Explicit High-Level Rules for the Customization of Web Services Management

María Agustina Cibrán
System and Software Engineering Lab
Vrije Universiteit Brussel (Belgium)
mcibran@vub.ac.be

Maja D'Hondt
Laboratoire d'Informatique Fondamentale
de Lille — INRIA Jacquard (France)
dhondt@lifl.fr

Abstract: Current Web Services integration approaches fail at supporting a highly flexible service selection and management. To overcome these problems, the Web Services Management Layer was proposed in previous work. However, this layer presents some limitations due to the implicit representation of service criteria that guide its customization. In this paper we propose externalizing these criteria as high-level business rules. Moreover, the specification of new unanticipated business rules is also supported. We do this by employing a high-level business rule language proposed in previous work. We show how this rule language can be used to customize the WSML, enhancing flexibility. The originality of our approach is the application of a general-purpose business rule language to the domain of Web Services management.

1 Introduction

In service-oriented computing, applications are often created by integrating third-party Web Services. However, in order for client applications to achieve a high flexibility in this integration, advanced support for selection and client-side service management is fundamental. This support is rarely provided in standard state-of-the-art service integration approaches and tools [BEA, Mic]. Moreover, we observe that the selection, integration and management of Web Services are driven by criteria based on non-functional service properties. For instance, the service integration can be guided by rules that prefer fast and reliable services or give priority to services with the least number of failures; other rules govern the way management should be carried out, e.g. advising the activation of a caching mechanism for services that are too slow. Many of these business rules depend on dynamic service properties that are only known at run time. The explicit specification of these business rules is crucial to achieve a highly flexible integration of services that best fit the client application's needs.

As a first step towards achieving these goals, the Web Service Management Layer (WSML) was proposed in previous work [CVV$^+$ar, VCJ04, VCV$^+$04]. The WSML is an AOP-based management framework that allows for the dynamic selection and integration of services into client-applications and the client-side service management. The WSML offers a reusable library of selection, management and monitoring concerns implemented as aspects that can be customized for different applications on which the layer is deployed. Although the WSML enhances the overall service management, we observe some limita-

tions with it: first of all, many decisions about how the framework has to be configured and customized are taken either at deployment time, i.e. at the moment the WSML framework is deployed on a concrete client application, or manually at run-time, i.e. through an interface that requires human interaction. Examples of these decisions are: choosing which aspects need to be plugged in, which parameters need to be used for their configuration and which services are to be composed. Moreover, even though some anticipated selection and management decisions are encapsulated in business rules, they are hardcoded in the implementation of the framework, with negative effects on maintainability. These limitations impede achieving highly configurable and flexible adaptation and customization of this framework. As business rules are tangled and scattered it is difficult to localize and change them. Moreover, new unanticipated rules cannot be easily added since it implies manually modifying or adding code to the framework at many places. Therefore, extra support is needed to — automatically and non-invasively — realize dynamic business rules that can vary at run time and that are unanticipated at deployment time.

In previous work we have conducted research on decoupling business rules and their connections to a core object-oriented application and propose a high-level domain model that allows for their specification in terms of domain concepts [CDJar, CDar]. The novelty of this approach is the use of AOP for *mapping* the domain model to implementation. Following a *Model-Driven Engineering* (MDE) approach, high-level rules and connections are automatically translated to object-oriented and aspect-oriented programs, respectively [CDar]. In this paper we show how this high-level business rule language can be used to express and enforce the dynamic business rules that guide the customization of the WSML. We validate this approach with two scenarios: i) we *refactor* existing rules from the core WSML implementation, and ii) we easily *evolve* the existing application by adding new rules. This approach has three clear advantages: first, we can — at run time — vary the conditions that guide the different selection, integration and management tasks offered by the WSML; second, we can non-invasively extend the core functionality of the WSML, abstracting from its technical complexity; third, it is possible to add rules that refer to run-time service properties that were not foreseen in the existing WSML implementation.

This paper is organized as follows: Section 2 presents the WSML, classifies the possible rules in this framework and describes which ones and how they are currently implemented. Section 3 gives an overview of our high-level business rule and connection languages. Section 4 shows how we can use these languages to express rules that customize the current WSML. Section 5 discusses the advantages and disadvantages of our approach. Finally, Sections 6 and 7 present related work and conclusions respectively.

2 WSML

2.1 Overview

The Web Services Management Layer (WSML) [CVV$^+$ar, VCV$^+$04, VCJ04] is an intermediate layer between the client applications and the world of web services. It allows

the dynamic selection and integration of services and client-side service management. It also supports the definition of criteria based on non-functional properties of services that govern their selection, integration and management.

The WSML is an AOP framework that facilitates the development of service-oriented applications. It enables client applications to invoke service functionality in a generic way by means of *service type* invocations. *Service Types* are unified interfaces exposing service functionality and making abstraction of how concrete services provide it. Service type generic requests are transparently redirected to concrete services by means of redirection aspects. The WSML also offers a library of generic and reusable management *templates* that can be personalized according to the requirements of the client applications. Examples of management templates are caching, pre and post billing and fallback strategies. Moreover, in order to monitor dynamic properties of services — for instance, average speed, number of invocations and number of failures — the WSML provides monitoring templates. The monitored properties are used to make decisions about the selection of services. These decisions are encoded in selection policy templates.

All these management, monitoring and selection templates are implemented as dynamic and reusable aspects in JAsCo [SVJ03], a dynamic AOP language that realizes the runtime addition and removal of aspects. JAsCo aspects are generic and can be deployed in different contexts using JAsCo connectors. For instance, in the WSML the management and monitoring aspects can be deployed on different invokable services, either service types, concrete web services or service compositions, obtaining this way different setups: global caching is achieved by deploying the *CachingAspect* on service type invocations, whereas local caching is obtained by deploying the aspects on concrete web service invocations (only caching results for that specific service). Analogously, depending on how the monitoring aspect is deployed, global or local monitoring can be realized. Selection aspects can also be deployed either on web service invocations or service composition invocations, obtaining different selection strategies acting at different levels. More details on the WSML and its implementation can be found in [CVV$^+$ar, VCV$^+$04, VCJ04].

2.2 Challenges: Distilling Potential Business Rules in WSML

We observe that the management, selection and integration of web services is rule intensive. Examples of this are: a caching strategy that needs to be plugged in only if the response time of the services drops under a certain threshold; a selection policy that selects services with minimum number of failures needs to be considered only if reliability is a crucial requirement; on the contrary, a different selection strategy can be considered when reliability is no longer a crucial requirement. We distill many actions of the core functionality of the WSML that are guided by business rules, including:

Action 1: enabling/disabling of monitoring, selection and management aspects

Action 2: execution of monitoring, selection and management aspects

Action 3: redirection of functional requests (to a single service or service composition)

Action 4: creation of service compositions

Action 5: classification of service types, services or compositions into categories

Action 6: configuration of monitoring, selection and management aspects

These actions are typically guided by conditions defined upon certain information stored in the WSML. Different sources of information are distilled:

Condition a: on values stored in WSML objects

Condition b: on values captured by and stored in WSML aspects

Condition c: on values that can be requested from the concrete web services

Condition d: check whether a monitoring, selection or management aspect is enabled

All the combinations of these conditions and actions are possible. In the following section we describe which concrete combinations are currently considered in the WSML and how they are supported.

2.3 Current Situation: Implicit Business Rules

Even though the WSML is a first step towards a more dynamic and configurable web service integration, we observe that it falls short when it comes to coping with changes in the business rules that guide how this framework needs to be configured. Currently, these decisions are either taken manually or are driven by business rules that are implicitly represented, resulting in tangled code in the implementation of the framework. This introduces all the problems discussed in section 1.

We identify concrete business rules example groups named BR1 to BR6 that are implicitly represented in the current implementation of the WSML:

BR1: the enabling of monitoring, selection and management aspects (action 1) is done manually by the WSML administrator either by means of an XML configuration language (interpreted at deployment time) or via an administration console (at run time). Thus, the business rules that guide these actions are implicitly represented in the WSML. We can imagine the situation in which enabling and disabling these aspects depend on unanticipated run-time conditions. In this case, the automatic enforcement of these conditions is pursued.

BR2: the execution of monitoring, selection and management aspects (action 2) is currently guided by conditions that are tangled in the implementation of the aspects themselves. These conditions check that a certain monitored property (values stored in WSML (condition a) and calculated by monitoring aspects (condition b)) falls into a certain range.

A concrete example in this category is the following: when a service type is invoked, the *ConditionalCaching* aspect first checks whether the average speed (i.e. throughput) of

that service type (meaning the average speed among all the invoked services that realise that service type) is smaller than 1000. If this is the case, the aspect starts caching the results of invoking that service type, which are then retrieved from the cache on every new invocation on that same service type. As soon as the average speed becomes greater than the specified threshold, the caching functionality is stopped and the results are retrieved from concrete web services again.

BR3: the redirection either to single services or to service compositions (action 3) is guided by conditions that are hardcoded in the redirection aspects themselves. As in case 2, they are conditions on monitored properties (cases a and b).

BR4: business rules that guide the creation of service compositions (action 4) are not supported.

BR5: implicit categories are considered that classify the services into *slow* or *fast* (action 5). Code to support this classification is included in the implementation of the monitoring aspect, resulting in tangled code.

BR6: the configuration of aspects (action 6) can be done through XML at deployment time or via the administration console at run time. However, in both cases human interaction is needed. We can imagine automating these decisions by considering business rules in charge of modifying certain parameters at run-time, this way influencing how monitoring, selection and management are carried out.

Table 1: Implicit business rules in the WSML

conditions/actions	(1)	(2)	(3)	(4)	(5)	(6)
(a)	-	X	X	-	-	-
(b)	-	X	X	-	X	-
(c)	-	-	-	-	-	-
(d)	-	-	-	-	-	-
manually	X	-	-	-	-	X

Table 1 summarizes the situation with current implicit rules in the WSML. The rows and columns refer respectively to the conditions and actions (distilled in section 2.2). We specify an "*X*" when a particular condition-action combination is supported in the current WSML implementation, either manually or hard-coded, and a "-" if it is not supported.

3 High-Level Business Rules

In [CDJar, CDar] we propose a *high-level domain model* consisting of: domain entities, business rules about domain entities, and connections of business rules to the core application in terms of domain entities. The domain entities represent the domain vocabulary of interest and are based on the typical modeling elements found in all data modeling approaches: classes, attributes, methods, associations and generalizations. The link from

domain concepts to implementation is encapsulated in a *mapping*. Differently to other high-level languages [JRu, Qui, Vis, Hal], this mapping can become very sophisticated in the case of domain concepts that are unanticipated in the existing application. Even though some anticipated mappings need to refer to implementation entities, others can be defined completely at the domain level. More details on these mappings can be found in [CDJar].

High-level business rules express relations between terms of the domain which are captured as domain entities. Thus, rules are independent of implementation details. A high-level business rule language is proposed in [CDJar], in which a rule is defined as an *IF ⟨condition⟩ THEN ⟨action⟩* statement where ⟨condition⟩ and ⟨action⟩ only involve domain entities. The condition denotes a boolean expression that can involve the invocation of domain methods, the retrieval of domain attributes and the reference to business objects (instances of domain classes) specified in the rule. Also these elements can be combined in logical or comparison expressions as well as in nested combinations. The action part denotes the invocation of domain methods that can involve accessors, reference to business objects and domain method invocations. Rule templates can be defined by means of the *PROPS* clause exposing the information that needs to be provided at instantiation time.

The *high-level business rule connections* specify the details of the rules' integration with the core application and typically denote an *event* at which the rule needs to be applied and the specification of the required information. A rule connection is specified as follows: *CONNECT ⟨brname⟩ BEFORE/AFTER/DURING ⟨eventname⟩*. If *brname* corresponds to a rule template, then the *PROPS ⟨value1⟩, ..., ⟨valueN⟩* clause is used to define the concrete values expected by the rule.

This high-level connection language is built on top of the connection aspects identified in previous work [CDJ03, CDS+03, CSD+04, DJ04]. In that work we observed that, at the implementation level, rule connections crosscut the core application and therefore we proposed AOP for encapsulating it. That work has shown that the aspects that encapsulate the rule connections consist of the same elements, that vary with certain situations: rule application time, contextual information and activation time. Thus, as the same issues recur in every connection aspect, we propose abstracting them in high-level features of a high-level rule connection language. This language allows expressing rule connections as separate and explicit entities at the domain level. Separating rules from their connections allows reusing both parts independently. Moreover, we also provide a set of variations for each different connection issue involved in the definition of the rule connections.

In order to make these rules executable and integrate them with the existing application according to the connections, we follow a *Model-Driven Engineering* (MDE) approach: the rules and connections are automatically translated to object-oriented and aspect-oriented programs, respectively. The transformations use the mapping to implementation from the domain entities. Our approach maintains separation of concerns from the domain level to the implementation level, thus facilitating traceability of the business rules and their connections. Moreover, the automatically generated code pertaining to rules and connections remains separated from the existing application code and therefore does not interfere with the development and maintenance of the application. More details on these transformations, which are outside the scope of this paper, can be found in [CDJar, CDar].

4 Explicit High-Level Rules for WSML Customization

This section presents our approach which consists of expressing WSML customization rules in the high-level business rule language introduced above. In order to validate our approach, we present two usage scenarios:

(1) a refactoring scenario, showing how implicit and tangled business rules can be made explicit and high-level. This scenario shows that: i) it is easier to reason about those rules when they are expressed in terms of the domain; ii) it becomes possible to reuse them by simply connecting them at different events; iii) the WSML code becomes more understandable and maintainable.

(2) an evolution scenario, showing that it is possible to extend the current WSML functionality in order to cope with new unanticipated business rules, This way, evolution is supported. The most important characteristic of this scenario is that those unanticipated rules can be enforced non-invasively in the WSML, without having to manually change or insert code.

We show how our approach supports these two scenarios. In section 4.2 we specify high-level domain entities that extract the domain knowledge required in (implicit) business rules, without having to change the existing WSML implementation. We then show how it becomes possible to express the implicit rules and their connections in terms of these high-level entities. In section 4.3 we show that the definition of new high-level domain entities, business rules and connections that were not anticipated, can also be expressed, without having to change the existing WSML implementation. Whereas the former requires mappings from domain elements to AOP elements, the latter requires the mapping to occur through AOP.

4.1 A Domain Model for the WSML and its Mapping to Implementation

The first step in extracting the business rules and making them explicit consists of defining the domain entities representing the required domain vocabulary. On the top part of Figure 1, the subset of the domain entities that are required for the examples presented in this paper is shown. Note that this is not a real-world domain but a web services one implying that the domain expert in charge of writing rules about this domain needs to be knowledgeable in web services terminology (e.g. web services, number of failures, service selection, monitoring and management).

The next step consists of defining a mapping to implementation per domain entity in the domain model. A full categorization of the possible mappings is presented in [CDJar]. In this section we show a few mapping examples that link the WSML domain to the existing WSML implementation. Since the WSML is an AOP framework, a domain entity can be realized by an entity implemented either in OOP or AOP. The lower part of Figure 1 depicts one-to-one mappings that link domain classes to existing classes and aspects in the framework. Also, some domain attributes and methods (e.g. the domain method getProperty() in domain class MonitoringConcern) are aliases for implementation attributes

and methods, and thus one-to-one mappings exist between them (omitted in Figure 1 for space reasons). Other domain entities however do not directly correspond to one existing implementation entity, implying the need for more sophisticated mappings (also supported by our approach). To illustrate these complex mappings, we present two examples:

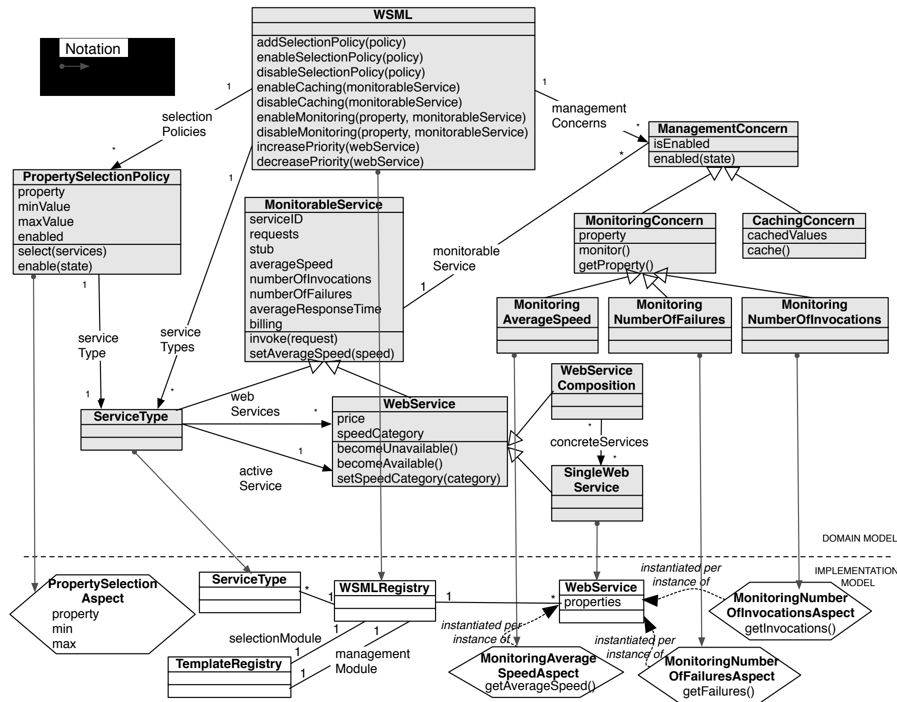

Figure 1: Domain entities in the WSML and their mappings to its implementation

Derived mappings. The domain method *enableCaching* defined in the *WSML* domain class, represents the action of enabling a caching aspect (as management concerns are implemented as aspects). However, at implementation level, management aspects are enabled and disabled via methods defined in the class *TemplateRegistry*. Thus, the domain method *enableCaching* is an alias for that enabling method in *TemplateRegistry*, as illustrated in Figure 2. As an association exists between the *WSMLRegistry* and the *TemplateRegistry* classes (as shown in Figure 1), the mapping for *enableCaching* is defined by means of navigating that association. The boxes in Figure 2 contain information kept in the mapping which specifies which parameters need to be used to obtain the correct mapping.

Composed mappings. The *WebService* domain class maps to the class with the same name existing in the implementation of the WSML, as depicted in Figure 3. The domain entities that represent static information about a web service (e.g. serviceID and price) map to corresponding attributes and methods on the WebService class. However, the domain attributes representing dynamic service properties (e.g. averageSpeed, numberOfInvoca-

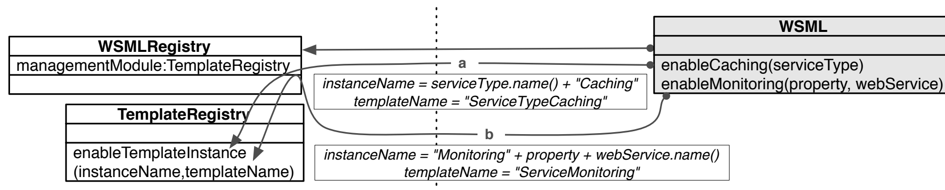

Figure 2: Fixed values in anticipated mapping from domain method to OO method

tions, numberOfFailures and averageResponseTime) do not have a corresponding implementation entity in the WebService class to which to map. This is because these dynamic properties are calculated by and stored in *monitoring aspects* predefined in the WSML. A different aspect exists per dynamic property. At the domain level, monitoring aspects are represented as monitoring concern domain classes, as shown by mappings **f**, **g** and **h**. Thus, the domain attributes representing dynamic service properties map to the *property* domain attributes in those monitoring concerns, as illustrated in mapping **e**. As a consequence, the mapping for the WebService domain class is a composition between the mapping to an implementation class (mapping **a**) and to several domain classes (mappings **b**, **c** and **d**).

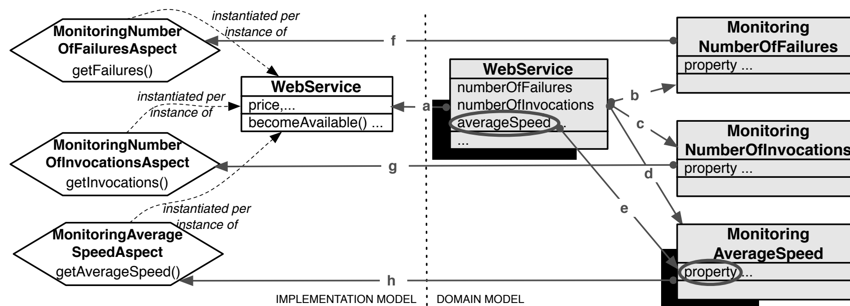

Figure 3: Example of composed mapping in the WSML

4.2 Refactoring Implicit Business Rules from the WSML

In this section we show how high-level business rules expressed in our high-level language can enhance flexibility and configurability of the WSML framework. We do this by showing how we can express some of the example rules presented in Section 2.3 in terms of the high-level WSML domain entities illustrated in Figure 1. In particular, we present concrete examples of the BR2, BR5 and BR6 categories.

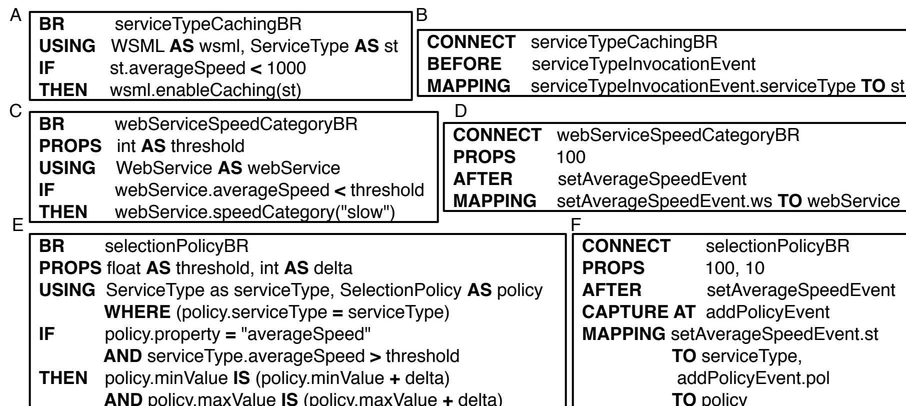

Figure 4: High-level rules for the WSML

BR2. Consider the following rule:

serviceTypeCachingBR: "**If** the average speed of a service type is smaller than 1000 **then** enable the caching of results for invocations on that service type"

In the WSML domain model, the average speed of a service type is represented by the *averageSpeed* domain attribute defined in *MonitorableService*; the action of enabling the caching concern is represented by the domain method *enableCaching(MonitorableService)* defined in the *WSML* domain class. Using these domain entities, we can express this rule at the domain level as expressed in part **A** of Figure 4.

The decision of whether the caching has to be enabled is taken before a certain functionality is requested on a service type, i.e. before the invocation of *invoke(request)* defined in *ServiceType*. The domain model allows defining events of interest at which rules might be applied. These definitions are completely specified at the domain level, in terms of domain entities. In this example, we are interested in capturing the moment the domain method *invoke(request)* is invoked on a service type. We name this event *serviceTypeInvocationEvent*. Events also expose the contextual information available at that point, i.e. receiver, parameters and return value. Imagine *serviceTypeInvocationEvent* exposes the target object under the name of *serviceType*. Once this event is defined, we can connect the *serviceTypeCachingBR* rule with the core application as expressed in part **B** of Figure 4. The service type available in the connection context is linked to the service type expected by the rule in the MAPPING clause.

BR5. Consider the following rule:

webServiceSpeedCategoryBR: "**If** the average speed of web service is less than 100 **then** the web service is *slow*"

Using the *averageSpeed* domain attribute defined in the *WebService* domain class as well

as its *speedCategoty(category)* domain method, this rule can be defined at the high-level as expressed in part **C** of Figure 4. This is an example of a rule template, as the concrete value for *threshold* is specified at rule instantiation time. This rule can be triggered at the moment the domain method *setAverageSpeed(speed)* is invoked on a web service, captured by an event that we refer to as *setAverageSpeedEventWS*. This event exposes the target web service as *ws*. Thus, the high-level connection of the *webServiceSpeedCategoryBR* on the *setAverageSpeedEventWS* event is defined in part **D** of Figure 4. Analogously to the previous two definitions, the rule "**if** average speed of web service is greater than 100 **then** the web service is *fast*" can also be expressed in terms of the domain.

BR6. Consider the following rule:

selectionPolicyBR: "**If** the average speed of a service type is greater than 100 **then** the minimum and maximum values considered by the selection policy for that service type that selects services based on their average speed is increased by 10"

A high-level specification of this rule is shown in part **E** of Figure 4. This rule can be triggered at the moment the domain method *setAverageSpeed(speed)* in invoked on a service type, captured by an event that we refer to as *setAverageSpeedEventST*. This event exposes the target service type as *st*. Additionally, the relevant selection policy needs to be captured at another event capturing the moment the *addSelectionPolicy(policy)* is invoked on the WSML. We name this event *addPolicyEvent(pol)*. The parameter is exposed by this event as *pol*. The high-level connection of the *selectionPolicyBR* on these events is shown in part **F** of Figure 4. Note that only the invocations of *setAverageSpeed* that occur after the addition of the corresponding selection policy will trigger this rule, as it is only then that all the information required by the rule is available at connection time.

4.3 Evolving the WSML by Adding Unanticipated Business Rules

Externalizing the domain vocabulary in the form of domain entities allows us to express and realize not only implicit rules that were foreseen in the WSML but also new business rules that appear as a result of domain evolution and that were not anticipated in the original WSML implementation. Many cases can occur:

1) Using existing domain entities in the domain model: **a)** new rules - same connections; **b)** new rules - new connections; **c)** same rules - new connections.

2) Extending the domain model with new domain entities that either *anticipated* or *unanticipated* in the WSML implementation. For each of these variation, we can have the following cases: **a)** new rules - same connections; **b)** new rules - new connections; **c)** same rules - new connections.

Case **1** shows that our approach enhances variability whereas case **2** shows that extensibility is achieved. Representative examples of some categories of these two cases are presented in Section 4.3.1 and 4.3.2.

4.3.1 Case 1: New rules in terms of existing domain entities

Using the domain entities already existing in our domain model of Section 4.1, we can specify new business rules that were not anticipated in the current WSML implementation, for example:

Rule A: ***if*** *service.numberOfFailures < X* ***then*** *WSML.increasePriority(service)*
Rule B: ***if*** *service.billing > X* ***then*** *WSML.reducePriority(service)*
Rule C: ***if*** *service.numberOfInvocations > X* ***then*** *WSML.enableCaching(service)*

For some of these rules we can reuse rule connections defined in Section 4.2 (for instance, *Rule C* can reuse the connection defined for *serviceTypeCachingBR*) or new connections can be defined for them. The domain entities involved in these rules require mappings that are defined similarly to the ones presented in 4.1. For instance, analogously to the average speed service property, the properties involved in these rules are also based on monitored and management results, and thus the corresponding domain entities are anticipated in the WSML implementation (section 4.2). The actions also correspond to existing operations in the WSML and thus there is an anticipated mapping for them as well. Therefore, it is clear that the high-level definition of these new business rules suffices to change the behaviour of the WSML.

4.3.2 Case 2: New rules in terms of new domain knowledge

The most interesting situation in this case is when the new domain knowledge is unanticipated in the WSML implementation, as they require more sophisticated mappings specifying how the new vocabulary has to be realized at the implementation level.

New rules in terms of unanticipated derived information. Consider the following rule example: ***if*** *service.numberOfSuccessfulInvocations > 10* ***then*** *increasePriority(service)*

This rule refers to a new domain attribute that specifies the *number of successful invocations* of a web service which is unanticipated in the WSML. However, this information can be derived from two existing domain attributes: the *numberOfInvocations* and *numberOfFailures*. Thus, this new domain attribute can be mapped to an expression that calculates the substraction between the two existing domain attributes, as shown in part **A** of Figure 5. The translation of this high-level mapping to a concrete implementation (in terms of values stored in aspects) is automatic and transparent for the domain expert.

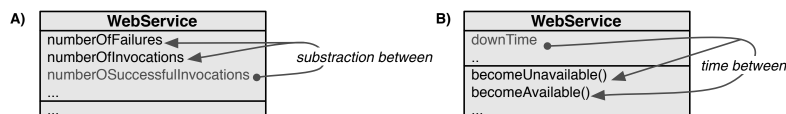

Figure 5: Derived mapping for unanticipated domain attributes

New rules encoding unanticipated categories. In Section 4.2 a rule that classifies services into the categories *fast* and *slow* according to their average speed is identified. These

categories were anticipated and hardcoded in the WSML. We can imagine other unanticipated categories in which it might be interesting to classify services, according to the values of their properties. For instance, we can classify services into *expensive* and *not expensive* depending on their price, or into *frequently invoked* or *not frequently invoked* according to the number of times they were actually invoked. Thus, new rules can be written which encode these categorizations:

if *service.price > X* ***then*** *service IS expensive*
if *service.price < X* ***then*** *service IS NOT expensive*
if *service.numberOfInvocations > X* ***then*** *service IS frequentlyInvoked*
if *service.numberOfInvocations < X* ***then*** *service IS NOT frequentlyInvoked*

In the domain model, the new categories are represented as new domain attributes in the domain classes that are being classified. In this case, new domain attributes *expensive* and *frequentlyInvoked* are required simply added to the WebService domain class. Less straightforward is the definition of their mappings, as they are unanticipated in the WSML implementation. In this case, the rules themselves control *how* and *when* to set the value of these domain attributes. Our approach realizes this classification using AOP [CDJar], again transparently for the domain expert.

New rules in terms of unanticipated dynamic properties. Consider the example:

if *service.downtime > X* ***then*** *WSML.enableCaching(service)*

This rule uses a *downtime* domain attribute which represents the amount of time a service is unavailable. This is a dynamic service property since it requires monitoring the time between the moment a service becomes unavailable and the moment it becomes available again. However, the current monitoring aspects in the WSML are not able to monitor this property as it requires different monitoring points. Although the WSML can be extended by introducing a new monitoring aspect, this aspect has to be written manually. Our solution performs the automatic generation of the aspect that realizes this unanticipated domain attribute. The input for this generation is a high-level mapping specification stating how this new dynamic property has to be monitored (as illustrated in case **B** of Figure 5). Again, AOP is transparently used to realize this high-level specification. Similarly to the *downtime*, we can define an *uptime* domain attribute representing the time between the moment the service becomes available until the moment it becomes unanvailable. This new domain vocabulary can be reused in other new rules, for instance: ***if*** *service.downtime > service.uptime* ***then*** *service is unreliable*.

5 Discussion

The above scenarios illustrate the following advantages:

Enhanced adaptability and variability. The management and selection aspects are now guided by externalised business rules instead of hard-coded policies. Moreover, generic rule templates can be defined for those implicit rules, allowing the deployment of the same logic with different parameters. As a result, we can vary the business rules that guide the

way management and selection are carried out, without having to adapt manually existing business rules nor the core WSML implementation. However, varying the rules might require defining new rule connections for which new events might need to be defined. Nevertheless, this can be easily done by using the proposed high-level connection language. As a consequence, different versions of the WSML implementation can be created by plugging in different sets of business rules.

Improved understandability. In addition to the above advantage, it is now possible for a domain expert to add new unanticipated business rules. Adding new rules is trivial if they can be specified in terms of existing domain entities and can reuse an existing connection. However, if new domain entities and/or connection are required, an initial investment is inevitable. Even though the domain expert is able to extend the domain model with high-level definitions of these elements, less straightforward from the point of view of the domain expert is providing the mapping of these new elements which might — in some cases — require detailed knowledge about the implementation. However, once these mappings are defined, the new domain concepts can be reused in as many rules as needed, making abstraction of their low-level realization.

Improved reusability. Our approach allows reusing the same business rules by connecting them at different events. Moreover, every connection can be configured in a different way (e.g. by specifying a different activation time, or a different mapping between the available/required information), changing this way the behavior of the WSML.

Run-time configuration. The variability of existing rule connections, the definition of new connections for existing rules, the addition or removal of rules and connections, can all occur at run-time thanks to the underlying dynamic AOP language.

First step towards traceability. The use of AOP in the transformations of our high-level rule and connection specifications allows for a well-modularized and localized implementation of the rules and their connections, avoiding invasive changes to the existing code. As the mapping from high-level rules and connections to implementation is made explicit, rule and connection traceability becomes possible. Analogously, traceability of domain knowledge is possible as well. However, one potential problem is *aspect interference* [EP01]. Conflicts between aspects that result from the translation of high-level rule connections can be treated at a high-level by means of the explicit specification of how to solve conflicts. We envision support for this in our high-level connection language. A more challenging interference is the one between the rule connection aspects and the aspects that exist in the core application (when this one is built using AOP, as it is the case in the WSML). More powerful feature interaction mechanisms are required in this case. Even though this is an important issue, a discussion on it is outside the scope of this paper.

6 Related work

Two lines of related research are identified: high-level business rule languages on one hand and web services management on the other hand. Our approach combines both lines of research by applying a general-purpose business rule language to the domain of web

services management in order to enhance customization. To our knowledge, this combination has not been explored so far. High-level rule languages are proposed to externalized rules in real-world domains [JRu, Qui, Vis, Hal]. However, in these approaches rules are expressed in terms of high-level domain concepts that are aliases for implementation entities and thus only simple anticipated rules can be externalized. Thus, there is no support for unanticipated rules. Other line of research proposes dedicated languages to address the explicit specification of QoS requirements for web services. For instance, the GlueQoS approach is proposed [WTM+04] which focuses on the dynamic reconciliation of QoS conflicts between interacting components. They propose a declarative language, based on the WS-Policy [wsp03] language, to specify QoS features, preferences and conflicts. These specifications are taken into account by a middleware-based mediator mechanism in charge of finding a compromise between the QoS of the communicating services. However, a fixed ontology of features with all possible interactions has to be explicitly and a-priori identified. Thus, differently to our approach, unanticipated features cannot be added at runtime. Another related approach is AO4BPEL [CM04] which supports selective web service composition by modularizing business rules as aspects. In this approach, however, the core service composition description is tangled with the definition of the business rules, as they are specified as part of the same process description. Moreover, contrary to our approach, AO4BPEL rules have to be programmed at a lower level.

7 Conclusion

In this paper we presented an approach that combines a general purpose high-level business rule language with the WSML layer in order to express and enforce the dynamic business rules that guide the customization of this layer. Existing rules can be externalized and new rules can be easily added, enhancing the adaptability of service-oriented applications. Moreover, these rules are defined in terms of the domain, abstracting technical complexity. We distill different categories of rules in the WSML, present concrete examples for them and express them in our high-level business rule language. A discussion on the advantages and limitations of our approach is included.

References

[BEA] BEA WebLogic Framework 8.1. http://www.bea.com/.

[CDJ03] M. A. Cibrán, M. D'Hondt und V. Jonckers. Aspect-Oriented Programming for Connecting Business Rules. In *Proceedings of BIS International Conference*, Colorado Springs, USA, Juni 2003.

[CDJar] M. A. Cibrán, M. D'Hondt und V. Jonckers. Mapping high-level business rules to and through aspects. *L'Objet*, 12(2-3), September 2006 (to appear).

[CDar] M. A. Cibrán und M. D'Hondt. A Slice of MDE with AOP: Transforming High-Level Business Rules to Aspects. *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Oktober 2006 (to appear).

[CDS+03] M. A. Cibrán, M. D'Hondt, D. Suvée, W. Vanderperren und V. Jonckers. JAsCo for Linking Business Rules to Object-Oriented Software. In *Proceedings of CSITeA International Conference*, Rio de Janeiro, Brazil, Juni 2003.

[CM04] Anis Charfi und Mira Mezini. Hybrid web service composition: business processes meet business rules. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, Seiten 30–38. ACM Press, November 2004.

[CSD+04] M. A. Cibrán, D. Suvée, M. D'Hondt, W. Vanderperren und V. Jonckers. Integrating Rules with Object-Oriented Software Applications using Aspect-Oriented Programming. In *Proceedings of ASSE'04, Argentine Conference on Computer Science and Operational Research*, Córdoba, Argentina, September 2004.

[CVV+ar] M. A. Cibrán, B. Verheecke, W. Vanderperren, D. Suvée und V. Jonckers. Aspect-Oriented Programming for Dynamic Web Service Selection, Configuration and Management. *World Wide Web Journal (WWWJ)*, 9, 2006 (to appear).

[DJ04] M. D'Hondt und V. Jonckers. Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge. In *Proceedings of the 3th International Conference on Aspect-Oriented Software Development*, Lancaster, UK, Marz 2004.

[EP01] J.O. Coplien E. Pulvermller, A. Speck. Proceedings of the 1st Workshop on Feature Interaction in Composed Systems. In *15th European Conference on Object-Oriented Programming (ECOOP)*, Budapest, Hungary, Juni 2001.

[Hal] HaleyRules. http://www.haley.com/products/HaleyRules.html.

[JRu] JRules. http://www.ilog.com/products/jrules/.

[Mic] Microsoft Visual Studio.NET 2003. http://msdn.microsoft.com/vstudio/previous/2003/.

[Qui] QuickRules. http://www.yasutech.com/.

[SVJ03] D. Suvée, W. Vanderperren und V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, USA, Marz 2003.

[VCJ04] B. Verheecke, M. A. Cibrán und V. Jonckers. Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection. In *European Conference on Web Services 2004 (ECOWS'04)*, Erfurt, Germany, September 2004.

[VCV+04] B. Verheecke, M. A. Cibrán, W. Vanderperren, D. Suvée und V. Jonckers. AOP for Dynamic Configuration and Management of Web services in Client-Applications. *International Journal on Web Services Research (JWSR)*, 1(3), 2004.

[Vis] Visual Rules. http://www.visual-rules.de.

[wsp03] *Web Services policy framework*, 2003. BEA and IBM and Microsoft S. AG.

[WTM+04] E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou und P. Devanbu. GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, Seiten 189–199, Washington, DC, USA, 2004. IEEE Computer Society.