Mapping high-level business rules *to* and *through* aspects

María Agustina Cibrán* — Maja D'Hondt — Viviane Jonckers***

* *Vrije Universiteit Brussel*
System and Software Engineering Lab
Pleinlaan 2, 1050 Brussels, Belgium

** *Université des Sciences et Technologies de Lille*
Laboratoire d'Informatique Fondamentale de Lille
59655 Villeneuve d'Ascq Cédex, France

mcibran@vub.ac.be, Maja.D-Hondt@lifl.fr, vejoncke@info.vub.ac.be

RÉSUMÉ.

ABSTRACT. *Many object-oriented software applications contain implicit* business rules. *Although there exist many approaches that advocate the separation of rules, the rules'* connections *still crosscut the core application functionality, which impedes reuse, either anticipated or not. Moreover, ultimately business rules are implemented in a programming language, which decreases understandability and accessibility by domain experts. We propose a high-level domain model for representing domain concepts, business rules about these concepts, and connections of business rules to the core application in terms of these concepts. The link to the implementation is invisible to domain experts and encapsulated in a* mapping. *The novelty and contribution of our approach is the use of* Aspect-Oriented Programming *(AOP) on two levels. First of all, elements from the high-level domain model are mapped* to *existing implementation entities of an application developed in Object-Oriented Programming (OOP) or AOP. Secondly, new implementation entities are generated in order to map domain model elements that do not have a direct realisation in the current implementation or that appear as a result of domain evolution. As the new implementation entities can result in crosscutting, the mapping occurs* through *AOP. We evaluate our approach in the* Web Services Management Layer *(WSML), a non-trivial application for creating applications using Web Services, by means of two scenarios: (1) extracting implicit business rules from the WSML and representing them in a high-level domain model, and (2) extending the WSML with unanticipated business rules.*

MOTS-CLÉS :

KEYWORDS: *Object-Oriented Programming, Aspect-Oriented Programming, Business Rules*

1. Introduction

Many object-oriented software applications contain implicit knowledge about the domain or business that they support. In this paper, we focus on knowledge in the form of rules or *business rules*. Examples can be found in e-commerce applications, where business rules guide customer preferences, personalized discounts, return and refund policies and so on. More sophisticated business rules are present in the domain of healthcare as complex legislation rules guiding the payment of medical costs by patients, or in the banking business as for example rules governing international bank transfers.

There is an increasing awareness with respect to the need for making business rules explicit throughout the software development cycle [B. 01] [G. 03]. As such, development of business rules also progresses from the discovery phase, to analysis, design and finally implementation, at all times keeping the rules separate from the core application functionality. Ultimately, the business rules are expressed in an implementation language that is able to execute the rules at certain events, i.e. points in the execution of the core application functionality. Although it considerably improves the more traditional object-oriented software development, we identify three problems with this approach.

First of all, at the implementation level, business rules are represented in some language that is executable, which is invariable a programming language. Moreover, the rules are expressed in terms of implementation elements of the core application. As a result, the business rules are no longer understandable by domain experts, who are typically not adept at programming. A second problem is one that we have identified and addressed in earlier work : the *connection* of the business rules crosscuts the core application and therefore *Aspect-Oriented Programming* (AOP) is a good technique for encapsulating it [CIB 03a] [CIB 03b] [CIB 04][D'H 04b]. However, as with the first problem, the business rule connection is expressed entirely at the programming level, and is thus again not understandable by the domain expert. Finally, a high-level — and executable — specification of business rules can be discrepant from the implementation of the core application functionality. This is due to business rules not always being anticipated in the core application. As such, there is not always a one-to-one mapping from some entities in the business rules to corresponding implementation entities, as current high-level business rule languages only support [JRu] [Qui] [Vis] [Hal].

In this paper we address the above problems and propose a *high-level domain model* consisting of domain concepts, business rules about these domain concepts, and connections of business rules to the core application in terms of the domain concepts (section 2). The link to the implementation is invisible to the domain experts and encapsulated in a *mapping* (sections 3 and 4). It maps elements from the high-level domain model to (1) existing implementation elements, which are implemented with OOP or AOP or (2) new implementation elements, which are implemented with OOP or AOP. It is important to note that the idea of a high-level domain model is not new,

but that the mapping we propose in this paper is our most important contribution. Our approach considers two different scenarios : 1) the domain model is constructed on top of an already developed application ; 2) following the ideas behind Model Driven Engineering (MDE), a first conceptual model exists which is then refined into an implementation model. In the latter case the entities in the conceptual model serve as the domain model entities used in the high-level rules.

We evaluate our approach in the *Web Services Management Layer* (WSML) [VER 04b] [VER 04a], a non-trivial service-oriented framework for creating applications by selecting, composing and integrating Web Services. The WSML contains many business rules, which have crosscutting connections and often require unanticipated changes, and is implemented in an AOP language. We evaluate our approach by means of two scenarios (section 5) : (1) extracting implicit business rules from the WSML and representing them in a high-level domain model, and (2) extending the WSML with unanticipated business rules. We conclude the paper with a discussion on the implementation (section 6), related work (section 7) and a conclusion (section 8).

2. Domain model

The characteristics pursued in the design of this domain model are : *high-level* and *declarative*. *High-level* means that no details are exposed about the concrete implementation of the core application where the rules are applied. *Declarative* means that the focus is put on *what* the rules do and not *how* they do it. Such a model allows expressing domain knowledge in terms of concepts of the real-world domain. Its high-level nature allows the reusability of the domain knowledge among different applications on the same domain or among different versions of an evolving application.

The domain model consists of three components : the domain entities, the business rules and the specification of the connection of the rules with the core application. The **domain entities** represent the vocabulary of the domain of interest. They are the building-blocks used in the definition of the high-level rules and their connection with the core application. The **high-level business rules** express relations between terms of the domain which are captured as domain entities. Thus, rules are independent of implementation details. 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.

2.1. *Domain entities*

In order to specify the domain vocabulary of our domain model, we propose domain entities that are based on very small subset of the typical modelling elements : *domain class*, *domain attribute* and *domain method*. A *domain class* defines a set of domain attributes and a set of domain methods and can have many instances. A *domain attribute* describes a property of the instances of a domain class whereas a

domain method represents a behavior that can be invoked on instances of the domain class. Examples domain classes found in the domain of an e-commerce application are *Customer*, *Product*, *ShoppingBasket*, *Account* and *Shop*. A *Customer* typically defines domain attributes such as the *name*, *age* and *account* and is able to respond to domain methods to *add a product to his/her shopping basket* and *check out*. Domain entities allow explicitly capturing domain knowledge. They can be either used to extract domain knowledge present in the implementation of an existing application or to express new domain vocabulary that needs to be constructed as a result of domain evolution.

2.2. *High-level business rules*

Rules express relations between domain entities which define or constraints some aspect of the business [Bus]. They are high-level and declarative since they express *what* to do and not *how*. The evaluation of the rules can vary depending on the context where they are integrated. Expressing rules at the level of implementation is not desirable since it affects expressivity and reusability of the logic behind those rules. Rules are driven by business decisions and thus generally specified by business analysts who are not necessarily aware of implementation details. Therefore, hiding technical complexity allows them to concentrate on the business logic itself, independently of the way it is implemented. The goal is to adapt (if possible non-invasively) the current implementation to the desired business logic, rather than having to change the logic to conform with the implementation.

The idea of defining rule-based knowledge in terms of a high-level rule language is not new and therefore present in some existing approaches [JRu] [Qui] [Vis] [Hal]. They provide support for declaratively specifying rules in terms of domain concepts described in a business model. Although these approaches allow specifying rules at a higher-level, the offered support is limited, as explained in Section 7.

Similarly to the way rules are specified in current approaches, we define a high-level rule as an *if condition then action* statement, meaning that the condition has to evaluate to true in order for the action to be performed. It is high-level because its condition and action are defined in terms of high-level domain entities. Part of the rule is also the specification of the required business objects that need to be provided at rule instantiation or connection time.

A business rule language is proposed which allows the expression of these high-level rules. In this language, 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 specified in the rule. Also these elements can be combined in logical or comparison expressions as well as in nested combinations. An example condition is : *customer.account.amountSpent() >= 100 OR customer.account.productsBought() >= 10*, where *customer.account* refers to the domain attribute *account* in *customer*, which is an instance of the *Customer* domain class ; *customer.account.amountSpent()* refers to the result of invoking the domain method *amountSpent()* defined in

the domain class *Account*, which is the type of the *account* attribute. The action part denotes the invocation of domain methods that can involve accessors, reference to business objects and domain method invocations. An example action is : *basket.setDiscountRate(discount) AND customer.becomeFrequent()* where *basket* and *customer* are instances of the domain classes *ShoppingBasket* and *Customer* respectively and where *setDiscountRate(discount)* and *becomeFrequent()* are domain methods defined in those domain classes.

The proposed business rule language is very simple, facilitating the writing of the rules. However, rules can be very powerful because they do not simply involve aliases to implementation entities but also entities that can have a very complex realisation at the level of the implementation. Contrary to current approaches where the rule itself can refer to complex OO constructs, in our approach the rules remain very simple since all the complexity is taken away from their specification and encapsulated in a mapping (Section 3).

2.3. *High-level business rule connections*

Defining the connection of a rule with the core application implies specifying the moment during the occurrence of a business process at which the rule needs to be applied. In our domain model this moment is captured by a high-level *event*. From the point of view of the business analyst, events represent points where it is likely to have business logic applied. Separating the event from the rule itself enables reusability of both parts.

Events are defined in terms of high-level domain entities, and thus are independent of implementation details. Two kinds of events are identified : *execution* and *contextual* events. An *execution event* denotes a moment in the execution of a domain method, i.e. *before*, *after* or *around* its execution. Before and after events denote specific points in the execution a domain method whereas around events designate the period of time around the execution of a domain method, from the moment the execution starts till it ends. An execution event denotes, for instance, the moment *before a customer is checking out*, meaning the point in time just before the domain method *checkOut(shoppingBasket)* defined in the domain class *Shop* is executed. Other examples of execution events are : *around obtaining the product price* and *after a customer logs in*. Execution events are used to denote the moment when a business rule needs to be applied : when a rule is applied at a before or an after execution event, the additional rule behavior is added to the core application. When the rule application ends, the original application flow continues normally. When a rule is applied at an around execution event, the additional rule behavior is superimposed on the original behavior performed by the domain method. Thus it can influence its original execution, either by augmenting the original behavior or replacing it.

A contextual event, instead of identifying a particular execution point, defines a context or situation in a business process. Examples are : *the context of the shipping*

process and *the context of the checkout process*. Typically a context is defined based on a domain method since it captures the control flow of the domain method's execution. The difference between a contextual event and an around event is that a rule cannot be applied at only a contextual event. A contextual event needs to be combined with an execution event in order for a rule to be applied on it. Events can be combined in *composite events* using logical operators, allowing one execution event to be combined with one or many contextual events ; for instance, *when obtaining the product price and not in the context of the checkout*.

The information available in the context of the event at which the rule is applied can be passed to the rule to enable its application. Analogously, information modified or calculated by the rule can be passed back to the context of that event, possibly influencing the original business process. When the information required by the rule is not available at the event where it is applied, *capture points* can be defined as part of the connection. A capture point is a domain entity in the high-level model that defines *when* and *which* required information needs to be captured. It identifies an event describing the moment the required information is available and specifies which information to capture from the context of that event. In the rule connection, a mapping is done between the business objects required by the rule and the information available at connection time.

3. Mapping of domain entities to the implementation

In previous sections, we presented the high-level entities that constitute our domain model. In this section, we focus on the realization of these high-level entities at the level of the implementation and present mapping categories of interest. We consider the cases where the target application is developed in OOP and eventually AOP. Both cases are possible : the target application already exists at the moment the domain model is created or the target application is developed afterwards, as a refinement the domain model. Orthogonal to this distinction, we also identify two cases of mapping, which are explained in detail in sections 3.1 and 3.2 : **anticipated mapping** from domain entities to implementation entities — either OO or AO — that are present (directly or indirectly via attribute navigation) in the core application ; and **unanticipated mapping** of domain entities that do not have an existing implementation in the current application. In the latter case, new implementation entities need to be constructed from existing implementation entities. The mapping encapsulates the knowledge of *how* to construct those entities, either using OO or AO. To clarify the distinction between anticipated and unanticipated mappings, the following example is considered : a domain attribute *age* is defined in the domain class *Customer* ; at implementation level, only the attribute *dateOfBirth* is provided in the *Customer* class. Thus, an unanticipated mapping is needed in order to define *how* the customer's age needs to be calculated in terms of his date of birth. In contrast, if a method for the calculation of the age was already anticipated in the implementation, then an anticipated mapping would be needed between the high-level attribute *age* and that existing method.

3.1. *Anticipated domain entities : making domain knowledge explicit*

Generally, a software application is written with the purpose of solving a certain problem of a domain of interest. Thus, in the implementation of a software application we find software entities that represent domain entities. However, the domain knowledge appears scattered all over the code and tangled with other technical concerns [D'H 02]. Thus, it is hard to identify which entities in the code implement domain concepts and which ones are inherent to the specific software solution and not important from the point of view of the domain. Therefore, the goal is to make the domain knowledge of a software application explicit and externalize it as high-level domain entities. This would facilitate reasoning about the domain in its own vocabulary through the definition of rules that only talk about domain terms. Thus, domain entities can be used to exhibit the domain knowledge on which a software application operates. A given domain concern can be implemented in many ways. In this section we analyze some possible implementations of a domain concern. We do this separately for an application developed using OOP and AOP. The domain entities can map to OO entities or to AO entities, in case they are crosscutting in the implementation.

3.1.1. *Mapping to OOP*

This mapping identifies the case where a given domain entity maps to an existing OO entity that implements that concept, namely an OO class, OO attribute or OO method. Different mappings from domain entities to implementation entities are possible which are depicted in Figure 1. The notation used in the diagrams throughout the paper is : the left-hand side contains domain model entities whereas the right-hand side implementation entities ; **a**, **b**, **c** and **d** represent independent cases (do not need to occur simultaneously).

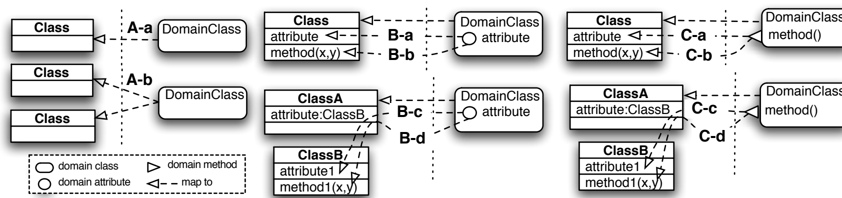

Figure 1. *Anticipated mapping : domain entities map to OO entities*

A) A domain class is mapped to : **a**) an OO class : all the domain entities in domain class map to implementation entities of the class ; **b**) many OO classes : domain entities of domain class map to domain entities of different OO classes.

B) A domain attribute is mapped to : **a**) an OO attribute : the value of the domain attribute corresponds to the value of the OO attribute ; **b**) an OO method : the value of the domain attribute corresponds to the result of invoking the OO method ; return

value expected ; OO method either parameter-less or all its parameters (x and y) are fixed at mapping time ; **c**) a referred OO attribute : the value of the domain attribute corresponds to the value of an OO attribute that is referred to by an instance of the OO class to which the domain class is mapped ; **d**) a referred OO method : analogous to previous case but mapping to the result of invoking a referred method.

C) A domain method is mapped to : **a**) an OO attribute : the result of invoking the domain method corresponds to the value of the OO attribute ; domain method must be parameter-less ; **b**) an OO method : both OO and domain methods with same number of parameters or extra parameters of OO method fixed at mapping time ; **c**) a referred OO attribute : analogous to **B-c** ; domain method must be parameter-less ; **d**) a referred OO method : analogous to **B-d**, idem **C-b** for parameter correspondance.

3.1.2. *Mapping to AOP*

When the core application is implemented in AOP, domain knowledge can appear implemented by aspects. Similar to OOP, it is desired to extract that knowledge from implementation entities to domain entities. The considered AOP entities that can be extracted as domain entities are : aspect, advice, aspect method (method defined in an aspect) and aspect attribute (attribute defined in an aspect). Even though introduction declarations are of great use when new units of structure or behavior need to be added to existing classes, their advantages are only exploited when obliviousness (i.e., elements from the aspects are not referred to by the base code since the latter is not aware of the existence of the former) is not a requirement. In this work we consider obliviousness as a requirement. Thus, the use of introductions has no advantage as the core application is not aware of the introduced features (only of interest for the aspect layer). Alternative solutions based on different aspect instances that would keep the added features encapsulated in the aspects themselves (even at runtime) can be possible involving only the considered AOP entities.

Figure 2 depicts the different mappings in this category.

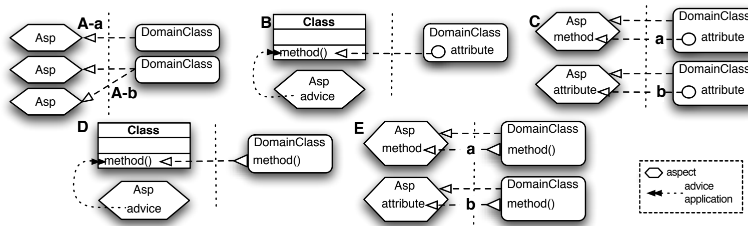

Figure 2. *Anticipated mapping : domain entities map to AOP entities*

A) A domain class is mapped to **a**) an aspect or **b**) many aspects : analogous to definitions **A-a** and **A-b** in 3.1.1.

B) A domain attribute is mapped to an OO method whose execution is modified by an advice. If it is a before or after advice, then this mapping is equivalent to an anticipated mapping to the OO method (without advices). If advice is around, then the value of the domain attribute corresponds to the value returned by the advice applied upon the method ; OO method is either parameter-less or have all its parameters fixed at mapping time.

C) A domain attribute is mapped to : **a**) an aspect method : the value of the domain attribute corresponds to the value returned by the aspect method ; **b**) an aspect attribute : the value of the domain attribute corresponds to the value of the aspect attribute ; in both cases the aspect instance corresponds to the domain class instance that defines the attribute.

D) A domain method is mapped to an OO method whose execution is modified by an advice. The result of invoking the domain method is defined analogously to case **B**.

E) Analogously to definitions **C-a** and **C-b**, a domain method is mapped to : **a**) an aspect method ; **b**) an aspect attribute ; idem **C** regarding instance correspondence.

3.2. *Unanticipated domain entities : defining new domain vocabulary*

Up until now we presented how domain entities can make existing domain knowledge explicit in order to define rules that reason about that knowledge. However, it can be desirable to express new domain knowledge that was unforseen in the core implementation. As domains evolve, new rules can involve domain entities that were not anticipated in the original domain model. These entities represent information or behaviors that are calculated from the existing application. The difficulty lies in realizing the implementation of these unanticipated domain entities, as adaptations and extensions to the core application may be needed. Moreover, the process of calculating the derived information or executing that derived behavior can result in crosscutting code in the core application. We want to preserve the transparency between the two worlds, the declarative domain layer and the implementation layer, and to avoid invasive changes to the core application. Our solution proposes the use of AOP to adapt the core application non-invasively in order to implement these new domain entities and envision their automatic code generation. As a result of the use of AOP, complex mappings can be realized. In this section we propose implementation solutions based on OOP and AOP for the realization of unanticipated domain entities.

3.2.1. *Mapping to OOP*

The new unanticipated entities are defined declaratively in terms of other domain entities. More specifically, a new domain entity can be defined in terms of an expression (involving an arithmetical or logical operator or a domain method invocation) that combines other domain entities. The implementation is transparently derived from this declarative specification. Figure 3 illustrates the different mappings to OO constructs for derived domain attributes and methods.

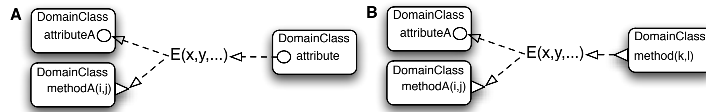

Figure 3. *Unanticipated mapping : domain entities map to derived expression*

A) A domain attribute is mapped to an OO expression in terms of existing domain entities ; domain methods involved must be either parameter-less or its parameters (i and j) need to be fixed at mapping time. Example : *customer.discount* ***maps to*** *percentage(10, (customer.getAccount().totalAmountSpent))*

B) A domain method is mapped to an OO expression in terms of existing domain entities ; parameters of domain method (k and l) can be used directly in the expression or in the invocation of other domain methods involved in the expression (methodA). Example : *customer.youngerThan(customer2)* ***maps to*** *customer.age < customer2.age*

3.2.2. *Mapping to AOP*

We can imagine that the definition of a domain entity could involve more complex processing than just evaluating a simple OO expression. For instance, a new domain attribute can be defined to represent the *customer's average amount spent in his/her last 10 transactions*. As the realization of this kind of domain entities can result in crosscutting code, we explore the use of AOP. Figure 4 depicts the possible mappings realized through aspects.

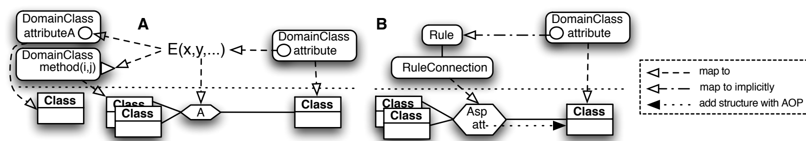

Figure 4. *Unanticipated mapping : domain entities map to AOP*

A) A domain attribute is mapped to the result of a *crosscutting* expression ; A crosscutting expression applies a crosscutting operator (pre-defined in the domain model) to existing domain entities. These operators are called crosscutting because their realisation would normally result in crosscutting code in the core application. Therefore, in our model they are implemented using AOP. This mapping is then realised by an aspect that calculates the value of the domain attribute. Example : *customer.averageTimeBetweenOrderAndPayment* ***maps to*** *average(****{timeBetween}*** *customer.addProduct(p)* ***{and}*** *customer.checkOut())*, where **timeBetween/and** is a crosscutting operator.

B) A domain attribute is *implicitly* mapped to a business rule's action : *when* and *how* to set the value of a domain attribute is determined by the application of a business rule ; at implementation level, the aspect that encapsulates the rule's connection adds an attribute to the base implementation class. Example : *customer.frequent* **implicitly *maps to*** *BR's action*, where BR specifies that *"if customer.account.productsBought() >= 10 then customer is frequent"*.

The mapping of an unanticipated **domain method** is analogous to mapping **A** with the difference that parameters can be defined to be used in the evaluation of the expression. A different aspect instance is needed per instance of the domain class where the method (or attribute) is defined. Example : *customer.averageAmountSpentInLast-Transactions(10)* ***maps to*** *average({**returnOf**} customer.checkOut() {**10 times**})*.

4. Mapping rules and their connection to implementation

Rules are mapped to OO classes that define methods for their conditions and actions, as illustrated in Figure 5. The connection of business rules crosscuts the core application, as observed in [CIB 02]. In previous work we identified the suitability of AOP for implementing the connection of rules [CIB 02] [CIB 03a] [CIB 03b] [CIB 04]. Thus, the declarative and high-level specification of the rule connection maps to the AOP solutions proposed in that work. Depending on the configuration of the connection, the mapping results in a different pattern that combines different building-block solutions regarding (e.g., application time, capturing of required information, rule combination strategy). Figure 5 depicts this situation. Basically the high-level connection is mapped as follows : an event is mapped to a pointcut ; a capture point is mapped to a pointcut that intercepts the application execution in order to get the desired information, which is then captured and stored in an advice on that pointcut ; the rule connection is mapped to an aspect that puts all the pieces together and triggers the application of the rule.

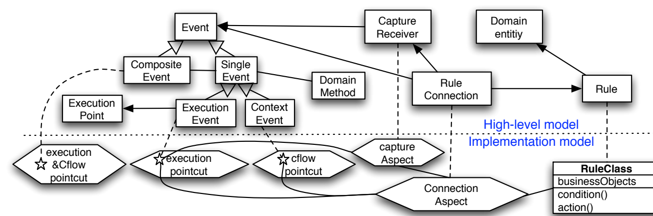

Figure 5. *Mapping high-level business rules and connection to implementation*

5. Evaluation : Web Services Management Layer

This section illustrates the contributions of our approach, i.e. the declarative and high-level properties of the business rules and their powerful mapping to implementation. In order to do so, the Web Services Management Layer (WSML) [VER 04b] [VER 04a] is used as a case study. The WSML 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 can be seen as an AOP framework that facilitates the development of service-oriented applications : it offers a library of generic and reusable selection and management *templates* that can be personalized according to the requirements of the client applications. These templates are implemented as dynamic and reusable JAsCo [SUV 03] aspects and can be deployed in different contexts using JAsCo connectors. 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 a monitoring template. These monitored properties are used to make decisions about the selection of services. More details on the WSML and its implementation be found in [VER 04b] [VER 04a].

We observe that the management, selection and integration of web services is rule intensive. In particular, the pluggability of the selection and management templates in the WSML is driven by rules. Examples are the need for a caching management strategy depending on the response time of services and the preference of services whose number of failures is zero. Other rule intensive tasks include, but are not limited to, guiding the adaptation of selection and management, performing calculations based on information captured as a result of management or monitoring (gathered by aspects), classifying services into categories and guiding the way service compositions are formed and adapted. Currently, this knowledge is either interpreted manually, with human interaction, or implicitly implemented, tangled in the implementation of the management and selection templates. This introduces all the problems discussed in section 1.

In this section we present two usage scenarios of our approach : (1) extracting implicit business rules from the WSML and (2) extending the WSML with new, unanticipated business rules. As such, we show that our approach supports the specification of high-level domain entities, business rules and connections that map onto the existing WSML implementation in OOP and AOP, without having to change it. Moreover, we show that the definition of new, high-level domain entities, business rules and connections that are not anticipated, can also be added to the original WSML implementation without having to change it. Whereas the former requires mappings of domain elements to AOP elements, the latter requires the mapping itself to be AOP.

5.1. *Extracting implicit business rules from the WSML*

In the current implementation of the WSML, there are implicit, low-level business rules scattered across the management and monitoring aspects. For instance, the aspect *ConditionalCaching* checks whether the average speed (i.e. throughput) of the service that is active for invocation is smaller than a certain value in order to start caching the results. The same condition is implemented in the aspect *ConditionalMonitoring* as a requirement to start monitoring the value of dynamic service properties.

The first step in extracting the business rules and making them explicit and high-level, consists of defining the domain model, i.e. the entities, business rules and their connections. Figure 6 shows the subset of the resulting domain entities that is relevant for this paper. Note that as this is an information system domain, the domain expert that would typically write rules about this domain needs to be familiar with the concepts involved in the WSML, such as web services, number of failures, service selection, monitoring and management.

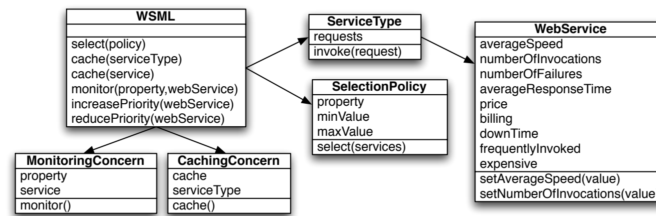

Figure 6. *Domain entities in the WSML*

The business rules are expressed at a high level in terms of these identified domain entities, shown below.

R1 : ***If*** *ST.activeService.averageSpeed* $<$ *1000* ***then*** *WSML.cache(ST)*

R2 : ***If*** *S.averageSpeed* $<$ *1000* ***then*** *WSML.monitor(numberOfFailures, S)*

R1's **application time** corresponds to the moment the client invokes a certain service functionality : *around the invocation of ServiceType»invoke(request)*. R2's **application time** corresponds to the moment the average speed of a service is changed : *after invocation of WebService»setAverageSpeed(value).*

The next step consists of specifying the mapping of the domain model elements to the existing WSML implementation. First, we start by mapping the domain entities. As they are extracted from an existing implementation, there is a one-to-one mapping from the extracted domain entities to corresponding implementation entities in the WSML implementation. Since the latter is implemented with OOP and AOP, a domain entity can map to an OOP or an AOP entity. Below are two example mappings to OOP. The first is a mapping to an OOP method enabling the caching aspect for the

service type received as parameter, whereas the second is a mapping to an OOP method enabling the monitoring aspect that calculates dynamic properties for the service received as parameter.

WSML»cache(ST) ***maps to*** *TemplateRegistry»public boolean enableTemplateInstance-(ST + "Caching", "ServiceTypeCaching")*

WSML»monitor(property, S) ***maps to*** *TemplateRegistry»public boolean enableTemplateInstance("Monitoring" + property + S, "ServiceMonitoring")*

We now present an example mapping to AOP. The monitored properties in the WSML are captured by and stored in monitoring aspects. The template ServiceMonitoring is a generic aspect that can be instantiated in different connectors for different properties and per different services. For example, the connector MonitoringAverageSpeed instantiates the aspect ServiceMonitoring with the property *average speed* per instance of the class *WebService*. Thus, the domain attribute *WebService» average speed* is mapped to the retrieval of the attribute *property* from the aspect instance associated to that specific service. This mapping is shown below.

service1.averageSpeed ***maps to*** *(IMonitoredProperty) AverageSpeedMonitoring.get-InstanceFor(service1).getProperty()*

A business rule is automatically mapped to a Java class with a condition method and an action method, implemented in terms of the mapping of the involved domain entities. The high-level connection is translated to an aspect that captures the application time of the business rule by means of a pointcut. The aspect's advice triggers the actual application of the rule.

5.2. *Extending the WSML with unanticipated business rules*

New business rules for guiding the management and selection are for example :

if *service1.numberOfFailures < X* ***then*** *WSML.increasePriority(S)*
if *service1.averageResponseTime < X* ***then*** *WSML.cache(S)*
if *service1.billing > X* ***then*** *WSML.reducePriority(S)*

Since these are based on monitored properties and management results, the corresponding domain entities are anticipated (section 5.1). Therefore, it is obvious that the high-level definition of these new business rules suffices to change the behaviour of the WSML.

However, there are other unanticipated business rules that require new domain entities :

if *service1.billing > X* ***then*** *service1 is expensive*
if *service1.numberOfInvocations > X* ***then*** *S is frequentlyInvoked*

The entities *expensive* and *frequently invoked* are domain attributes that are not anticipated in the WSML, but are required by the new business rules. In particular,

these rules guide *how* and *when* to set these attributes, so the mapping to implementation of the domain attributes is closely linked to the application of the rules which is encapsulated in rule connection aspects. Also, AOP is needed to non-invasively introduce these new attributes to the base implementation class that maps to the *WebService* domain class. Thus, the rule connection aspects are extended to encapsulate the implementation of these domain attributes.

5.3. *Discussion*

As a result of performing the two above scenarios, we achieve the following advantages, which we discuss :

– *enhanced adaptability and variability*
The management and selection aspects are now guided by externalised business rules instead of hard-coded policies. This means that alternative business rules can be specified and applied in order to adapt the guidance of management and selection, without having the adapt existing business rules or the core WSML implementation. Moreover, different versions of the WSML implementation can be created by plugging in different sets of business rules. This advantage comes at a certain cost, however, since for each business rule a — possibly unanticipated — connection to the core application needs to be specified.

– *improved understandability*
In addition to the above advantage, it is now possible for a domain expert to add new business rules. However, his advantage requires an initial investment : the domain entities need to be constructed first. Adding new rules is trivial if it can be specified in terms of existing domain entities and can reuse an existing connection. Moreover, even if new domain entities and/or connection are required, 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. The mapping does not have to be implemented entirely by hand since we provide templates guided by wizards for generating the mapping implementation. Nevertheless, input from one knowledgeable about the implementation is required for providing the correct parameters for the generation.

– *improved reusability*
The same business rules can be reused to guide the pluggability of different management, selection or monitoring aspects.

6. Implementation

The current implementation supports the entire domain model, as explained in section 2. As such, it supports the definition of domain entities, high-level rules in terms of those domain entities, and connection of the rules also in terms of the high-level entities.

A parser for the high-level business rule language is implemented. The high-level rules specified in this language are syntactically parsed and automatically translated to an object-oriented rule implementation, more specifically in Java. This translation process is carried out in cooperation with the domain model in order to validate the domain entities used in the definition of the rules. The domain model is also consulted to get information about how the high-level rules map to a concrete implementation, following their mapping specifications (as presented in section 4). Note that, as explained in that section, some mappings directly point to implementation entities (in the case of domain entities that are aliases of implementation entities and thus the mapping represents a one-to-one relation between them) and others are defined in terms of domain entities. In the latter, the translation process becomes more complex as nested mappings need to be explored in order to calculate a representation of the mapping only in terms of implementation entities, which is then used in the generation of the rule implementation. This translation from a high-level mapping to an implementation-based mapping is done automatically.

An implementation of the parser for the high-level rule connection language is also provided. This language allows to connect a high-level rule at an event. Also, using this language it is possible to map information required by the rule with contextual information available at the application event (and eventually extra capturing events). Moreover, information modified by the rule can be passed back to the context on which the rule is applied, potentially modifying the control flow of the core application as a result of applying the rule. The high-level connection specified in this language is automatically translated to a JAsCo aspect bean, which puts all the component pieces together : it defines a hook for the application event associated with an advice (before, after or around depending on the application event specification) that triggers a rule instance at the specific joinpoint using the required contextal information and a different hook per capture point with an associated advice that captures the desired information and makes it available to the rule. JAsCo connectors are automatically generated to deploy the aspect bean at the concrete events, using their mapping information. For this, mapping restrictions (such as fixed values encapsulated in the mapping) are taken into account in order to generate aspects that only hook at very specific method invocations (for instance, invocations where the parameters coincide with the fixed values specified in the mapping).

We are currently investigating further which of the four categories of mappings of domain entities that we identify and discuss in section 3 can be supported fully automatically and which others have to be defined semi-automatically or even manually. In any case, adapting the implementation of particular mappings does not affect the basic setup of our approach, which consist of separating the high-level and declarative specification of the domain model from the mapping to a particular implementation. We also envision the definition of a language for specifying the mapping, which at the moment has to be manually defined by instantiating mapping classes in the domain model.

Currently, the notation used in the high-level business rule language is based on the typical syntax used in modeling languages : a *dot* is used to refer to domain attributes and to invoke domain methods on instances of domain classes, and parentheses are used to specify the arguments of method invocations. Although it could be desirable to enhance this notation in order to make the language more human-readable (e.g., by adopting a more smalltalk-like syntax which would allow infix specification of the parameters), the discussion on this matter is not a contribution of this paper. This is a field extensively explored in the domain of artificial intelligence and also in existing high-level business rule languages (e.g., JRules [JRu] and HaleyRules [Hal]). Instead, the focus of this paper is put on the declarative and high-level properties of the suggested language and its powerful mapping to implementation.

7. Related work

Several state-of-the-art systems that support business rules were studied and analyzed, among them JRules [JRu], OPSJ [ops01], NéOpus [PAC 95], HaleyRules [Hal], QuickRules [Qui], RuleML [Rul], IRules [IRu], OptimalJ [Opt] and VisualRules [Vis]. Some approaches, such as JRules, OPSJ, NéOpus, HaleyRules and QuickRules provide a rule-based language for expressing rules, which is more declarative than for example an object-oriented programming language. However, rule-based languages are *programming* languages, requiring the user to have programming skills, as opposed to high-level languages, which can be used by domain experts. Nevertheless, some approaches provide a high-level, declarative language for expressing rules in addition to the rule-based programming language. The former is typically translated into the latter. Examples of systems that support this are JRules, QuickRules and HaleyRules. The high-level rules are also defined in terms of domain concepts captured by a business model. This model is the result of either manually or automatically extracting domain knowledge from an existing object-oriented implementation or XML schemas. Basically, the business model define the OO classes and methods to which the business rules are applied, and maps the natural language syntax of the business rule language to these implementation entities. Thus, the domain entities are simply aliases for implementation entities, requiring a one-to-one mapping between them. As a consequence a tight coupling exists between the domain model and the implementation model.

Our approach extends this idea in two dimensions : first of all, we also consider the case where domain concepts are mapped *to* a crosscutting feature, encapsulated in an aspect-oriented implementation, and secondly, the mapping itself can be aspect-oriented, which allows for unanticipated mappings between domain and implementation entities. Thus, among the mapping categories presented in this paper, current approches only support the basic category of *anticipated mapping to OOP* (more specifically category A-a, B-a and C-b), but they fail at supporting the other categories of *anticipated mapping to OOP* and do not support at all *unanticipated mapping to OO*, *anticipated mapping to AOP* and *unanticipated mapping to AOP*.

Rule-based programming languages (e.g., the ILOG Rule Language (irl) language supported in JRules) typically allow writting object-oriented code directly in the rules themselves. This makes the rules more powerful but also more complex, since the whole complexity of the object-oriented paradigm is added. Rules are not high-level since they directly refer to implementation entities. In our approach, rules are very simple because only references to domain attributes and domain method invocations of a domain model are possible. All the complexity that would otherwise appear in the rules themselves, is encapsulated in a mapping. This mapping can be very sophisticated, making the rules very powerful.

In some existing approaches, an intermediate language is used during the rule translation. This is the case of JRules for instance, where high-level rules expressed in terms of a business model map to low-level executable rules expressed in irl, the language understood by JRule's engine. This implies that translated rules can only be reused in JRules-enabled applications. In our approach, there is no intermediate language : our high-level rules are directly translated to a low-level OO language. This facilitates the reusability of the translated rules in any other OO application.

Moreover, all previous approaches are based on rule engines that are in charge of asserting and executing the rules. The rule engines need to be triggered explicitly from the applications that integrate the rules. Also, application objects must be asserted into, retracted from or updated in the working memory before pattern matching in rules can begin. This is typically done either by using keywords within rules or through APIs invoked within application objects themselves. As a consequence, the connection of the rules hampers the reusability and maintainability of the application code, since every time rules/connection change, the application has to do so.

In previous work we observe that the connection of business rules crosscuts the core application [CIB 02] and we evaluate the suitability of AOP for implementing the connection of rules. A set of requirements for decoupling the business rules connection is identified and experiments are done in AspectJ [Asp] and JAsCo [SUV 03] illustrating their suitability [CIB 03a] [CIB 03b]. More generally, we have identified a number of aspect-oriented features, both at the language and the technology level, required for addressing the identified requirements and we have examined several state-of-the-art AOP approaches against those features [CIB 04]. Other related research focuses on the suitability of AOP for integrating object-oriented and rule-based programming languages [D'H 04a]. Hybrid aspects are proposed [D'H 04b] to achieve a very loose coupling between both paradigms at the program and the language level.

Even though this previous work shows the suitability of AOP for addressing the problems of encapsulating the business rules connection code, a main problem in the adoption of such a solution is the difficulty and complexity raised by the use of AOP. Some work has been done to support the generation of aspects that are identified in [CIB 02] for connecting business rules through wizards [ELS 04]. This is a first step towards facilitating the adoption of an AOP-based approach for the connection of the rules. However, these wizards are not high-level since they work with elements of the implementation.

8. Conclusion

In this paper we propose a high-level domain model consisting of domain entities, a business rule language and connections of rules to domain entities. The elements of the domain model are mapped to the implementation of the core application functionality. The objective of this work is to relieve the business analysts and domain experts from requiring to have technical skills in order to define business rules. The use of AOP is explored in various regards, mainly as a means to non-invasively realize the implementation of domain model elements (crosscutting at implementation level) that are unanticipated in the core application, and to support a transparent connection of the rules with the core application.

The high-level nature of our approach helps enhancing the understandability of the business rules. By applying our approach to the WSML, we show that extracting implicit business rules from the implementation of the core application is supported, as well as extending the domain model with unanticipated business rules without having to change the implementation of the core application functionality. In this case, constructing the domain model implies a certain initial overhead. However, once the model is in place, adding business rules requires only minimal changes to the model. Of course, in the case where the domain entities are going to be choosen from an existing conceptual model, less initial effort is required for the construction of the domain model.

9. Bibliographie

[Asp] AspectJ, http ://eclipse.org/aspectj/.

[B. 01] B. V. H., *Business Rules Applied*, Wiley, 2001.

[Bus] The Business Rules Group, « Defining Business Rules : What Are They Really ? », http ://www.businessrulesgroup.org/.

[CIB 02] CIBRÁN M. A., « Using aspect-oriented programming for connecting and configuring decoupled business rules in object-oriented applications », Master's thesis, Vrije Universiteit Brussel, Belgium, 2002.

[CIB 03a] CIBRÁN M. A., D'HONDT M., JONCKERS V., « Aspect-Oriented Programming for Connecting Business Rules », *Proceedings of BIS International Conference*, Colorado Springs, USA, juin 2003.

[CIB 03b] CIBRÁN M. A., D'HONDT M., SUVÉE D., VANDERPERREN W., JONCKERS V., « JAsCo for Linking Business Rules to Object-Oriented Software », *Proceedings of CSITeA International Conference*, Rio de Janeiro, Brazil, juin 2003.

[CIB 04] CIBRÁN M. A., SUVÉE D., D'HONDT M., VANDERPERREN W., JONCKERS V., « Integrating Rules with Object-Oriented Software Applications using Aspect-Oriented Programming », *Proceedings of ASSE'04, Argentine Conference on Computer Science and Operational Research*, Córdoba, Argentina, 2004.

[D'H 02] D'HONDT M., CIBRÁN M. A., « Domain Knowledge as an Aspect in Object-Oriented Software Applications », *workshop on Knowledge-based Object-Oriented Soft-*

ware Engineering at the 16th European Conference on Object-Oriented Programming, Málaga, Spain, juin 2002.

[D'H 04a] D'HONDT M., « Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality », PhD thesis, Vrije Universiteit Brussel, Belgium, 2004.

[D'H 04b] D'HONDT M., JONCKERS V., « Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge », Lancaster, UK, mars 2004.

[ELS 04] ELSOCHT J., « Wizard and code generation support for linking Business Rules to Object Oriented applications using AOP », Master's thesis, Vrije Universiteit Brussel, Belgium, 2004.

[G. 03] G. R. R., *Principles of the Business Rule Approach*, Addison-Wesley Publishing Company, 2003.

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

[IRu] IRules, http ://www.eas.asu.edu/ irules/.

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

[ops01] « OPSJ 4.1 », 2001, Manual by Charles L. Forgy from Production Systems Technologies Inc.

[Opt] OptimalJ Business Rules, http ://www.compuware.com/products/optimalj/.

[PAC 95] PACHET F., « On the Embeddability of Production Rules in Object-Oriented Languages », *Journal of Object-Oriented Programming*, vol. 8, n° 4, 1995, p. 19-24.

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

[Rul] RuleML, http ://www.ruleml.org/.

[SUV 03] SUVÉE D., VANDERPERREN W., JONCKERS V., « JAsCo : an Aspect-Oriented approach tailored for Component Based Software Development », Boston, USA, mars 2003.

[VER 04a] VERHEECKE B., CIBRÁN M. A., JONCKERS V., « Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection », *European Conference on Web Services 2004 (ECOWS'04)*, Erfurt, Germany, 2004.

[VER 04b] VERHEECKE B., CIBRÁN M. A., VANDERPERREN W., SUVÉE D., JONCKERS V., « AOP for Dynamic Configuration and Management of Web services in Client-Applications », *International Journal on Web Services Research (JWSR)*, vol. 1, n° 3, 2004.

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