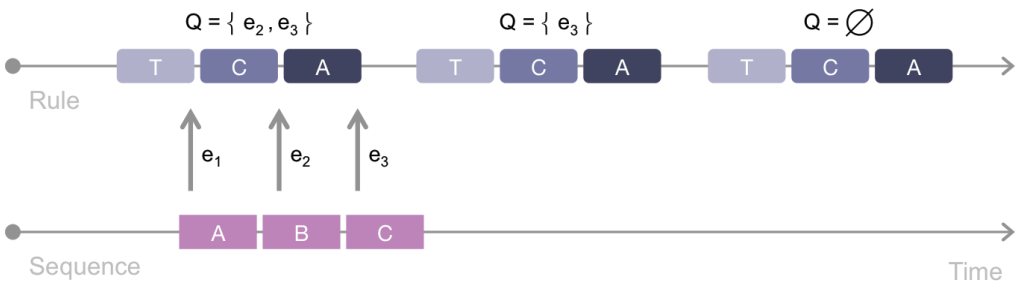

Figure 4.26 Example of a rule with instance-oriented semantics

Event Consumption Mode

The **event-consumption mode** specifies when a rule can be evaluated and A-executed with respect to the triggering policy. If a rule is immediate, it is evaluated immediately after it has been triggered. When a rule *consumes* the event, it is considered only for one execution of the rule and then it is not preserved for triggering other rules.

Transitions from states triggered and evaluated to states evaluable and A-executable respectively, depend precisely on the following parameters: execution mode, execution instant, and coupling mode.

Action Execution Mode

The **action-execution mode** specifies the relationship between the execution of the action of a rule and the entity triggering its policy. A *synchronous* rule pauses the execution of the activity associated before executing the action. Once the action terminates, the rule resumes the execution of the activity. When a rule is asynchronous, the execution of the action and the activity are done in parallel.

Figure 4.27 illustrates the execution of a synchronous rule. In the example the activity **Invoke Activity** produces the event e_1, which triggers the rule. Since the rule is *synchronous* the activity is *paused* during the processing of e_1. Once the rule ends the execution of its action, the activity *resumes* its execution.

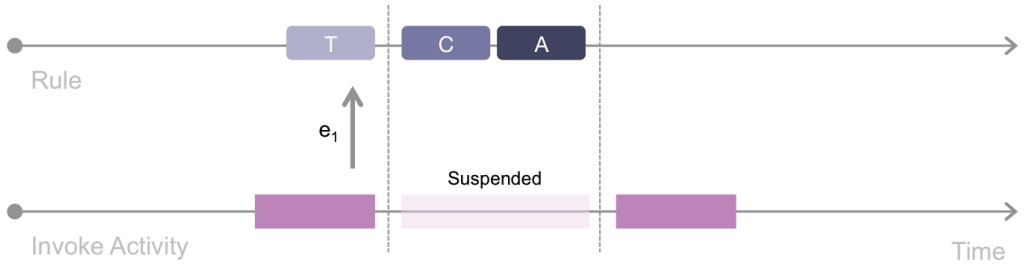

Figure 4.27 Synchronous rule execution example

Figure 4.28 illustrates the execution of an asynchronous rule. In the example the rule receives the notification of e_1 and the rule starts its execution. Note that in this example the activity continuous its execution in parallel. For example, assume that the activity **UpdateStatus** signals a message containing publicity about a concert of the group that is currently playing, when the activity is initialized. The action can be done in parallel without any implications on the atomic management of the status of a user in her different social networks.

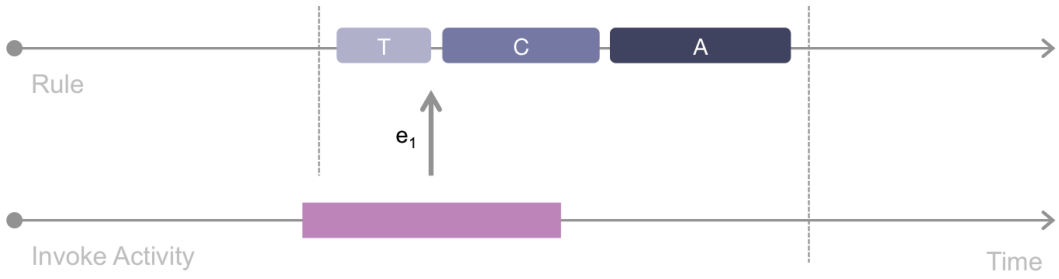

Figure 4.28 Asynchronous rule execution example

Execution Instant

A *trigger-able* rule becomes triggered as soon as a triggering event is notified. However, a triggered rule is not necessarily evaluated immediately. Similarly, an evaluated rule is not necessarily A-executed immediately.

- **Immediate**: The condition (or action) is evaluated instantly interrupting the execution of the current activity.
- **Deferred**: The condition (action) is evaluated (executed) within the same process but not at the earliest opportunity.

Coupling Mode

The evaluation of the rule can be done within the execution of the **same workflow** where it is triggered or in a **new workflow instance**. The new process can be causally independent, i.e. the rule is processed completely asynchronously with the triggering workflow, or causally dependent, where the processing of the rule is undertaken equally in a separate process but not before the completion of the triggering workflow.

4.2.3 Executing Several Rules

During the execution of an active policy based workflow it is possible that multiple rules become *triggered* at the same time. Therefore, rules have to be scheduled based on a local execution plan. The diagram of **figure 4.29** illustrates the general process used for scheduling the execution of several rules. The diagram is interpreted as follows: at certain point in time no rules are triggered (see state S_0). Then, once an event is notified, the scheduling process determines (i) whether the event *activate/deactivate* a policy and thus the rules of the policy (see **AP* Management** state) and (ii) whether the event is a *triggering event* of one or more of the activated rules (see **R* Triggering** state). If the event triggered more than one rule, the rules are ordered based on some criteria (see **R* Ordering** state). After that the process executes each of the rule actions one following the established order (**A* Dispatching**).

In this work we consider that triggered rules can be ordered based on the criteria shown in **table 4.3**:

- **Priorities**. Rules are ordered based on the priorities specified when defining the policies (i.e., using the priorities among rules and the priorities among policies).
- **FIFO**. Rules are ordered according to the order in which they were triggered.
- **Parallel**. Rules can be executed simultaneously (i.e., there is no order dependency among the execution of the rules).

- **Mixed**. Rules are ordered using Priorities/FIFO/Parallel criteria (e.g., rules can be ordered first by priorities and then all the rules having the same priority can be executed in parallel).

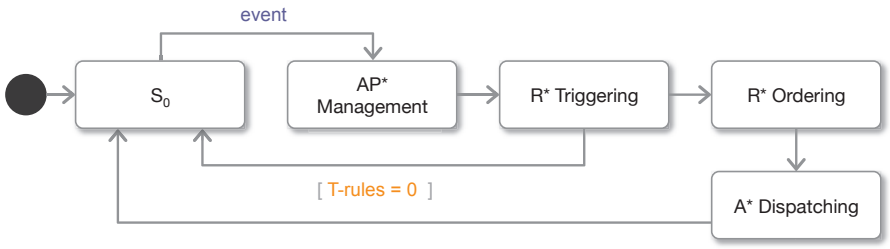

Figure 4.29 Scheduling the execution of several rules (UML state machine)

	Dimension	Valid Values
AP Rules	Scheduling Mode	{ Priority Based, FIFO, Parallel, Mixed}
Rules of different APs	Scheduling Mode	{ Priority Based }

Table 4.3 Order criteria for scheduling several rules

Note that it is also possible that the execution of a rule triggers a second rule, which in turns triggers a third rule and so on (i.e., the execution of the action workflow associated to a rule may produce a *cascade* of triggering events). In this scenario the scheduling process determines the order among the rules based on the following strategies **[Coup96]**:

- **Depth-first** execution. Prioritizes the execution of newly triggered rules.
- **Execution cycles**. Prioritizes the execution of rules based on the *cycle* where they were triggered.

As an example consider the *triggering event tree* shown in **figure 4.30**. The figure is interpreted as follows: $\mathbf{e_1}$ represents an event that triggers rules $\mathbf{R_1}$ and $\mathbf{R_2}$; $\mathbf{e_2}$ represents an event produced by $\mathbf{R_1}$ that triggers rules $\mathbf{R_{1a}}$, and $\mathbf{R_{2b}}$; $\mathbf{e_4}$ represents an event produced by $\mathbf{R_2}$ that triggers rule and $\mathbf{R_{2a}}$. When the scheduler process follows the **depth-first** strategy, the scheduler dispatches the rules as follows:

$$\mathbf{R_1 \rightarrow R_{1a} \rightarrow R_{2b} \rightarrow R_2 \rightarrow R_{2a}}$$

In contrast, when the scheduler follows the **execution cycle** strategy, the scheduler dispatches the rules as follows:

$$\mathbf{R_1 \rightarrow R_2} \text{ (cycle 1)}$$

$$\mathbf{R_{1b} \rightarrow R_{1a} \rightarrow R_{1b} \rightarrow R_{2a}} \text{ (cycle 2)}$$

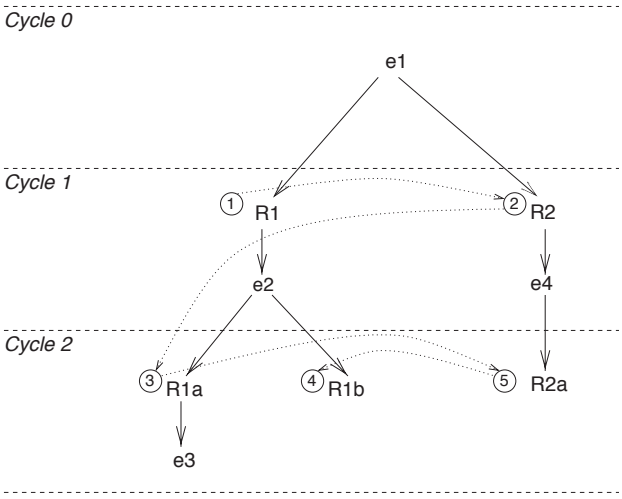

Figure 4.30 Example illustrating the scheduling of rules triggered by cascade triggering events

4.3 Reinforcing Non-Functional Properties at Runtime

From a general point of view a workflow can be seen as a *process* that executes *read/write* operations (i.e., activities) into a *memory space*. For example, **figure 4.31** illustrates a workflow that operates on its own memory space (i.e., activities **A**, **B** and **C** read/write workflow' variables v_1, v_2 and v_3). In this work we consider that workflows can operate over two kinds of memory spaces: (i) the space composed of the set of variables belonging to a workflow and (ii) the space composed of the *data structures* representing the activities of a workflow and their order relationship (i.e., the workflow activity tree). Thus, in order to ensure a non-functional property using active policies, we have to determine what kind of relationship exist between (i) the memory space of a workflow implementing a services' coordination and (ii) the operations executed by the action workflows implementing a non-functional property (i.e., whether they read and/or write the workflow memory space). We identify 3 kinds of relationships among *workflows* and *action workflows*: **non-intrusive**, **intrusive** and **control-flow and memory intrusive**.

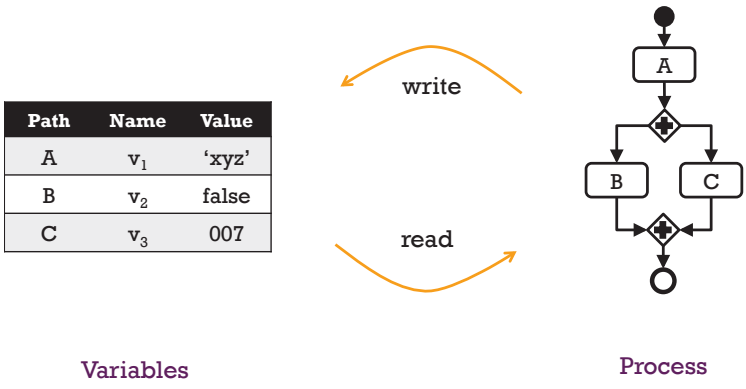

Figure 4.31 Workflow operating over its own memory space

4.3.1 Non-Intrusive

As shown in **figure 4.32**, in the **non-intrusive** relationship between a workflow and an action workflow, and action workflow only reads the variables of a workflow for implementing a non-functional property. For instance, let us consider that a developer wants to add *logging* capabilities to the *Status Updater* workflow (i.e., the developer wants to store into a log the values of the variables of the workflow at different moments of the workflow execution).

As shown in **figure 4.33**, in the **Status Updater FR** workflow, assume that the activity **Update Facebook** has an associated state management NFP for logging its state in a log. The activity starts its execution: it receives the value **'xyz'** associated to the variable **new_Status,** and assigns the value **'abc'** to the variable **old_status.** The Logging NFP workflow starts its execution reading these values from the execution state of the activity **Update Facebook** and stores them in the log. Meanwhile, the activity **Update Facebook** continues its execution.

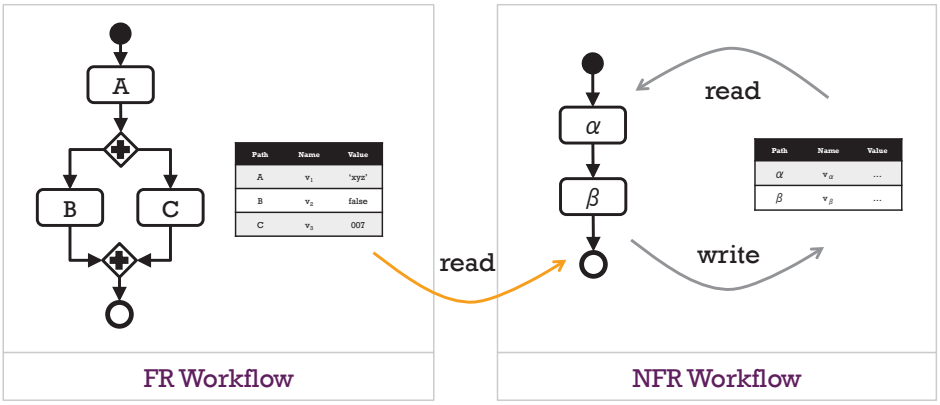

Figure 4.32 Non-intrusive FR and NFR workflows synchronization

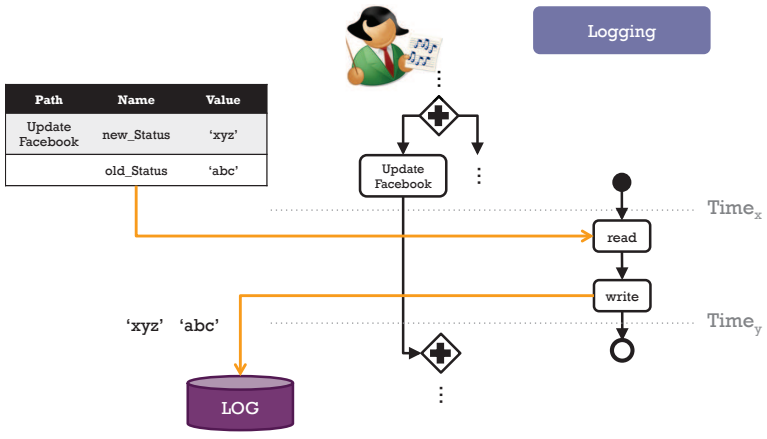

Figure 4.33 Logging: example of non-intrusive FR and NFR workflows synchronization

4.3.2 Memory Intrusive

In this case the FR workflow shares its execution state with the NFR workflow that reads and writes on it when executed. The NFR, instead, does not share its execution state with

the FR workflow (see **figure 4.34**). The synchronization of both workflows is done giving access the their respective execution state.

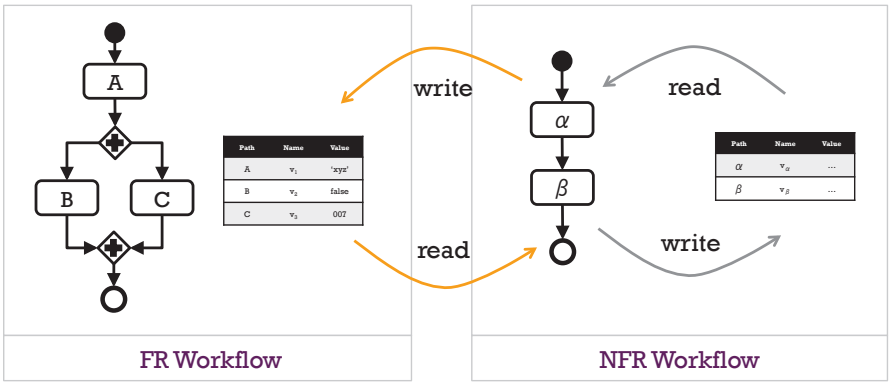

Figure 4.34 Memory intrusive NF and NFR workflows synchronization

In the **Status Updater** example, the activity **Update Facebook** has an associated authentication NFP, where the activity first sends a new request. Then, it prepares an URI with the variable **new_Status** as output parameter. Before executing the request, the **Authentication** workflow that implements the Open Authentication Protocol, reads the user login and the password from the execution state of the activity **Update Facebook** shared with the **workflow** Authentication (see **figure 4.35**). Then, it produces a token stored in the variable **new_Status** of the state of the activity **Update Facebook**. The activity **Update Facebook** can call the service Facebook requesting a status update (see **req.sendUpdate()** in **figure 4.35**).

Type	Name	Value
Request	req	new Request()
	uri	www.facebook.com/me/{**new_Status**}
Response	resp	req.sendUpdate()

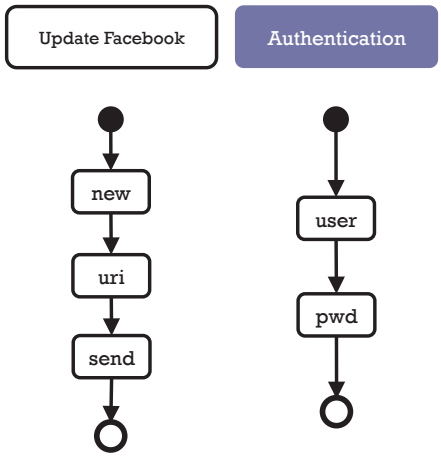

Figure 4.35 Authentication: example of memory intrusive FR and NFR workflow synchronization

4.3.3 Control Flow and Memory Intrusive

In this case the workflows FR and NFR are synchronized by sharing their execution states in shared memory spaces and by eventually modifying the NFR control flow. Both workflows read and write on the shared memory spaces.

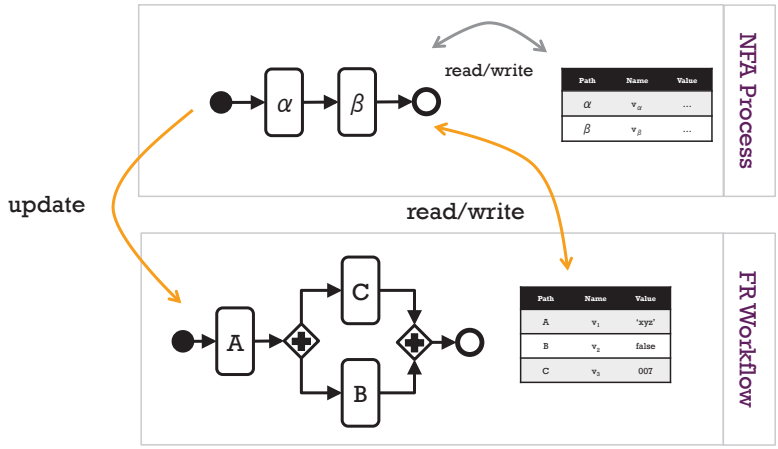

Figure 4.36 Control flow and memory intrusive FR and NFR workflows synchronization

For example, assume now that the service **Facebook** updates its authentication protocol and implements now an HTTP protocol, where a login and password are required. The **Updater Status FR** and NFR workflows are now synchronized as follows. The activity Update Facebook of the FR workflow starts creating a new request, then, provides an URI with the variable **status**. Doing this the control flow is now deviated for executing the authentication protocol by the workflow **Authentication**. It executes the activity user that requests the user **login** (see **figure 4.37** variable **username** with value **"john doe"**), written on the execution state, and then the activity **pwd** for requesting the password that is also written on the execution state (see **figure 4.37** variable **password** with value **"secret"**). Once executed, the control of the activity **Update Facebook** is resumed, and it executes the last activity **send**.

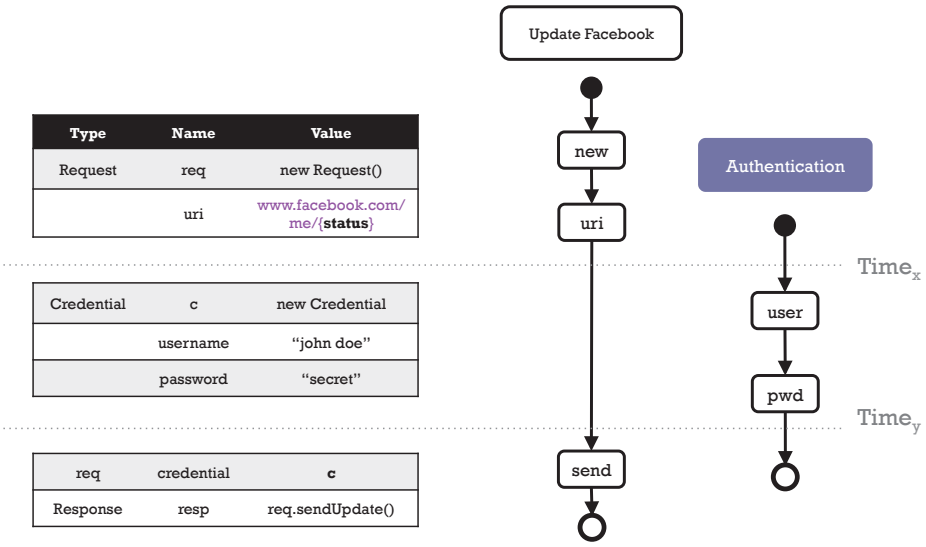

Figure 4.37 Authentication: example of control flow and memory intrusive FR and NFR workflows synchronization

Adding exception handling to the **Status Updater** workflow is also an example of memory and control flow intrusive NFP, where the execution of the FR workflow must be observed. Then, the NFP must modify the control flow for catching the exception produced and up-

dating the control flow for recovering from the exception: signaling the exception, re-executing the activity that produced the exception, rolling back the execution to a state previous to the production of the exception.

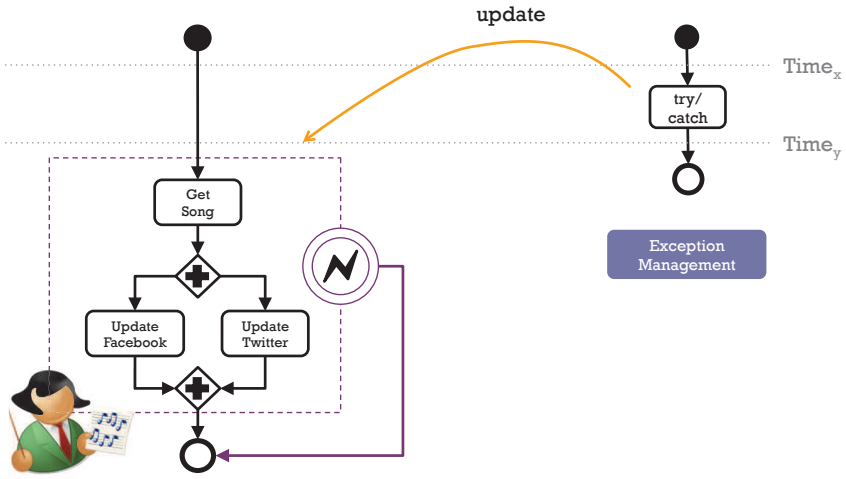

Figure 4.38 Exception handling: example of control flow and memory intrusive FR and NFR workflows

4.4 Conclusions

This chapter introduced the execution of the entities of the AP Model: **Activity**, **A-Policy** and **Rule**. It described the execution simple and complex activities specifying their execution states and the conditions in which they pass from one state to another.

Rules are executed within the execution of active policies and policies are executed within the execution of a workflow. Thus the chapter showed how to synchronize the execution of these entities among each other. It defined all the aspects that should be considered for deciding when to trigger an AP with respect to the notification of an event, and when to activate its associated rules. It also described when rules are triggered with respect to the notification of their triggering event, when to evaluate the condition and when to execute the action. The action can modify the workflow execution state or its control flow. Therefore the chapter proposed patterns showing how NFP can be reinforced.

A WF Engine and an AP Engine execute workflows and active policies, respectively. The WF Engine manages the *execution lifecycle* of a workflow instance (i.e., the lifecycle of a composite activity). The AP engine manages the lifecycle of a policy and it also plans the execution of several policies and policy' rules. Both engines interact synchronously and asynchronously. **Chapter 6** presents the system that we proposed for executing workflow and policies.

5 Reliable Services' Coordinations

We define a *reliable services' coordination* as a coordination that tolerates *failures* at runtime. This chapter introduces our approach for adding *reliability* to services' coordinations using active policies. It describes how we specialize the **Active Policy** type for tolerating *semantic* and *system* failures. We tolerate semantic failures by adding *atomicity* (i.e., the principle of "*all or nothing*") to the execution of a coordination using the **Activity Exception Management** and **Atomicity** policy types. We tolerate system failures by *persisting* and *recovering* the execution state of a coordination using the **Activity State Management** and **Persistency** policy type. When these policies are associated to a workflow implementing a services' coordination, they form a *policy tree* that ensures the reliability of the workflow execution.

The chapter is organized as follows. **Section 5.1** explains the basis of our approach for adding reliability to a services' coordination. **Section 5.2** presents our approach for adding an *atomic behavior* to a coordination. In particular, this section describes the structure and use of the **Activity Exception Management** and **Atomicity** policy types. **Section 5.3** presents our approach for adding *persistency* to a coordination. In particular, this section describes the structure and use of the **Activity State Management** and **Persistency** policy types. Finally, **section 5.4** concludes the chapter.

5.1 Reliability

In this work we consider a *reliable services' coordination* as a coordination that tolerates *failures* at runtime. Failures can be of two kinds: *semantic* and *system* failures. For instance, dividing a number by zero or passing a value of a type *X* when a value of type *Y* is expected are examples of *semantic failures*. An application crash produced by a stack overflow is an example of a *system failure*.

We address *semantic failures* by adding *atomicity* to the execution of a services' coordination (i.e. by providing an atomic behavior to the workflow implementing the coordination). This strategy was first used in the database domain where the atomic behavior associates the principle of *"all or nothing"* to database operations (i.e., either all operations *commit* or *fail*, and if committed, they can be rolled back). In the context of workflows, activities represent the operations that must be executed atomically and thus, we implement atomicity by specifying:

- How to treat failures on a *single activity*. For instance, in the *Status Updater* workflow the information retrieved by the *Get Song* activity is vital for the execution of

the *Update Facebook* and *Update Twitter* activities (cf. **section 3.1**). Thus, we specify that the *Get Song* activity must be retried several times in case of failures before stopping the workflow execution.

- How to provide an *atomic behavior* to a *set of activities*. For instance, we specify that the execution of the activities *Update Facebook* and *Update Twitter* depend on each other: either both activities update a user social networks or none of them. This ensures that both social networks contain the same information (i.e., this ensures data consistency).

We address *system failures* by providing *persistency* to the execution of a services' coordination. This ensures that the coordination survives despite system failures. For this we keep track the *execution history* of the workflow implementing the coordination (i.e., we track the states of the activities composing the workflow and the workflow itself at different moments of the workflow execution). Key elements to consider when keeping an execution history are (i) *where* to store the execution history (e.g., cache or disk) and (ii) *how* to do the recovery process using it. Thus we implement persistency by specifying:

- How to treat persistency and recovery of *single activity*. For instance, if an activity can be retried, then the recovery process can re-execute the activity for bringing back the workflow to the state previous to the failure.
- How to handle the execution state of *several activities*. For instance, if a set of activities must be executed in an atomic way, we consider this dependency during the recovery process.

Our approach for providing *atomicity* and *persistency* to a services' coordination is based on the work of **[Port06b]** that implements the atomic and persistent behaviors (i) assuming a specific activity execution model, (ii) classifying workflow activities based on their *exception* and *state management properties* and (ii) associating *execution strategies* to the workflow activities based on their types.

The *activity execution model* assumed by the author is shown in **figure 5.1**. The diagram is interpreted as follows: after being created, an activity (i) *prepares* its input and local variables, (ii) *starts* the call to its associated service' operation, (iii) *terminates* the call either (with error or success) after receiving the operation result and finally (iv) commits. Note that this behavior is similar to the one found in database transactions: either the execution of an activity *commits* or *fails*.

Figure 5.1 Activity execution model assumed in [Port06b] (UML state machine diagram)

The *exception* and state *management properties* are properties inherent to all activities that can be used for guiding the implementation of the *atomic* and *persistent* behaviors. The following list enumerates the exception management properties identified by the author:

- **Compensate-able** – specifies whether an activity can be undone by compensating it with the execution of another activity. For instance, let us assume that it is impossible to delete the status of a Facebook user once it has been updated. In order to compensate the execution of the activity *Update Facebook*, we have to execute another activity that updates the user status to a previews value (i.e., the activity is compensated semantically).
- **Side-effects** – specifies whether undoing an activity generates side-effects or not. For instance, let us assume that calling a Facebook operation has a cost. Compensating the execution of the *Update Facebook* activity can lead to an extra cost due to the fact that the compensation requires calling the service a second time with a new status.
- **Retry-able** – specifies whether an activity can be executed several times. For instance, it is possible to execute several times the *Get Song* activity in order to obtain the information about the last played song.

Based on these properties activities are classified as follows (see **table 5.1**):

- **Critical** – represents an activity that cannot be compensated or retried (i.e., represents an activity that is vital for a workflow execution).
- **Non-Vital** – represents an activity that can be retried and/or compensated without producing side effects.
- **Undoable** – represents an activity that can be compensated without producing side effects. An undoable activity may be retried.
- **Compensate-able** – represents an activity that can be compensated causing side effects. A compensate-able activity may be retried.

For instance, the *Get Song* activity is an example of *non-vital* activity since it can be retried several times in order to obtain the information about the last played song and it has no need to be compensated. The *Update Twitter* and *Update Facebook* activities are examples of *compensate-able* activities since they can be semantically undone by executing activity that update the user status with a previews value.

	Critical	Non Vital	Undoable	Compensate-able
compensate-able	No	No / Yes	Yes	Yes
side-effects	-	No	No	Yes
retry-able	No	No / Yes	No / Yes	No / Yes

Table 5.1 Classification of activities based on their Exception Management Properties

The following list enumerates the activity state management properties identified in **[Port06b]**:

- **isStateAccesible** – specifies whether the memory space manipulated by an activity can be accessed by other activities or not (e.g., the **Update Facebook** activity is *state acces-*

sible because it is possible to retrieve and modify the status of a Facebook user using other activities).

- **isIdempotent** – determines whether executing the same activity more than once produces the same result (e.g., the **GetMood** activity is *idempotent* because the same song title will compute the same mood).
- **presumedFinalState** – specifies the *activity final state* that has to be assumed in the presence of a system failure (e.g., *committed*, *failed* or *unknown*).

Based on these properties activities are classified as follows (see **table 5.2**):

- **Non-Reliable** – represents an activity (i) whose memory space is inaccessible to other activities, (ii) when executed does not produces the same result and (iii) its final state is unknown in advance.
- **Predictable** – represents an activity that has a predicable behavior (i.e., its final state is known in advance).
- **Idempotent** – represents an activity (i) whose memory space is inaccessible to other activities and (ii) which can be re-executed several times producing the same result.
- **Verifiable** – represents an activity whose memory space can be accessed by other activities. A verifiable activity may be idempotent and its final state may be known in advance.

For instance, the **Update Facebook** activity is an example of a *verifiable activity* since (i) the memory space modified by the activity (i.e., the user status) can be *accessed* by other activities, (ii) when executed more that once the activity updates a user status with the same information (i.e., the activity is *idempotent*) and (iii) nothing can be assumed about its *final state* (i.e., it is *unknown* whether the activity fails or commit constantly).

	Non Reliable	Predictable	Idempotent	Verifiable
isStateAccessible	No	No	No	Yes
isIdempotent	No	No	Yes	No / Yes
presumedFinalState	No	Yes	No / Yes	No / Yes

Table 5.2 Classification of activities based on their State Management Properties

Based on the *exception* and *state management properties* and their associated activity types, we have defined a set of active policy types that implement the *execution strategies* identified in **[Port06b]** for providing *atomicity* and *persistency* to the execution of workflows. These policy types are described in the following sections.

5.2 Active Policies for Atomic Behavior

We use the **Activity Exception Management** and **Atomicity** policy types for providing atomicity to the execution of a workflow. The former defines rules for dealing with activity failures. The latter defines rules for compensating a set of committed activities (i.e., for undoing the

work done by the activities). **Figure 5.2** shows the relationship between the **Activity Exception Management** and **Atomicity** policy types.

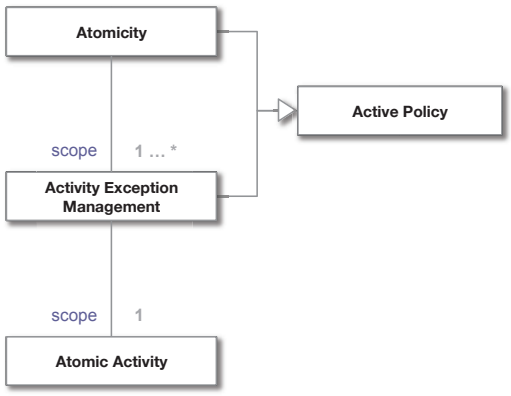

Figure 5.2 Activity Exception Management and Atomicity policy types relationship (UML class diagram)

As shown in the figure, the **Activity Exception Management** policy type is a policy that can be associated only to *atomic activities* (i.e., the policy is an *activity policy*). In contrast, the **Atomicity** policy type is a policy type that can be associated to any number of *exception management* policies (i.e., the policy is a *policy of policies*). When an *atomicity policy* is associated to a set of *exception management policies* they form a *policy tree* where the root (i.e., the atomicity policy) ensures the atomic behavior of the tree leafs (i.e., the atomic activities). This is illustrated in **figure 5.3** where the *atomicity policy* $\mathbf{ap_3}$ ensures the atomic behavior of the *Update Facebook* and *Update Twitter* activities through the *exception management* policies $\mathbf{ap_1}$ and $\mathbf{ap_2}$ respectively.

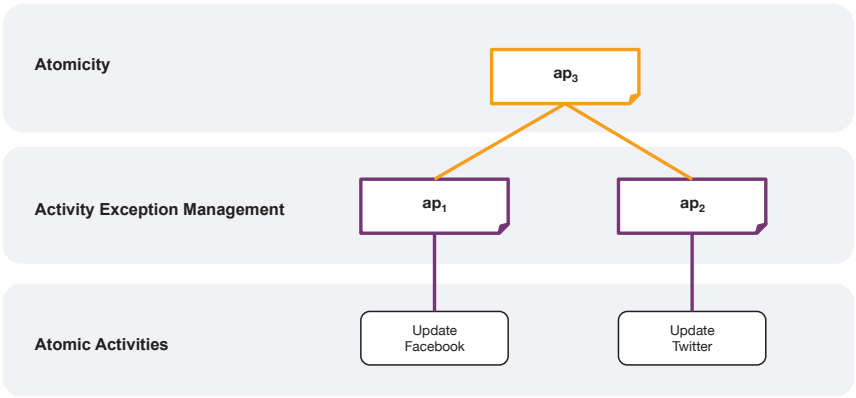

Figure 5.3 Example of an Atomicity policy tree

When **Activity Exception Management** and **Atomicity** policies are associated to each other in a policy tree, they interact among them by producing events. **Figure 5.4** illustrates the general interaction among instances of these policy types. The diagram is interpreted as follows: when an activity **act** belonging to the scope of an *exception management policy* $\mathbf{ap_{ex}}$ fails, **act** produces an event **ActivityFailed** that triggers one of the rules of $\mathbf{ap_{ex}}$ (see **step 1**). At this point $\mathbf{ap_{ex}}$ tries to handle the failure locally (e.g., by retrying the activity). If $\mathbf{ap_{ex}}$ does not arrive to

handle the failure, **ap$_{ex}$** produces an event **CustomScopeFailure** that represents the policy scope failure (see **step 2**).

If **ap$_{ex}$** belongs to the scope of an *atomicity policy* **ap$_{atom}$**, the notification of the event **CustomScopeFailure** triggers the rule of **ap$_{atom}$** in charge of ensuring the atomic behavior. For this, the rule produces an event **CompensationRequest** that triggers the compensation of the activities associated to all the *exception management policies* belonging to the scope of **ap$_{atom}$**. Let us assume that only **ap$_{ex}$** belongs to the scope of **ap$_{atom}$**. Thus, the notification of the event **CompensationRequest** triggers a rule in **ap$_{ex}$** that compensates **act** (see **step 3**). If **act** is compensated then the atomic behavior is respected. If for any reason the activity cannot be compensated (e.g., the activity is not compensate-able), **ap$_{ex}$** produces an event **CustomActivityPolicyFailure** that represents the fact that the activity compensation failed (see **step 4**). In turn this event triggers a rule in **ap$_{atom}$** that notifies the environment about the failure of the compensation procedure (see **step 5**).

In what follows we describe the structure of the **Activity Exception Management** and **Atomicity** policy types.

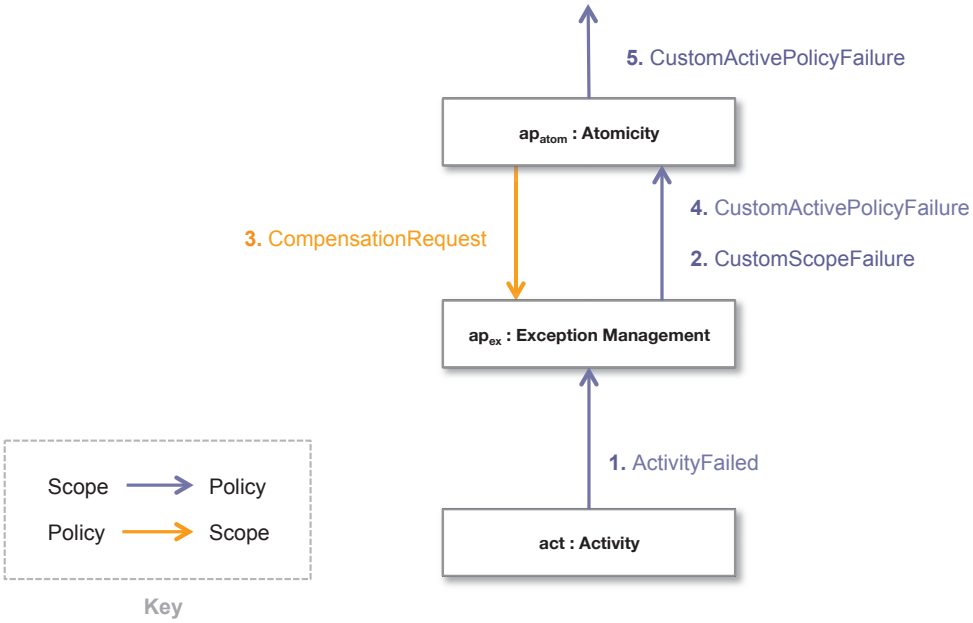

Figure 5.4 Activity Exception Management and Atomicity policies general interaction (UML interaction diagram)

5.2.1 Activity Exception Management Active Policy

We use the **Activity Exception Management** policy type for handling (i) activity failures and (ii) compensation of committed activities. As shown in **figure 5.5**, we have specialized the **Activity Exception Management** type in the **Critical**, **Non-Vital**, **Undoable** and **Compensate-able** subtypes in order to encapsulate the logic responsible of handling failures and compensation of *critical*, *non-vital*, *undoable* and *compensate-able* activities (respectively). First, we will describe the structure of the **Activity Exception Management** policy type. Then, we will describe the structure of each of its subtypes.

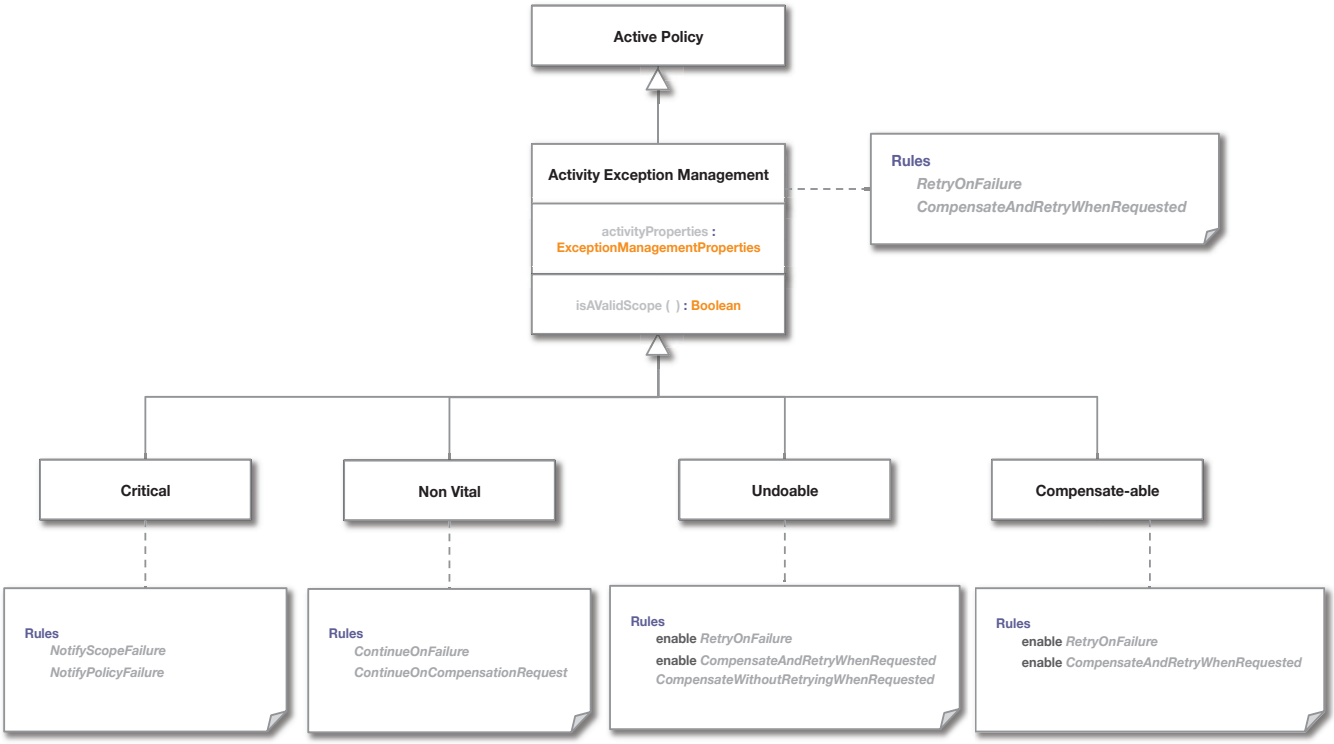

Figure 5.5 Exception Management Active Policy type (UML class diagram)

Activity Exception Management Policy

The definition of the **Activity Exception Management** policy is shown in **listing 5.1**:

- **Line 1** defines the *scope* of the policy. Note that the scope is composed of a single execution unit called *activity*. Also note that the type of *activity* is **Atomic Activity** (i.e., an instance of an **Activity Exception Management** can be applied to any *atomic activity*).
- **Line 2** defines the variable *activityProperties* of type **ExceptionManagementProperty**. This variable contains the *exception management properties* of *activity* (i.e., the variable contains the properties of the activity in the policy scope).
- **Lines 3–8** define the policy constructor. Note that the constructor receives as input the *exception management properties* of the activity that has to be associated to the new policy instance. If these properties describe a valid scope, the constructor creates a new policy instance applying to *activity*. Otherwise, the constructor throws an exception.
- **Line 9** defines the method *isAValidScope*. This method determines whether a policy instance (based on its properties) can be applied to a specific activity. For instance, if the policy is expecting a *critical activity*, and a developer applies the policy to a *non-vital activity*, the method return *false*. Note that this method is used inside the policy constructor.
- **Lines 10–13** define the rule *RetryOnFailure*. When *activity* fails (see **step 1** in **figure 5.4**), this rule tries to re-execute the activity for handling the failure. The rule works as

follows: on the notification of an event **ActivityFailed**, if *activity* is *retry-able* and the event was *produced by activity*, the rule will execute (synchronously) an instance of the **Retry** workflow. Intuitively, a **Retry** instance proceeds as follows: first it pauses the execution of the *main workflow* (i.e., the workflow executing *activity*). Then it re-executes the failed activity (i.e., *activity*). Finally, it returns the control to the main workflow (who does not notice the failure) for continuing its execution. If for any reason the **Retry** instance does not arrive to re-execute the activity, it produces an event **CustomScopeFailure** representing the activity failure (see **step 2** in **figure 5.4**). Note that **Retry** receives as input (i) the *id* of the failed activity (i.e., the id of the event producer) and (ii) the *maximum number of times* that the activity can be retried.

- **Lines 14–17** define the rule *CompensateAndRetryWhenRequested*. When an atomic policy begins the compensation process (see **step 3** in **figure 5.4**), this rule compensates and re-executes *activity* (i.e., the activity in the scope of the policy). The rule works as follows: on the notification of an event **CompensationRequest**, if *activity* is *retry-able* and *compensate-able*, and the event was *produced by* an atomicity policy applying to the **Activity Exception Management** policy (i.e., produced by the *ancestor* of the policy in a *policy tree*), the rule will execute (synchronously) an instance of the **CompensateAndRetry** workflow. Intuitively, a **CompensateAndRetry** instance proceeds as follows: first it pauses the execution of the *main workflow*. Then, it compensates and re-executes *activity*, which causes *activity* to re-enter the *committed* state. Finally, it resumes the main workflow execution. If for any reason the workflow does not arrive to retry or compensate *activity*, it produces an event **CustomActivePolicyFailure** that represents the failure of the compensation process (see **step 4** in **figure 5.4**). Note that **CompensateAndRetry** also receives as input (i) the *id* of the event producer and (ii) the *maximum number of times* that the activity can be retried.

Note that the **Activity Exception Management** policy type is an *abstract* type (see **line 1** in **listing 5.1**). This implies that the policy type cannot be instantiated and thus we only use it as a *base type* (i.e., the **Activity Exception Management** policy type defines the scope, variables, methods and rules inherited by the **Critical**, **Non-Vital**, **Undoable** and **Compensate-able** subtypes). Finally note that the **Activity Exception Management** policy type is based on the types shown in **listing 5.2**. The **ExceptionManagementProperties** type represents the *exception management properties* of an activity. The **CompensationRequest**, **CustomActivePolicyFailure** and **CustomScopeFailure** represent the events produced during the implementation of the atomic behavior. Since these event types are not part of the AP Model event types (cf. **figure 3.23**), we define them as specializations of the type **Event**.

```
1  abstract policy class ActivityExceptionManagement |[ AtomicActivity activity ]| : ActivePolicy {
2      ExceptionManagementProperties activityProperties ;
3      ActivityExceptionManagement ( ExceptionManagementProperties activityProperties ) {
4        if ( isAValidScope ( activityProperties ) ) {
5            this.activityProperties = activityProperties ;
6        } else
7            throw Exception ;
8      }
9      Boolean isAValidScope ( ExceptionManagementProperties activityProperties ) { return false; }
10     sync rule RetryOnFailure
11       on ActivityFailed
12       if event.producedBy( scope ) && this.activityProperties.retry-able
13       do Retry ( event.producer, maxNumberOfRetries )
14     sync rule CompensateAndRetryWhenRequested
15       on CompensationRequest
16       if event.producedByAncestorAction( scope ) && this.activityProperties.compensate-able && this.activityProperties.retry-able
17       do CompensateAndRetry ( event.producer , activityProperties.maxNumberOfRetries )
18 }
```

Listing 5.1 Definition of the Activity Exception Management policy type

```
class ExceptionManagementProperties {

   Boolean  compensate-able ;
   Boolean  side-effects ;
   Boolean  retry-able ;
   Integer  maxNumberOfRetries ;

}

class CompensationRequest : Event { ... }

class CustomActivePolicyFailure : Event { ... }

class CustomScopeFailure : Event { ... }
```

Listing 5.2 Definition of the Exception Management Properties and custom Event types

Critical Active Policy

Since *critical activities* cannot be retried nor compensated (see **table 5.1**), we use the **Critical** policy type for notifying failures of this kind of activities. The definition of the **Critical** policy type is shown in **listing 5.3**:

- **Line 1** defines the **Critical** type as a specialization of the **Activity Exception Management** type.
- **Lines 2–8** redefine the method *isAValidScope* for ensuring that only *critical activities* are associated to a **Critical** policy instance. Recall that this method is inherited from the **Activity Exception Management** policy type.
- **Lines 9–12** define the rule *NotifyScopeFailure*. When *activity* fails (see **step 1** in **figure 5.4**), the rule notifies the failure to the execution environment. The rule works as follows: on the notification of an event **ActivityFailed**, if the event was *produced by activity*, the rule executes an instance of the **NotifyScopeFailure** workflow that produces the event **CustomScopeFailure** that represents the activity failure (see **step 2** in **fig-**

ure 5.4). Note that **NotifyScopeFailure** receives as input the *id* of the event producer (i.e., the id of the failed activity).

- **Lines 13–16** define the rule *NotifyPolicyFailure*. When an atomic policy begins the compensation process (see **step 3** in **figure 5.4**), this rule notifies that the policy cannot compensate or retry the policy scope. The rule works as follows: on the notification of an event **CompensationRequest**, if the event was *produced by* an atomicity policy applying to the **Critical** policy, the rule executes an instance of the **NotifyActivePolicyFailure** workflow that produces an event **CustomActivePolicyFailure** that represents the activity compensation failure (see **step 4** in **figure 5.4**). Note that the **NotifyActivePolicyFailure** receives as input the *id* of the event producer and a reference to the policy.

```
1  policy class Critical : ActivityExceptionManagement {
2      Boolean isAValidScope ( ExceptionManagementProperties activityProperties ) {
3        if ( activityProperties.compensate-able == false &&
4             activityProperties.retry-able == false )
5              return true;
6        else
7              return false;
8      }
9      sync rule NotifyScopeFailure
10       on ActivityFailed
11       if  event.producedBy( scope )
12       do NotifyScopeFailure ( event.producer )
13     sync rule NotifyPolicyFailure
14       on CompensationRequest
15       if  event.producedByAncestorAction( scope )
16       do NotifyActivePolicyFailure ( event.producer, this )
17 }
```

Listing 5.3 Definition of the Critical policy type

As an example consider the activity *Update Facebook* of the Status Updater workflow. Let us assume that this activity cannot be retried nor compensated. Therefore, the *Update Facebook* activity is an example of a *critical* activity. **Listing 5.4** illustrates the use the **Critical** policy type for handling failures of the *Update Facebook* activity. The expression is interpreted as follows:

- **Line 1** defines the variable *wf* that holds a reference to the *Status Updater* workflow. **Lines 2–7** create and assign an instance of type **ExceptionManagementProperties** to the variable *activityProperites*. This variable represents the exception management properties of the *Update Facebook* activity.
- **Line 8** creates and assigns an instance of the policy type **Critical** to the variable *ap*. Note that *ap* is applied to the activity *Update Facebook*. Also note that the policy constructor receives as input the variable *activityProperites*. Since this variable represents a critical activity, the constructor of **Critical** considers *Update Facebook* a valid scope and creates the new policy instance.

```
1   var wf = new StatusUpdater { ... }
2   var activityProperties = new ExceptionManagementProperties {
3           compensate-able = false,
4           side-effects = null,
5           retry-able = false,
6           maxNumberOfRetries = null
7   }
8   var ap = new Critical |[ wf.activities[ "UpdateFacebook" ] ]| ( activityProperties ) ;
```

Listing 5.4 Example of a Critical policy instance

Non-Vital Active Policy

We use the **Non-Vital** policy type for handling failures and compensating *non-vital activities*. The definition of this type is shown in **listing 5.5**:

- **Line 1** defines the **Non-Vital** type as a specialization of the **Activity Exception Management** type.
- **Lines 2–7** redefine the method *isAValidScope* for ensuring that only *non-vital activities* are associated to a **Non-Vital** policy instance (cf. **table 5.1**).
- **Lines 8–11** define the rule *ContinueOnFailure*. When *activity* fails (see **step 1** in **figure 5.4**), this rule forces the main workflow to continue its execution despite the failure. The rule works as follows: on the notification of an event **ActivityFailed**, if the event was *produced by activity*, the rule executes an instance of the workflow **Continue**. Intuitively, a **Continue** instance proceeds as follows: first it pauses the execution of the *main workflow*. Then, it forces the main workflow to ignore the failed activity and move to the next activity in the workflow' control flow. Finally, it resumes the main workflow for continuing its execution. Note that **Continue** receives as input the *id* of the failed activity.
- **Lines 12–15** define the rule *ContinueOnCompensationRequest*. When an atomic policy requests the compensation of *activity* (see **step 3** in **figure 5.4**), this rule simple forces the main workflow to ignore the activity and move to the next activity. The rule works as follows: on the notification of an event **CompensationRequest**, if the event was *produced by* a policy applying to the **Non-Vital** policy, the rule executes an instance of the workflow **Continue**.

As an example consider the activity *Compute Mood* of the Status Updater workflow. Since we assume that the *mood of a user* is not necessary for generating the message to be posted in a user social network, *Compute Mood* is an example of a *non-vital* activity. **Listing 5.6** illustrates the use of the **Non-Vital** policy type for managing failures of the *Compute Mood* activity. As shown in the expression, we first create an instance describing the state management properties of the *Compute Mood* activity. Then, we use this instance for creating a **Non-Vital** policy instance applied to *Compute Mood*.

```
1  policy class NonVital : ActivityExceptionManagement {
2      Boolean isAValidScope ( ExceptionManagementProperties activityProperties ) {
3        if ( activityProperties.side-effects == false )
4            return true;
5        else
6            return false;
7      }
8      rule ContinueOnFailure
9        on ActivityFailed
10       if  event.producedBy( scope )
11       do Continue ( event.producer )
12     rule ContinueOnCompensationRequest
13       on CompensationRequest
14       if  event.producedByAncestorAction( scope )
15       do Continue ( event.producer )
16 }
```

Listing 5.5 Definition of the Non-Vital policy type

```
var ap = new NonVital |[ wf.activities[ "Compute Mood" ] ]| (
      new ExceptionManagementProperties {
          side-effects = false,
      }
) ;
```

Listing 5.6 Example of a Non-Vital policy instance

Undoable Active Policy

We use the **Undoable** policy type for handling failures and compensating *undoable activities*. The definition of this type is shown in **listing 5.7**:

- **Line 1** defines the **Undoable** type as a specialization of the **Activity Exception Management** type.
- **Lines 2–10** redefine the method *isAValidScope* for ensuring that only *undoable activities* are associated to an **Undoable** policy instance (cf. **table 5.1**).
- **Lines 11** and **12** enable the rules *RetryOnFailure* and *CompensateAndRetryWhenRequested*. Recall that these rules are inherited from the **Activity Exception Management** type (see **figure 5.5**).
- **Lines 13–16** define the rule *CompensateWithoutRetryingWhenRequested*. In contrast to the rule *CompensateAndRetryWhenRequested*, this rule compensates the work done by *activity* without retrying it. The rule works as follows: on the notification of an event **CompensationRequest**, if *activity* is not *retry-able* and the event was *produced by* a policy applied to the **Undoable** policy (see **step 3** in **figure 5.4**), the rule executes an instance of the workflow **Compensate** which undoes (i.e., rollback) the committed *activity*.

As an example consider the activity *Update Twitter* of the Status Updater workflow. Let us assume that this activity cannot be retried once it commits. Since the activity can be compensated but not retried, *Update Twitter* is an example of an *undoable* activity. **Listing 5.8** illustrates the use of the **Undoable** policy type for managing failures of the *Update Twitter* activity. As shown in the expression, (i) we create an instance describing the state management properties of the *Update Twitter* activity and (ii) we use the instance for creating an **Undoable** policy instance associated to the activity *Update Twitter*.

```
1  policy class Undoable : ActivityExceptionManagement {
2      Boolean isAValidScope ( ExceptionManagementProperties activityProperties ) {
3        if ( activityProperties.side-effects == false &&
4             activityProperties.compensate-able == true &&
5             activityProperties.retry-able != null &&
6             activityProperties.maxNumberOfRetries > 0 )
7               return true;
8        else
9               return false;
10     }
11     enable rule RetryOnFailure
12     enable rule CompensateAndRetryWhenRequested
13     rule CompensateWithoutRetryingWhenRequested
14       on CompensationRequest
15       if  event.producedByAncestorAction( scope )  &&  this.activityProperties.compensate-able  && ! this.activityProperties.retry-able
16       do Compensate ( event.producer )
17 }
```

Listing 5.7 Definition of the Undoable policy type

```
var ap = new Undoable |[ wf.activities[ "Update Twitter" ] ]| (
      new ExceptionManagementProperties {
            side-effects = false,
            compensate-able = true,
            retry-able = false,
      }
);
```

Listing 5.8 Example of an Undoable policy instance

Compensate-able Active Policy

We use the **Compensate-able** policy type for handling failures and compensating *compensate-able activities*. The definition of this type is shown in **listing 5.9**:

- **Line 1** defines the **Compensate-able** type as a specialization of the **Activity Exception Management** type.
- **Lines 2–10** redefine the method *isAValidScope* for ensuring that only *compensate-able activities* are associated to a **Compensate-able** policy instance (cf. **table 5.1**).

- **Lines 11** and **12** enable the rules *RetryOnFailure* and *CompensateAndRetryWhenRequested* (see **figure 5.5**).[1]

As an example consider the activity *Update Facebook* of the Status Updater workflow. Since this activity can be retried and compensated (with side effects), the activity is *compensate-able*. **Listing 5.10** illustrates the use of the **Compensate-able** policy type for managing failures of the *Update Facebook* activity using the previous pattern: first we create an instance describing the state management properties of the *Update Facebook* activity. Then, we use this instance for creating a **Compensate-able** policy applying to the activity *Update Facebook*.

```
1   policy class Compensate-able : ActivityExceptionManagement {
2       Boolean isAValidScope ( ExceptionManagementProperties activityProperties ) {
3         if ( activityProperties.side-effects == true &&
4             activityProperties.compensate-able == true &&
5             activityProperties.retry-able != null &&
6             activityProperties.maxNumberOfRetries > 0 )
7               return true;
8         else
9               return false;
10      }
11      enable rule RetryOnFailure
12      enable rule CompensateAndRetryWhenRequested
13  }
```

Listing 5.9 Definition of the Compensate-able policy type

```
var ap = new Compensate-able |[ wf.activities[ "Update Facebook" ] ]| (
      new ExceptionManagementProperties {
            side-effects = true,
            compensate-able = true,
            retry-able = false,
            maxNumberOfRetries = 3,
      }
) ;
```

Listing 5.10 Example of a Compensate-able policy instance

5.2.2 Atomicity Active Policy

We use the **Atomicity** policy type for providing an atomic behavior to the execution of the activities participating in an atomicity policy tree (cf. **figure 5.3**). In particular, the **Atomicity** policy type can provide three types of atomic behaviors:

- **Strict** atomicity. This behavior conforms to the classical "all or nothing" principle.

[1] Note that the **Compensate-able** policy type does not define any new rules (i.e., the functionality of this policy type resides in the rules inherited from the **Activity State Management** policy type).

- **Alternative** atomicity. This behavior captures the possibility of using alternative execution paths for committing.
- **Exception** atomicity. This behavior indicates that if something goes wrong an exception must be launched.

Listing 5.11 shows the definition of the **Atomicity** policy type. The expression is interpreted as follows:

- **Line 4** defines the *scope* of the policy. In this case, the scope can be composed of a set of execution units of type **Activity Exception Management**.
- **Line 6** defines the variable *atomicityType* of type **AtomicityType**. This variable specifies the type of atomic behavior provided by an instance of the **Atomicity** policy type (e.g., *strict* or *alternative* atomicity*).* **Line 1** shows the definition of the type **AtomicityType**.
- **Lines 8–10** define the policy constructor. Note that the constructor receives as input the type of atomic behavior that has to provide the new policy instance.
- **Lines 12–15** define the rule *StrictBackwardRecovery*. This rule implements the *strict atomic behavior*. The rule works as follows: on the notification of an event **CustomScopeFailure** (see **step 2** of **figure 5.4**), if *atomicityType* is *strict* and the producer of the event is an *exception management policy* belonging to the policy scope (i.e., the producer is a descendent of the policy in a policy tree), the rule will execute an instance of workflow **BackwardRecovery**. Intuitively, a **BackwardRecovery** instance produces an event **CompensationRequest** that triggers the compensation of the activities participating in the policy tree (see **step 4** of **figure 5.4**).
- **Lines 17–21** define the rule *FordwardRecovery*. This rule implements the *alternative atomic behavior*. The rule works as follows: on the notification of an event **CustomScopeFailure** (see **step 2** of **figure 5.4**), if (i) *atomicityType* is *alternative*, (ii) there exist an alternative path for committing and (iii) the producer of the event belongs to the policy scope, the rule will execute an instance of the **FordwardRecovery** workflow. Intuitively, a **FordwardRecovery** instance proceeds as follows: first it pauses the execution of the *main workflow* (i.e., the workflow executing the activities). Then it analyses the main workflow' control flow for determining the alternative path to take. Finally it executes all the activities that are in the new path until committing the workflow. **Figure 5.6** illustrates the *forward recovery* principle using the status updater workflow. In the example, the *Update Twitter* activity fails, which triggers the rule *FordwardRecovery* of $\mathbf{ap}_{\mathbf{atom}}$. Since there is another path to commit, the rule "forces" the main workflow to follow this path.
- **Lines 23–27** define the rule *BackwardRecovery*. In contrast to the *FordwardRecovery* rule, this rule undoes all the committed activities when no alternative path exist.[2] **Figure 5.7** illustrates the *backward recovery* principle using the Status Updater workflow. In the example, the *Update Twitter* and Update Facebook activities both fail

[2] Note that *alternative atomicity* is similar to *strict atomicity* when no other path exists to commit.

so there is no other path to commit. In consequence, the *BackwardRecovery* rule of $\mathbf{ap}_{\mathbf{atom}}$ undoes the committed activities *Get Song* and *Compute Mood*.

- **Lines 29–32** define the rule *NotifyException*. This rule implements the *exception atomicity behavior*. The rule works as follows: on the notification of an event **CustomScopeFailure**, if *atomicityType* is *exception*, and the producer of the event belongs to the policy scope, the rule will execute an instance of the workflow. Intuitively, a **NotifyException** instance notifies the environment about the workflow failure and waits for instructions for treating the failure.
- **Lines 34–38** define the rule *NotifyPolicyFailure*. This rule notifies the environment about the failure of the compensation procedure. The rule works as follows: on the notification of an event **CustomActivePolicyFailure**, if the event producer is the action of one of the policy rules belonging to the scope of the policy, the rule will execute an instance of the workflow **NotifyActivePolicyException** that notifies the compensation failure (see **step 5** of **figure 5.4**). Recall that a compensation procedure may fail because (i) an activity cannot be compensated or (ii) its maximum number of retries is exceeded.

```
1   enum AtomicityType { Strict, Alternative, Exception }
2
3   [ AfterPolicy ActivityExceptionManagement ]
4   policy class Atomicity |[ ActivityExceptionManagement * ]| : ActivePolicy {
5
6     AtomicityType atomicityType
7
8     Atomicity ( AtomicityType atomicityType ) {
9       this.atomicityType = atomicityType ;
10    }
11
12    rule StrictBackwardRecovery
13      on CustomScopeFailure
14      if event.producedByDescendent( scope ) && atomicityType.Strict
15      do BackwardRecovery ( event.producer )
16
17    rule FordwardRecovery
18      on CustomScopeFailure
19      if event.producedByDescendent( scope ) && atomicityType.Alternative &&
20         this.existAlternativePath( )
21      do FordwardRecovery ( event.producer )
22
23    rule BackwardRecovery
24      on CustomScopeFailure
25      if event.producedByDescendent( scope ) && atomicityType.Alternative &&
26         ! this.existAlternativePath( )
27      do BackwardRecovery ( event.producer )
28
29    rule NotifyException
30      on CustomScopeFailure
31      if event.producedBy( scope ) && atomicityType.Exception
32      do NotifyException ( event.producer )
33
34    rule NotifyPolicyFailure
35      on CustomActivePolicyFailure
36      if event.producedByDescendent( scope ) &&
37         compare( event.delta [ "Policy" ].type, ActivityExceptionManagement )
38      do NotifyActivePolicyFailure ( event.producer, this )
39  }
```

Listing 5.11 Atomicity policy type expression

Finally note that the **Atomicity** policy type has *lower priority* than the **Activity Exception Management** policy type (see **line 3** in **listing 5.11**). This ensures that, even if a developer adds rules to the exception policy type, the rules in an exception policy instance will be executed before the rules of an atomicity policy instance. This maintains the principle of our approach: atomicity is build on top of activity exception management.

As an example consider the *Update Facebook* and *Update Twitter* activities of the Status Updater workflow. Let us assume that these activities can be executed using *alternative atomicity*. **Listing 5.12** illustrates the use of the **Atomicity** policy for adding an atomic behavior to these activities. In the example, (i) ap_1 is an instance of a **Critical** policy applying to the *Update Facebook* activity; (ii) ap_2 is an instance of an **Undoable** policy applying to the *Update Twitter* activity and (iii) ap_n is a policy instance of type **Atomicity** that applies to $\{ap_1, ap_2\}$ and which associates an alternative atomic behavior to *Update Facebook* and *Update Twitter* (see **figure 5.6**).

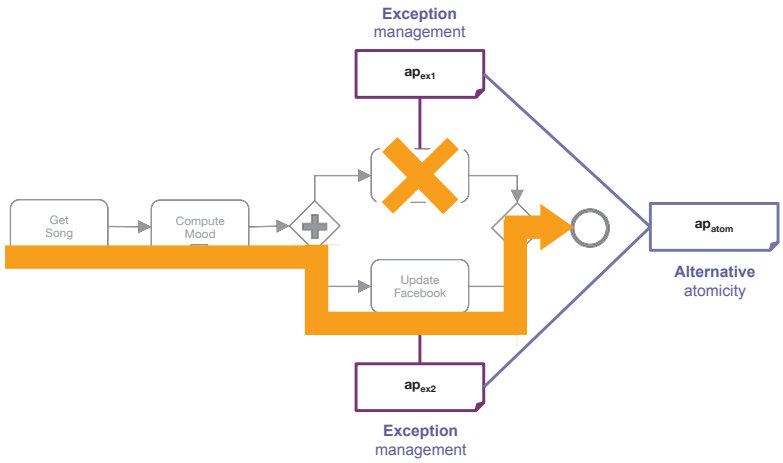

Figure 5.6 FordwardRecovery principle

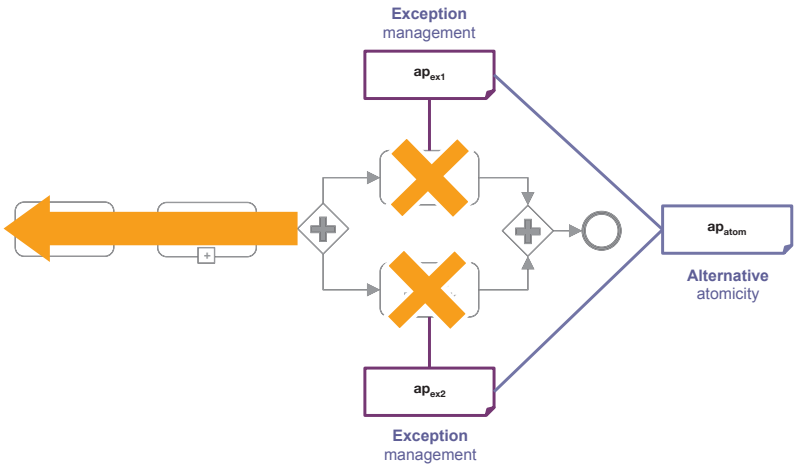

Figure 5.7 BackwardRecovery principle

```
var ap1 = new Critical |[ wf.activities[ "UpdateFacebook" ] ]| (
      new ExceptionManagementProperties {
            compensate-able = false,
            retry-able = false,
      }
) ;

var ap2 = new Undoable |[ wf.activities[ "Update Twitter" ] ]| (
      new ExceptionManagementProperties {
            side-effects = false,
            compensate-able = true,
            retry-able = false,
      }
) ;

var apn = new Atomicity |[ ap1, ap2 ]| ( AtomicityType.Alternative ) ;
```

Listing 5.12 Atomicity policy instance example

5.3 Active Policies for Persistence

We use the **Activity State Management** and **Persistency** policy types for providing persistency to the execution of a workflow. The former defines rules for keeping track of the *execution history* of an activity. The latter defines rules for (i) persisting the execution history of a set of activities into a stable storage and (ii) recovering the state of the activities in the presence of system failures. **Figure 5.8** shows the relationship among the **Activity State Management** and **Persistency** policy types.

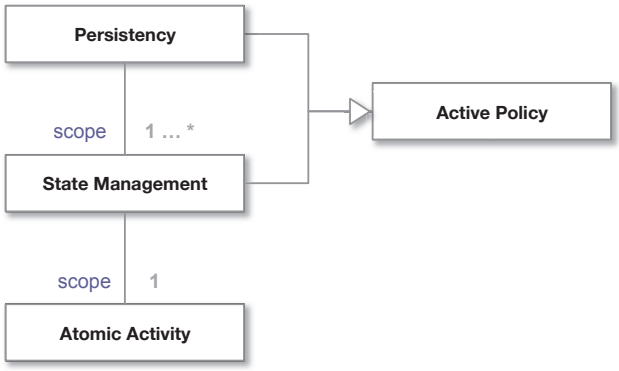

Figure 5.8 Persistency and State Management types relationships (UML class diagram)

As shown in the figure, the **Activity State Management** policy type is a policy that can be associated only to *atomic activities* (i.e., the policy is an *activity policy*). In contrast, the **Persistency** policy type is a policy type that can be associated to any number of *state management* policies (i.e., the policy is a *policy of policies*). When a *persistency policy* is associated to a set of *state management policies* they form a *policy tree* where the root (i.e., the persistency policy) ensures the persistency and recovering the tree leafs (i.e., the atomic activities). This is illustrated in **figure 5.9** where the *atomicity policy* $\mathbf{ap_3}$ ensures the persistency of the

Update Facebook and *Update Twitter* activities through the *state management* policies $\mathbf{ap_1}$ and $\mathbf{ap_2}$ (respectively).

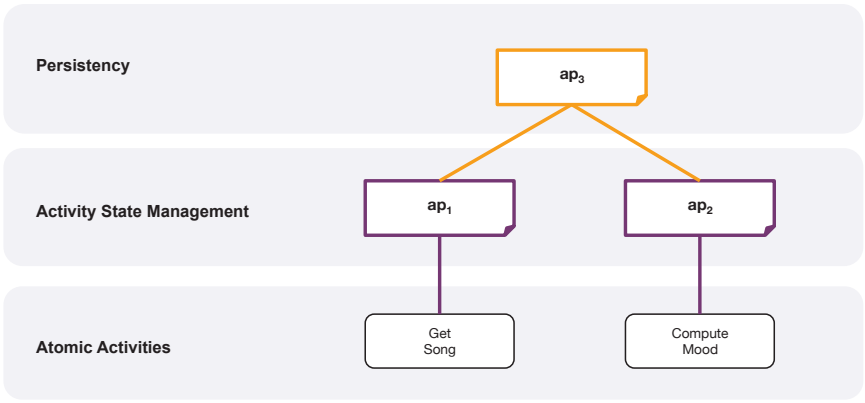

Figure 5.9 Example of a Persistency policy tree

When **Activity State Management** and **Persistency** policies are associated to each other in a policy tree, they interact among them by producing events. **Figure 5.10** illustrates the general interaction among instances of these policy types. The diagram is interpreted as follows: when an activity belonging to the scope of a state management policy *starts* (*completes*) its execution, the state management policy saves the *initial* (*final*) state of the activity into an internal structure (see **step 1**). Then, if the state management policy is already associated to the scope of a persistency policy, the persistency policy saves these states into a stable storage (e.g., disk or cache). This is done every time the rule in charge of saving the activity' state ends its execution (see **step 2**). When the workflow associated to the persistency policy fails, the execution environment detects the failure and produces a **WorkflowFailed** event that triggers the recovery process (see **step 3**). Then, this process recovers each of the states of the activities participating in the persistency' policy tree to the state previous to the failure (see **step 4**).

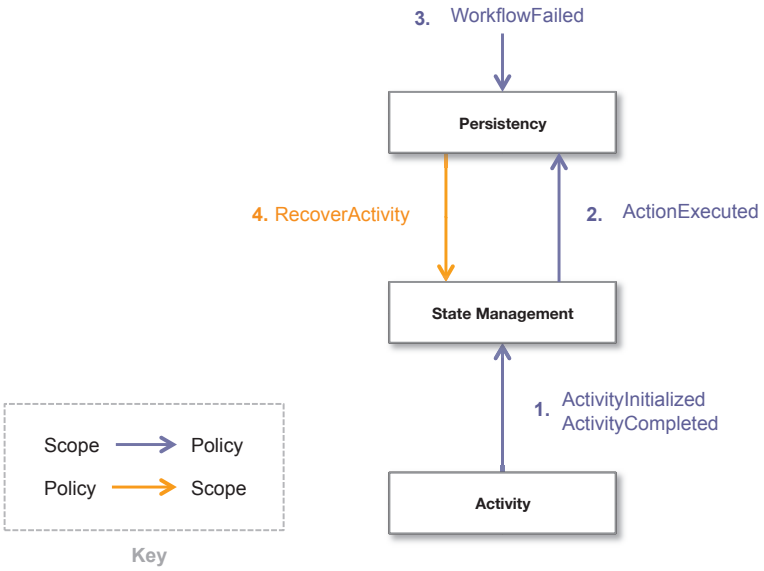

Figure 5.10 Persistency and State Management policies general interaction (UML interaction diagram)

In what follows we describe the structure of the **Activity State Management** and **Persistency** policy types.

5.3.1 Activity State Management Active Policy

We use the **Activity State Management** policy type for (i) constructing the *execution history* of an activity and (ii) recovering the state of the activity using this history. As shown in **figure 5.11**, we have specialized the **Activity State Management** type in the **Non-Reliable**, **Predictable**, **Idempotent** and **Verifiable** subtypes in order to encapsulate the logic responsible of saving and recovering the state of *non-reliable*, *predictable*, *idempotent* and *verifiable* activities (respectively). First, we will describe the structure of the **Activity State Management** policy type. Then, we will describe the structure of each of its subtypes.

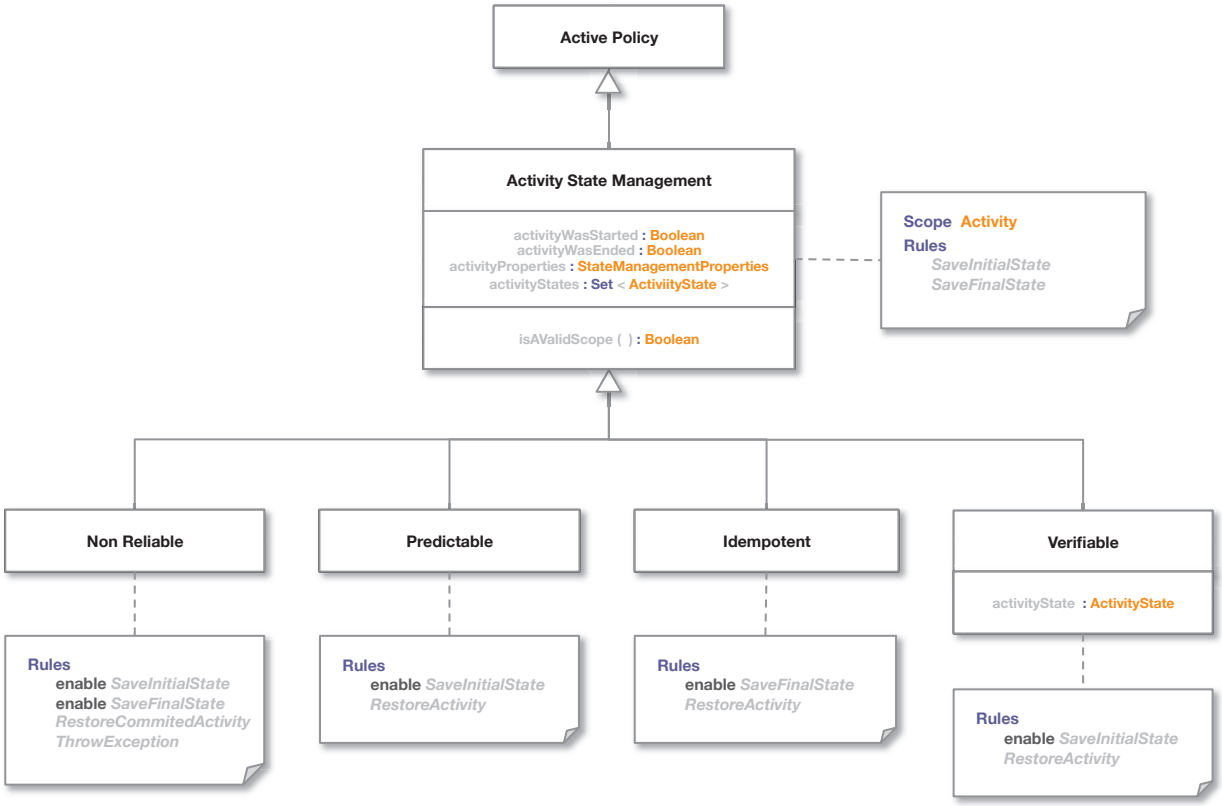

Figure 5.11 Activity State Management policy type (UML class diagram)

Activity State Management Policy

Listing 5.13 shows the definition of the **Activity State Management** policy type using the AP Language.

- **Line 1** defines the **Activity State Management** type as a subtype of the **Active Policy** type. This line also defines the *scope* of the policy as a scope composed of a single execution unit called *activity*. Note that the type of *activity* is **AtomicActivity** (i.e., the state management policy type applies to any atomic activity).

- **Line 3** defines the boolean variables *activityWasStarted* and *activityWasEnded*. These variables are used for determining whether an activity was started or ended (respectively).
- **Line 4** defines the variable *activityProperties* of type **StateManagementProperties**. This variable contains the *state management properties* of *activity* (i.e., the variable contains the properties of the activity in the policy scope). The definition of the **StateManagementProperties** type is shown in **listing 5.14**.
- **Line 5** defines the variable *activityStates*. This variable is used for containing the set of states through which an activity goes through (i.e., the execution history of *activity*). Note that we use the **ActivityState** type for representing an activity state. The definition of this type is shown in **listing 5.14**.
- **Lines 7–13** define the policy constructor. Note that the constructor receives as input the *state management properties* of the activity that has to be associated to the new policy instance. If the properties correspond to the activity' properties expected by the constructor (i.e., if the activity is a valid scope), the constructor creates a new policy instance. Otherwise, it throws an exception.
- **Line 15** defines the method *isAValidScope*, which determines whether a policy instance (based on its properties) can be applied to a specific activity or not. For instance, if the policy is expecting a *predictable activity*, and a developer applies the policy to a *non-reliable activity*, this method will return *false* and the constructor will not create the policy instance.
- **Lines 17–25** define the rules *SaveInitialState* and *SaveFinalState*, which are responsible of saving the initial and final states of *activity* (respectively). The rules work as follows: on the notification of an event **ActivityInitialized** / **ActivityCompleted**, if the event was produced by *activity* (i.e., the activity in the policy scope), the rules will execute an instance of the **SaveState** workflow. Intuitively, this workflow will save the activity' *current state* (i.e., the values of the activity' variables when initialized) into the variable *activityState*. Then, the rule will set the variable *activityWasStarted* / *activityWasEnded* to *true*.

Note that the **Activity State Management** policy type is an *abstract* type (see **listing 5.13**). Recall that this implies that the policy type cannot be instantiated. Thus we only use it as *base type* for the subtypes **Non-Reliable**, **Predictable**, **Idempotent** and **Verifiable**.

```
1   abstract policy class ActivityStateManagement |[ AtomicAtivity activity ]| : ActivePolicy {
2
3     Boolean activityWasStarted, activityWasEnded ;
4     StateManagementProperties activityProperties ;
5     Set <ActivityState> activityStates ;
6
7     ActivityStateManagement ( StateManagementProperties activityProperties ) {
8       if ( isAValidScope ( activityProperties ) ) {
9         this.activityProperties = activityProperties ;
10        this.activityStates = new Set <ActivityState> ( );
11      } else
12        throw Exception ;
13    }
14
15    Boolean isAValidScope ( StateManagementProperties activityProperties ) { return false; }
16
17    rule SaveInitialState
18      on ActivityInitialized
19      if event.producedBy( activity )
20      do activityWasStarted = SaveState ( activity, activityStates ) ;
21
22    rule SaveFinalState
23      on ActivityCompleted
24      if event.producedBy( scope )
25      do activityWasEnded = SaveState ( activity, activityStates ) ;
26  }
```

Listing 5.13 Definition of the Activity State Management policy type

```
class StateManagementProperties {
  Boolean isStateAccessible;
  Boolean isIdempotent;
  ActivityState presumedFinalState ;
}

enum ActivityState {
  Prepared, Failed, Commited, Unknown
}
```

Listing 5.14 Definition of the StateManagementProperties and ActivityState types

Non Reliable Policy

We use the **Non-Reliable** policy type for constructing the execution history and recovering the execution state of *non-reliable* activities. **Listing 5.15** shows the definition of the **Non-Reliable** type:

- **Line 1** defines the **Non-Reliable** type as a specialization of the **Activity State Management** type (i.e., an instance of the **Non-Reliable** policy type inherits the scope, methods, variables and rules defined in the **Activity State Management** policy type).
- **Lines 3–10** redefine the method *isAValidScope* for ensuring that only *non-reliable activities* are associated to a **Non-Reliable** policy instance (see **table 5.2**).
- **Lines 12–13** enable the rules *SaveInitialState* and *SaveFinalState*. Recall that these rules are inherited from the **Activity State Management** policy type (see **figure 5.11**).

- **Lines 15–18** define the rule *RestoreCommittedActivity*. On the notification of an event **RecoverActivity**, if the activity associated to the policy was *started* and *ended*, and the event was produced by a policy applying to the **Non-Reliable** policy (i.e. produced by an *ancestor* of the policy in a *policy tree*), the rule executes an instance of the workflow **Restore**. Intuitively, this workflow restores an activity to a specific state. In this case, the workflow restores the activity associated to the policy to the *committed* state.
- **Lines 20–23** define the rule *ThrowException*. On the notification of an event **RecoverActivity**, if *activity* is still running (i.e., *activity* was *started* but not *ended*), and the event producer is a policy applying to the **Non-Reliable** policy, the rule executes an instance of the workflow **ThrowException**. This workflow notifies the environment that the activity cannot be restored because the activity does not commit before the system failure.

As an example consider the activity *Update Twitter* of the Status Updater workflow. Let us assume that a user status on Twitter can be written but not read. Since the memory space modified by the activity *Update Twitter* is not accessible to other activities, the activity is not idempotent and nothing can be assumed about its final state, *Update Twitter* is an example of a *non-reliable* activity. **Listing 5.16** illustrates the use of the **Non-Reliable** policy type for recovering the execution of the *Update Twitter* activity. As shown in the expression, we first create an instance describing the state management properties of the *Update Twitter* activity. Then, we use the instance for creating a **Non-Reliable** policy instance associated to the activity *Update Twitter*.

```
1   policy class NonReliable : ActivityStateManagement {
2
3       Boolean isAValidScope ( StateManagementProperties activityProperties ) {
4         if ( activityProperties.stateIsAccesible == false &&
5             activityProperties.isIdempotent == false &&
6             activityProperties.presumedFinalState == ActivityState.Unknown )
7             return true;
8         else
9             return false;
10      }
11
12      enable rule SaveInitialState
13      enable rule SaveFinalState
14
15      rule RestoreCommitedActivity
16        on RecoverActivity
17        if  event.producedByAncestorAction( scope ) && activityWasStarted && activityWasEnded
18        do Restore ( activity, ActivityState.Commited )
19
20      rule ThrowException
21        on RecoverActivity
22        if  event.producedByAncestorAction( scope ) && activityWasStarted && !activityWasEnded
23        do ThrowException
24  }
```

Listing 5.15 Definition of the Non Reliable policy type

```
var wf = new StatusUpdater { ... }

var ap = new NonReliable |[ wf.activities[ "Update Twitter" ] ]| (
      new StateManagementProperties {
            stateIsAccessible = false,
            isIdempotent = false,
            presumedFinalState = ActivityState.Unknown
      }
);
```

Listing 5.16 Example of a NonReliable policy instance

Predictable Policy

We use the **Predictable** policy type for constructing the execution history and recovering the execution state of *predictable* activities. **Listing 5.17** shows the definition of the **Predictable** type:

- **Line 1** defines the **Predictable** type as a specialization of the **Activity State Management** type.
- **Lines 3–9** redefine the method *isAValidScope* for ensuring that only *predictable activities* are associated to a **Predictable** policy instance (see **table 5.2**).
- **Line 11** enables the rule *SaveFinalState*. Recall that this rule is inherited from the **Activity State Management** type (see **figure 5.11**).
- **Lines 13–17** define the rule *RestoreActivity*. On the notification of an event **RecoverActivity**, if the activity associated to the policy was *started*, and the event was produced by a policy applying to the **Predictable** policy, the rule executes an instance of the workflow **Restore** for restoring the activity to its *presumed* state. In contrast, if the activity was not started, the rule restores the activity to the *prepared* state.

As an example consider the activity *Update Facebook* of the Status Updater workflow. Since we assume that this activity always commits, the activity is an example of a *predictable* activity. **Listing 5.18** illustrates the use of the **Predictable** policy type for recovering the execution of the *Update Facebook* activity. As shown in the expression, we first create an instance describing the state management properties of the *Update Facebook* activity. Then, we use the instance for creating a **Predictable** policy instance associated to the activity *Update Facebook*.

```
1   policy class Predictable : ActivityStateManagement {
2
3       Boolean isAValidScope ( StateManagementProperties activityProperties ) {
4         if ( activityProperties.presumedFinalState == ActivityState.Commited ||
5              activityProperties.presumedFinalState == ActivityState.Failed )
6                return true;
7         else
8                return false;
9       }
10
11      enable rule SaveFinalState
12
13      rule RestoreActivity
14        on RecoverActivity
15        if  event.producedByAncestorAction( scope ) && activityWasStarted
16        do Restore ( activity, activityProperties.presumedFinalState )
17        else do Restore ( activity, ActivityState.Prepared )
18  }
```

Listing 5.17 Definition of the Predictable policy type

```
var ap = new Predictable |[ wf.activities[ "Update Facebook" ] ]| (
      new StateManagementProperties {
            stateIsAccessible = false,
            isIdempotent = false,
            presumedFinalState = ActivityState.Committed
      }
) ;
```

Listing 5.18 Example of a Predictable policy instance

Idempotent Policy

We use the **Idempotent** policy type for constructing the execution history and recovering the execution state of *idempotent* activities. **Listing 5.19** shows the definition of the **Idempotent** type:

- **Line 1** defines the **Idempotent** type as a specialization of the **Activity State Management** type.
- **Lines 2–7** redefine the method *isAValidScope* for ensuring that only *idempotent activities* are associated to an **Idempotent** policy instance.
- **Line 8** enables the rule *SaveFinalState*. Recall that this rule is inherited from the **Activity State Management** type (see **figure 5.11**).
- **Lines 9–12** define the rule *RestoreActivity*. On the notification of an event **RecoverActivity**, if the activity associated to the policy does not complete, and the event was produced by a policy applying to the **Idempotent** policy, the rule executes an instance of the workflow **Restore** for restoring the activity to the *prepared* state.

As an example consider the activity *Compute Mood* of the Status Updater workflow. Since we assume that this activity always produces the same mood based on the music the user is currently listen to, *Compute Mood* is an example of an *idempotent* activity. **List-**

ing 5.18 illustrates the use of the **Idempotent** policy type for recovering the execution of the *Compute Mood* activity using the typical pattern: first we create an instance describing the state management properties of the *Compute Mood* activity. Then, we use the instance for creating an **Idempotent** policy instance associated to the activity *Compute Mood*.

```
1   policy class Idempotent : ActivityStateManagement {
2       Boolean isAValidScope ( StateManagementProperties activityProperties ) {
3           if ( activityProperties.isIdempotent == true )
4               return true;
5           else
6               return false;
7       }
8       enable rule SaveFinalState
9       rule RestoreActivity
10          on RecoverActivity
11          if  event.producedByAncestorAction( scope ) && !activityWasEnded
12          do Restore ( activity, ActivityState.Prepared )
13  {
```

Listing 5.19 Definition of the Idempotent policy type

```
var ap = new Idempotent |[ wf.activities[ "Compute Mood" ] ]| (
    new StateManagementProperties {
        isIdempotent = true,
    }
);
```

Listing 5.20 Example of an Idempotent policy instance

Verifiable Policy

We use the **Verifiable** policy type for constructing the execution history and recovering the execution state of *verifiable* activities. Listing 5.21 shows the definition of the **Verifiable** type:

- **Line 1** defines the **Verifiable** type as a specialization of the **Activity State Management** type.
- **Line 2** defines the variable *activityState* of type **ActivityState**. This variable will contain the activity state previous to the failure.
- **Lines 3–8** redefine the method *isAValidScope* for ensuring that only *verifiable activities* are associated to a **Verifiable** policy instance.
- **Line 9** enables the rule *SaveInitialState*. Recall that this rule is inherited from the **Activity State Management** type (see figure 5.11).
- **Lines 10–17** define the rule *RestoreActivity*. On the notification of an event **RecoverActivity**, if *activity* was *started* and the event was produced by a policy applying to the **Verifiable** policy, the rule executes a workflow that retrieves the activity' last state

from the stable storage and saves this state into the variable *activityState*. Then, the workflow uses the value in *activityState* for restoring the activity to that state (e.g., if the last state was the *committed* state, the workflow will retrieve the *committed* state from the stable storage and it will restore the activity to the committed state). In contrast, if the activity was not started, the workflow restores the activity to the *prepared* state.

As an example consider the activity *Get Song* of the Status Updater workflow. Since we assume that the information about the played songs is accessible to other activities, this activity is an example of a *verifiable* activity. **Listing 5.22** illustrates the use of the **Verifiable** policy type for recovering the execution of the *Get Song* activity.

```
1   policy class Verifiable : ActivityStateManagement {
2       ActivityState activityState ;
3       Boolean isAValidScope ( StateManagementProperties activityProperties ) {
4         if ( activityProperties.stateIsAccesible == true )
5             return true;
6         else
7             return false;
8       }
9       enable rule SaveInitialState
10      rule RestoreActivity
11        on RecoverActivity
12        if event.producedByAncestorAction( scope ) && activityWasStarted
13        do Sequence (
14            GetSate ( activity, activityState ),
15            Restore ( activity, activityState )
16        )
17        else do Restore ( activity, ActivityState.Prepared )
18  }
```

Listing 5.21 Definition of the Verifiable policy type

```
var ap = new Verifiable |[ wf.activities[ "Get Song" ] ]| (
      new StateManagementProperties {
          stateIsAccessible = true,
      }
) ;
```

Listing 5.22 Example of a Verifiable policy instance

5.3.2 Persistency Active Policy

We use the **Persistency** policy type for providing persistency to the execution of the activities participating in a persistency policy tree (cf. **figure 5.9**). In particular, the **Persistency** policy type can provide two types of persistent behaviors:

- **Best-Effort** persistency. This behavior saves the activities' execution history into fast and volatile storage unit (e.g., a cache). Since the storage unit is volatile, the execu-

tion history may not be available for the recovery process in the presence of a total system failure (e.g., power outage).

- **Guaranteed** persistency. This behavior saves the activities' execution history into a slow and durable storage unit (e.g., a disk). Thus, in case of a total system failure, the execution history may survive and it will be available for the recovery processes.

Listing 5.23 shows the definition of the **Persistency** policy type. The expression is interpreted as follows:

- **Line 3** defines the *scope* of the policy. In this case, the scope can be composed of a set of execution units of type **Activity State Management**.
- **Line 4** defines the variable *persistencyType*, which determines the type of persistency that an instance of the **Persistency** policy implements (e.g., *guaranteed persistency).* The type of this variable is **PersistencyType**. The definition of this type is shown in **line 1**.
- **Lines 5–7** define the policy constructor. The constructor receives as input the type of persistent behavior that must implement a new **Persistency** instance (i.e., best-effort or guaranteed).
- **Lines 8–11** define the rule *RecoverWorkflow*. On the notification of an event **WorkflowFailed** (see **step 3** of **figure 5.10**), if the producer of the event was produced by the workflow associated to the policy, the rule will execute an instance of the **RecoveryProcess** workflow for recovering the state of each of the activities participating in the policy tree (see **step 4** of **figure 5.10**).
- **Lines 12–17** define the rule *BestEffortStrategy*. On the notification of an event **ActionExecuted** (see **step 2** of **figure 5.4**), if (i) the event producer belongs to the policy scope (i.e., the producer is a *descendent* of the policy in a policy tree), (ii) the event represents the execution of the rule *SaveInitialState* (or *SaveFinalState*) and (iii) the policy implements a *best effort* behavior, the rule will execute an instance of the **SaveIntoCache** workflow. Intuitively, this workflow will save into *cache* the *activity states* associated to the policy producing the event. Note that the rule will execute its action asynchronously. This is due to the fact that the best effort behavior does not guarantees the durability of the execution history, thus the main workflow can continue its execution independently of the rule' action.
- **Lines 22–27** define the rule *GuaranteedStrategy*. This rule works similar to the *BestEffortStrategy* rule except that, when the rule implements the *guaranteed persistent* behavior, it executes an instance of the workflow **SaveIntoDisk** for saving the *activity states* into disk. Note that this rule executes its action synchronously. This ensures that the main workflow cannot continue its execution until the activities states are truly saved on disk.

Finally note that the **Persistency** policy type has *lower priority* than the **Activity State Management** policy type (see **line 2** in **listing 5.23**). This ensures that, even if a developer adds rules to the state management policy type (or one of its subtypes), the rules of the persistency poli-

cy will be executed before the rules of the state management policies thus maintaining the coherence of the approach (i.e., persistency is build on top of state management).

```
1   enum PersistencyType { BestEffort, Guaranteed }
2
3   [ AfterPolicy ActivityStateManagement ]
4   policy class Persistency |[ ActivityStateManagement * ]| : ActivePolicy {
5
6     PersistencyType persistencyType ;
7
8     Persistency ( PersistencyType persistencyType ) {
9       this.persistencyType = persistencyType ;
10    }
11
12    rule RecoverWorkflow
13      on WorkflowFailed
14      if event.producedByAssociatedWorkflow( scope )
15      do RecoveryProcess
16
17    async rule BestEffortStrategy
18      on ActionExecuted
19      if event.producedByDescendent( scope ) && persistencyType.BestEffort
20          ( compare( event.delta [ "Rule" ].name, "SaveInitialState" ) ||
21          compare( event.delta [ "Rule" ].name, "SaveFinalState" ) )
22      do SaveIntoCache ( ( (ActivityStateManagement) event.producer ).activityStates )
23
24    sync rule GuaranteedStrategy
25      on ActionExecuted
26      if event.producedByDescendent( scope ) && persistencyType.Guaranteed &&
27          ( compare( event.delta [ "Rule" ].name, "SaveInitialState" ) ||
28          compare( event.delta [ "Rule" ].name, "SaveFinalState" ) )
29      do SaveIntoDisk ( ( (ActivityStateManagement) event.producer ).activityStates )
30  }
```

Listing 5.23 Definition of the Persistency policy type

As an example consider the *Status Updater* workflow. Let us assume that we want to add a *best effort* persistent behavior to the *Get Song* and *Compute Mood* activities, and a *guaranteed* persistent behavior to the *Update Twitter* and *Update Facebook* activities. **Listing 5.24** illustrates the use of the **Persistency** policy type for adding these behaviors to the *Status Updater* activities. In the example, (i) ap_1 is an instance of a **Verifiable** policy applying to the *Get Song* activity; (ii) ap_2 is an instance of an **Idempotent** policy applying to the *Compute Mood* activity; (iii) ap_3 is an instance of a **Predictable** policy applying to the *Update Facebook* activity; (iv) ap_4 is an instance of a **NonReliable** policy applying to the *Update Twitter* activity, and (v) $ap_{bestEffort}$ and $ap_{guaranteed}$ are **Persistency** policy instances applying to $\{ap_1, ap_2\}$ and $\{ap_3, ap_4\}$ respectively (see **figure 5.12**).

```
var ap1 = new Verifiable |[ wf.activities[ "Get Song" ] ]| (
        new StateManagementProperties {
                stateIsAccessible = true,
        }
) ;

var ap2 = new Idempotent |[ wf.activities[ "Compute Mood" ] ]| (
        new StateManagementProperties {
                isIdempotent = true,
        }
) ;

var ap3 = new Predictable |[ wf.activities[ "Update Facebook" ] ]| (
        new StateManagementProperties {
                presumedFinalState = ActivityState.Committed
        }
) ;

var ap4 = new NonReliable |[ wf.activities[ "Update Twitter" ] ]| (
        new StateManagementProperties {
                stateIsAccessible = false,
                isIdempotent = false,
                presumedFinalState = ActivityState.Unknown
        }
) ;

var apBestEffort = new Persistency |[ ap1, ap2 ]| ( PersistencyType.BestEffort ) ;

var apGuaranteed = new Persistency |[ ap3, ap4 ]| ( PersistencyType.Guaranteed ) ;
```

Listing 5.24 Persistency policy instance example

Now let us assume an execution of the Status Updater workflow: after the execution of the *Get Song* activity, the song information is sent to cache according to the $ap_{bestEffort}$ policy. Then, when the *Compute Mood* ends computing the user mood, the mood is also sent to the cache according to the $ap_{bestEffort}$ policy. Since there were no problems during the execution of these two activities, the workflow continues with the execution of the *Update Twitter* and *Update Facebook* activities. According to their associated policies (i.e., guaranteed policies), their initial state is sent immediately to disk.

Now assume that the *Update Facebook* activity commits but immediately after the workflow crash. This failure triggers the execution of the $ap_{bestEffort}$ and $ap_{guaranteed}$ policy and thus a crash recovery process is started. The execution state of previously executed activities is recovered according to their associated policies (e.g., the state of the activity *Compute Mood* is assumed to be committed. The states of the activities *Update Twitter* and *Update Facebook* are retrieved from disk). After recovering the activities, the workflow continues its execution, the *Update Twitter* commits and the workflow terminates.

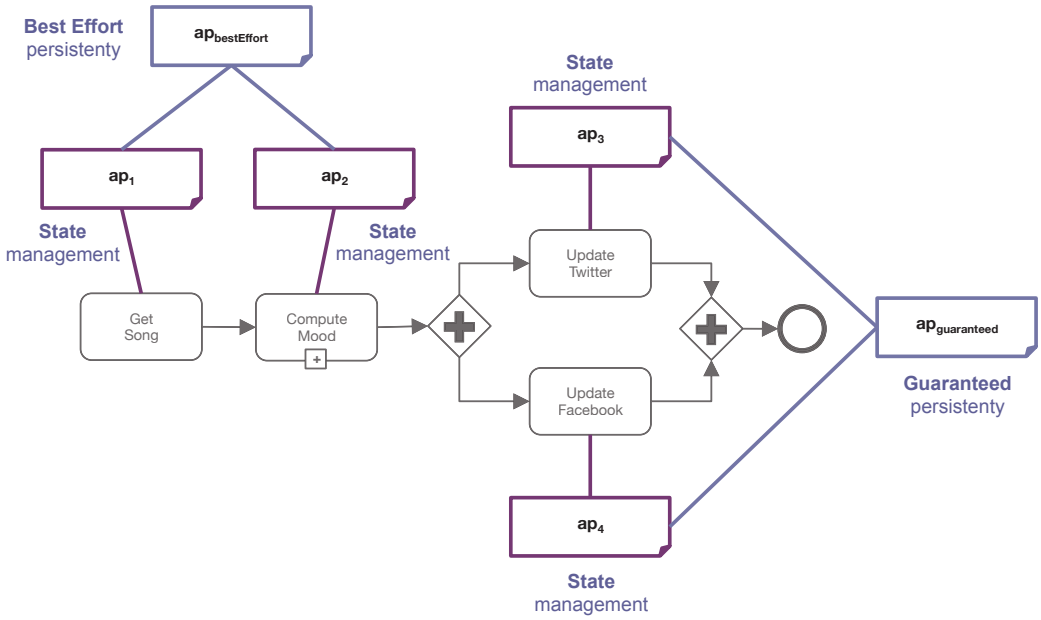

Figure 5.12 Example illustrating the Status Updater workflow with Persistency policies

5.4 Conclusions

This chapter described how to use the AP Model and language for defining exception handling, atomicity, state management and persistency NFP and how to associate them to a concrete service's coordination. Associating these types of AP to a service's coordination makes it reliable. According to our work a reliable service's coordination is able to maintain its functionality in unexpected situations (i.e., failures). Failures are of two kinds: semantic and system failures. Reliability is relevant due to current business oriented nature of services based applications. It provides QoS to applications by means of ensuring access to resources in a continuous way.

Several works have been devoted to provide reliability to applications using several approaches (e.g., see **[AFHL99, BhGP05, BoCR05, Bhir05, Boni00,PiBM03, SABS02, TMWD04, ViVo04, ZHMS06, DFDB05, HrWi05, LASS00, LiWe03, Lome04, NFGJ05, Oasi02b, Oasi09a, Oasi09b, ViVo04, SABS02, Wc00a]**). The way in which reliability is addressed, impacts the definition itself of the aspects (i.e., the level of abstraction used for its definition) and the type of developed application (i.e., the level of adaptability, maintenance and evolution of the application). In our approach we showed that it is possible to define NFP by defining AP types and then composing them with other AP types. For example we used exception handling AP types for defining atomicity AP types. An application developer can choose to associate only exception handling to an application or atomicity, just by specifying the appropriate associations.

Intentionally left blank

6 Implementation and Validation

This chapter describes the general architecture of our system Violet for executing active based workflows. It describes the *experimental validation* we conducted for adding reliability and freshness to *data services' coordinations* at runtime using the policies we introduced in **Chapter 5**.

The chapter is organized as follows. **Section 6.1** describe the general architecture of Violet our system for executing active policy based workflows. **Section 6.2** introduces the notion of query workflow and illustrates its use through the Visual Music Player application. Finally **section 6.3** concludes the chapter.

6.1 Violet: an Active Policy Based Workflow Engine

In order to execute *active policy based workflows* (i.e., workflow with associated active policies) we designed and implemented **Violet**, an *active policy based workflow engine*. The diagram of **figure 6.1** shows **Violet** general architecture. As shown in the diagram, **Violet** is composed of 3 components:

- **Workflow Engine**. Responsible of executing *workflow instances* and controlling their execution.
- **Active Policy Engine**. Responsible of managing and executing *active policy* and *rule* instances.
- **Event Service**. Responsible of monitoring and notifying the events produced by workflow and active policy instances.

Note that **Violet** has an external component called **Definition Tool**. This component is responsible of (i) receiving workflows and active policy definitions and (ii) parsing them for generating the corresponding intermediate representations used by the **Workflow Engine** and **Active Policy Engine**. Also note that the **Workflow Engine** and **Active Policy Engine** components are the main components of **Violet** since they are responsible of executing the AP Model execution units. In what follows we describe the architecture of these components.

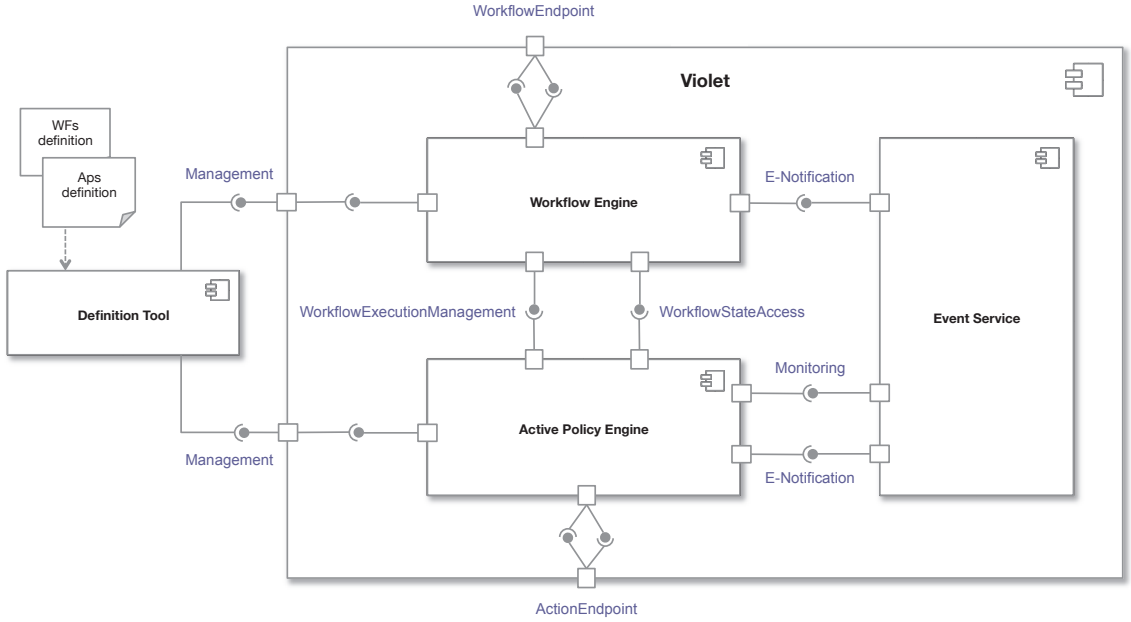

Figure 6.1 Violet general architecture (UML component diagram)

6.1.1 Workflow Engine

As shown in the diagram of **figure 6.2**, the **Workflow Engine** is composed of three main components:

- **Manager**. Responsible of *controlling* the lifecycle of workflow instances (e.g., it can *pause* and *resume* a workflow and/or its associated activities). The **Manager** is also responsible of controlling the execution of the **Scheduler** and **Tracker** components (e.g., it can interrupt their execution).
- **Scheduler**. Responsible of *executing* workflows' activity instances in the appropriate order. In particular, the **Manager** decides when to prepare workflow' atomic activities and when to call their associated services' operations.
- **Tracker**. Responsible of *monitoring* the execution of workflow and activity instances (e.g., when a workflow produces an event the **Tracker** detects the event and then it forwards the event to the **Event Service**.

The diagram of **figure 6.2** also shows that the **Workflow Engine** exposes several interfaces. The operations offered by each interface are shown in **figure 6.3**. The following list describes the **Workflow Engine** interfaces:

- **Management Interface** offers operations for (i) controlling the execution of a **Workflow Engine** and (ii) creating workflow instances. For instance, the operation *StartEngine* starts the execution of the **Workflow Engine** and specifies the type of the instances that the **Workflow Engine** will execute (e.g., instances of the **Status Updater** workflow). The operation *CreateWorkflow* creates a workflow instance of a specific *WorkflowType* that is uniquely identified by a *WorkflowInstanceID*.
- **WorkflowExecutionManagement Interface** offers operations for *controlling* the execution of a specific workflow instance. For instance, the operation *StartWorkflow* begins

the execution of the workflow instance identified by a *WorkflowInstanceID*. The operation *PauseWorkflow* stops the execution of a workflow instance identified by an ID while the operation *ResumeWorkflow* continues its execution from the point the workflow left before been paused.

- **WorkflowStateAccess Interface** offers operations for *retrieving* and *updating* the execution state of workflow instances. For instance, the operation *UpdateWorkflow* changes the state of a workflow identified by a *WorkflowInstanceID* with custom *WorkflowChanges*. The *GetWorkflow* operation retrieves reference to a *WorkflowInstance* based on its ID.
- **Monitoring Interface** offers operations for *monitoring* the execution of workflow instances. For instance, the operation *Subscribe* allows an external entity (e.g., the **Event Service**) to express its will for receiving the *events* produced during the execution of a workflow. Then, when the **Tracker** detects an event, it notifies the entity by using a *Callback*.

Finally note that the diagram of **figure 6.2** shows an interface called *WorkflowEndpoint*. This interface is not part of the **Workflow Engine** itself but an interface representing the capability of a workflow for *sending* and *receiving* messages (representing operation calls) to third party entities. For instance, using this interface a workflow can call the operations exported by a **Workflow Engine** for controlling the execution of another workflow in *synchronous* or *asynchronous* way.

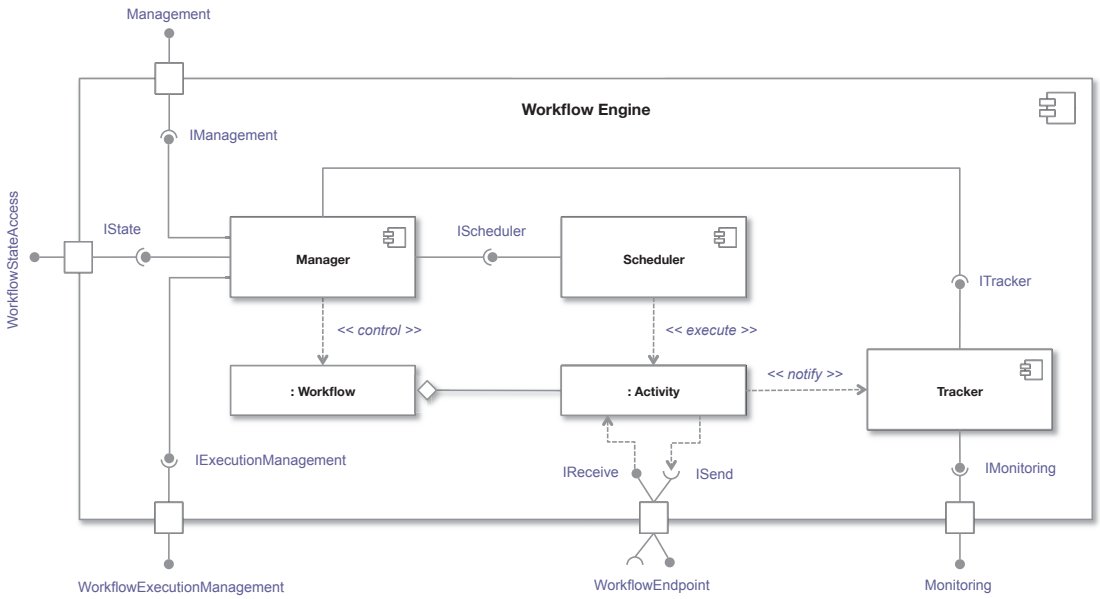

Figure 6.2 Workflow Engine Architecture (UML component diagram)

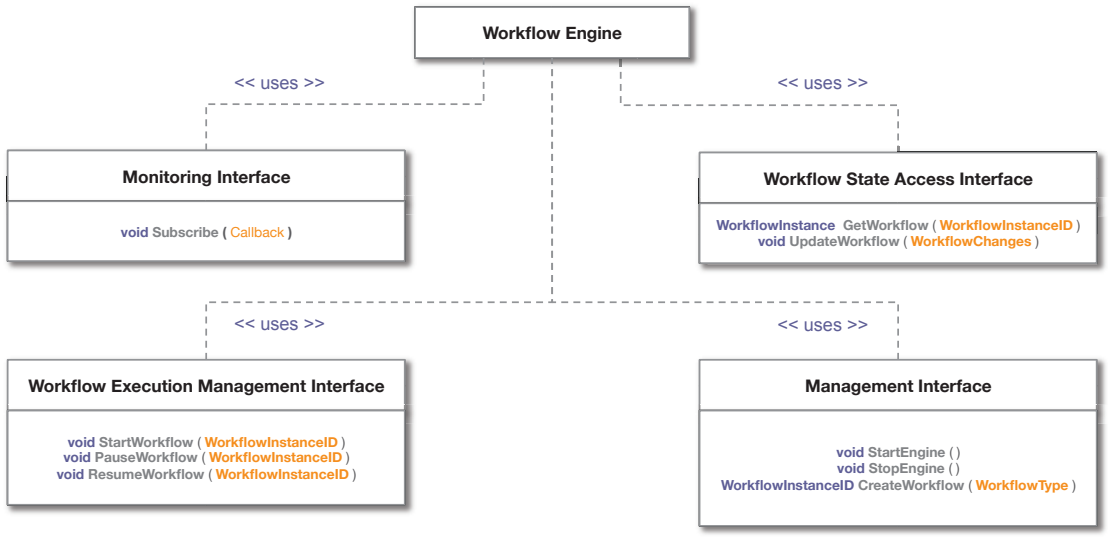

Figure 6.3 Workflow Engine exported interfaces (UML class diagram)

In this work we consider that any existing **Workflow Engine** respecting the architecture described above (the components and the interfaces) can interact with the **Active Policy Engine** of **Violet** as long as it complies with the following characteristics:

- **Preemption right**. This enables the *interruption* of the execution a workflow instance at different points in time. Thus it is possible to synchronize the execution of an active policy (and its rules) with the execution of the workflow' activity instances.
- **Activity atomic execution**. The execution of workflow activities is *atomic* (i.e., the activity either *commits* or *fails*).
- **Mutable state**. The execution state of the workflow can be modified arbitrarily. This is necessary because after the execution of some rule actions the workflow execution state may change (e.g., when an activity is successfully retried with different input values).

6.1.2 Active Policy Engine

As shown in the diagram of **figure 6.4**, the **Active Policy Engine** is composed of three main components:

- **Manager**. Responsible of *creating* active policy instances and *managing* their lifecycle (e.g., it activate and deactivate the policies).
- **Scheduler**. Responsible of processing *event notifications* for determining (i) whether a policy rule has to be triggered and (ii) the order of execution among the triggered rules. This component is also responsible of *synchronizing* the execution of a rule with the execution of the entity producing the event (e.g., if a workflow produced an event that triggered a rule, the **Scheduler** uses the **WorkflowExecutionManagement** interface for *pausing* the workflow execution and *resuming* it after the rule completes).
- **Action Engine**. Workflow engine responsible of executing the *action* of a policy rule (i.e., it executes the workflow implementing a rule' action).

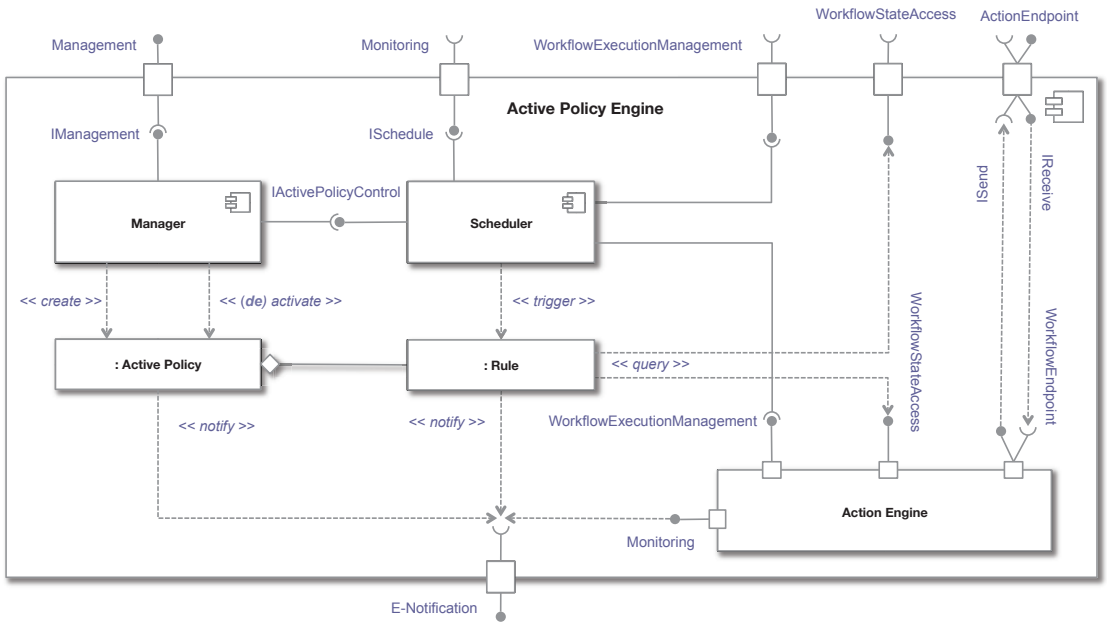

Figure 6.4 Active Policy Engine architecture (UML component diagram)

Note that the **Active Policy Engine** offers several interfaces. The following list enumerates the most important interfaces:

- **Manager Interface**. Offers operations for controlling the **Active Policy Engine**. In particular this interface is used for controlling the **Manager** component (e.g., using this interface it is possible to deactivate *all* the policies associated to a workflow).
- **E-Notification Interface**. Offers operations for notifying the events produced by *active policies*, *policy' rules* and *workflow' action* instances to the **Event Service** component (cf. **figure 6.1**).
- **Monitoring Interface**. Offers operations for receiving the events coming from the **Event Service** (cf. **figure 6.1**). These events are then sent to the **Scheduler** component for processing.

In order to execute active policies the **Active Policy Engine** considers the following aspects:

- When to evaluate the rule conditions with respect to the notification of an event.
- How policies triggered at the same time are ordered.
- How to interact with the execution of a workflow for executing its associated policy rules (preemption).

Based on these aspects the **Active Policy Engine** works as follows: given a workflow implementing a services' coordination and a set of active policies representing a non-functional property (cf. **figure 6.1**), the **Active Policy Engine** generates the code for evaluating every policy as well as the synchronization interface that interacts with the **Workflow Engine**. The code generation process is done by a *policy compiler*, which validates whether type declarations and policy expressions are well formed. The compiler also implements the transformation rules of AP Language expressions into AP Model types. The **Active Policy Engine** imports these types and implements evaluation strategies for executing the policies. Then it uses the **Event**

types defined in the policy types for constructing the **Monitoring** and **E-Notification** interfaces since *event detection* and *event notification* are specified on these interfaces.

Active policies express non-functional properties that must be evaluated at execution time and within the execution of a workflow. We represent the execution of a workflow by a *plan* consisting of a set of *execution units* (i.e. activities and policies) and an order function. This plan is generated and used by the **Scheduler** when policies are triggered. The evaluation of a policy is done within two processes: *events detection* and *execution strategies*.

Events detection is used to observe the execution of an *active policy based workflow* and *produce*, *detect* and *store* the events produced by this execution in a log. The log stores ordered instances of events produced during the execution of a workflow. The log also represents the execution state of a workflow and is used to decide *how* to evaluate policies.

While a workflow is being executed, the **Scheduler** consumes the events that can trigger the policies associated to the workflow. Using the properties specified in these policies, the **Scheduler** uses the execution strategies for filtering the events and deciding which policies are triggered. Recovery actions are seen as activities that must be inserted into the execution of a workflow.

The evaluation of policies within the execution of a workflow introduces a *best effort* execution strategy. Indeed, the active policy evaluation is a process executed before the execution of the activities of a workflow and can lead to the *cancellation*, *re-execution* or *compensation* of an activity. If for any reason an event representing an exception is detected, the set of recovery actions will be triggered to treat the exception.

6.2 Visual Music Player Application

In order to validate our approach we developed the **Visual Music Player** application, a music player web application that offers controls for searching and playing songs and which is implemented as a set of *active policy based workflows*.

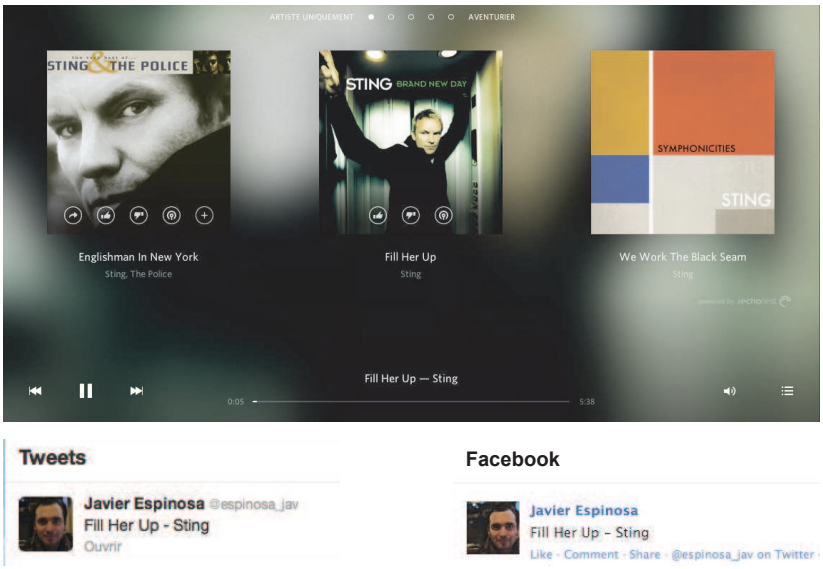

Figure 6.5 Visual Music Player example

The **Visual Music Player** works as follows: once a user has selected the music he/she wants to play, the player starts receiving the music and its associated metadata (e.g., song title, album and artist names). Then, based on the played song metadata, the application (i) constructs and displays a *slide show* with images related to the artist and (ii) updates the user' social network status for informing his/her friends about his/her musical activity. For instance, if the user is listening the live version of *Fill Her Up* of *Sting*, the application will construct a slide show with images of Sting. Then, it will update the user Facebook and Twitter status with the string *"Fill Her Up – Sting"*. This is illustrated in **figure 6.5**.

6.2.1 Architecture

Figure 6.6 shows the architecture of the **Visual Music Player** application. As shown in the figure, our application is a *client-server* application. The **client** (web browser) is responsible of handling the *presentation logic* used for displaying images and the player controls. The **server** is responsible of handling the application *business logic* (i.e., the logic that is not related with presentation aspects and that is shared among multiple clients). In particular, the *client* is responsible of retrieving the images used for constructing the slide-show, and the *server* is responsible of *filtering* and *ordering* the images as well as updating a user' social networks status.

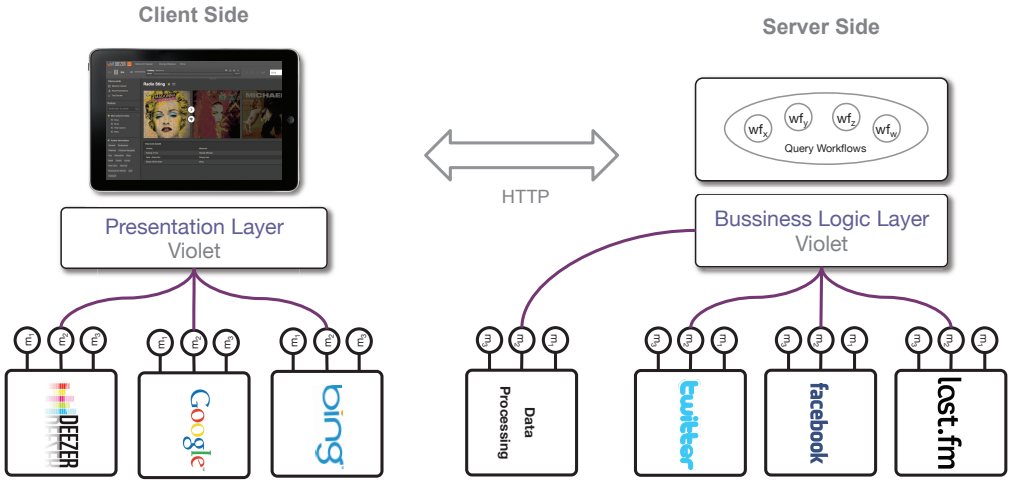

Figure 6.6 Architecture of the Visual Music Player application

Figure 6.6 also shows the *services* in which the **Visual Music Player** application is based on. The following list enumerates and describes each of these services.

- **Deezer** service. Exports operations for listening music continuously. The service also provides operations for accessing metadata about the played songs (e.g., song' duration, artist and album).
- **LastFM** service. Exports operations for accessing and updating the history of songs played by a user on a music service (e.g., Deezer, Rdio, Spotify).
- **Google** and **Bing** services. Export operations for searching images, videos and documents based on keywords.
- **Facebook** and **Twitter** services. Export operations for accessing and updating a user profile (e.g., the status of a Facebook user).

- **Data Processing** service. Exports operations for *filtering* and *ordering* collections. For instance, for filtering a collection of images based on their *resolution*.

Based on the operations used by the **Visual Music Player** application, these services can be classified in the *data service types* shown in **table 6.1** (cf. **section 2.1** service taxonomy). For instance, the services **Google**, **Bing**, **Deezer** and **LastFM** are examples of *data provision services* since the application retrieves information from them. In particular the **Google**, **Bing** and **LastFM** services are examples of *on-demand* services since access to images and to the played songs history is done explicitly every time they are required. In a similar way, the **Deezer** service is an example of a *streaming service* because it subscribes to the service before starts receiving the music.

Also note that even if the **Facebook** and **Twitter** services offer operations for accessing and updating user profiles, they are classified as *data administration* services and not as *data services*.[1] This is due to the fact that the application only uses the operations intended for updating the user profiles and not for retrieving them. In contrast, the **LastFM** service is an example of a *data service* since the application retrieves and updates the user history of played songs (i.e., the application reads and writes the history). Finally note that **Data Processing** is the only *data processing service* used in the application.

Service	Data Provision		Data Administration	Data Processing
	Streaming	On-demand		
Google		X		
Bing		X		
Deezer	X			
LastFM		X	X	
Facebook			X	
Twitter			X	
Data Processing				X

Table 6.1 Data services used by the Visual Music Player

6.2.2 Application Logic

As stated at the beginning of the chapter, we implemented the **Visual Music Player** application using *active policy based workflows*. **Figure 6.7** shows the workflows implementing the application:

- The **Visualization Workflow** is responsible of constructing and displaying the *slide show* on a client.
- The **Query Workflow** is responsible of filtering and ordering the images used for constructing the slide show. This workflow resides on the server side.

[1] Recall that a *data service* is a service classified as a *data provision* and *data administration* service (i.e., a service that offers operations for reading and modifying a collection of data).

- The **Status Updater Workflow** is responsible of updating the social networks status of a user with information about the song he/she is currently listening to. This workflow also resides in the server side.

Figure 6.8 illustrates the general interaction among these workflows. The figure is interpreted as follows: once the application starts receiving the music from the **Deezer** service, the **Visualization Workflow** retrieves the metadata of the current played song using the *Get Song* activity. Then, using this meta-data, the workflow constructs the slide show and update the user social networks in parallel.

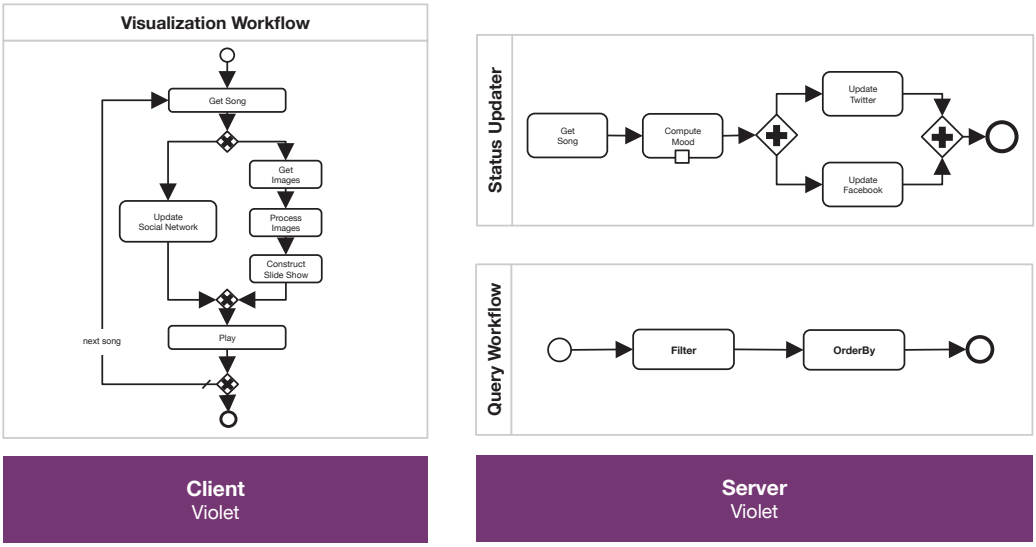

Figure 6.7 Workflows implementing the Visual Music Player

For updating the user social networks, the **Visualization Workflow** sends the *song title* and *artist name* to the **Status Updater Workflow** using the *Update Social Network* activity.[2] Then, the **Status Updater Workflow** uses the **Facebook**, **Twitter** and **LastFM** services for updating the user Facebook and Twitter profiles as well as the history of *user' played song* in LastFM. Once the **Status Updater Workflow** completes its execution, the **Visualization Workflow** receives a notification and continues its execution.

For constructing the slide show the **Visualization Workflow** retrieves the artist related images from the **Google** service using the *Get Images* activity. Then it uses the *Process Images* activity for sending the images to the **Query Workflow**. At this point the **Query Workflow** filters and orders the images based on their *size* and *resolution* using the **Data Processing** service. Then the **Query Workflow** sends back the processed images to the **Visualization Workflow**. At this point the **Visualization Workflow** uses the *Construct Slide Show* activity for constructing the slide show using the filtered and ordered images. Finally the workflow uses the *Play* activity for playing the slide show until the next song starts.

[2] Recall that the *Status Updater* workflow was introduced and described in **chapter 1**.

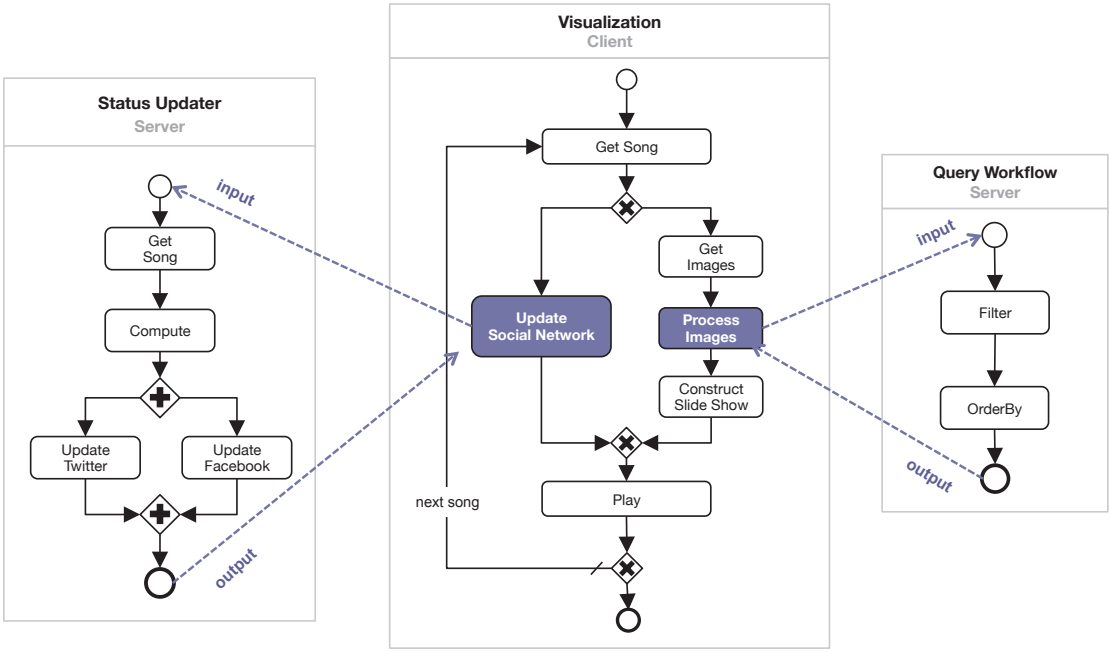

Figure 6.8 Visual Music Player workflows interaction

As shown in figure 6.8, all the logic used for filtering and ordering the images used by the application resides in **Query Workflow**, a special kind of workflow used for processing collections of data **[Cuev11]**. A query workflow has the following characteristics:

- Receives (produces) as input (output) a *collection of data*.
- Only coordinates calls to *data processing services*.
- It is composed of atomic activities called *data operator activities* that only call operations of data processing services (e.g., join activity, filter activity, order-by activity).

For instance, figure 6.9 illustrates the internals of **Query Workflow**. As shown in the figure, **Query Workflow** is composed of two activities: the **Filter** and **OrderBy** activities, both calling operations of the **Data Processing** service. The **Filter** activity calls the *filter* operation and passes as input (i) the *collection of images* to be filtered and (ii) a string representing the *filtering condition* (e.g., "image.size > screen.resolution"). The **OrderBy** activity calls the *orderBy* operation and also passes as input (i) a collection of images and a (ii) string representing the criteria used for ordering the images (e.g., "order by size asc"). Note that both operations (i.e., the *filter* and *orderBy* operations) produce as result a collection of elements of type **T**. In this case **T** represents the filtered and ordered images received by the **Filter** and **OrderBy** activities (respectively).

Figure 6.10 further details the **Query Workflow** by showing how data flows among the workflow activities. The figure is interpreted as follows:

1. **Query Workflow** receives from **Visualization Workflow** the images to be filtered and the client' *screen resolution*. They are saved in the workflow *images* and *screenResolution* input variables (respectively).

2. **Query Workflow** computes the value for the local variable *conditionExp*. This variable is used for containing the images filtering condition (e.g., for filtering the images having a resolution similar to the client screen-resolution, the workflow uses the value in *screenResolution* for constructing the expression '*image.size==1024x768*'). Then, the workflow passes the values of *images* and the *conditionExp* variables as input of the **Filter** activity.
3. When the **Filter** activity completes its execution (i.e., once it receives the filtered images from the *filter* operation), **Query Workflow** uses the activity output as input for the **OrderBy** activity. At the same time, the workflow assigns the ordering criteria to the variable *orderByExp* (e.g. in order to sort the images ascending order, the workflow assigns *orderByExp* the value "*order by image.size asc*"). Then it passes the value of *orderByExp* to the **OrderBy** activity.
4. Finally, when the **OrderBy** activity completes its execution, **Query Workflow** takes the set of images produced by the activity (i.e. **OrderBy** output) and sends them back ordered to the **Visualization Workflow**.

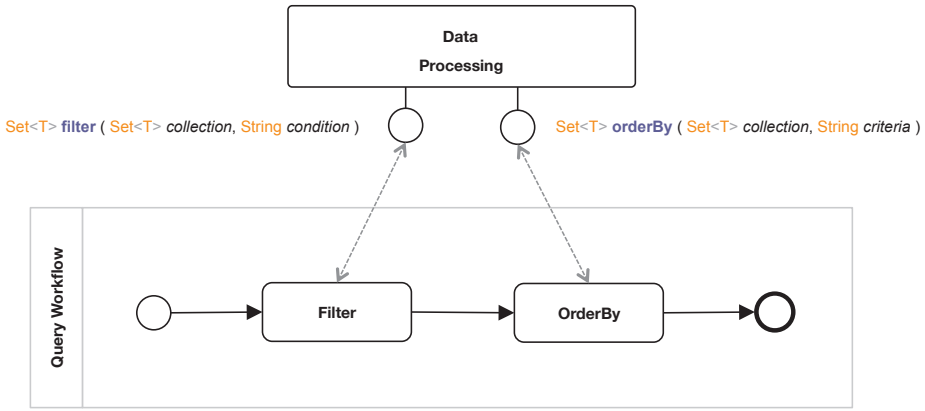

Figure 6.9 Visual Music Player Query Workflow

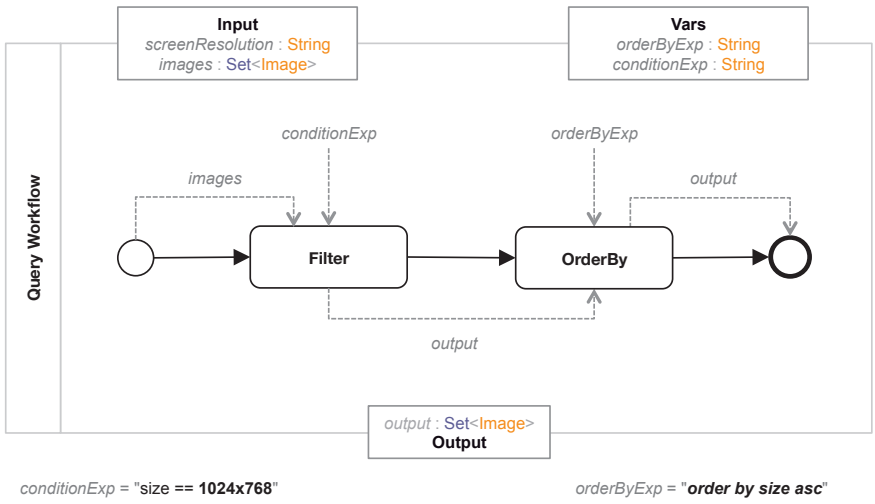

Figure 6.10 Visual Music Player Query Workflow Dataflow

As an example, let us assume that **Query Workflow** receives as input a collections of images having the *structure* shown in **listing 6.1** (i.e., a collection of images is represented as a set of values of type **Image**, where each value has a *name*, a *size* and a *resolution*). Also assume that the expressions used for *filtering* and *ordering* the images are the string values "*image.resolution == 1024x768*" and "*order by image.size asc*" respectively (i.e., **Query Workflow** filters images not having a resolution equal *1024x768* and it orders the resulting images based on their size: first the smaller images, then the largest). Finally assume that **Query Workflow** receives as input the set of values shown in **listing 6.2**. **Listing 6.3** shows the collection of images received by **Visualization Workflow** once **Query Workflow** completes its execution

```
Images {
    Image < name : String, uri : String, size : Number, resolution : String >
}
```

Listing 6.1 Query Workflow input structure

```
Images {
        Image < name = "P1", uri = "/Photos/P1", size = 1000, resolution = "1024x768"> ,
        Image < name = "P2", uri = "/Photos/P3", size = 1200, resolution = "640x960"> ,
        Image < name = "P3", uri = "/Photos/P2", size = 300, resolution = "1024x768">
}
```

Listing 6.2 Query Workflow input example

```
Images {
        Image < name = "P3", uri = "/Photos/P2", size = 300, resolution = "1024x768"> ,
        Image < name = "P1", uri = "/Photos/P1", size = 1000, resolution = "1024x768">
}
```

Listing 6.3 Query Workflow output Example

In our approach, we assume that operations exported by *data services* are also implemented as workflows coordinating calls to services' operations. For instance, **figure 6.11** shows the implementation of the *filter* operation exported by the **Data Processing** service as a workflow called **Filter**. As shown in the figure, **Filter** (i) *receives* as input a *collection* of **N-Tuples** and a string representing the *filtering expression*, and (ii) *produces* as output another collection of **N-Tuples**. The workflow works as follows: when **Filter** is called, the workflow uses the *condition expression* for determining whether a tuple in the input collection has to be filtered or not. If a tuple has to be filtered, it is just discarded and then the workflow continues with the next tuple in the input collection. Otherwise it inserts the tuple into the output collection and continues this process until all the tuples in the input collection are analyzed.

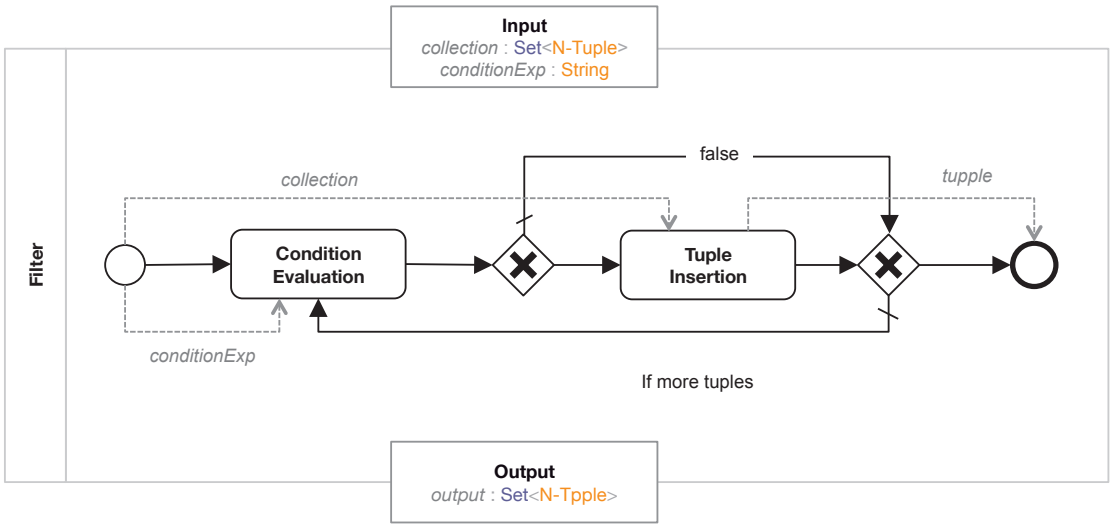

Figure 6.11 Workflow Implementing a Filtering Operation

6.2.3 Active Policies for Data Services

As shown in **figure 6.7**, it is possible to define active policies types for associating the following non-functional properties to the **Visual Music Player.**

Authentication

In order to protect users' privacy the **Facebook** and **Twitter** services restrict access to their data to authorized users by using authentication protocols. Since these services may change their protocols over time for reinforcing users' privacy, the **Visual Music Player** has to be able to adapt it self at runtime for coping with different authentication protocols. For instance, let us consider that Twitter has decided to change its basic authentication protocol (i.e., the protocol based on username and password) for the *OAuth protocol*. In order to handle the new authentication requirements, the **Status Updater Workflow** has to implement the *OAuth protocol* on the fly in order to continue working. Thus we use the **OAuth** policy type defined in **section 3.6.3** for implementing this authentication protocol. This is illustrated in **figure 6.12**.

Fault Tolerance

In order to provide fault tolerance (i.e., reliability) to the **Visual Music Player** we address the following aspects:

- **Adaptability**. If for any reason the **Google** service is not available (e.g., due to network unavailability), the **Visualization Workflow** should be able to replace the service with the **Bing** service. For this we defined the **Adaptability** policy type shown in **listing 6.4**. The policy works as follows: once an **Adaptability** policy instance is associated to the *Get Images* activity, the rule *ReplaceGoogleService* replaces the **Google** service with the **Bing** service in the presence of a failure of the *Get Images* activity. **Figure 6.12** illustrates the association of the **Adaptability** policy to the **Visualization Workflow**.

- **Retry**. In a similar way, if the services **Twitter** and **Facebook** are not available, the **Status Updater Workflow** has to postpone the update of the user' status for some time and then retry the update as many times as possible as long as the song has not ended (e.g., if the song has a duration of 5 minutes and the server retries the update every minute, the server may retry the update up to 4 times). We shown in **section 5.2.1** the use of the **Activity Exception Management** for retrying activities in the case of failures. **Figure 6.12** shows the association of the **Activity Exception Management** to the **Status Updater Workflow** for implementing the required behavior.
- **Rollback**. Since the **Query Workflow** in charge of filtering and ordering the images may fail, the server should be able to restart the workflow from the last stable point (i.e., last *checkpoint*). For instance, if the **Query Workflow** ends filtering the images but fails before ordering them, the workflow has to be able to restart the query workflow at the ordering stage and using the already filtered images. We shown in **section 5.3.2** the use of the **Persistency** and **Activity State Management** policy types for providing persistency and recovery to the execution of a workflow. **Figure 6.14** illustrate the use of these policies for implementing the persistent behavior required by the **Query Workflow**.

Atomicity

In order to have a consistent view about the user musical activity, the server has to ensure that the user Twitter and Facebook status contain both the same song' information. As an example assume that both status display the string "*Exogenesis (Part I) - Muse*" and that the server tries to update them with the string "*Exogenesis (**Part II**) - Muse*". Let us assume that due to a Twitter error only the Facebook status is updated. In this scenario the server has to *undo* the update in order to display the previous string in both status and assure the atomic behavior (i.e. either all of the status are updated or none of them). We already showed in **section 5.2.2** how the use of the **Atomicity** policy type for providing different kinds of atomic behavior. **Figure 6.12** illustrates the use of the **Atomicity** policy for providing atomicity to the **Status Updater Workflow**.

```
policy class Adaptability |[ Activity activity ]| : ActivePolicy {

  String bingService = "http://www.bing.com/api";

  rule ReplaceGoogleService
    on ActivityFailed
    if event.producedBy ( scope )
    do ChangeService ( event.producer, bingService )

}
```

Listing 6.4 Definition of the Adaptability policy type

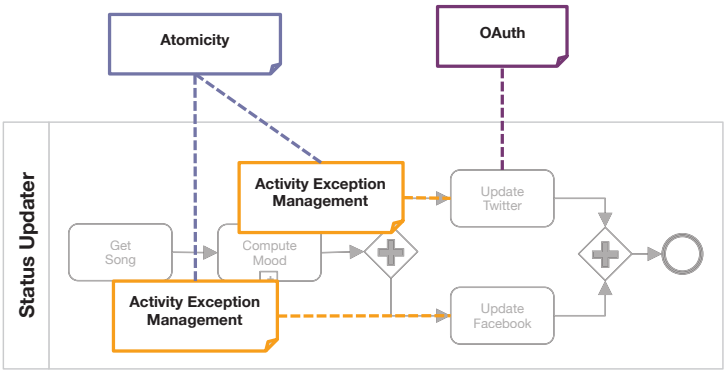

Figure 6.12 Status Updater Workflow with associated active policies

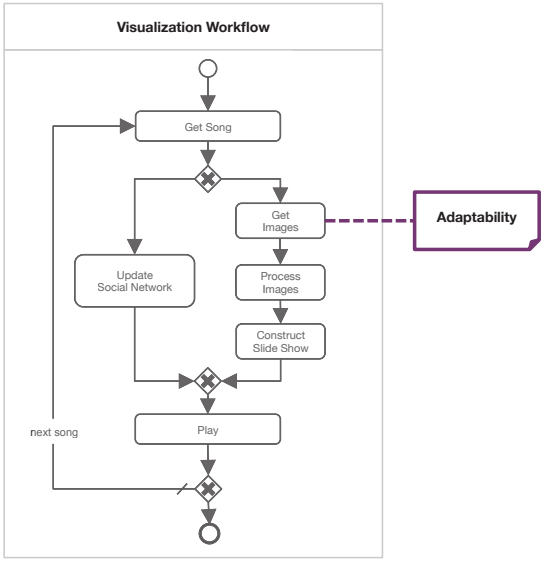

Figure 6.13 Visualization Workflow with associated active policies

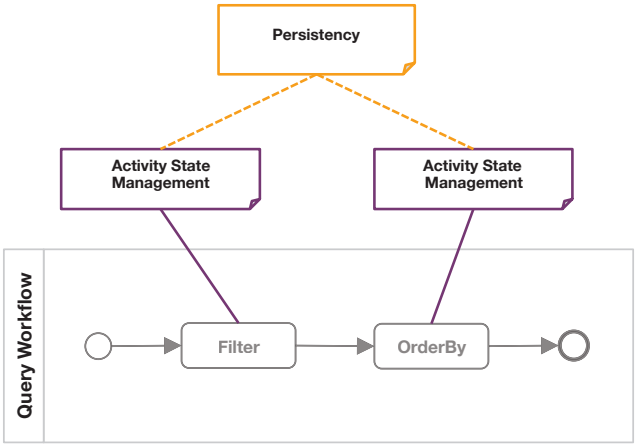

Figure 6.14 Query Workflow with associated active policies

6.3 Conclusions

This chapter first presented the system Violet for executing AP based workflows that consists of a **WF Engine** and an **AP Engine.** The **AP Engine** manages the execution of a set of AP instances and controls the execution of a **WF Engine**. Violet's architecture respects the separation between the executions of a-policies and the workflow. The system can work with any workflow engine as long as it provides an interface for controlling the execution of a workflow. Thus, the WF engine can be coupled with an AP engine for ensuring NFPs.

The chapter then presented the experiment we conducted for validating our approach. It focused on a web application where the operations were executed ensuring fault tolerance (exception handling and recovery properties that lead to atomicity models) and security properties. We developed a second scenario focusing on data processing ensured by coordinating services described in **Appendix B**. The objective was to focus on QoS properties to be ensured to data collections produced by services, and processed through workflows. For example ensuring their freshness and their validity. We showed how to use policies for ensuring these properties.

7 Conclusions and Perspectives

A services' coordination has associated functional (i.e. what services actually do) and non-functional properties (i.e. the expected behavior of a service at execution-time). NFPs are properties that are not directly related to the functionality offered by a service (e.g., cost, security, reliability, efficiency, complexity and adaptability). Often NFPs are associated to the communication infrastructure, the service provider or the way a service behaves at execution time. For example, message encryption for protecting the information exchanged; verification of service authenticity for identified trusted services; exception management for warrantee a minimum level of Quality of Service (QoS).

When defining an application based on services' coordination, it is fundamental to model, manage and perform the discovery, composition, negotiation and the agreement based on the NFPs. This is achieved through the specification, reinforcement and managing of agreements known as Service Level Agreements (SLAs). Respecting a set of SLAs assures that a set of services will work in an acceptable way. Besides, given the increase of services being offered by different service providers and the popularity of the Internet, it becomes necessary to have models of NFPs to construct application based on services' coordination taking into account the different SLA of each application.

The layered classification of current standards shows that functional, non-functional and communication aspects in services' coordination are implemented by combining different languages and protocols. The objectives of this work were to propose a model that could enable adding and modifying NFP without having to modify the application logic implemented by the service coordination.

7.1 Results and Contribution

In this thesis we presented the AP Model for associating non-functional properties (NFP) to a services' coordination by means of *active policies*. We also presented strategies for synchronizing the executions of a service coordination implemented by a workflow and its associated active policies. In particular we focused on providing reliability and properties to services coordinations. Finally we proposed a system for showing the AP policies definition and evaluation strategies at run time. As an experimental validation we implemented a reliable services coordination execution engine called Violet that adds NFPs to a given services coordination. Our results were experimentally validated in use cases where services coordinations are used for querying, processing and mashing up data. In both use cases we validated the use of **Active Policy** types related to *fault tolerance*, *authentication*,

adaptability and QoS properties associated to the data they process (freshness, validity, durability). In our experience, using the AP Model and its associated language for specifying AP types that implement an NFP can be easy as long as the programmer knows the protocol to express: (i) the events that must be observed for triggering an AP, (ii) the conditions to be verified in the execution state of the coordination and (iii) the recovery actions to enforce the NFP. Policies provide independence of application logic and NFP and this eases the maintenance of the service coordination. This is important because services can change their interfaces and their requirements (e.g., they can change the authentication protocol used for executing operations). Having AP modularizes this aspects and help to maintain a service' coordination only by associating new policies to it.

The main contribution of our work is provide an AP Model and associated language for expressing policies that can be associated to activity oriented workflow models used for processing data and for implementing business processes. The AP model and language are also associated to strategies for executing APs and synchronizing their execution with the execution of a workflow. NFP are thereby added to service coordination without modifying their definition. NFP can be modified according to requirements associated to workflows and to the characteristics of their execution context (service modification or evolution).

7.2 Perspectives

Future work includes adding NFP to service coordinations used for implementing data management functions (e.g., querying, storage, cache, duplication).

7.2.1 Reliable Hybrid Query Evaluation

In the experimental validation of this work we started studying the way NFP can be added to query workflows that implement *hybrid queries*. Hybrid queries can be *recurrent* and *continuous*. They are implemented by query workflows that coordinate calls to *data providers* and *data processing* services. Query workflows execution for the time being are executed by an ad hoc engine called Hypatia. For the time being the hypothesis for executing query workflows is that services are available and that they do not have associated constraints regarding NFP (e.g., authentication protocols for calling data providers, exception handling, fault tolerance).

Besides, query workflows do not reason about data freshness, service unavailability, atomicity, and adaptability. Given that hybrid queries are executed under the hypothesis that no full-fledge DBMS is available for doing that. We believe that AP can be used for providing NFP to their execution. We will explore the extension of the query language HSQL **[Cuev11]** with our AP Language, and also the extension of the query engine Hypatia with an adapted AP execution engine. The work will require revisiting the architecture of the hybrid query and AP engines for avoiding overhead on the execution time due to the execution of APs.

7.2.2 Service Lookup and Recommendation

The proliferation of services available in Internet and in cloud platforms makes is difficult to have a clear idea of their purpose, exported function and quality. Building applications on top of these services is risky since the quality of the resulting system hardly depends on the quality of its components. Besides, given the autonomy of services, they can evolve or

be unavailable during certain periods of time and applications using them must resist to these kinds of exceptions by replacing them. Having efficient recommendation systems that can provide abstract views of annotated services and that can help to find services is still an open issue. Developing a research on this problem can be important particularly if the "user" requirements are coupled with technical requirements for guiding the recommendation process.

Today, with the emergence of cloud platforms that use services as construction units, recommendation systems can be useful to explore available services deployed through the different levels of these platforms. Particularly, approaches that rely on service coordination for evaluating queries can benefit from this kind of systems for integrating service discovery phases in the evaluation process for example if services are not available. The challenge is to provide service recommendation strategies based on QoS measures and SLA contracts in an efficient way for avoiding overhead to the query evaluation process.

The application of such a recommendation system can go beyond business applications, it can also be integrated in the construction of value added services willing to implement data management issues.

7.2.3 Policy Based Cloud Services Coordination

With the popularity of cloud computing it is now necessary to understand service coordination in such a context. Today DBMS architecture has evolved to the notion of service-based infrastructure where services are adapted and coordinated for implementing ad hoc data management functions (e.g., storage, fragmentation, replication, analysis, decision making, data mining). Services can be distributed and be made redundant by using several computers connected through a network. Having several services deployed in the cloud implies making decisions on how to coordinate them dynamically for fulfilling applications requirements considering the consumption of resources provided by the cloud. Having policies associated to services coordination can provide means to support automatic control of the execution of the service coordination. This control can ease the execution of applications on the cloud, that implies taking into consideration several variables for avoiding extra economic costs imposed by the cloud business model.

Intentionally left blank

Bibliography

[AAAB07] ALVES, Alexandre; ARKIN, Assaf; ASKARY, Sid; BARRETO, Charlton; BLOCH, Ben; CURBERA, Francisco: *Web services business process execution language version 2.0.* URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[ADHW02] VAN DER AALST, W.; DUMAS, M.; HOFSTEDE, A.H.M. ter; WOHED, P.: *Pattern-based analysis of BPML (and WSCI)*, Technical report FIT-TR-2002-05, Queensland University of Technology, Brisbane, 2002

[AgLS09] AGARWAL, Sudhir; LAMPARTER, Steffen; STUDER, Rudi: Making web services tradable: a policy-based approach for specifying preferences on Web service properties. In: *Journal of Web Sem.* 7 (2009), Nr. 1

[AHKB03] VAN DER AALST, W M P; TER HOFSTEDE, A H M; KIEPUSZEWSKI, B; BARROS, A P: Workflow patterns. In: *Journal of Distributed Parallel Databases* 14. Hingham, MA, USA (2003), Nr. 1

[BaKS10] BABAMIR, S.M.; KARIMI, S.; SHISHECHI, M.R.: A broker-based architecture for quality-driven web services composition. In: *Proc. of the Int. Conference on Computational Intelligence and Software Engineering (CiSE)*. Wuhan, 2010

[BaWR09] BARKER, A; WALTON, C D; ROBERTSON, D: Choreographing web services. In: *Journal of IEEE Transactions on Services Computing* 2 (2009), Nr. 2

[BBKL78] BOEHM, Barry W.; BROWN, J.R.; KASPAR, H.; LIPOW, M.; MACLEOD, G.J.; MERRITT, M.J.: *Characteristics of software quality* : North-Holland, 1978

[BDSN02] BENATALLAH, B.; DUMAS, M.; SHEN, M.; NGU, A. H. H.: Declarative composition and peer-to-peer provisioning of dynamic web services. In: *Proc. of the 18th Int. Conference on Data Engineering*. USA, 2002

[BGGL06] BUSI, Nadia; GORRIERI, Roberto; GUIDI, Claudio; LUCCHI, Roberto; ZAVATTARO, Gianluigi: Choreography and orchestration conformance for system design. In: *Proc. of the 8th Int. Conf. on Coordination Models and Languages*. Berlin,

Heidelberg : Springer-Verlag, 2006 — ISBN 3-540-34694-5, 978-3-540-34694-4

[BhGP05] Bhiri, Sami; Godart, Claude; Perrin, Olivier: Reliable web services composition using a transactional approach. In: *Proc. of the Int. Conference on e-Technology, e-Commerce and e-Service*. France, 2005

[BoCR05] Bocchi, Laura; Ciancarini, Paolo; Rossi, Davide: Transactional aspects in semantic based discovery of services. In: *Proc. of the 7th Int. Conference COORDINATION*. Belgium, 2005

[Bpmi04] BPMI: *Business process modeling language (BPML)*. URL http://www.bpmi.org

[BuSF06] Bultan, T; Su, Jianwen; Fu, Xiang: Analyzing conversations of Web services. In: *Journal of IEEE Internet Computing* 10 (2006), Nr. 1

[CaGP09] Castagna, Giuseppe; Gesbert, Nils; Padovani, Luca: A theory of contracts for Web services. In: *Journal of ACM Transactions on Programming Languages and Systems* 31 (2009), Nr. 5

[ChLa09] Chollet, Stéphanie; Lalanda, Philippe: An extensible abstract service orchestration framework. In: *Proc. of the IEEE Int. Conference on Web Services (ICWS)*. USA, 2009

[ChMe04] Charfi, Anis; Mezini, Mira: Hybrid web service composition: business processes meet business rules. In: *Proc. of the 2nd Int. Conference on Service Oriented Computing (ICSOC'04)*. USA, 2004

[ChMe07] Charfi, Anis; Mezini, Mira: AO4BPEL: An aspect-oriented extension to BPEL. In: *Journal of World Wide Web* 10 (2007), Nr. 3

[CIJK00] Casati, Fabio; Ilnicki, Ski; Jin, Li-jie; Krishnamoorthy, Vasudev; Shan, Ming-Chien: eFlow: a platform for developing and managing composite e-services. In: *Proc. of the Academia/Industry Workshop on Research Challenges (AIWoRC)*, 2000

[Coup96] Coupaye, Thierry: *Un modèle d'exécution paramétrique pour systèmes de bases de données actifs*, Université Joseph Fourier - Grenoble 1, 1996

[CSHM06] Charfi, Anis; Schmeling, Benjamin; Heizenreder, Andreas; Mezini, Mira: Reliable, secure, and transacted web service compositions with AO4BPEL. In: *Proc. of the 4th European Conference on Web Services (ECOWS '06)*, 2006

[Cuev11] Cuevas-Vicenttín, Víctor: *Evaluation de requêtes hybrides basée sur la coordination de services*, Thèse de doctorat, Université de Grenoble, 2011

[CyPr04] CYSNEIROS, Luiz Marcio; DO PRADO LEITE, Julio Cesar Sampaio: Non-functional requirements: from elicitation to conceptual models. In: *Journal of IEEE Transactions on Software Engineering* 30 (2004), Nr. 5

[DaLF93] DARDENNE, Anne; VAN LAMSWEERDE, Axel; FICKAS, Stephen: Goal-directed requirements acquisition. In: *Journal of Science of Computer Programming* 20, Elsevier Science Publishers (1993), Nr. 1-2

[Darp00] DARPA: *The DARPA agent markup language*. URL http://www.daml.org/ontologies/

[DeWe07] DECKER, Gero; WESKE, Mathias: Local enforceability in interaction Petri nets. In: *Proc. of the 5th Int. Conference on Business Process Management (BPM'05)*. Berlin, 2007

[DFDB05] DUARTE, Helga; FAUVET, Marie-Christine; DUMAS, Marlon; BENATALLAH, Boualem: Vers un modèle de composition de services web avec propriétés transactionnelles. In: *Journal of Ingénierie des systèmes d'information* 10 (2005), Nr. 3

[Dhon04] D'HONDT, Maja: *Hybrid aspects for integrating rule-based knowledge and object-oriented functionality*, Vrije Universiteit, Brussel, 2004

[DiSD02] DIMITROV, E.; SCHMIETENDORF, A.; DUMHE, R.: UML-based performance engineering possibilities and techniques. In: *Journal of IEEE Software* 19 (2002), Nr. 1

[DKLW07] DECKER, G; KOPP, O; LEYMANN, F; WESKE, M: BPEL4Chor: Extending BPEL for modeling choreographies. In: *Proc. of the IEEE Int. Conference on (ICWS)*. Salt Lake City, USA, 2007

[DuSc05] DUSTDAR, Schahram; SCHREINER, Wolfgang: A survey on web services composition. In: *International Journal of Web and Grid Services* 1 (2005), Nr. 1

[ELLR90] ELMAGARMID, Ahmed K.; LEU, Yungho; LITWIN, Witold; RUSINKIEWICZ, Marek: A multidatabase transaction model for InterBase. In: *Proc. of the 16th Int. Conference on Very Large Data Bases (VLDB '90)*. Brisbane, Australia, 1990

[FaLo10] FAHNDRICH, Manuel; LOGOZZO, Francesco: Static contract checking with abstract interpretation. In: *Proc. of the Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)* : Springer, 2010

[FePf96] FENTON, Norman E.; PFLEEGER, Shari Lawrence: *Software metrics: a rigorous and practical approach* : International Thomson Computer Press, 1996

[Frie12] FRIEDMANN-HILL, E.J.: *JESS: The java expert system shell*. URL http://herzberg.ca.sandia.gov/jess/

[GaMy00] GAL, Avigdor; MYLOPOULOS, John: Supporting distributed autonomous information services using coordination. In: *International Journal of Cooperative Information Systems* 9 (2000), Nr. 3

[GaSa87] GARCIA-MOLINA, Hector; SALEM, Kenneth: Sagas. In: *Proc. of the Int. Conference on Management of data (SIGMOD '87)*, 1987

[GaUW00] GARCIA-MOLINA, Hector; ULLMAN, Jeffrey D.; WIDOM, Jennifer: *Database system implementation* : Prentice Hall, 2000

[GeHS95] GEORGAKOPOULOS, Dimitrios; HORNICK, Mark F; SHETH, Amit P: An overview of workflow management: from process modeling to workflow automation infrastructure. In: *Journal of Distributed and Parallel Databases* 3 (1995), Nr. 2

[GeKT96] GEPPERT, A.; KRADOLFER, M.; TOMBROS, D.: *EVE: Event engine for workflow enactment in distributed environments*, Technical Report 96.05, University of Zurich, Switzerland, 1996

[GGKK91] GARCIA-MOLINA, Hector; GAWLICK, Dieter; KLEIN, Johannes; KLEISSNER, Karl; SALEM, Kenneth: Modeling long-running activities as nested sagas. In: *Journal of Data Engineering* 14 (1991), Nr. 1

[GrYu01] GROSS, Daniel; YU, Eric: From non-functional requirements to design through patterns. In: *Journal of Requirements Engineering* 6 (2001), Nr. 1

[GuRF09] GUTIÉRREZ, Carlos; ROSADO, David G; FERNÁNDEZ-MEDINA, Eduardo: The practical application of a process for eliciting and designing security in web service systems. In: *Journal of Information and Software Technology* 51 (2009), Nr. 12

[Hall01] HALLE, Barbara von: *Business rules applied: building better systems using the business rules approach* : John Wiley & Sons, 2001

[Hay00] HAY, David et al: *Defining business rules: what are they really?*, 2000

[HeBW06] HENKEL, Martin; BOSTRÖM, Gustav; WÄYRYNEN, Jaana: Moving from internal to external services using aspects. In: *Interoperability of Enterprise Software and Applications* : Springer, 2006 – ISBN 978-1-84628-151-8

[Hern94] HERNANDEZ, D: *Qualitative representation of spatial knowledge* : Springer, 1994 – ISBN 978-3-540-58058-4

[HrWi05] HRASTNIK, P.; WINIWARTER, W.: TWSO-Transactional web service orchestrations. In: *Proc. of the Int. Conference on Next Generation Web Services Practices (NWeSP'05)*. Korea, 2005

[Ibm00a] IBM: *Web Intermediaries (WBI)*. URL http://www.almaden.ibm.com/cs/wbi/

[Ibm00b] IBM: *ILOG JRules*. URL http://www.ilog.com/products/jrules/

[JeCL09] JEONG, Buhwan; CHO, Hyunbo; LEE, Choonghyun: On the functional quality of service (FQoS) to discover and compose interoperable web services. In: *International Journal of Expert Systems Applications* 36 (2009), Nr. 3

[KaPi06] KAZHAMIAKIN, Raman; PISTORE, Marco: Analysis of realizability conditions for web service choreographies. In: *Proc. of FORTE'06*. Berlin, 2006

[KaRR98] KAPPEL, G.; RAUSCH-SCHOTT, S.; RESSCHITZEGGER, W.: *Workflow Management Frameworks* : Wiley & Sons, 1998

[KiDa96] KIRNER, Tereza G.; DAVIS, Alan M.: Non-functional requirements of real-time systems. In: *Journal of Advances in Computers* 42 (1996)

[KPRR95] KAPPEL, G.; PROLL, B.; RAUSCH-SCHOTT, S.; RESSCHITZEGGER, W.: TriGSflow: Active object-oriented workflow management. In: *Proc. of the Hawaii Int. Conference on System Sciences*. USA, 1995

[Ladd03] LADDAD, Ramnivas: *AspectJ in action: practical aspect-oriented programming* : Manning Publications, 2003 – ISBN 1930110936

[Lams01] VAN LAMSWEERDE, A.: Goal-oriented requirements engineering: a guided tour. In: *Proc. of the 5th IEEE Int. Symposium on Requirements Engineering*. Washington, USA : IEEE Computer Society, 2001

[LASS00] LAZCANO, A; ALONSO, G; SCHULDT, H; SCHULER, C: The WISE approach to electronic commerce. In: *International Journal of Computer Systems Science & Engineering, special issue on Flexible Workflow Technology Driving the Networked Economy* 15 (2000)

[LCRS01] LEYMANN, Frank; CURBERA, Francisco; ROLLER, Dieter; SCHMIDT, Marc-Thomas; KLOPPMANN, Matthias; SKRZYPCZAK, Frank: *Web services flow language (WSFL 1.0)*, IBM, 2001

[LiWe03] LITTLE, Mark; WEBBER, Jim: Introducing WS-Coordination. In: *Journal of Web Services* 3 (2003), Nr. 5

[Lome04] LOMET, David: Robust web services via interaction contracts. In: *Proc. of the 5th Int. Conference on Technologies for E-Services (TES'04)*. Canada, 2004

[Lyu96] LYU, Michael R. (ed.): *Handbook of software reliability engineering* : Mcgraw-Hill, 1996 – ISBN 978-0070394001

[MaZN10] MAIRIZA, Dewi; ZOWGHI, Didar; NURMULIANI, Nurie: An investigation into the notion of non-functional requirements. In: *Proc. of the ACM Symposium on Applied Computing (SAC'10)*. New York, USA : ACM, 2010

[MCHP08] MANCIOPPI, Michele; CARRO, Manuel; HEUVEL, Willem-Jan; PAPAZOGLOU, Mike P: Sound multi-party business protocols for service networks. In: *Proc. of the 6th Int. Conference on Service-Oriented Computing*. Berlin : Springer, 2008

[Merr06] MERRILL, Duane: *Mashups: The new breed of Web app*, IBM, 2006

[MGKS96] MYLOPOULOS, John; GAL, Avigdor; KONTOGIANNIS, Kostas; STANLEY, Martin: A generic integration architecture for cooperative information systems. In: *Proc. of the 1st IFCIS Int. Conference on Cooperative Information Systems (COOPIS '96)*. USA : IEEE Computer Society, 1996

[MSMR08] MENDONÇA, Nabor C.; SILVA, Clayton F.; MAIA, Ian G.; RODRIGUES, Maria Andréia F.; OLIVEIRA VALENTE, Marco Tulio: A loosely coupled aspect language for SOA applications. In: *International Journal of Software Engineering and Knowledge Engineering* 18 (2008), Nr. 2

[MuIO90] MUSA, John D.; IANNINO, Anthony; OKUMOTO, Kazuhira: *Software reliability: measurement, prediction, application* : McGraw-Hill, 1990 – ISBN 978-0070441194

[NFGJ05] NEPAL, Surya; FEKETE, Alan; GREENFIELD, Paul; JANG, Julian; KUO, Dean; SHI, Tony: A service-oriented workflow language for robust interacting applications. In: *Proc. of the Confederated OTM Int. Conferences CoopIS, DOA and ODBASE*. Cyprus : Springer, 2005

[Oasi02] OASIS: *Business transaction protocol (version 1.0)*. URL https://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf

[Oasi04a] OASIS: *UDDI (version 3)*. URL http://uddi.org/pubs/uddi-v3.0.2-20041019.pdf

[Oasi04b] OASIS: *WS-Security*. URL https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[Oasi09a] OASIS: *Web services atomic transaction (WS-AtomicTransaction) version 1.2*. URL http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os/wstx-wsat-1.2-spec-os.html

[Oasi09b] OASIS: *Web services coordination (WS-Coordination) version 1.2*. URL http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html

[Oasi12] OASIS: *WS-Trust*. URL http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html

[Omg04] OMG: *UML profile for modeling quality of service and fault tolerance characteristics and mechanisms*. URL http://www.omg.org/spec/QFTP/1.1/PDF/

[OrHe07] ORTIZ, G.; HERNANDEZ, J.: A case study on integrating extra-functional properties in Web service model-driven development. In: *Proc. of the 2nd Int. Conference on Internet and Web Applications and Services (ICIW '07)*. Washington, USA : IEEE Computer Society, 2007

[Pelt03] PELTZ, Chris: Web Services Orchestration and Choreography. In: *Journal of Computer* 36 (2003), Nr. 10

[PiBM02] PIRES, Paulo F; BENEVIDES, Mario R. F; MATTOSO, Marta: Building reliable Web services compositions. In: *Proc. of Revised Papers from the NODe Web and Database-Related Workshops on Web, Web-Services, and Database Systems* : Springer, 2002

[Port06a] PORTILLA-FLORES, Alberto: *Services coordination with transactional properties*, Master's Thesis, UDLA-INPG, 2006

[Port06b] PORTILLA-FLORES, Alberto: *Une approache à base de contrats pour la coordination fiable des services*, PhD Thesis, Université de Grenoble, 2006

[PoVC06] PORTILLA-FLORES, Alberto; VARGAS-SOLAR, Genoveva; COLLET, Christine: Towards a transactional services coordination model. In: *Proc. of the 10th Int. Database Engineering and Applications Symposium (IDEAS'06)*. Delhi, India, 2006

[PVGC08] PORTILLA-FLORES, Alberto; VARGAS-SOLAR, Genoveva; GARCIA-BANUELOS, Luciano; COLLET, Christine; ZECHINELLI-MARTINI, José-Luis: Verifying atomicity requirements of services coordination using B. In: *In Proc. of the Mexican Int. Conference on Computer Science (ENC'08)*. Mexicali, Mexico : IEEE Computer Society, 2008

[QZCY07] QIU, Zongyan; ZHAO, Xiangpeng; CAI, Chao; YANG, Hongli: Towards the theoretical foundation of choreography. In: *Proc. of the 16th Int. Conference on World Wide Web*. New York, USA : ACM, 2007

[ReWä92] REUTER, Andreas; WÄCHTER, Helmut: The ConTract model. In: *Database transaction models for advanced applications* : Morgan Kaufmann Publishers Inc., 1992 – ISBN 1-55860-214-3

[RHAM06] RUSSELL, Nick; TER HOFSTEDE, Arthur H M; VAN DER AALST, Wil M P; MULYAR, Natalya: *Workflow control-flow patterns: a revised view*, BPM Center Report BPM-06-22 , BPMcenter.org, 2006

[RiGu99] RIBEIRO, C.; GUEDES, P.: SPL: An access control language for security policies with complex constraints. In: *Proc. of the Network and Distributed System Security Symposium*. San Diego, USA, 1999

[SABS02] SCHULDT, Heiko; ALONSO, Gustavo; BEERI, Catriel; SCHEK, Hans-Jörg: Atomicity and isolation for transactional processes. In: *Journal of ACM Transactions on Database Systems* 27 (2002), Nr. 1

[SBFZ08] SU, Jianwen; BULTAN, Tevfik; FU, Xiang; ZHAO, Xiangpeng: Towards a theory of web service choreographies. In: *Proc. of the 4th Int. Conference on Web services and formal methods*. Berlin : Springer, 2008

[ScCM10] SCHMELING, Benjamin; CHARFI, Anis; MEZINI, Mira: Non-functional concerns in web services: requirements and state of the art analysis. In: *Proc. of the 12th Int. Conference on Information Integration and Web-based Applications & Services*. New York, USA : ACM, 2010

[Schm05] SCHMIDT, Karsten: Controllability of open workflow nets. In: *Proc. of Enterprise Modelling and Information Systems Architectures (EMISA'05)*. Klagenfurt, Austria, 2005

[Slom94] SLOMAN, M. S.: Policy driven management for distributed systems. In: *Journal of Network and Systems Management* 2 (1994), Nr. 4

[Souz12] DE SOUZA NETO, Plácido Antônio: *A methodology for building reliable service-based applications*, PhD. Thesis, Universidade Federal do Rio Grande do Norte, 2012

[StGr05] STELLMAN, Andrew; GREENE, Jennifer: *Applied software project management* : O'Reilly, 2005 – ISBN 978-0596009489

[That01] THATTE, S.: *XLANG: Web Services for Business Process Design*, Microsoft, 2001

[TMWD04] TAI, Stefan; MIKALSEN, Thomas; WOHLSTADTER, Eric; DESAI, Nirmit; ROUVELLOU, Isabelle: Transaction policies for service-oriented computing. In: *Journal of Data & Knowledge Engineering* 51 (2004), Nr. 1

[TPSS98] THEODORIDIS, Y; PAPADIAS, D; STEFANAKIS, E; SELLIS, T: Direction relations and two-dimensional range queries: optimization techniques. In: *Journal of Data and Knowledge Engineering* 27 (1998), Nr. 3

[TsNA12] TSADIMAS, Anargyros; NIKOLAIDOU, Mara; ANAGNOSTOPOULOS, Dimosthenis: *Extending SysML to explore non-functional requirements: the case of information system design*. New York, USA : ACM, 2012

[UBJS03] USZOK, A.; BRADSHAW, J.; JEFFERS, R.; SURI, N.; HAYES, P.; BREEDY, M.; BUNCH, L.; JOHNSON, M.; et al.: KAoS policy and domain services: toward a description-logic approach to policy representatin, deconfliction and enforcement. In: *Proc. of 4th IEEE Workshop on Policies for Networks and Distributed Systems (Policy'03)*. Lake Como, Italy : IEEE, 2003

[VeVJ06] VERHEECKE, B.; VANDERPERREN, W.; JONCKERS, V.: Unraveling crosscutting concerns in Web services middleware. In: *Journal of IEEE Software* 23 (2006), Nr. 1

[ViVo04] VIDYASANKAR, K.; VOSSEN, G.: A multilevel model for Web service composition. In: *Proc. of the Int. Conference on Web Services (ICWS'04)*. Canada : IEEE Computer Society, 2004

[VSVC05] VANDERPERREN, Wim; SUVÉE, Davy; VERHEECKE, Bart; CIBRÁN, María Agustina; JONCKERS, Viviane: Adaptive programming in JAsCo. In: *Proc. of the 4th Int. Conference on Aspect-Oriented Software Development (AOSD'05)*. New York, USA : ACM, 2005

[VZEC13] VARGAS-SOLAR, Genoveva; ZECHINELLI-MARTINI, José-Luis; ESPINOSA-OVIEDO, Javier A.; COLLET, Christine; MOTZ, Regina; RETTICH, Alvaro: *Mashing up Web data as spatio-temporal presentations using M-QLIST*, Thecnical Report, Laboratory of Informatics of Grenoble, 2013

[Wc00a] W3C: *Web Services Description Language (WSDL) 1.1*. URL http://www.w3.org/TR/wsdl/

[Wc00b] W3C: *Simple Object Access Protocol (SOAP) 1.1*. URL http://www.w3.org/TR/soap/

[Wc00c] W3C: *XML Path Language*. URL http://www.w3.org/TR/xpath/

[Wesk07] WESKE, Mathias: *Business process management: concepts, languages, architectures* : Springer, 2007 – ISBN 3540735216

[Wmc96] WMC: *Workflow management coalition specification: terminology and glossary*. URL http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf

[XCZB08] XIAO, Hua; CHAN, B; ZOU, Ying; BENAYON, J W; O'FARRELL, B; LITANI, E; HAWKINS, J: A framework for verifying SLA compliance in composed services. In: *Proc. of the 2008 Int. Conference on Web Services (ICWS'08)*, 2008

[ZBDH06] ZAHA, Johannes Maria; BARROS, Alistair P.; DUMAS, Marlon; TER HOFSTEDE, Arthur H. M.: Let's dance: a language for service behavior modeling. In: *Proc. of the OTM Conferences*. Montpellier, France, 2006

[ZHMS06] ZHANG, Xianan; HILTUNEN, Matti; MARZULLO, Keith; SCHLICHTING, Richard: Customizable service state durability for service oriented architectures. In: *Proc. of the 6th European Dependable Computing Conference (EDCC'06)*. Portugal : IEEE Computer Society, 2006

[ZNBB94] ZHANG, Aidong; NODINE, Marian; BHARGAVA, Bharat; BUKHRES, Omran: Ensuring relaxed atomicity for flexible transactions in multidatabase systems.

In: *Proc. of the ACM SIGMOD Int. Conference on Management of Data*. New York, USA, 1994

Appendix A Workflow Execution Environment

Business process management includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes can be enacted. The process enactment phase encompasses the actual run time of the business process. Business process instances are initiated to fulfill the business goals of a company. Initiation of a process instance typically follows a defined event. The business process management system actively controls the execution of business process instances as defined in the business process model.

Workflow management systems architectures organize the subsystems that are involved in the design and enactment of both system workflows and human interaction workflows. A generic workflow management systems architecture is shown in **figure a.1**. The generic workflow management systems architecture proposed by the Workflow Management Coalition. This architecture provides a high-level systems architecture blueprint for workflow management systems.

Process enactment guarantees that the process activities are performed according to the execution constraints specified in the process model. A monitoring component of a business process management system visualizes the status of business process instances. Process monitoring is an important mechanism for providing accurate information on the status of business process instances.

Interface 1: gives access to tools used for modeling workflows. The goal of this interface is to enable tools developed by different workflow system vendors to work in a standardized representation of a business process. Interface 1 is specified in the XML language XML Process Definition Language, or XPDL. This language is based on a meta-model approach, in which the concepts of interacting business processes of multiple participants are defined. The XPDL package meta-model can be used to represent business process diagrams expressed in the BPMN. The meta-model includes classes for pools, lanes, processes, participants, and message flow. The internal structure of processes is represented by a process meta-model. In addition to processes and activities, different types of activities and transitions are modeled to represent control flow in process models. The specification document also defines a serialization of process models to XML that includes conceptual information as well as graphics information used for rendering business process diagrams.

Interface 2: During the enactment of human interaction workflows, the persons involved receive work items that inform them about activities due for execution. This functionality is realized by workflow client applications, attached to the enactment service by Interface 2. The goal of standardizing Interface 2 is to allow workflow client applications of different vendors to talk to a given workflow enactment service.

Interface 3 provides the technical information to invoke applications that realize specific workflow activities. This interface should facilitate invocation of applications across heterogeneous software platforms. The reference architecture also provides an interface to other workflow enactment services.

Interface 4 is used for interoperability between different workflow enactment services. The administration and the monitoring of workflows are handled by a dedicated component, accessed by Interface 5. These interfaces are not specified in detail, and their implementations remain in the prototypical stage.

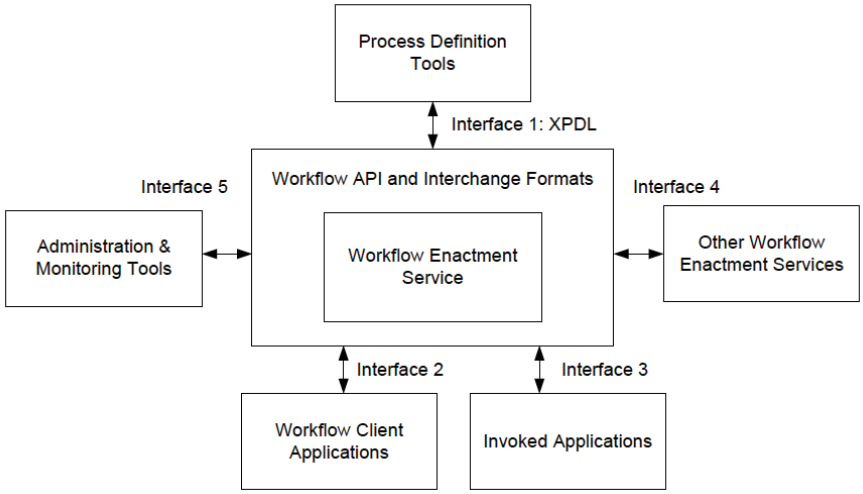

Figure A.1 WFMC general architecture

Recent achievements in service-oriented architectures contribute to the standardization efforts. The Workflow Management Coalition and the development of the XML Process Definition Language are important steps towards solving the workflow management system interoperability issue.

ActiveVOS is built on the BPEL, BPEL4People and WS-Human Task open standards to allow developers the ability to orchestrate various systems and services. Apache ODE (Orchestration Director Engine) is a WS-BPEL 1.1 and 2.0 compliant business process management (BPM) engine that supports two communication layers: one based on Apache Axis2 (Web Services HTTP transport) and another one based on the Java Business Integration standard (using Apache ServiceMix). It is capable of Hot-deployment, and features a management interface for processes, instances, and messages. Oracle BPEL Process Manager provides a framework for easily designing, deploying, monitoring, and administering processes based on BPEL standards. BPEL Process Manager is the service orchestration solution on Oracle's SOA Suite.

Intervoice Media Exchange contains an orchestration engine that has been designed to initiate and manage media interactions. It is the industry's first commercially available product that has implemented State Chart eXtensible Markup Language (SCXML) as the framework for building complex multi-modal interactions. TIBCO BusinessWorks is an orchestration, integration and transformation tool that supports BPEL, Web Services, common integration activities and visual modeling of orchestration processes. Microsoft System Center Orchestrator contains an orchestration engine which can be used for IT Process Automation as well as Business process management (BPM), allowing developers to quickly orchestrate complex IT and business processes involving multiple disparate systems. The Orc language is an academic language for describing and implementing orchestrations. IBM WebSphere Process Server contains an orchestration engine, able to execute BPEL. Juju is a service orchestration framework for Ubuntu Linux.

Trigsflow **[KaRR98, KPRR95]** is a WFMS based on an active extension of the ODBMS Gemstone. In Trigsflow, activities are assigned to agents with respect to their role and/or load. Activities can be delegated to other agents. In the Brokers/Services approach, agents are brokers that execute services. Agents are specified and implemented as reactive components based on the event engine EVE **[GeKT96]**. Agents interact asynchronously (using events) to ensure workflow execution. CoopWARE **[GaMy00, MGKS96]** focuses on interaction aspects between workflow servers and agents. Different to Brokers/Services they adopt a centralized architecture where the WFMS coordinates agents.

A.1 Executing Services Coordinations

Activities can be found in the leaves of the functional decomposition. An activity model describes a set of similar activity instances, analogously to a process model describing a set of process instances with the same structure. While process models are typically expressed in graph-like notations activity models can be expressed in different forms, for instance, by plain text or by some formal specification or references to software components that implement them. Activity instances represent the actual work conducted during business processes, the actual units of work.

A.1.1 Activity Life Cycle

Each activity instance during its lifetime is in different states. These states and their respective state transitions can be represented using a *state transition diagram*. A simple state transition diagram is shown in **figure a.2**. The states that an activity instance adopts during its lifetime are described as follows: when it is created it enters the **init** state. By the enabled state transition the activity instance can enter the ready state.

If a particular activity instance is not required, then the activity instance can be skipped, represented by a skip state transition from the not started state to the skipped state. From the ready state, the activity instance can use the begin state transition to enter the running state. When the activity instance has completed its work, the terminate state transition transfers it to the terminated state. When an activity instance is in the terminated or the skipped state, then it is closed.

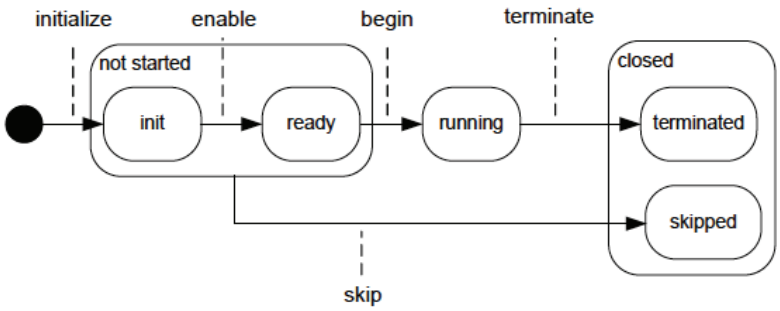

Figure A.2 Transition diagram of simple activity instances [Wesk07]

Activity instances are likely to expose a more complex behavior including disabling and enabling activities, suspending running activities, and skipping or undoing activities. The respective state transition diagram is shown in **figure a.3**. The activity instance is initiated, and it enters the ready state before entering the running state. Finally, the activity instance terminates. However, the sub-states of the termination state are not represented. The simple diagram does not provide different states for successful and unsuccessful termination.

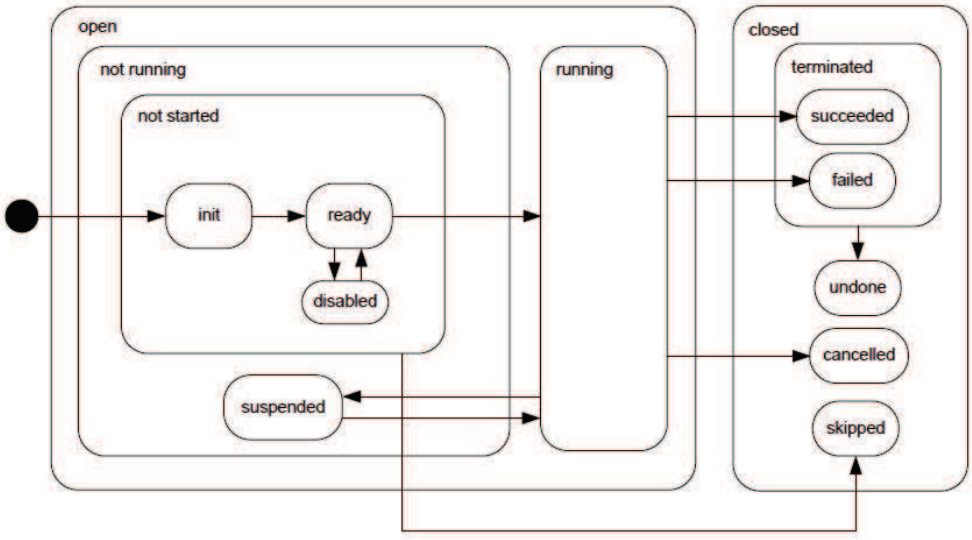

Figure A.3 Complex activity instances state diagram [Wesk07]

When an activity instance can be started, it enters the **ready** state. If during the execution of a process instance certain activity instances are currently not available for execution, they can be disabled. Activity instances that are initialized, disabled, or enabled are in the not **started** state. Once an activity instance is ready, it can be started, entering the **running** state. Running activities can be temporarily suspended, to be resumed later. An activity instance can terminate either successfully or in failure. Terminated activity instances can be undone, using compensation or transactional recovery techniques.

Events and event orderings are introduced to properly represent the essence of activity instances. Events are points in time, i.e., events do not take time. The time interval in which

an activity instance is in one state is delimited by two events, the event representing the state transition to enter the state and the event representing the state transition to leave the state. For example, the time interval in which the activity instance is in the running state is delimited by the begin and terminate events.

The basic idea of using events for representing activity instances is that each state transition of an activity instance is represented by an event. An event diagram for a particular activity instance is shown in figure a.4 that forms a directed acyclic graphs, where the nodes are events and the edges reflect causal ordering between events.

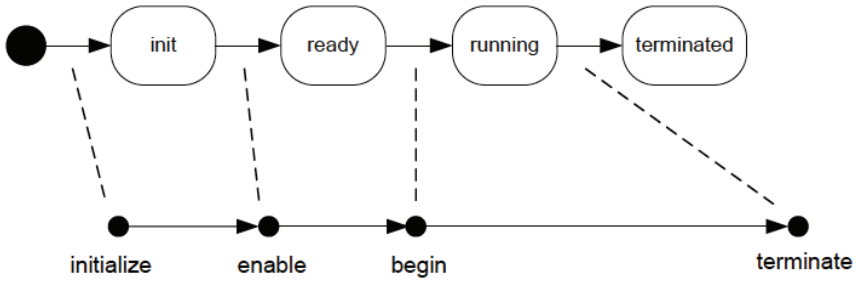

Figure A.4 Event diagram for the execution of an activity instance [Wesk07]

The activity instance starts with a state transition to the init state. This state transition is represented by an initialize event in the event diagram. An enable state transition brings the activity instance in the state ready; this state transition is represented by an event enable. An activity instance in the ready state can be started, represented by the state begin transition. Finally, the state terminate transition completes the activity instance.

A.1.2 Data Patterns

Workflow data patterns formulate characteristics on how to handle data in business processes. They are organized according to the dimensions data visibility, data interaction, data transfer, and data based routing.

Data visibility is very similar to the concept of scope in programming languages because it characterizes the area in which a certain data object is available for access. The most important workflow data patterns regarding data visibility are as follows **[Wesk07]**.

- Task data: The data object is local to a particular activity; it is not visible to other activities of the same process or other processes.
- Block data: The data object is visible to all activities of a given sub process.
- Workflow data: The data object is visible to all activities of a given business process, but access is restricted by the business process management system, as defined in the business process model.
- Environment data: The data object is part of the business process execution environment; it can be accessed by process activities during process enactment.

Data interaction patterns describe how data objects can be passed between activities and processes. Data objects can be communicated between activities of the same business

process, between activities and sub processes of the same business process, and also between activities of different business processes.

Data can also be communicated between the business process and the business process management system. Data transfer is the next dimension to consider. Data transfer can be performed by passing values of data objects and by passing references to data objects. These data transfer patterns are very similar to call-by-value and call by -reference, concepts used in programming languages to invoke procedures and functions **[Wesk07]**.

In data based routing, data can have different implications on process enactment. In the simplest case, the presence of a data object can enable a process activity. Data objects can also be used to evaluate conditions in business process models, for instance, to decide on the particular branch to take in a split node. Workflow data patterns are an appropriate means to organize aspects of business processes related to the handling of data **[Wesk07]**.

Appendix B Volcano Observer Application

In order to validate our approach in the context of *data integration* and *presentation*, we developed the application **Volcano Observer**, an application for integrating environmental information about the behavior of the active volcano Popocatepetl[1] in Puebla, Mexico. The objective of the application is to integrate information produced by (i) the *National Centre for Preventing Disasters*[2], (ii) the *Meteorology Service*[3], (iii) the *Index of the Air Quality*[4] and (iv) the social observation generated in *Twitter* and *Facebook* by civilians and other governmental institutions in charge of informing about the alter state evolution (e.g., the volcano is in phase 3/3).

We implemented the **Volcano Observer** application using the concept of *Mashup* **[Merr06]**, a special kind of applications that combines data produced by different web sources and which presents it in a 2-dimensional space. A mashup is modeled using 2 concepts:

- **Mashlet** – minimum content unit providing a homogeneous view of data and their providers (i.e., the building block).
- **Mashup** – represents the spatial organization in which building blocks will be presented in a 2D space.

In our experiment we extended the notion of mashup for considering (i) data coming from *data services* and (ii) data presentations in a 3-dimensional space (i.e., *X*, *Y* and time). We also allow the association of active policies to mashlets and mashups.

---

[1] This work was done in the context of the projects **e-CLOUDSS** (program LACCIR-MICROSOFT), **Red-Shine** (program BQR of Grenoble INP) and **CLEVER** (program STICAMSUD).

[2] http://www.cenapred.unam.mx/

[3] http://smn.cna.gob.mx

[4] http://www.calidadaire.df.gob.mx/calidadaire/

B.1 Spatio-Temporal Mashup Language (M-QLiST)

In order to define mashups we defined **M-QLiST**, a Spatio-Temporal Mashup Language **[VZEC13]**. **Listing B.1** shows our general expression for defining a mashup. As shown in the expression, a mashup definition is composed of (i) a mashup name, (ii) a data flow expression, (iii) a spatio-temporal expression and (iv) a set of active policies expressions.

```
define mashupName as mashup dataflow
where spatio-temporal-expression
on    active-policies
```

Listing B.1 M-QLIST mashup general definition expression

Dataflow specifies the dependencies between the produced and consumed data of a set of *mashlets* that are mashed up. The dataflow of **M-QLiST** is defined on top of a simple data model that includes the atomic and complex data of the AP Model (**section 3.2**). The data model subsumes the JSON[5] model that is widely used for retrieving data on the web (e.g., in REST architectures).

Inspired in systems like Yahoo Pipes[6] and Montage[7], we expressed the dataflow of a mashup using *data* and *control flow* operators. *Data operators* are used for processing data. The following list enumerates the *data operators* considered in **M-QLiST**:

- **Filter**. Filters the data collection produced by a service operation call.
- **Count**. Counts the number of elements in a collection.
- **Rename**. Renames one (or several) attribute of a complex data type.
- **Reverse**. Reverse the order of the elements composing a data collection.
- **Union**. Includes all the elements of a collection *A* in a collection *B* (e.g., implements the union set operation).
- **Tail**. Retrieves the last element in a data collection.
- **Truncate**. Obtains a subset of the elements contained in a collection.

Control flow data operators define how data flow among mashlets. The following list enumerates the *control flow data operators* considered in **M-QLiST**:

- **Sequence**. The output of the data retrieved in a mashlet is used as input of another mashlet.
- **Split**. The output of a mashlet feeds two or more independent mashlets.
- **Join**. The output of one or several mashlets feeds a mashlet.
- **Loop**. Executes a set of mashlets several times considering the results of their previous execution.

[5] http://www.json.com

[6] http://pipes.yahoo.com/pipes/

[7] http://fuse.microsoft.com/projects/montage

Spatio-Temporal-Expression defines spatio-temporal properties of a mashup (see **listing b.1**). The spatio-temporal operators used in **M-QliST** are the following:

- **Locate At**. Specifies the *location* and *size* of a mashup in a 2D space (e.g., the expression *Locate At 50, 50 with 300px, 300px* locates the origin of the mashup at x=50, y=50 with a size of 300 pixel width/height).
- **After**. Specifies the instant of presentation of a mashup of the screen and the duration of the presentation (e.g., the expression *After 5 sec during 10 min* specifies that the mashup has to be presented (i) 5 seconds after the mashup starts running and (ii) during 10 minutes. After that the mashup is hidden).

Active-Policies define the active policy instances associated to a mashup. Policy instances are expressed using the AP Language (see **chapter 2**).

B.2 Application Logic as Mashup

Figure B.1 shows the order in which data is retrieved from the data services composing the **Volcano Observer** mashup. As shown in the figure, the application retrieves data in two parallel steps: the first one intended to retrieve in parallel the volcano tweets and posts from the **Twitter** and **Facebook** services (see activities *GetVolcanoTweets* and *GetVolcanoAlerts*); the second one retrieves data about the semaphore (activity *GetSemaphore*), the air quality (activity *GetAirQuality*) and the meteorology of the regions around the volcano (activity *GetMeteorology*).

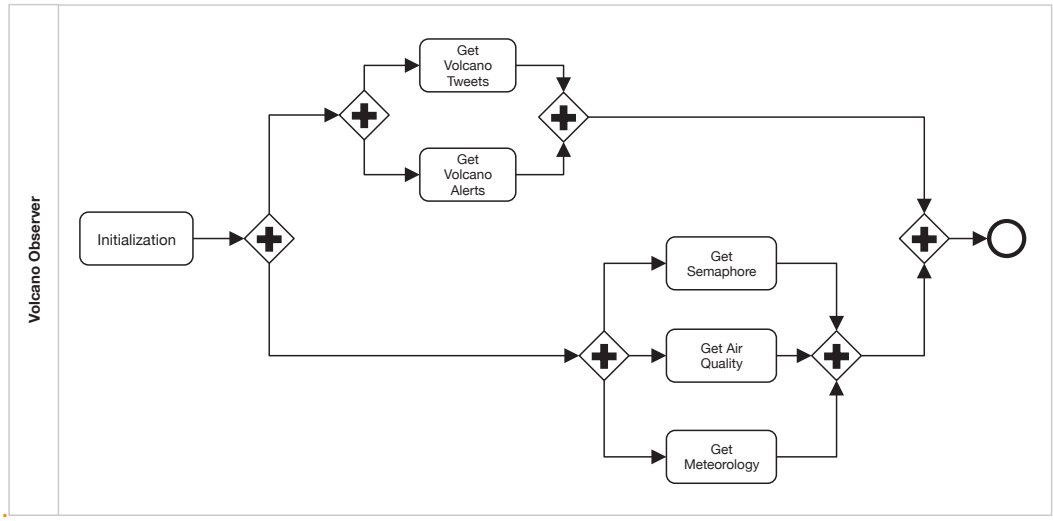

Figure B.1 Workflow implementing the Volcano Observer data retrieval

Figure B.2 shows the 5 mashlets composing the **VolcanoObserver** mashup. For instance, the mashlet $\mathbf{M_1}$ presents the Twitter posts of the observations of the civilians that live in the region of Puebla and that observe whether the volcano produces exhalations of fumes. These posts are presented in a rectangle of 200x300 pixels in the center of an html page during the whole execution of the mashlet. **Listing B.2** shows the definition of mashlet $\mathbf{M_1}$. In a similar way the mashlet $\mathbf{M_2}$ presents the posts coming from Facebook with information about the exhalation color and magnitude.

```
define M1 as mashlet "https://twitter.com/john.doe"
every 5 min
locate at 5, 10 after 10 sec with 300px, 200px
```

Listing B.2 Example of a mashlet definition

In a similar way $\mathbf{M_3}$ defines a mashlet that presents the description of the level of alert (semaphore) and the volcano evolution during the last 24 hours. Note that this information is presented in a rectangle located at the bottom-left hand side of a page. We define two equivalent mashlets presenting the information of the quality of the air, and the meteorology service ($\mathbf{M_4}$ and $\mathbf{M_5}$).

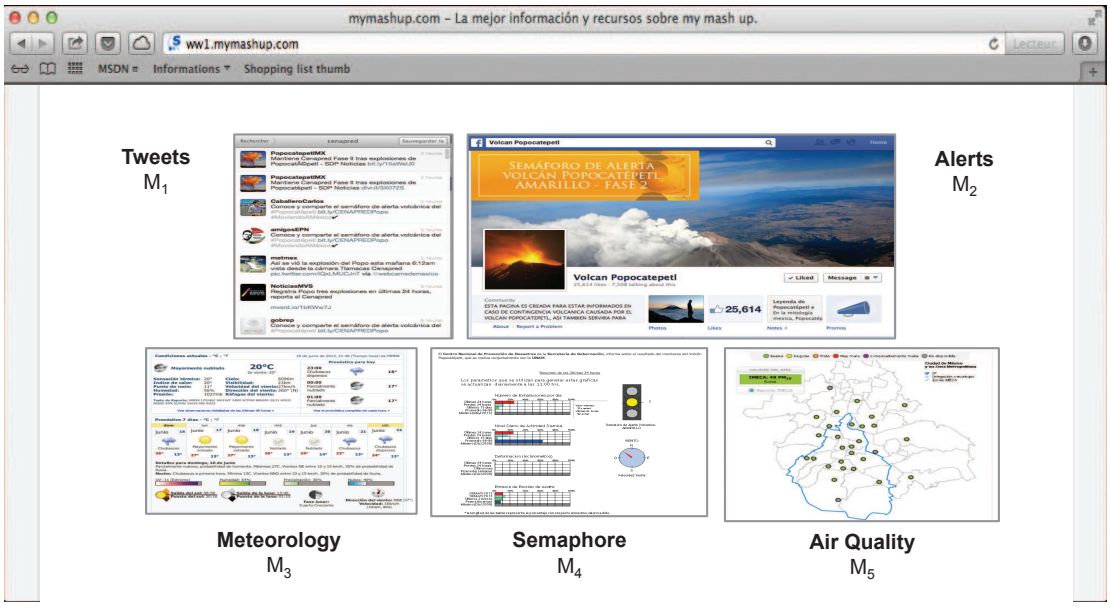

Figure B.2 Volcano Observer Mashup

The overall mashup logic works as follows: first each mashlet interacts with a data service for retrieving data and presenting them with specific spatio-temporal properties as described before. Then, the mashup expression defined the spatio-temporal organization of each mashlet defines the mashlet positions. In our model, spatial relations among mashlets are specified using the approach proposed by **[Hern94, TPSS98]** and the associated semantics (e.g., centered, middle, to the right/left, on top, bottom). For example, in the **Volcano Observer** mashup we define that mashlet (i) $\mathbf{M_2}$ is presented to the right of $\mathbf{M_1}$, (ii) $\mathbf{M_4}$ is presented to the right of $\mathbf{M_3}$ and $\mathbf{M_5}$ is presented to the right of $\mathbf{M_4}$ and (iii) $\mathbf{M_3}$, $\mathbf{M_4}$ and $\mathbf{M_5}$ are presented to the south of $\mathbf{M_1}$ and $\mathbf{M_2}$.

B.3 Active Policies for Mashups

Besides the challenge of coordinating the data retrieval within a mashup, it is also important to consider that the quality of the data retrieved (e.g., reliability, freshness, pertinence). We handle data quality by associating *management active policies* to the mashlets

retrieving the data and by considering the characteristics of the services it accesses. In the case of the **Visual Observer** mashup the data comes from different services and Web sites that update their information with different frequency. In order to maintain a consistent view of the official information, information updates are synchronized. In contrast, social networks information is updated continuously.

Figure B.3 shows the general approach that we proposed for specifying mashup management policies to the **Volcano Observer** mashup. For instance, the atomicity policy applying to mashlets M_3, M_4 and M_5 ensure that these mashlets are executed following a strict atomic behavior (cf. **section 5.2.2**). In a similar way, the authentication policies applying to mashlets M_1 and M_2 ensure the implementation of the authentication protocols required to access the Twitter and Facebook accounts (cf. **sections 3.6.2**). **Listing B.3** and **Listing B.4** illustrates the definition of these policy types using the AP Language adapted to the context of mashups.

The **StrictAtomicityPolicy** policy type defines a policy that implements the strict atomicity behavior. In general the policy works as follows: on the failure of a service call executed by a mashlet (represented by the event type **ServiceCallFailure**), if the number of times this call has been retried is less than a threshold, then the service is invoked again. Otherwise the mashlets that were executed without failures are compensated. We have proposed atomicity policies in other works that are applied to define the semantics of the compensation operation **[PoVC06, PVGC08]**. In the case of mashlets, the compensation can mean to present the data they retrieved the last time that all of them were executed correctly.

The **HTTPAuthPolicy** policy type defines a policy that uses the HTTP authentication protocol for calling the service operation associated to a mashlet. Note that the scope of the policy is the type **Mashup**. In general the policy works as follows: when the mashlet is prepared for calling the service' operation (represented by the event type **ServiceCallPrepared**), the policy ask the user its credentials (i.e., username and password) and injects this credentials in the code handling the operation call.

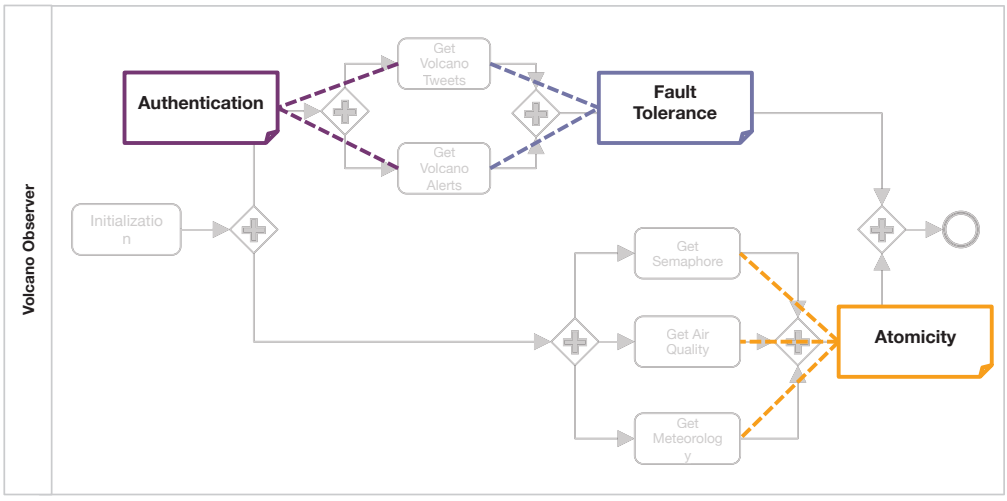

Figure B.3 Policies for dealing with mashups data management

```
policy class StrinctAtomicityPolicy |[ Mashlet * ]| : ActivePolicy {

    int numberOfTimes, NoT_threshold;

    rule R1
        on ServiceCallFailure
        if event.producedByDescendent ( scope ) && numberOfTimes < NoT_threshold
        do Sequence (
            Continue (event.producer ),
            Increment ( NoT_threshold, 1)
        )

    rule R2
        on ServiceCallFailure
        if event.producedByDescendent ( scope ) && numberOfTimes >= NoT_threshold
        do Compensate ( scope )


}
```

Listing B.3 Strict Atomicity Policy type for a mashlets

```
policy class HttpAuthPolicy |[ Mashlet mashlet ]| : ActivePolicy {

  String username, password;

  rule R1
    on ServiceCallPrepared
    if event.producedBy ( scope ) && numberOfTimes < NoT_threshold
    do Assign ( mashlet.httpRequest.credentials, new NetworkCredential ( username, password ) )

}
```

Listing B.4 Http Authentication Policy type for mashlets