*The International Journal of Parallel, Emergent and Distributed Systems*
Vol. 00, No. 00, Month 2011, 1–25

RESEARCH ARTICLE

# A distributed architecture for policy-customisable multi-tenant Processes-as-a-Service

MingXue Wang[a] and Kosala Yapa Bandara[b] and Claus Pahl[c]*

[a]*Ericsson Ltd, Athlone, Ireland*; [b]*SLIIT, Sri Lanka*; [c]*School of Computing, Dublin City University, Dublin 9, Ireland*

(*Received 00 Month 200x; in final form 00 Month 200x*)

Service-based business processes are often developed and deployed by single organizations. In distributed, shared resource environments like the cloud on the other hand, consumers share resources owned by cloud providers. This requires multi-tenancy capability for service processes that provides customized behaviour for on shared process implementations to meet the varying needs of different process consumers as tenants of the process resource. In this paper, we define a distributed multi-tenant architecture for BPEL processes provided as a service. A single-version BPEL process is deployed by a provider and offered for all process consumers, combined with a customization and management functionality to create a unique experience for different consumers (process tenants). We provide two core components: a policy model for consumers to express customization/business requirements of service processes and a coordination framework for policy enforcement between consumers and providers to achieve on-the-fly customization of service processes.

**Keywords:** Multi-tenancy; Services; Processes; Policy; Coordination; Governance

## 1. Introduction

A *business process* is a collection of interrelated tasks or activities, which are designed to deliver a particular result or complete a business goal. In Service-Oriented Architecture (SOA), business processes are often implemented as BPEL processes, which are composed from individual Web services. Because of requirements such as monitoring or security at service level and typically different business requirements or policies of organisations with regard to business processes, processes generally are scoped and reside within one organisation [1]. Consumers would also develop and host business processes based on services from service providers. Business processes are not available for sharing between cross-organisational, distributed service consumers in SOA currently [2]. Shared ready-to-use business processes for on-demand requirements of consumers is an important topic in the cloud computing and software-as-a-service (SaaS) context. We propose a solution for BPEL business-processes-as-a-service where part of the provided process service is the process customization. Business requirements of processes are expressed as policies by process consumers, and policies are enforced on process executions through for on-the-fly process customization coordinated remotely by the process consumer.

The essence of multi-tenancy in a software system is sharing, but also isolating resources between different tenants and their application users [3]. The tenants are the contractual partners of the process providers. Multi-tenancy means higher

*Corresponding author. Email: Claus.Pahl@dcu.ie

ISSN: 1744-5760 print/ISSN 1744-5779 online
© 2011 Taylor & Francis
DOI: 10.1080/17445760.YYYY.CATSid
http://www.informaworld.com

levels of resource sharing, resulting in more economy-of-scale for providers [4]. These benefits also apply to the tenants in the form of lower service fees, quicker access to new functionality, etc. [5][3][6]. Our work supports multi-tenancy capable business processes. A single version BPEL process is assumed to be developed, hosted and shared for all process consumers, but tenant customization functionality is also offered at the same time to offer a unique experience for tenants to meet their varying requirements. Thus, different and isolated process execution instances are the result based on a customization of the same deployed BPEL process across the organizational boundries between provider and consumer.

Our solution for multi-tenancy BPEL processes in a cloud-like architecture is based on two core components: a policy model and a coordination framework.

- *Business policies* describe business requirements that are expressed as formal *policy* statements and are focused on SOA governance. Technologies such as WS-Policy [7] in the SOA governance domain do not address policies with respect to services or tasks for cross-organisational consumers. SOA treats policies of business processes as an internal organisational problem. Hence, we need a policy model that can be facilitated for process consumers on shared business processes as customization metadata.
- Defined policies of process consumers must be enforced on the process execution when consumers use the processes. For reasons such as policy centralization and privacy concerns of cloud consumers [8][9][10], policies should reside at the consumer side. Hence, governance directly from process consumers needs a coordination protocol as the basis for a service contract between providers and consumers to address the policy enforcement. However, WS-Coordination, WS-BA and their extensions [11][12][13] cover only transactional activity control for distributed Web services. They are not designed for transactions of BPEL processes and other aspects arising from our policy model.

The remainder of the paper is structured as follows. After describing the problem through a case study and related work in Sections 2 and 3, we give an architecture overview in Section 4, then detail the policy model and coordination in Sections 5 and 6. Finally, we evaluate and conclude in Sections 7 and 8, respectively.

## 2. Use case scenario

There are many business processes which could be used across many application domains and organisations. For example, a *purchase order checkout* process contains *order inspection*, *shipping*, and *payment* business activities [14]. This could be used in supply chain systems, online retail applications, etc. The services implementing these business activities could come from the same or different external providers. Service consumers can compose business services into composite services or business processes themselves, such as the order checkout process. However, an organisation as a *process provider* might like to develop and host such a business process and to offer this to other external organisations as a *process consumers*. The business process, which generally has an increased business capability, is difficult to use by a process consumer, if it does not meet unique detailed requirements or business policies of a specific organisation [1]. For example, one organisation or process consumer could have policies such as "*Free parcel shipping for orders with a total over 2000 euro*"; "*All payment transactions should be processed by Bank of Ireland*"; another process consumer could have very different policies, such as "*Credit card processing should be completed quickly (expected in less than 700 ms) without fault*". These policies of different consumers can generally be easily addressed when consumers develop their own unique processes, but with a scenario of a shared, ready-to-use business process offered by the process provider in a cloud environment, it becomes a challenge that has not been sufficiently addressed.

We will return to the case study throughout the paper and also in a separate evaluation section where the case study serves to support an emprical validation.

## 3. Related work

Policies for Web services, such as XACML [15] and WS-Policy [7], are not process-centric policy languages. They only deal with independent Web services and do not cover the range of aspects for process customization across individual service invocations. XACML only covers security aspects. To offer a process-as-a-service solution, policies are usually added for process consumers after the business processes are defined. This makes the policy-first process development approach, such as the conventional business rules approach [16][17], and tightly coupled process and policy approaches, such as [18][19][20], not applicable for sharing processes that are hosted outside an organisation with multi-tenancy capability. Generally, organisation-internal processes only need to comply with single party policies.

Process adaptation is closely related to policy systems, as changing of processes is normally realized by enforcing policies. [21] utilizes BPEL event handling to provide adaptation by performing predefined alternative actions if certain events occur. [22] propose a service relevance and replacement framework. The Dynamo project [19] developed a supervision framework for the ActiveBPEL engine. Similar frameworks such as [23][24][25][26][27][28] extend a BPEL engine for process adaptation. However, these also suffer from the same problem as policy models. Multi-tenancy requirements missing at the design stage are usually difficult to extend for multi-party policy enforcement on a single BPEL engine.

The Service Transaction (WS-TX) specifications contain a set of specifications – WS-Coordination [29], WS-AtomicTransaction (WS-AT) [30], WS-BusinessActivity (WS-BA) [31] – for service transactions. WS-Coordination defines an overall framework, and WS-AT and WS-BA are two protocols for atomic and long-running transactions (LRT), respectively. These protocols are designed for Web service transactions. However, there is no WS-BA-based interaction assumed between a process and contained services. It is impossible for a BPEL process to participate in a WS-BA coordination [32]. However, without standard protocols, it is impossible to coordinate a transaction with various processes distributed in one or many different providers. Still, different aspects of policies as requirements needs a more comprehensive protocol rather than those that only deal with transaction management. Such work on coordination with policy enforcement for consumers and process providers is still lacking, but is needed for business process sharing in the cloud paradigm. Additionally, the multi-tenancy capability needs to be taken care of in a coordination framework implementation.

[33][6] aim to solve the problem for the cloud platform layer (PaaS) by providing a shared BPEL engine rather than BPEL processes for consumers. Process consumers still need to develop and maintain their own BPEL processes in that case. At the SaaS layer, the Cafe project [34][35] offers a front-end ready Web-based application for process consumers rather than software components, where the BPEL process is wrapped in an end user-based application. QoS configuration is available at the provider side for consumers. This approach is limited with regard to policy centralization and reuse, and concerns over confidentiality of policies exist [8]. Currently, the multi-tenancy capability of BPEL processes is achieved by the Web services dynamically bound to the process instances, resulting in different QoS

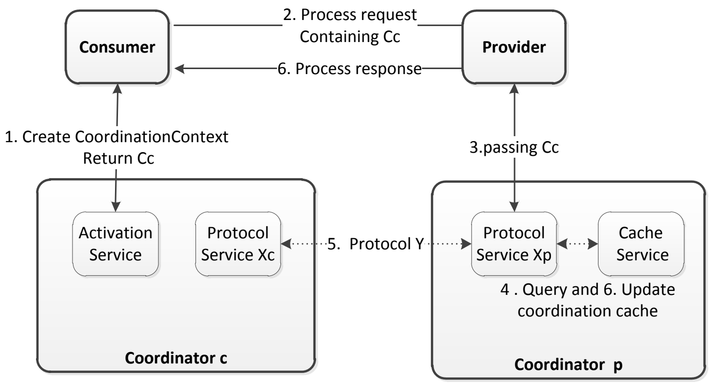

Figure 1. Schematic coordination example

behaviour for different tenants. [33] [6] present the WSO2 Business Process Server based on this idea. The Axis2 service engine intercepts and injects the message into the extended ODE BPEL engine at runtime, which takes care of creating process instances or routing messages to an already running instance.

Business policies might cover a wide range of requirements on business processes in addition to common QoS properties addressed in current work on multi-tenant business processes, e.g., WSO2 or the Cafe project. Adaptive processes are needed so that processes can be changed to meet the various change requirements of tenant customization, i.e., various business policies of process consumers, as the basis of customization. However, the multi-tenancy problem is not covered in current adaptive process approaches [19] [22].

## 4. Architecture overview

Organisations as process providers shall be able to offer a ready-to-use customizable business process to be shared by multiple organisations or process consumers (tenants) as a service. Consumers can then customize the process execution behaviour according to their own business policies by defining a set of policies. Then consumers can directly invoke or use the process which is hosted inside the provider, without the need for new process development and execution environment maintenance. The overall architecture to enable this consists of (see Fig. 1):

- *Process components* at the process provider side offer business processes. They execute the activities in the process logic to serve a particular goal, and hold process runtime information resources needed for policy evaluation or weaving.
- *Governance components* govern processes at runtime for process consumers at the consumer side. They hold the policies of process consumers and evaluate or weave the policies.
- A *coordination protocol* within a service or process contract defines the connectors and behaviour between any process and governance components.

### *4.1 Policy model and governance*

Process customization is achieved by means of runtime governance of processes. The policy modelling is a technique for consumers in our framework. The policy model can be viewed as a customization language for prepared business processes. The defined policies are enforced in business processes at the provider side on behalf of the consumers. This is a high-level process contract between process consumers and providers. The mechanism for process providers to carry out this contract is a contract-defined coordination protocol. Hence, the development of the policy model is based on a coordination protocol for runtime governance between process consumers and providers to achieve on-the-fly customization. The process components are not tied to any governance components, but comply with coordination protocols, which define the provider capability of customization. One governance component is responsible for one process consumer that has a separate set of policies. The connections from any governance component to any process component are dynamic, created on demand through coordination protocols.

### *4.2 Coordination model and protocol*

The coordination model is inspired by the WS-Coordination and XACML policy frameworks, and is redefined for the specific need of our coordination protocol and mechanism for policy enforcement. The coordination model defines two types of subcoordinators for process consumers and providers (Figure 1). Thus, each participant only interacts with its own type of coordinator. The model is defined as $< COOR, COOR_{context} >$, where

- $COOR = COOR^c \cup COOR^p$. $coor^c \in COOR^c$ is a coordinator associated with the consumers or process governance component $PG$. $coor^p \in COOR^p$ is a coordinator associated with the provider or business process $BP$.
- $coor_{context} \in COOR_{context}$ is coordinaton context information.

$coor^p \in COOR^p$ is required for all process providers, including the subprocesses. Figure 1 illustrates how $coor^c$ and $coor^p$ interact in a coordination conversion. The process consumer initializes $coor_{context}$ (Cc) that contains information needed for starting a coordination conversation, then sends a process request to the provider or business process containing the $coor_{context}$. The protocol $X$ and services $X_c$ and $X_p$ are specific to a coordination protocol. Cached coordination data is queried and updated through the cache service if required (note that steps 4, 5 and 6 are not necessarily executed in sequence). The coordination conversation ends by completing the process execution or the business transaction.

### *4.3 Coordination context*

The $COOR_{context}$ defines the data structure of the coordination message exchange as an XML schema. All process consumers and providers must understand this information to enable coordination conversions. $coor_{context} \in COOR_{context}$ contains data such as identification, protocol service reference, initialized by the consumer at the start of a coordination conversation for a process request or business transaction instance. It can only be assigned by a $coor^c$. For activities as subprocesses in a business process, the $coor_{context}$ data is propagated to participants, i.e., the subprocesses, whether they belong to the same process provider or different providers. Providers do not initialize a new context for subprocesses. This is important for multi-tenancy as the source of $coor_{context}$ symbolizes the source of the business policies, i.e., all processes include subprocesses that are governed by the policies defined by the original process consumers, not by the policies from process providers. A $coor_{context}$ can be initialized by a provider for subprocesses in a transaction, the subprocesses would then be enforced with policies defined by the provider. Here, the activities in the subprocesses of the overall process are Web services, i.e., atomic activities for process consumers. This is different from the distributed coordination of WS-Coordination, which is achieved by a chaining coordination [29].

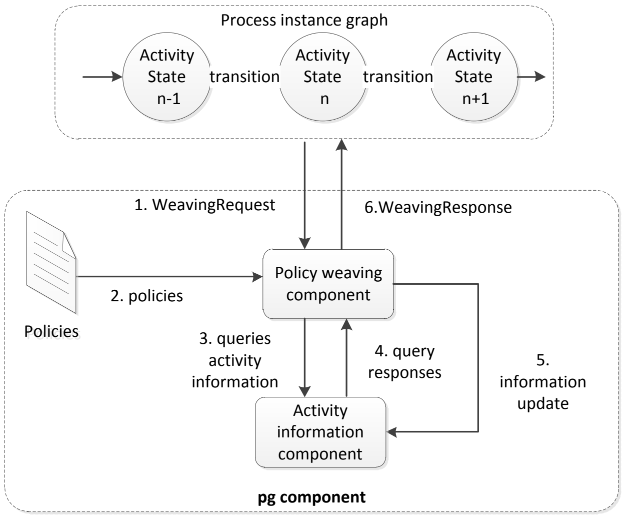

Figure 2. Information model and framework elements

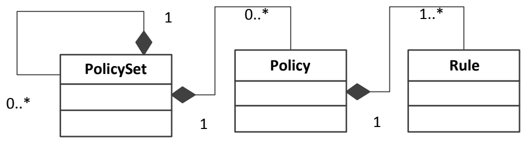

Figure 3. Core components of the policy language model

## 5. Policy model

The policy model for process consumers formalizes business policies as a user customization of provider-side business processes. As discussed in Section 3, existing policy models and approaches are not applicable for sharing pre-developed processes in multi-tenant cloud environment. We need a new policy model for consumers to formalize the business policies in pre-developed processes. The core of our new policy model provides a language model for consumers to express business policies for existing processes of providers as process customization metadata.

Before we describe the policy language model, we first introduce the basic information architecture (Figure 2). This also introduces the basic elements of the framework with a component for policy weaving of process governance $pg$.

- *Process instance graph* - is an execution instance derived from an activity-based process coordination protocol (Section 6). It sends a weaving request $w_{request} \in W_{request}$ to the policy weaving component for governance states of activities.
- *Policies* - the requirements or customization of processes are described in policies, which will be carried out by both the process consumer and provider through a coordination framework.
- *Policy weaving component* - weaves defined policies of the process consumer at process runtime into governance states of process activities. The weaving response $w_{response} \in W_{response}$ containing a policy decision is sent back to the process instance as a part of a contract which needs to be implemented by the provider of the business process.
- *Activity information component* - operates on information sources of processes for policy weaving. Information sources cover the *service profile* $SP$ - service endpoint reference and service context information; the *weaving history* $WH$ - $W_{response}$ history of policy weaving; *user logs* $UL$ - recording user actions.

### *5.1 Rule categorisation*

Our policy model is influenced by the XACML specification, which also influenced other proposed SOA policy models [36][37]. We adopt the three-level structure (*Rule*, *Policy*, *PolicySet*) to support nested policies for different administrative levels (Figure 3). Still, XACML only focuses on access control, and extensions such as [36] are not process-aware. Since rules are used as the basic policy elements of the policy model, our policy modelling starts with different categories of rules needed for different aspects of business policies. Business policies can be formalized as business rules for SOA governance. Business rule classifications [16][17] only show different types of formal expressions of business rules. The goal is formulating the business policies in a formal rule language for rule engines. To develop our own policy language, we need a classification to find a common connection between rules and processes that can be used for our policy model and coordination protocol development later on. Based on aspects of autonomic computing [38] and state-action policy modelling [39], we have developed a categorization schema that allows us to categorise rules *RU* for processes into four different categories based on the safe, controllable boundary of a business process execution. The safe boundary is defined in terms of rules, derived from business regulations and requirements which the business must conform to. Figure 4 explains the rule categories. It shows a process which has nine execution steps, represented by circles.

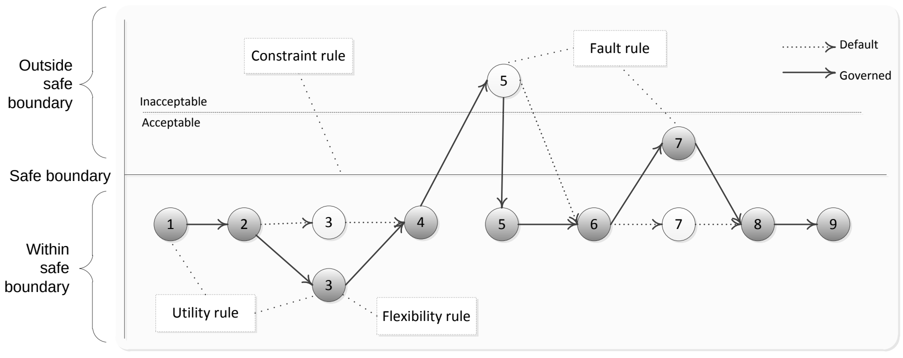

Figure 4. Rule categorization related to process execution

We define rules $RU = RU^{flexiblity} \cup RU^{constraint} \cup RU^{fault} \cup RU^{utility}$ with:

1) *Flexibility rules* ($RU^{flexiblity}$) - this category expresses business decisions *within the safe boundary* of the execution. The execution steps continue after the decisions are made. It is used to specify variable business decision logic for various expected business scenarios such as different customer types or frequently changing strategies (e.g., different discount rates over times). The business dynamics is the driving force. The purpose of this rule category is configuring business operations for business versatility and different business conditions.

2) *Constraint rules* ($RU^{constraint}$) - this category defines the *safe (controllable) boundary* of the process execution to restrict business behaviour. Constraint rules specify assertions that must be satisfied in all steps of the process execution, e.g., the availability of the payment service must be above 99%. The purpose of this category is to ensure that the business complies with relevant laws or regulations.

3) *Fault rules* ($RU^{fault}$) - this type defines system responses when a process crosses the safe boundary and constraints are violated, i.e., *outside the safe boundary.* Remedial strategies are required to avoid subsequential failure of the goal. Since constraint violations are viewed as 'faults' of process executions, this category is the fault rule. The fault rule can be further divided for acceptable and unacceptable business cases outside the safe boundary. Its purpose is handling the violations of business regulation compliance that may have occurred.

4) *Utility rules* ($RU^{utility}$) are rules that do not control or affect process execution. They define additional or utility actions that might need to be associated with process execution. The purpose of utility rules are for instance event notifications.

## *5.2 Rules*

Our policy model can now be modelled based on the different categories of rules. A *Rule* element $ru \in RU$ specifies the actual conditions under which defined governance actions are allowed to be performed. It follows the ECA paradigm like other policy models [40] [27]. Each rule contains *applicability predicates* and/or *condition predicates* as conditions to determine whether governance actions defined in the rule will be performed for a weaving request $w_{request}$. Rules are building blocks of a policy. A rule is a tuple $< \underline{os, ss,}\ \underline{cs,}\ \underline{acs,}\ \underline{fh,}\ \underline{ruid, pr, de} >$ with:

- An *Objects* element $os \in OS$ and an *ActivityStates* element $ss \in SS$ define the *applicability predicates* of the rule, i.e., the E part of the ECA.
- A *Conditions* element $cs \in CS$ defines the *condition predicates* of the rule, i.e., the C part of the ECA.
- If either the *applicability* or the *condition* predicate evaluate to *false* or *fault*, the governance actions contained in the *Actions* element $acs \in ACS$ of the $ru$ will not be performed, i.e., the A part of the ECA.
- A fault handler element $fh \in FH$ which contains actions when faults occur during rule weaving.

In addition to the main elements, a $ru$ has the following attributes and elements. Rule-Id *ruid* is a string identifier. Priority *pr* assigns a positive integer denoting the priority weight. Description $de \in DE$ is provided by the policy developer.
**Objects:** An *Objects* element $os \in OS$ defines the governance targets of the business process. It specifies what the rule applies to. $os = \{(os^{'}_{k}, sma) | k = 1, ..., n;\ os^{'}_{k} \in OS^{'};\ sma \in SMA\}$, where $sma \in SMA$ is a *SemanticMatchingAlgorithm* element, which will be described later. $os^{'}_{k} \in OS^{'}$ is a disjunctive sequence element *ObjectsAnyOf*. $os^{'}_{k} = \{os^{''}_{k}, sma | k = 1, ..., n;\ os^{''}_{k} \in OS^{''};\ sma \in SMA\}$, where $os^{''}_{k} \in OS^{''}$ is a conjunctive sequence element *ObjectsAllOf*. $os^{''}_{k} = \{o_n, sma | k = 1, ..., n;\ o_n \in O\ , sma \in SMA\}$, where *Object* element $o_n \in O$ represents a process element as a single governance target. $O = A \cup P \cup R \cup V$ with:

- *Activity* $a \in A$ is an implementation of a business task by a service, defined by $< na, sma >$. $a$ is identified by its name $na$. $sma \in SMA$ is defined as above.
- *Process* $p \in P$ contains a set of activities executed in a specific sequence, defined by $< wso, wsa, sma >$. $p$ is implemented by a composite service, identified by *WSoperation wso* and/or *WSaddress wsa* of the service reference of the process and $sma \in SMA$. A process itself could be an activity not different from other activities, but here, it is used to specify the policy scope. A $p$-specific policy will only apply to the process $p$ itself, but not to subprocesses of $p$.
- *Resource* $r \in R$ is a business object in a process for transferring data between business partners or activities, and defined by a tuple $< na, sma >$. A resource is identified by the *Name na* of the business object.
- *Violation* $v \in V$ is an occurrence of violating constraints, defined by a tuple $< tp, sma >$. A violation is identified by the constrained aspect of a business process, i.e., $tp \in TP$, a violation type.

**ActivityStates:** An *ActivityStates* element $ss \in SS$ defines the governance states of activities of the process. $ss = s_1 \vee ... \vee s_n\ for\ s_k \in S_g$ and $k = 1, ..., n$ is a disjunctive set of elements. $S_g = S_g^{flexibility} \cup S_g^{constraint} \cup S_g^{fault}$, where,

- ○ $S_g^{flexibility} = \{s_{man_{pre}val_{pre}}, s_{man_{pre}val_{post}}, s_{man_{post}val_{pre}}, s_{man_{post}val_{post}}\}$ is a set of governance states for $RU^{flexibility} \cup RU^{utility}$.
- ○ $S_g^{constraint} = \{s_{validating_{pre}}, s_{validating_{post}}\}$ is a set of governance states for $RU^{constraint} \cup RU^{utility}$.
- ○ $S_g^{fault} = \{s_{handling_{pre}}, s_{handling_{post}}, s_{cancelling}\}$ is a set of governance states for $RU^{fault} \cup RU^{utility}$.

These states are defined as follows:

(1) $s_{validating_{pre/post}}$ is a state of an activity execution for a *pg* component which enforces constraint rules defined for the activity. 'pre' and 'post' denote validation before or after the activity execution.
(2) $s_{man_{pre/post}val_{pre/post}}$ is a state of an activity execution for a *pg* which enforces flexibility rules defined for the activity through message manipulation. It contains a condition, 'pre' and 'post' denote that the manipulation happens before or after a validating pre/post state.
(3) $s_{handling_{pre/post}}$ is a state of an activity execution for a *pg* enforcing fault rules defined for the activity when violations occur. 'pre' and 'post' denote handling violations occurring at the $s_{validating_{pre}}$ or $s_{validating_{post}}$ states.
(4) $s_{cancelling}$ is a state of an activity execution for a *pg* enforcing the fault rules defined for the process if it cancels its previous execution effect.

There are a number of states defined by the coordination protocol for a process activity. $S_g$ comprises the nine governance states for *PG* components. The remaining activity states of the protocol are also associated with policies, but processes do not interact with *PG* components in the remaining states.

**Conditions:** A *Conditions* element $cs \in CS$ defines additional conditions for triggering actions on business processes. $cs = ce_1 \wedge ... \wedge ce_n$ for $ce_k \in CE$ and $k = 1, ..., n$ as a conjunctive sequence. A *ConditionExpression* element $ce \in CE$ is an XPath expression specifying a condition requirement on a data source $ds \in DS$. It returns a Boolean value on its evaluation. $ds =< w_{request}, SP, UL, WH >$.

**Actions:** An *Action* element $acs \in ACS$ defines a sequence of governance actions on processes. $acs = \{ac_k | k = 1, ..., n;\ ac_k \in \{CA \cup PA\};\ \#\{ac_k | ac_k \in PA\} \leq 1\}$. An individual *Action* element $ac \in AC$ defines a type of governance action. An *ac* can be either a consumer action *CA* or provider action *PA*, but there can be at most one provider action for an *acs*. These actions are designed for different rule categories with different parties to offer compensative customization.

A *ConsumerAction* element $ca \in CA$ is an action performed within *PG* components or available on the consumer side for governance without directly controlling process executions. They are needed for $RU_{utility}$ policies. For example, it is used to collect data required for subsequence control or monitoring. All consumer action elements as direct children of the *acs* will be weaved and executed immediately within a *pg* component when the rule is weaved. $CA = CA_{log} \cup CA_{suspend} \cup CA_{alert}$ is a set of supported consumer actions included in the policy model:

- ○ A log action $ca_{log} \in CA_{log}$ stores weaving request in the user log *UL*. A log level is an attribute to specify how much information needs to be stored.
- ○ A suspend action $ca_{suspend} \in CA_{suspend}$ suspends the current service for the consumer by updating the *ActiveTime* of the service in service profile *SP*. The *Time* attribute $t \in ca_{suspend}$ specifies a suspending time from the current time for the service. If $currentTime + t \leq ActiveTime$, then $ca_{suspend}$ is ignored.

- An alert action $ca_{alert} \in CA_{alert}$ notifies a relevant process stakeholder about the situation. Its *MailTo* attribute specifies a stakeholder email address.

A $ProviderAction$ element $pa \in PA$ is an action to directly control process executions on the provider side for governance needed for $RU_{flexiblity}$, $RU_{constraint}$, and $RU_{fault}$ policies. It also includes provider action types $PA^{'}$ resulting from policy combination or weaving, but they are not part of the policy language model for policy developers. $PA$ and $PA^{'}$ comply with the coordination protocol.

$PA$ contains a set of different provider action types in the policy model that are designed for different rule categories, thus are expected for different activity states. The activity states with rule categories are described above. The following table defines expected provider actions for $PA$ from policy developers.

| $S_g$ | **expected** $PA$ |
|---|---|
| $S_g^{flexibility}$ | $PA_{manipulate}$ |
| $S_g^{constaint}$ | $PA_{validate} \cup PA_{violate}$ |
| $s_{handling_{pre}} \in S_g^{fault}$ | $PA_{ignore} \cup PA_{replace} \cup PA_{cancel} \cup PA_{skip}$ |
| $s_{handling_{post}} \in S_g^{fault}$ | $PA_{ignore} \cup PA_{replace} \cup PA_{cancel} \cup PA_{retry} \cup PA_{compensate}$ |
| $s_{cancelling} \in S_g^{fault}$ | $PA_{compensate}$ |

These provider action elements are described as follows:

- $pa_{manipulate} \in PA_{manipulate}$ is a manipulation action to prepare the *Resource* data $r$ in a $w_{request}$ for the message adaptation requirement of the flexibility rules. The $pa_{manipulate}$ is executed immediately during rule weaving with the consumer, but the manipulated resource will be sent back to the provider containing a set of *Copy* operations to be executed in an *all or nothing* manner. The *Copy* operation is the same as BPEL <copy>, which is used for manipulating XML data. Exceptions caused by $pa_{manipulate}$ trigger the fault handler.
- $pa_{validate} \in PA_{validate}$ is a validation action for constraint rules to allow process execution steps to continue, if the process instance is within the safe boundary.
- $pa_{violate} \in PA_{violate}$ is a violation action for constraint rules to guide the process execution into a violated state, if the current process instance is outside the safe boundary. $pa_{violate}$ contains a set of child elements denoting a set of violation types $TY$ of the current process being violated.
- $pa_{ignore} \in PA_{ingore}$ is a remedial action defined for fault rules to guide the current process instance back to the business safe boundary without additional recovery. It ignores specified faults which do not affect the overall business goal.
- $pa_{replace} \in PA_{replace}$ is a remedial action for fault rules to guide the process instance to replace the service reference of the current activity by an alternative. $pa_{replace} =< io, scs >$, where the *InstanceOnly* $io$ attribute denotes two types of replace actions: 1) a process instance adaptation with temporary replacement (service replacement is only applied for current activity in process instance) and 2) a process adaptation action for continuous process improvement with permanent replacement. The activity replacement is applied to the current and following request instances in the current process. The *ServiceConditions* $scs$ element is used to specify a service reference for the activity implementation.
- $pa_{compensate} \in PA_{compensate}$ is a remedial action for process instances to take a compensation action for the current activity by executing a compensation activity. $scs \in pa_{compensate}$ is used to specify the service reference for the compensation. Services are implementations of activities that must be assigned for every activity. A service should be specified with $PA_{replace}$ or $PA_{compensate}$ action type, which contain service references of activities. A service selection mechanism is used to specify a service through the *ServiceConditions* $SCS$ element.

A $scs \in SCS$, $scs = \{sce_k | k = 1, ..., n;\ sce \in SCE\}$ is defined as a conjunctive sequence of conditions, which a service needs to satisfy for the activity to be executed. A $sce \in SCE$ is a *ServiceConditionExpression*, $sce =< ex, foc >$ is defined as a single condition on a service. The *expression* attribute $ex$ is an XPath 2.0 expression with the service profile $SP$ as the data source. The *Force* attribute $foc$ indicates if this condition is mandatory for a service selection.

- ◦ $pa_{cancel} \in PA_{cancel}$ is a remedial action for fault rules to cause a current process instance to cancel the process execution.
- ◦ $pa_{retry} \in PA_{retry}$ is a remedial action to wait a period of time before retrying the current fault causing activity. It has a *waitFor* attribute complying with BPEL time expressions. Immediate retry without waiting can be achieved by setting zero as the waiting time of a $pa_{retry}$.

Provider action types of $PA^{'}$ are not defined in the policy language model, but can result from policy rule combination and weaving. They are needed for the coordination protocol to support additional, composite provider actions for a cache mechanism and fault handling. More details will be provided later.

- ◦ $pa_{com+ign} \in PA_{com+ign}$ is a composite provider action which is composed of a $pa_{compensate}$ and a $pa_{ignonre}$ action in a sequence.
- ◦ $pa_{com+rep} \in PA_{com+rep}$ is a composite provider action which is composed of a $PA_{compensate}$ and a $pa_{replace}$ action in a sequence.
- ◦ $pa_{undefined} \in PA_{undefined}$ indicates that no policy/rule is defined for the related activity state of an activity on the policy weaving, i.e., all policies or rules fail on an activity state evaluation for a $w_{request}$.
- ◦ $pa_{unexpected} \in PA_{unexpected}$ indicates policies or rules which do not have any expected provider action in policy weaving. Thus, all specified provider actions in the rules or policies are not expected for a $w_{request}$. A $pa_{unexpected}$ becomes the result provider action for a $w_{request}$ on weaving in this case.
- ◦ $pa_{undetermined} \in PA_{undetermined}$ indicates a situation which cannot determine between $pa_{undefined}$, $pa_{unexpected}$ and expected provider actions for a $w_{request}$.

### *5.3 Fault handler*

A *FaultHandler* element $fh \in FH$ specifies what should be done if exceptions occur when evaluating a *Conditions* element $cs \in CS$, or executing a $pa_{manipulate} \in PA_{manipulate}$ action. Since these elements involve XPath and XSLT expressions defined by policy developers, exceptions may occur during rule weaving because of incorrect expressions. If exceptions occur, the fault handler will be called and involves the current rule weaving. A fault handler contains the *Actions* element $acs \in ACS$, which specifies a set of actions for fault handling on policy weaving. $acs = \{ac_k | k = 1, ..., n;\ ac_k \in \{CA \cup PA^{fh}\}, \#\{ac_k | ac_k \in PA^{fh}\} \leq 1\}$, where $PA^{fh} \subset PA$. The following table defines the expected provider actions for fault handling with regard to different activity states of a weaving request.

| $S_g$ | **expected** $PA^{fh}$ |
|---|---|
| $S_g^{flexibility} \cup S_g^{constraint}$ | $PA_{validate} \cup PA_{violate}$ |
| $s_{handling_{pre}} \in S_g^{fault}$ | $PA_{ignore} \cup PA_{replace} \cup PA_{cancel} \cup PA_{skip}$ |
| $s_{handling_{post}} \in S_g^{fault}$ | $PA_{ignore} \cup PA_{replace} \cup PA_{cancel} \cup PA_{retry} \cup PA_{compensate}$ |
| $s_{cancelling} \in S_g^{fault}$ | $PA_{compensate}$ |

A provider action of a fault handler of a rule is expected to be in the same rule category as the provider action of the rule, except $RU^{flexibility}$. $PA_{manipulate}$ can not be defined in a fault handler. For handling exceptions for $PA_{manipulate}$ of a rule, $PA_{validate}$ or $PA_{violate}$ is expected in a fault handler.

If the fault handler is *Absent*, or an *Expected* provider action $pa^{fh}$ is not included in the defined fault handler of a rule, a $pa_{undetermined}$ is the provider action when the exceptions occur, i.e., a $pa_{undetermined}$ is the default provider fault action.

## 5.4 Obligations

An *Obligations* element $obs \in OBS$ contains a set of obligations $obs = \{ob_k | k = 1, ..., n;\ ob_k \in OB\}$ with an *Obligation* element $ob \in OB$, $ob = \{(ca_k,\ pa\_t) |\ k = 1, ..., n; ca_k \in CA; pa\_t \in \{typeOf(PA \backslash PA_{manipulate})\}\}$ specified as a set of consumer actions, which is only executed on the consumer side when a type of provider action is executed on the provider side for a policy weaving request. The type attribute $pa\_t$ specifies a type of provider action.

A rule component should have at most one obligation, as a rule can only have at most one provider action. It cannot be associated with $PA_{manipulate}$ provider actions as they are executed on the consumer side. The obligation elements might be merged when a Rule/Policy/PolicySet is weaved for a weaving request. For example, if two obligations for logging details when a process instance is cancelled, then the details are only logged once when the process is cancelled.

## 5.5 Policy and PolicySet

A *Policy* element $po \in PO$ is a tuple $< os,\ ss,\ RU,\ obs,\ sa,\ cca, rca, poi, pr, de >$, where $RU$ is a set of *Rules*. A *PolicySet* element $pos \in POS$ is used to combine separate policies into a single policy. It allows policy developers to have nested policies. A policy set is a tuple $< os, ss, PS, POS, obs, sa, cca, rca, posi, pr, de >$, where $PS$ is a set of *Policy* elements and $POS$ a set of *PolicySet* elements. For the above, $os \in OS$ is an *Objects* element, $ss \in SS$ is an *ActivityStates* element, $obs \in OBS$ is an *Obligations* element, *poi* and *posi* are identifications, *pr* is the priority weight, $de \in DE$ is a description element. $sa \in SA$ is a *SequencingAlgorithm*, $cca \in CCA$ is a *ConstraintCombiningAlgorithm*, $rca \in RCA$ is a *RemedyCombiningAlgorithm*. The algorithms $SA, CCA, RCA$ will now be introduced.

## 5.6 Related algorithms

A set of algorithms, defined in the policy model, allow policy developers to specify or configure the weaving behaviour. We briefly summarise algorithms of the policy model (details are available in [41]).

**Semantic matching algorithm** $sma \in SMA$ specifies the algorithms used for semantic similarity measurement between policy objects and attributes of a weaving request. A policy can target a wide range of objects without matching the exact identification. For $sma =< ty,\ de >$, $ty$ denotes the type of built-in algorithms for semantic similarity and $de$ denotes the matching degree.

Some policy objects, such as activities, can have similar or equal semantics, but not the same identification with different process providers. Through a semantic matching configuration, the policy can easily be applied to interesting objects of all processes from different providers. For example, a new policy can be applied to 'payment' related activities. A semantic matching configuration could easily apply the policy to 'process payment', 'pay' or 'repayment' activities.

**Sequencing algorithm** $sa \in SA$ specifies the weaving sequence on a collection of *Rule/Policy/PolicySet* components within a *Policy/PolicySet* component. This allows to specify an action execution sequence in a governance state. A $sa \in SA$ has a type attribute indicating the type of algorithm for sequencing. *Ordered* (based

on the order they are listed in a component) and *PriorityBased-QuickSort* (based on the *priority* attribute of each component) are two built-in algorithms.
**Policy combining algorithms:** For multiple policies (developed at different times or by different developers) and nested policies, a potential problem are conflicting action types from multiple rules on a weaving request. As a consequence, we need algorithms to combine multiple provider actions into a single joint provider action.

The following table associates the algorithms to different activity states. The algorithms are ignored for an unrelated activity state.

| $S_g$ | **Defined combining algorithm** |
|---|---|
| $S_g^{flexiblity}$ | n/a |
| $S_g^{constraint}$ | $CCA$ |
| $S_g^{fault}/s_{cancelling}$ | $RCA$ |
| $s_{cancelling} \in S_g^{fault}$ | default |

One way of solving policy conflicts is to assign priority values to policies to define a precedence [42]. This is done by two sets of combining algorithms in XACML for constraint and fault rules, respectively. The policy combining algorithms allows to give priority to different types of actions for constraint and fault rules.

The combining algorithm is not required for $S_g^{flexiblity}$, as $PA_{manipulate}$ actions are executed immediately when weaving the rules. For the $s_{cancelling}$ state, since there is only one type of a provider action ($PA_{compensate}$) that is expected, a simple default combining algorithm is assigned without policy developer involvement. It does not combine different types of actions, but merges the same type of actions. In this case, the combination is based on the union of the child elements of $PA_{compensate}$ actions. Similarly, consumer actions of obligations are also merged and executed when a provider action is decided for a $weaving_{response}$.

The *ConstraintCombiningAlgorithm* $cca \in CCA$ element is defined for combining provider actions with constraint rules. A $cca$ has a type attribute denoting the type of built-in algorithms that have different behaviour resulting in different combining conclusions. A $cca$ is defined to be one of following types:

(1) *Pa-Violate-Override-Through-All* gives priority to $PA_{violate}$ actions over $PA_{validate}$ actions. The *Through-All* means that all of rules or policies are weaved, even when the type of action has been decided. The purpose is a) gathering complete violation information needed for violation handling and b) making sure all necessary consumer actions are weaved.
(2) *Pa-Validate-Override-Through-All* is similar to the above, but gives priority to a $PA_{validate}$ action.
(3) *Pa-Violate-Unless-Pa-Validate-Through-All* gives a final decision with $PA_{violate}$ as the default.
(4) *Pa-Validate-Unless-Pa-Violate-Through-All* gives a final decision with $PA_{validate}$ as the default.

The *RemedyCombiningAlgorithm* $rca \in RCA$ element is defined for combining provider actions resulting from fault rules. For $rca =< ty, DS >$, $ty$ is an attribute denoting the type of algorithms which have different behaviour resulting from different combining conclusions and $DS$ specifies a defined sequence of provider actions as an input parameter of one type of algorithm. A $rca$ is of one of following types:

(1) *Defined-Sequence-Overrides-Through-All* gives a priority ranking according to the sequence of the defined provider action types for fault rules. The first action type in the sequence has the highest priority. Hence, when a list of remedies is available from defined related policies, the one with the highest priority is chosen. The *Through-All* denotes that all rules or policies are

weaved, even if the type of provider action has been decided.

(2) *Pa-Ignore-Unless-Defined-Sequence-Through-All* gives a strict final decision with $PA_{ignore}$ as default. In this case, if no remedy is found from defined related policies or rules, the violations are ignored.

(3) *Pa-Cancel-Unless-Defined-Sequence-Through-All* is similar. It cancels the process instance which has violations not covered by fault policies or rules.

## 6. Coordination

A coordination framework with protocols as contracts makes process consumers and providers work together for governance to ensure that defined policies are enforced in multi-tenant environments. We have already introduced the coordination model in the architectural overview. We now focus on message exchange protocols for policy enforcement between participants and coordinators, and also a cache mechanism to reduce the overhead of coordination conversations caused by message exchange. BPEL templates are offered to implement the protocols with multi-tenancy capability for providers.

### *6.1 Process activity protocol*

The process activity protocol defines a coordination type for coordination conversations. It relies on our coordination model. A coordination conversation is established upon coordination of all activities which are within the overall process and subprocesses for the consumer. The conceptual modelling of the coordination protocol is activity-centric, so it can be applied to any process regardless of the flow logic and without losing aspects related to business processes. This coordination protocol applies to all activities of business processes. A coordination protocol comprises three components that define $coor_{context}$ which re described in the following subsections: a protocol message schema, a Finite State Machine (FSM) of $COOR^c$ and $COOR^p$ and a cache function specification of $COOR^p$.

#### *6.1.1 Protocol message schema*

The protocol message schema defines the message data structure for the communication between $COOR^c$ and $COOR^p$ for the extension element of $COOR_{context}$. Two elements defined in the schema are $PAP_{request}$ and $PAP_{response}$ (request and response of protocol services). A $pap_{request} \in PAP_{request}$ is defined as a tuple $< p, a, r, v', s >$, which extends the $COOR_{context}$ to form the $W_{request}$, where

- process $p \in P$ contains process name and a service reference for the process.
- activity $a \in A$ contains its name and the service reference of the service which implements the activity.
- resource $r \in R$ contains the business object involved for the current activity state, as a free extensible element (xsd:any) for any type of business objects.
- set of violations $v' \subseteq V$ contains available violation information for the activity.
- state $s \in S_g$ is the current governance state of the activity. $S_g$ is defined in the FSM part of the protocol.

A $pap_{response} \in PAP_{response}$ extends $COOR_{context}$ to form $W_{response}$. It is defined as a transition action $ta \in TA$, i.e. an abstract type of a set of concrete transition actions mapped to provider actions, which are described in the policy model.

The $W_{request}$ is defined for request messages of $COOR^p$, the $W_{response}$ is defined for response messages of protocol services of $COOR^c$ (Figure 5). All messages are

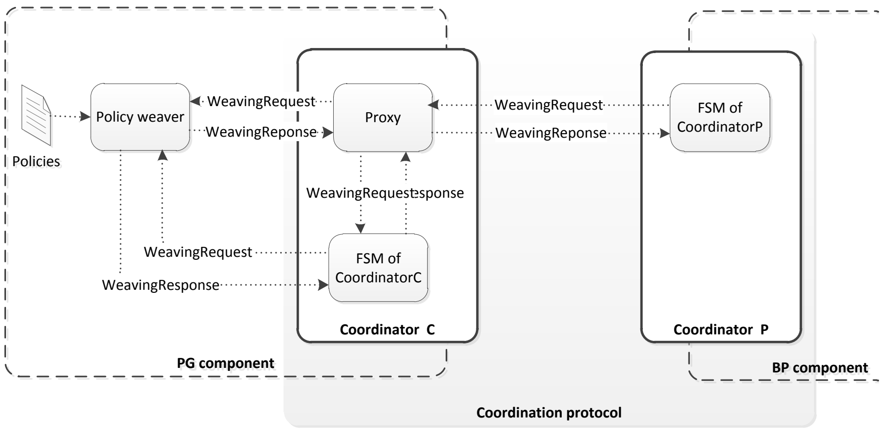

Figure 5. Message flow diagram

wrapped as coordination context information. The policy weaver can be viewed as an implementation of a protocol service of a $coor^c$ depending on the coordination protocol, because of the difference between the policy model and the coordination message definition. The $PA$ needs to be transformed to $TA$ as the $W_{response}$ before sending out from the policy weaver. Still, the policy weaver does not directly communicate with a $coor^p$, because the FSM of the coordination protocol is divided into two parts. A proxy service sends a $w_{request}$ to a policy weaver or a FSM of $COOR^c$ depending on the activity state.

### *6.1.2 FSMs $COOR^c$ and $COOR^p$ of the protocol*

The process activity protocol defines a given level of runtime governability for processes and the responsibilities of process providers and consumers as a contract. Governability should deal with all rule categories. We formalize this as a finite state machine (FSM) for the protocol. We define a separate FSM for every activity in a process and describe the behaviour of $COOR^c$ and $COOR^p$ in conversations. FSMs help to instrument the governance states into the process flow.

A complete FSM is divided into two parts for a protocol, which are responsible for $COOR^c$ and $COOR^p$, respectively. The FSM of $COOR^c$ is a submachine state of the FSM of $COOR^p$. The process providers only follow the part of the protocol which is defined for $COOR^p$. The consumers follow the FSM of $COOR^c$. Since the implementation of the FSM will be executed at the consumer and provider separately, the $COOR^c$ must have sufficient information about the process execution for its part of the state machine execution, as the process executes on the provider side. In our design of the entire FSM, the FSM of $COOR^c$ defined for the submachine state in the FSM of $COOR^p$ is isolated from the business process. As a result, the protocol message schema only covers the complete information about the activity rather than the process state information. The execution of the FSM of $COOR^c$ does not require information other than the $W_{request}$, which is defined in the protocol message schema. The execution of the FSM of $COOR^p$ does not require information other than $W_{response}$. The reason behind this design is that, firstly, the same protocol message schema can be used for different coordination protocols. A process consumer can customize the FSM of $COOR^c$ for itself without affecting the FSM of $COOR^p$ and other process consumers. Secondly, it avoids possible complexity in state machine implementation for both sides. One side does not need to know the implementation details of the other side.

The purpose of the two-part design is that it reduces the number of governance states in the FSM of $COOR^p$, hence reduces the protocol message exchange times required between $COOR^c$ and $COOR^p$ for coordination conversations. It reduces performance overhead caused by communication between the protocol services, as network-based communication between a process consumer and providers can be expensive. The disadvantage is that it increases the complexity on the consumer side, because of the FSM of $COOR^c$ is implemented by consumers. However, a different protocol can be defined with $COOR^p$ that has a complete FSM.

The FSM of $COOR^p$ specifies the protocol which is responsible for $COOR^p$. It is defined as a 5-tuple $(S, s_{start}, F, TA, \delta)$, where

- $S = S_g \bigcup S_{\neg g}$ is a set of states. $S_g$ are governance states $\{s_{man\_val_{pre}}$, $s_{man\_val_{post}}$, $s_{handling_{pre}}$, $s_{handling_{post}}$, $s_{cancelling}\}$ directly involved with process consumers or policies. $S_{\neg g}$ is the set of non-governance states $\{s_{start}$, $s_{violated_{pre}}$, $s_{executing}$, $s_{replacing}$, $s_{waiting}$, $s_{skipping}$, $s_{violated_{post}}$, $s_{compensating}$, $s_{com+rep}$, $s_{com+ign}$, $s_{completed}$, $s_{end}\}$ not directly involved with process consumers.
- $s_{start} \in S_{\neg g}$ is an initial state. The activity coordination can only be started by the process provider, and is not directly involved with consumers.
- $F \subseteq S_{\neg g}$ is a set of final states $\{s_{end}\}$.
- $TA = TA_g \bigcup TA_{\neg g}$ is a set of input symbols of transaction actions. $TA_g$ is a set of transition actions $\{ta_{violate}$, $ta_{validated}$, $ta_{ignore}$, $ta_{replace}$, $ta_{skip}$, $ta_{cancel}$, $ta_{compensate}$, $ta_{retry}$, $ta_{com+ign}$, $ta_{com+rep}\}$ expected from process consumers. $TA_{\neg g}$ is a set of transaction actions, which are not expected from consumers $\{0,\ 1\}$. The input stream of the FSM regarding $TA_{\neg g}$ is decided by the process provider based on the process state information which is not covered by the FSM, as the FSM is only activity scoped.
- $\delta$ is a transition system $\delta : S \times TA \rightarrow S$ (a transition graph, see Fig. 6).

The FSM of $COOR^p$ introduces two submachine states: $s_{man\_val_{pre}}$ before $s_{executing}$, and $s_{man\_val_{post}}$ after $s_{executing}$. They contain the FSM for $COOR^c$. It enables message adaptations and constraint validations before and after $s_{executing}$.

The FSM for the $s_{man\_val_{pre}}$ submachine state specifies the protocol which is responsible for $COOR^c$. It is defined as a 5-tuple $(S, s_{start}, F, TA, \delta)$, where

- $S = S_g \bigcup S_{\neg g}$ is a set of states. $S_g$ is a set of governance states, $\{s_{man_{pre}val_{pre}}$ $s_{man_{post}val_{pre}}$, $s_{validating_{pre}}\}$. The $S_{\neg g}$ is a set of non-governance states $\{s_{start}$, $s_{replacing}$, $s_{violated_{pre}}$, $s_{executing}\}$ from the parent FSM.
- $s_{start} \in S_{\neg g}$ is an initial state from the parent FSM.
- $F \subseteq S_{\neg g}$ is a set of final states $\{s_{violated_{pre}}$, $s_{executing}\}$ from the parent FSM.
- $TA = TA_g \bigcup TA_{\neg g}$ is a set of input symbols of transaction actions. $TA_g$ is a set of transaction actions $\{ta_{validated}$, $ta_{violate}\}$ expected from process consumers or policies. $TA_{\neg g}$ actions are not expected from process consumers $TA_{\neg g} = \{1\}$.
- $\delta$ is a transition system $\delta : S \times TA \rightarrow S$ (a transition graph, see Fig. 7).

The submachine state consists of governance states allowing constraint rule validation before $s_{executing}$, and message adaptation before and after $s_{validating_{pre}}$. When a $coor_c$ receives a $w_{request}$, indicating that an activity of a process is in the $s_{man\_val_{pre}}$ state, the proxy service of $coor_c$ enters the submachine of the $coor_c$ implementation defined for the $s_{man\_val_{pre}}$ state. The FSM of $coor_c$ sends its $w_{request}$ to the policy weaver. The $s$ of $w_{request}$ received by the policy weaver is a governance state defined in the FSM of $COOR_c$. After completing the FSM of $COOR_c$, the final $w_{response}$ is sent to $COOR_p$. The FSM of $COOR_c$ for the $s_{man\_val_{post}}$ submachine state is identical to $s_{man\_val_{pre}}$, except $S_g = \{s_{man_{pre}val_{post}}$, $s_{man_{post}val_{post}}$, $s_{validating_{post}}\}$.

### *6.1.3 Cache of the process activity protocol*

The coordination cache mechanism is designed to improve coordination efficiency by reducing protocol message communications between two types of subcoordina-

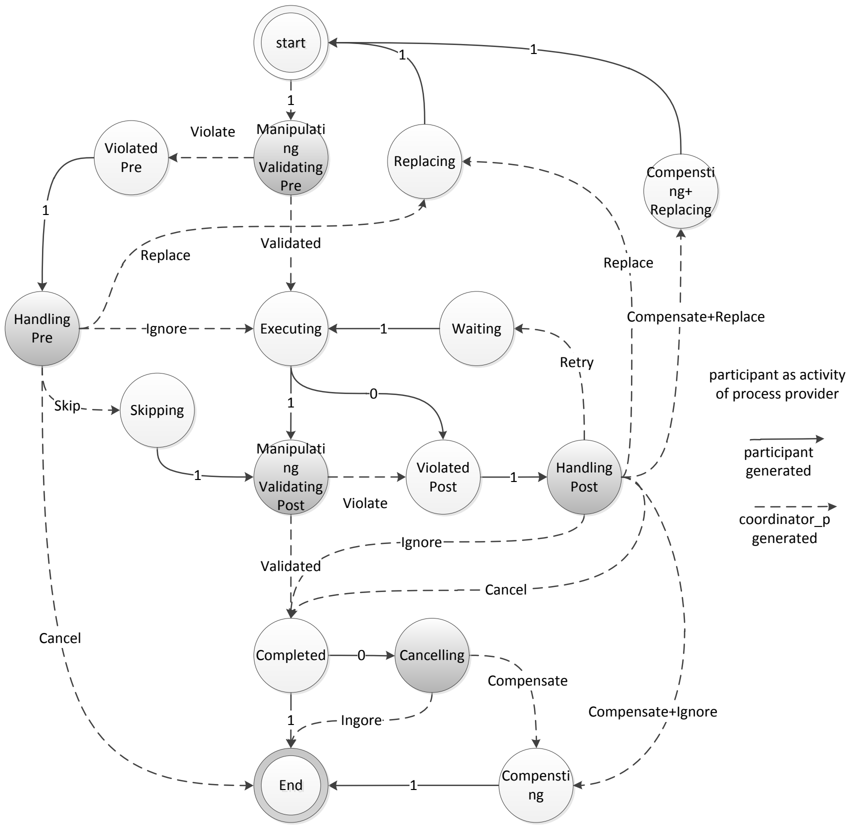

Figure 6. Transition graph of FSM in $COOR^p$

tors. It caches message responses of a coordination protocol of coordination conversations. For a $coor^p$, it remembers policy information defined or a final provider action at a particular point of a business process by consumers to decide what interaction pattern is needed between a $coor^c$ and $coor^c$ at the point.

There are three types of interaction patterns for the cache mechanism (Table below). It results in three default extra transition actions for all coordination protocols: $TA_c = \{ta_{undefined}, ta_{unexpected}, ta_{undetermined}\}$. They are mapped to $PA_{undefined}$, $PA_{unexpected}$, $PA_{undetermined}$ (Section 5). The three extra actions are mapped to transition actions specified in the coordination protocol on governance.

| **Transition action - Interaction pattern and description** |
|---|
| $ta_{undefined}$ - *No interaction* The protocol service of $coor^p$ does not try to communicate with $coor^c$ on a governance state. A mapped transition action is applied. |
| $ta_{unexpected}$ - *One-way notification* The protocol service of $coor^p$ sends a message or $w_{request}$ to $coor^c$ using a one-way interaction mode in a governance state. Then, the consumer actions defined in the policies are executed. A mapped transition action is applied. |
| $ta_{undetermined}$ and others from specific protocol $TA_g$ - *Synchronous request/response* The protocol service of $coor^p$ communicates with $coor^c$ and waits for a provider action. The cache is also updated with the provider action from the consumer. A mapped transition action is applied if the returned transition action is one of $ta_{undefined}$, $ta_{unexpected}$, $ta_{undetermined}$. |

In the first two cases of the table, the $COOR^p$ does not interact with the $COOR^c$ or uses a one-way notification interaction mode. Thus, process execution is not blocked to wait for the consumer to complete the weaving. Hence, the performance overhead is reduced in these cases. Depending on the coordination protocol defined

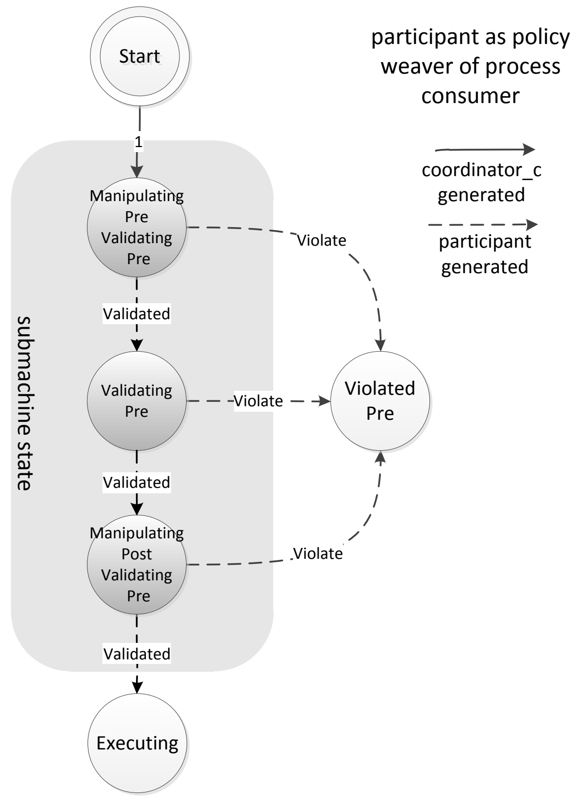

Figure 7. Transition graph of FSM in $COOR^c$

for the coordination, a cache mechanism may be implemented in $coor^c$. In this case, it does not reduce the communication overhead between the $coor^c$ and $COOR^p$, but the overhead of policy weaving. Since the weaving is not in the scope of coordination protocols, the caching in the $COOR^c$ is not defined in the coordination model.

The cache of the process activity protocol defines the cache function on $COOR^p$. It includes a protocol-specific data set of cached data ($PS \subset CAD$) and an action mapping table ($TA_c \rightarrow TA_g$). $PS$ specifies additional conditions for cached results. These additional conditions are relevant for the elements of a request of a protocol message of $COOR^p$ ($W_{request}$). In this protocol, a $pa \in PS$ is defined as $<s>$ and $s \in S_g$. In other words, a final provider action resulting from policy weaving is expected to be of the same interaction type (described in the coordination cache function) for any $w_{request}$ with the same $s$ regardless of other elements when the cache is enabled. More conditions can be added for different types of coordination protocols, such as the activity name or a service reference for a activity.

In the FSM definition of the protocol, transition actions are explicitly defined for transitions from governance states to governed states. However, process consumers have three types of extra transition actions $TA_c$ (defined in the coordination cache) for the cache function. They are not included in the $TA_g$ defined in the process activity protocol. Hence, the protocol also defines the mapping ($\mapsto$) from $TA_c$ to $TA_g$. In this protocol, the mapping for $COOR^p$ is defined as follows:

$$\forall s \in \{s_{man\_val_{pre}},\ s_{man\_val_{post}}\}, ta \in TA_c.\exists ta_{validated} \in TA_g.(ta \mapsto ta_{validated})$$
$$\forall s \in \{s_{handling_{pre}},\ s_{handling_{post}}, s_{cancelling}\}, ta \in TA_c.\exists ta_{ignore} \in TA_g.$$
$$(ta \mapsto ta_{ignore})$$

The mapping for $COOR^c$ is defined as follows:

$$\forall s \in \{s_{man_{pre}val_{pre}}, s_{man_{post}val_{pre}}, s_{man_{pre}val_{post}}, s_{man_{post}val_{post}}\}, ta \in TA_c. \\ \exists ta_{validated} \in TA_g.(ta \mapsto ta_{validated})$$
$$\forall s \in \{s_{validating_{pre}}, s_{validating_{post}}\}, ta \in TA_c. \exists ta_{validated} \in TA_g. \\ (ta \mapsto ta_{validated})$$

The mapping for $COOR^c$ is not restricted for a consumer, if *coor*$^c$ is on the consumer side. The consumer can change at any time, as the process providers does not need to be aware of changes and does not affect other process consumers.

For the protocol implementation, we designed a set of templates (Chapter 6.4 of [41], [43]) for BPEL development to avoid platform dependency. In this case, the protocol is implemented with a BPEL process as a *coor*$^p$ for activities. The BPEL contains the flow logic to be executed and can be driven by protocol messages. A process instance, not the BPEL process, is associated with a coordination conversation belonging to a consumer to enable multi-tenancy. The BPEL transaction scope concept is applied for implementing the protocol with BPEL for supporting LRTs. LRTs in BPEL are centred on scopes and scopes can be nested. Nested scopes can be standalone BPEL subprocesses which are activities of the parent process. When a fault occurs in a BPEL process, all previously committed activities can either be compensated within the fault process, or compensated as an activity in its parent.

## 7. Use Case and Evaluation

The evaluation covers both the policy model and coordination framework, but with different objectives.

### *7.1 Use case - Policy model evaluation*

The objective is to demonstrate the validity of the policy model with use case examples to show how business policies are expressed in our policy language, and to evaluate if different aspects of business policies are covered for a comprehensive customization. The aspects referred to here are aspects of autonomic computing [38] – configuration, healing, optimization, protection – which can be specified as policies here. These aspects are similar to functional areas in distributed system management [18][44], i.e., configuration, fault, performance, security management, and adaptive systems [28] with configuration, correction, optimization, prevention.

We briefly describe a simple business policy example for the optimization and healing aspects: *Credit card processing should be completed quickly (expected in less than 700 ms) without fault.* It demonstrates various aspects business policies that are expressible in our policy language. We have three rules in a policy - *cardProcessingPolicy10* (Listing 1) for the business policy. The policy targets both *Visa* and *MasterCard Card Processing* activities.

Listing 1 cardProcessingPolicy10

```
<p1:Policy policyId="cardProcessingPolicy10" priority="0">
 <p1:Objects><p1:ObjectsAnyOf><p1:ObjectsAllOf><p1:Activity><Name>Visa Card
      Processing</Name></p1:Activity></p1:ObjectsAllOf><p1:ObjectsAllOf><p1:Activity>
      <Name>MasterCard Card Processing</Name></p1:Activity></p1:ObjectsAllOf></
      p1:ObjectsAnyOf></p1:Objects>
 <p1:ActivityStates/>
 <p1:Rule priority="0" ruleId="constraintRule10">...</p1:Rule>
 <p1:Rule priority="0" ruleId="retryRemedyRule10">...</p1:Rule>
 <p1:Rule priority="0" ruleId="replaceRemedyRule10">...</p1:Rule>
 <p1:ConstraintCombiningAlgorithm type="Pa-Violate-Unless-Pa-Validate-Through-All"/>
```

```
8   <p1:RemedyCombiningAlgorithm type="Pa-Cancel-Unless-Defined-Sequence-Through-All"><
        DefinedSequenceElement>Pa-Retry</DefinedSequenceElement><DefinedSequenceElement
        >Pa-Replace</DefinedSequenceElement> ... </p1:RemedyCombiningAlgorithm>
9   <p1:SequencingAlgorithm type="Ordered"/>
10 </p1:PolicySet>
```

The first rule is a constraint rule - *constraintRule10* (Listing 2), specifying a condition to check the service performance of the current activity (line 3). If the performance is slower than required, a $pa_{violate}$ action is executed to guide the process instance to a performance violation state. A $ca_{suspend}$ consumer action is also defined to avoid the service to be selected for any activity or process for the next 5 hours. The rule has a fault handler (line 6) specifying the performance violation which is expected if condition checking of the rule is faulty, but the $ca_{suspend}$ actions is not performed, as this is not defined in the fault handler.

Listing 2 constraintRule10

```
1 <p1:Rule priority="0" ruleId="constraintRule10">
2   <p1:ActivityStates><p1:ActivityState>Validating-Pre</p1:ActivityState></
        p1:ActivityStates>
3   <p1:Conditions><p1:ConditionExpression>exists(//ServiceProfile/ServiceReference[(
        child::Ws-address =//WeavingRequest/Activity//Ws-address and child::Operation
        =//WeavingRequest/Activity//Operation) and descendant::Performance&gt;700])</
        p1:ConditionExpression></p1:Conditions>
4   <p1:Actions><p1:Ca-Suspend Time="P0Y0M0DT5H"/><p1:Pa-Violate><p1:Violation><Type>
        QoS:Performance</Type></p1:Violation></p1:Pa-Violate>
5   </p1:Actions>
6   <p1:FaultHandler><p1:Pa-Violate><p1:Violation><Type>QoS:Performance</Type></
        p1:Violation></p1:Pa-Violate></p1:FaultHandler>
7 </p1:Rule>
```

The second and third rules are fault rules (Listing 3), defining remedy actions for both effect violation and performance violations. The second rule *retryRemedyRule10* targets the *Functional:Effect* violation (line 2). It specifies a $pa_{retry}$ remedial action for the violation with a condition, which specifies for a current service that a $pa_{retry}$ action is executed less than 5 times on the provider side within the last minute. Otherwise, $pa_{retry}$ is not expected. The third rule - *replaceRemedyRule10* defines a $pa_{replace}$ for both *Functional:Effect* and *QoS:Performance* violations. The replacement service is defined with a mandatory condition on the context, a weak condition on preferred service performance for the activity. Hence, a fast performance service would be selected. An obligation is also defined (line 10) to log the replacement event, if $pa_{replace}$ is executed on the provider side. The remedy combining algorithm of the policy (Listing 1 line 8) specifies that $pa_{replace}$ is the preferred remedy over the $pa_{retry}$ if both remedies are applicable.

Listing 3 retryRemedyRule10 and replaceRemedyRule10

```
1  <p1:Rule priority="0" ruleId="retryRemedyRule10">
2    <p1:Objects><p1:ObjectsAnyOf><p1:ObjectsAllOf><p1:Violation><Type>Functional:Effect<
         /Type></p1:Violation></p1:ObjectsAllOf></p1:ObjectsAnyOf></p1:Objects>
3    <p1:Conditions><p1:ConditionExpression>count(//Pa-ActionLog/Pa-Action[@type="Pa-
         Retry" and @time > (current-dateTime()- xdt:dayTimeDuration('PT1M')) and
         descendant::ServiceReference])&lt;=5</p1:ConditionExpression></p1:Conditions>
4    <p1:Actions><p1:Pa-Retry WaitFor="PT0M"/></p1:Actions>
5  </p1:Rule>
6  <p1:Rule priority="0" ruleId="replaceRemedyRule10">
7    <p1:Objects><p1:ObjectsAnyOf><p1:ObjectsAllOf><p1:Violation><Type>QoS:Performance</
         Type></p1:Violation></p1:ObjectsAllOf><p1:ObjectsAllOf><p1:Violation><Type>
         Functional:Effect</Type></p1:Violation></p1:ObjectsAllOf></p1:ObjectsAnyOf></
         p1:Objects>
8    <p1:Conditions/>
9    <p1:Actions><p1:Pa-Replace InstanceOnly="false"><p1:ServiceConditions><
         p1:ServiceConditionExpression force="false" expression="/Context//Performance&
         lt;700"/><p1:ServiceConditionExpression force="true" expression="/Context//
         Trust&gt;5"/></p1:ServiceConditions></p1:Pa-Replace></p1:Actions>
10   <p1:Obligations><p1:Obligation Type="Pa-Replace"><p1:Ca-Log level="4"/></
         p1:Obligation></p1:Obligations>
11 </p1:Rule>
```

The examples demonstrate that our policy modelling can deal with thr issues that affect policy modelling, such as different aspects of business policies or re-

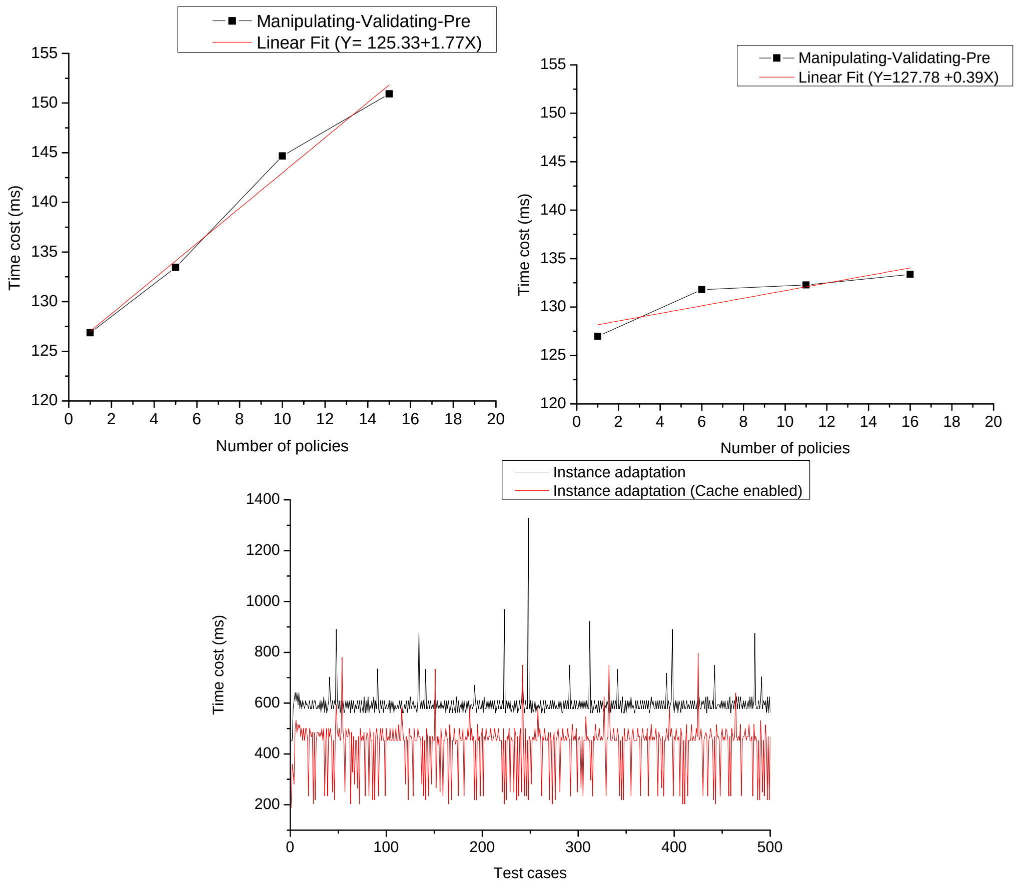

Figure 8. 1) Overhead with related policies, 2) Overhead with unrelated policies, 3) Overhead with process instance adaptation

quirements, distributed to consumers and targeting business processes. Along with other empirical evaluations [41], this confirms the comprehensiveness of the model.

## *7.2 Experiments - Coordination framework evaluation*

Our discussion of the coordination framework focuses on evaluating the effectiveness and performance overhead. Effectiveness means that the process can be governed in a distributed and multi-tenant environment, i.e., policies are enforced on business process executions for multiple consumers while enabling business goals to be achieved. We designed test cases involving two consumers on a process and configuration. Our study of effectiveness is divided in two stages: first, we validate effectiveness for a single process consumer; then, we validate effectiveness for two concurrent process consumers, examining its multi-tenancy capability. The *purchase order checkout BPEL process* is developed for the experimental setup. A set of alternative services are also available for test cases related to $PA_{replace}$ remedies. The service context information required for constraint validation and service selection is manually and randomly assigned. A test case comprises of five parts of information (see Chapter 6 [41] for details).

(1) Process: target process of test case (some cases address a sub-process level).
(2) Defined input: a section of SOAP message of the business process input that contains the business object information.
(3) Defined policies: policies defined for the business process.
(4) Expected process activity log: refers to expected activities and states information log in a process instance.
(5) Expected output: SOAP message referring to output from process instance.

In the first step, we developed 21 test cases for Consumer_1 only. These cases were designed to cover four categories of rules with different scenarios: e.g., a test case with three constraints for validating the security context of activities. Then, we compared process execution and coordination log following the execution with the expected process activity log to verify whether the validations have occurred. In another similar test case, we manually changed the security context information for a constraint rule. We traced process execution to verify whether the defined fault rules are weaved, and the final remedy is applied in the process execution.

In the second step, we developed 10 test cases that involved two consumers (Consumer_1 and _2). Both had different defined inputs, policies, expected process activity logs and expected outputs. For these test cases, we made two consumers continually and simultaneously send a number of process requests to verify if the policies of each consumer were enforced and whether there was interference between each other. We also forced the slowdown of the policy weaver on one of the consumers and on one process request instance of one consumer to ensure messages received by BPEL processes do not follow a particular sequence. With the successful test cases, we can demonstrate that our approach provides an effective coordination solution for governance in a distributed and multi-tenant environment.

The execution aspect of our approach is inherently time consuming, but a performance overhead is also expected on coordination conversations. Once the activities of a process instance are in a governance state, the process would be blocked and wait for a provider action or policy decision from the process consumer. Two governance states ($s_{man\_val_{pre/post}}$) must be passed for all activities in the FSM of $COOR_p$ to reach the $s_{completed}$ state. These two governance states are considered to be the coordination overhead in violation-free situations. We used a local machine for an in-lab experiment. The setup used a 3.0 GHz single core process with 1GB ram WinXP VMware VM (ActivateBPEL server, JAX-WS XML web services). The coordination overhead of the activity was around 245 ms with the cache disabled and around 127 ms with the cache enabled in a violation-free situation. The actual overhead also depends on the number of policies defined by consumers, as more policies result in a greater overhead. It is less than 2 ms for a new related policy in this case (Figure 8.1). For a new policy unrelated to the governance state, the overhead is even less (Figure 8.2). In a violation situation, the coordination overhead mean value for instance adaptation is around 598ms with the cache disabled (Figure 8.3). However, this is not expected to happen on a regular basis and the overhead is then normally considered to be the necessary price to pay to fix the problem in such cases [19]. The overall overhead can increase when we apply it in real-world networks with consideration of the network latency. However, we still consider the performance overhead as quite small, as long running business activities take a few hours or even a few days for execution in a process with LRT.

## 8. Conclusion

Business processes are assets of enterprises. A shareable process would increase the reuse potential for providers and would increase profits from external consumers through customised business processes. Following the cloud computing principles, these business processes could be offered as-a-service. Consumers benefit from a customisation approach for these shared processes if they are offered in the form of a service that they can manage and control themselves through policy-based process customisation.

Consumer requirements in the form of business policies affecting the business processes can be expressed in a formal policy language, which acts as customization metadata of business processes. We have provided a policy model to formalize four categories of rules of policies, which we identified from different aspects related to business process execution (flexibility, constraint, fault and utility rules).

On-the-fly process customization and adaptation can be achieved for process consumers through process runtime governance based on process elements (business activity). A coordination framework and protocol can be used for activities within processes or subprocesses from different providers to work together on process execution for business transitions within a multi-tenancy environment. We have provided a coordination framework and protocol correlated with a policy model.

Two wider concerns can be identified behind the two core solution components we identified. The first contribution, the policy model, is a vehicle that allows business domain-specific concerns to be captured as policies and a service process implementation to be instrumented and governed through the policies beyond the usual more technical quality aspects. Consequently, more effort shall be put into developing domain-specific policy templates. The second contribution is the remote coordination protocol, which causes challenges in terms of interoperability between consumer and provider, but also between service and process providers, in particular if a migration from one provider to another is considered by the consumer. Future work will also focus on high-level policy modelling and enhancements of the coordination protocol for deeper, but also interoperable process customization in multi-tenancy environments.

## References

[1] Yi Wei and M. Brian Blake. Service-oriented computing and cloud computing: Challenges and opportunities. *IEEE Internet Computing*, 14(6):72–75, 2010.

[2] Alessio Gambi and Cesare Pautasso. Restful business process management in the cloud. In *Principles of Engineering Service-Oriented Systems (PESOS), 2013 ICSE Workshop on*, pages 1–10. IEEE, 2013.

[3] Frederick Chong and Gianpaolo Carraro. Architecture strategies for catching the long tail, 2006. `http://msdn.microsoft.com/en-us/library/aa479069.aspx`.

[4] Stefan Walraven, Dimitri Van Landuyt, Eddy Truyen, Koen Handekyn, and Wouter Joosen. Efficient customization of multi-tenant software-as-a-service applications with service lines. *Journal of Systems and Software*, 91:48–62, 2014.

[5] George Reese. *Cloud Application Architectures - Building Applications and Infrastructure in the Cloud.* O' Reilly, 2009. pages 3.

[6] Milinda Pathirage, Srinath Perera, Indika Kumara, and Sanjiva Weerawarana. A multi-tenant architecture for business process executions. In *IEEE International Conference on Web service*, 2011.

[7] W3c web services policy 1.2 - framework (ws-policy). `http://www.w3.org/Submission/WS-Policy`.

[8] MingXue Wang, Kosala Yapa Bandara, and Claus Pahl. Process as a service - distributed multi-tenant policy-based process runtime governance. In *IEEE International Conference on Services Computing*, pages 578–585, 2010.

[9] MingXue Wang and Claus Pahl. User-customisable policy monitoring for multi-tenant cloud architectures. In *European Conference on Service-Oriented and Cloud Computing*, 2012.

[10] MingXue Wang, Kosala Yapa Bandara, and Claus Pahl. Integrated constraint violation handling for dynamic service composition. In *IEEE International Conference on Services Computing*, 2009.

[11] Sanjay Dalal, Sazi Temel, Mark Little, Mark Potts, and Jim Webber. Coordinating business transactions on the web. *IEEE Internet Computing*, 7(1), 2003.

[12] Michael von Riegen, Martin Husemann, Stefan Fink, and Norbert Ritter. Rule-based coordination of distributed web service transactions. *IEEE Transactions on Service Computing*, 3(1):60–70, 2010.

[13] Frank Leymann and Stefan Pottinger. Rethinking the coordination models of ws-coordination and ws-cf. In *IEEE European Conference on Web Services*, 2005.

[14] Lakshmi Ramachandran, Nanjangud C. Narendra, and Karthikeyan Ponnalagu. Dynamic provisioning in multi-tenant service clouds. *Service Oriented Computing and Applications*, 6(4):283–302, 2012.

[15] Oasis extensible access control markup language (xacml) 3.0, 2010. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.html`.

[16] Barbara von Halle. *Business Rules Applied - Business Better Systems Using the Business Rules Approach.* John Wiley and Sons, Inc., New York, 2001. pages 33-35, 15.

[17] Florian Rosenberg, Christoph Nagl, and Schahram Dustdar. Applying distributed business rules - the vidre approach. *IEEE International Conference on Services Computing*, 2006.

[18] Abdelkarim Erradi. *Policy-Driven Framework for Manageable and Adaptive Service-Oriented Processes.* PhD thesis, 2008. Computer Science and Engineering, The University of New South Wales.

[19] Luciano Baresi and Sam Guinea. Self-supervising bpel processes. *IEEE Transactions on Software Engineering*, 37(2):247 – 263, 2011.

[20] Soumaya Marzouk and Mohamed Jmaiel. A policy-based approach for strong mobility of composed web services. *Service Oriented Computing and Applications*, 7(4):293–315, 2013.

[21] Yunzhou Wu and Prashant Doshi. Making bpel flexible adapting in the context of coordination constraints using ws-bpel. In *IEEE International Conference on Services Computing*, 2008.

[22] Kareliotis Christos, Costas Vassilakis, Efstathios Rouvas, and Panayiotis Georgiadis. Exception resolution for bpel processes: a middleware-based framework and performance evaluation. In *International Conference on Information Integration and Web-based Applications and Services*, 2008.

[23] Danilo Ardagna, Marco Comuzzi, Enrico Mussi, Barbara Pernici, and Pierluigi Plebani. Paws: A framework for executing adaptive web-service processes. *IEEE Software*, 24(6):39–46, 2007.

[24] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33(6):369 – 384, 2007.

[25] Adina Mosincat and Walter Binder. Transparent runtime adaptability for bpel processes. In *International Conference on Service-Oriented Computing*, 2008.

[26] Gerhard Friedrich, Mariagrazia Fugini, Enrico Mussi, Barbara Pernici, and Gaston Tagni. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, 36(2):198 – 215, 2010.

[27] Sattanathan Subramanian, Philippe Thiran, Nanjangud C. Narendra, Ghita Kouadri Mostefaoui, and Zakaria Maamar. On the enhancement of bpel engines for self-healing composite web services. In *International Symposium on Applications and the Internet*, pages 33–39, 2008.

[28] Abdelkarim Erradi, Piyush Maheshwari, and Vladimir Tosic. Policy-driven middleware for self-adaptation of web services compositions. In *ACM/IFIP/USENIX International Middleware Conference*, 2006.

[29] Oasis web services coordination (ws-coordination) - accessed online, 2014. `http://docs.oasis-open.org/ws-tx/wscoor/2006/06`.

[30] Oasis web services atomic transaction (ws-atomictransaction) - accessed online, 2014. `http://docs.oasis-open.org/ws-tx/wsat/2006/06`.

[31] Oasis web services business activity (ws-businessactivity) - accessed online, 2014. `http://docs.oasis-open.org/ws-tx/wsba/2006/06`.

[32] Stefan Pottinger, Ralph Mietzner, and Frank Leymann. Coordinate bpel scopes and processes by extending the ws-business activity framework. In *International Conference on Cooperative Information Systems*, 2007.

[33] Afkham Azeez, Srinath Perera, Dimuthu Gamage, Ruwan Linton, Prabath Siriwardana, Dimuthu Leelaratne, Sanjiva Weerawarana, and Paul Fremantle. Multi-tenant soa middleware for cloud computing. In *IEEE International Conference on Cloud Computing*, 2010.

[34] Christoph Fehling, Frank Leymann, and Ralph Mietzner. A framework for optimized distribution of tenants in cloud applications. In *IEEE International Conference on Cloud Computing*, 2010.

[35] Tobias Unger, Ralph Mietzner, and Frank Leymann. Customer-defined service level agreements for composite applications. *Enterprise Information Systems*, 3(3):369–391, 2009.

[36] Anne H. Anderson. An introduction to the web services policy language (wspl). In *IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004.

[37] Hamdi Yahyaoui, Azzam Mourad, Mohamed Almulla, Lina Yao, and Quan Z. Sheng. A synergy between context-aware policies and aop to achieve highly adaptable web services. *Service Oriented Computing and Applications*, 6(4):379–392, 2014.

[38] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[39] Jeffrey O. Kephart and William E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004.

[40] Liangzhao Zeng, Hui Lei, Jun-jang Jeng, Jen-Yao Chung, and Boualem Benatallah. Policy-driven exception-management for composite web services. In *IEEE International Conference on E-Commerce Technology*, 2005.

[41] MingXue Wang. *A policy based governance framework for cloud service process architectures (Dublin City University)*. Phd thesis, 2012.

[42] Emil Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852 – 869, 1999.

[43] Kosala Yapa Bandara, MingXue Wang, and Claus Pahl. An extended ontology-based context model and manipulation calculus for dynamic web service processes. *Service Oriented Computing and Applications*, 2013.

[44] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360, 1994.