LINKING BUSINESS RULES TO OBJECT-ORIENTED SOFTWARE USING JASCO

María Agustina Cibrán[1], Maja D'Hondt[1], Davy Suvée[1], Wim Vanderperren[2], Viviane Jonckers

{mcibran, mjdhondt, dsuvee, wvdperre}@vub.ac.be, vejoncke@info.vub.ac.be

System and Software Engineering Lab

Vrije Universiteit Brussel

Pleinlaan 2, 1050 Brussel, Belgium

ABSTRACT

Object-oriented software applications that support a particular business or domain consist of substantial core application functionality and business rules. Since business rules tend to evolve frequently, it is important to separate them from the core application. However, current approaches that support business rules at the implementation level only separate the business rules themselves and not the code that links them to the core application. We observe that this code crosscuts the core application. As a result, Aspect-Oriented Programming is required to separate and encapsulate the linking code. In addition to this, we identify several other requirements for obtaining highly flexible and configurable business rules. In previous work we conducted an experiment with AspectJ for separating the business rule links. Although this delivered satisfactory results for some of the requirements, many others were not fulfilled. This paper shows how JAsCo, an aspect-oriented implementation language combining the advantages of AspectJ's expressiveness with the plug-and-play characteristics of components, succeeds in fulfilling the remaining requirements.

Keywords: Object-Oriented Software Engineering, Business Rules, Aspect-Oriented Programming

1. INTRODUCTION

Software that supports and manages business domains and processes – such as found in electronic commerce, the financial and legal fields, television and radio broadcasting – comes in a wide variety: information systems that are inherently data-oriented [11], rule systems that automate knowledge-intensive domains [30], and software that has a substantial core application functionality supporting the user in his or her tasks without fully automating them. In this paper we focus on the latter kind of software applications, developed using object-oriented or component-based software development techniques.

In this context it is increasingly important to consider *business rules* as a means to capture some business policies explicitly. The Business Rules Group defines a business rule as *a statement that defines or constraints some aspect of the business. It is intended to assert business structure or to control the behaviour of the business* [5]. Business rules tend to evolve more frequently than the core application functionality [20][1][34]. Therefore, it is crucial to *separate* business rules from the core application, in order to *trace* them to business policies and decisions, *externalize* them for a business audience, and *change* them [34]. A business rule is applied at an event, which is a well-defined point in the execution of the core application functionality.

However, approaches that advocate and support the separation of business rules at the implementation level, fail to separate and encapsulate the code that *links* the business rules to the core application. One has to adapt the source code of the core application manually at different places each time business rules change. This phenomenon is known as *crosscutting* code in the area of *Aspect-Oriented Programming* (AOP) [3][9]. AOP advocates extending standard modularization constructs of a programming language with additional constructs to encapsulate crosscutting code. Although AOP is usually employed for encapsulating implementation-level issues like logging

[1] Funded by the *Flemish Institute for the Improvement of the Scientific-Technological Research in Flanders* (Belgium).

[2] Funded by a doctoral scholarship from the *Fund for Scientific Research* (Belgium).

and synchronization, we introduce the idea of domain knowledge as an aspect in [13] and [12]. In [7] and [6] we conducted an experiment which uses *AspectJ* [21] for encapsulating the crosscutting business rule links.

However, separating and encapsulating the business rule links is not sufficient in order to achieve highly flexible and configurable business rules. We identify other requirements, which are presented in the next section. AspectJ addresses some of these issues successfully because of its expressiveness with respect to describing and manipulating events in the core application. Some other requirements however, are not adequately satisfied. This paper reports on our efforts to meet the requirements for the business rule links using *JAsCo* [32], which is an aspect-oriented implementation language integrating the ideas of AOP into Component-Based Software Development (CBSD) [31]. JAsCo combines the advantages of AspectJ's expressiveness with the idea of fully reusable and highly configurable plug-and-play characteristics of components.

After introducing AOP and our requirements in Section 2, we discuss the JAsCo language in Section 3. We show how JAsCo fulfils the requirements for linking business rules in Section 4. Business rules for price personalization in e-commerce are used as a running example throughout the paper since personalization is an increasingly important issue [10] and e-commerce is a favoured case of most business rules approaches [17][2][28]. Finally, we discuss related work in Section 5 and conclude in Section 6.

2. ASPECT-ORIENTED PROGRAMMING FOR BUSINESS RULES

2.1 Introduction to Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) argues that some concerns of a system, such as synchronization and logging, cannot be cleanly modularized using current software engineering methodologies as they are scattered all over the different modules of the system. Similar logic is thus repeated in different modules. Due to this code duplication, it becomes very hard to add, edit and remove such a *crosscutting* aspect in the system. The ultimate goal of AOP is to achieve a better separation of concerns. To this end, AOP approaches introduce a new concept that is able to modularize crosscutting concerns, called an *aspect*. An aspect defines a set of *join points* in the target application where the normal execution is altered. Aspect *weavers* are used to weave the aspect logic into the target application.

Nowadays, several AOP approaches, such as AspectJ, Composition Filters [4], HyperJ [25][33] and DemeterJ [23] are available. These technologies have already been applied on large industrial projects by for instance Boeing, IBM and Verizon Communications. For more information about AOP in general, we refer to [3] and [9].

2.2 Requirements for Business Rule Links

Business rule logic can be seen as the combination of the business rules themselves and the specification of the business rule link with core application events. Ideally, business rules are represented as an "***if*** *condition* ***then*** *action*"-statement. However, at the implementation level of object-oriented applications business rules are typically modelled as classes [1][27]. A business rule class defines operations for the condition and the action. Hence, no aspect-oriented support is needed for the business rules themselves, because object-oriented techniques suffice for encapsulating and reusing them.

However, not only the reusability of business rules is required. The business rule links should also be encapsulated in order to enable reusability. As business rule links crosscut the core application, AOP techniques are required. Moreover, we identify a set of requirements that should be satisfied for an AOP approach to be suitable [7]:

- ✓ connect business rules to core application events which depend on run-time properties,
- ✓ pass necessary business objects to an event in order to make business rules applicable at that event,
- ✓ reuse a business rule link at different events,
- ✓ combine, prioritize and exclude business rules when they interfere with one another,
- ✓ control the instantiation, initialization and execution of business rule links,
- ✓ and preferably accomplish the above dynamically without interrupting the application execution.

AspectJ, which is able to describe and manipulate events in a very expressive way, fulfils the first two requirements successfully, whereas the other requirements are only met partially or not at all [7]. The next section introduces JAsCo, after which we show in Section 4 how it addresses the last four requirements successfully.

3. JASCO

3.1 Introduction to JAsCo

Originally JAsCo was designed to integrate aspect-oriented ideas into Component-Based Software Development. However, JAsCo has some characteristics that are also useful in an object-oriented context:

- ✓ Aspects are described independent of a concrete context, making them highly reusable.
- ✓ JAsCo allows easy application and removal of aspects at run time.
- ✓ JAsCo has extensive support for specifying aspect combinations.

The JAsCo language itself stays as close as possible to the regular Java syntax and concepts and introduces two new entities: *Aspect Beans* and *Connectors*. An aspect bean is an extension of a regular Java Bean component that is able to encapsulate crosscutting behaviour. A connector on the other hand is responsible for applying the crosscutting behaviour of the aspect beans and for declaring how several of these aspects collaborate. On a technical level we introduce a new, backward compatible component model that enables run-time application and removal of connectors. The next two sections introduce aspect beans and connectors in more detail. For more information about JAsCo and its component model, we refer to [32].

3.2 Aspect Beans

Aspects Beans describe some behaviour that would normally crosscut several parts of a system. An aspect bean is an extended version of a regular Java Bean that defines one or more logically related hooks as a special kind of inner classes. The aspect bean itself is used to implement the business rule and to specify the hooks that are used to describe the linking of the rule with the core application. Hence, an aspect bean is able to combine the two parts of the business rule logic in the same module, but is still able to maintain the desired separation and independence between the business rule and the concrete linking within the base application.

```
1   class BRPriceDiscount{
2
3     public void setDiscount(Float aDiscount) {…
4     public Float getDiscount(){…
5     public Float applyDiscount(Float aPrice) {…
6     abstract public Boolean discountCondition(Customer);
7
8     hook BRPriceDiscountHook {
9
10      BRPriceDiscountHook(Float method(Customer c)){
11        execute(method);
12      }
13
14      isApplicable(){
15        return discountCondition(c);
16      }
17
18      replace(){
19        Float price = method(c);
20         return applyDiscount(price);
21      }
22    }
23  }
```

Figure 1: The implementation of the abstract business rule

Figure 1 illustrates a discount aspect bean from which all discount business rules inherit. The `BRPriceDiscount`-aspect bean describes the business rule (lines 3 to 6) and declares a `BRPriceDiscount`-hook (lines 8 to 22) that describes the linking of the business rule with the core application. A hook specifies **when** the normal execution of the base program should be interrupted, and **what** extra behaviour should be executed at that precise moment in time. In order to define when the functionality of a hook should be executed, the hook is equipped with at least one constructor (lines 10 to 12) that takes one or more *abstract method parameters* as input. These abstract method parameters are used for describing the abstract context of a hook. This generic specification of the context of an aspect makes business rule links reusable and as a result deployable in different contexts. The `BRPriceDiscount`-hook specifies that its behaviour is deployable on every method that takes a *Customer* as input and that returns a *Float*-value. The constructor body describes how the joinpoints of a hook initialization should be computed. In this particular case, the constructor-body (line 11) specifies that the functionality of the `BRPriceDiscount`-hook should be performed whenever *method* is executed. The advices of a hook on the other hand, are used for specifying the various actions a hook needs to perform whenever one of its calculated joinpoints is encountered. Three types of advices are available: *before*, *after* and *replace*. The replace behaviour method of the `BRPriceDiscount`-hook (lines 18 to 21) specifies that some discount is given, whenever the `isApplicable`-method returns true. The `isApplicable`-method specifies a dynamic condition that is executed at run-time, to check whether the advices of an aspect should be executed. The specific discount-percentage and the `discountCondition`-method are undetermined at the moment, because this information is specific to each business rule that extends the `BRPriceDiscount` aspect bean.

```
class ChristmasBR extends BRPriceDiscount{

  public boolean discountCondition(Customer customer){
    //return true if Christmas
  }

}
```

Figure 2: The Christmas business rule

Figure 2 illustrates the Christmas business rule, which is a concrete implementation of the discount business rule presented in Figure 1. The `ChristmasBR`-rule only implements the `discountCondition`-method, since the logic behind this method is specific for each discount business rule. In this particular case, the `discountCondition`-method returns true if it is Christmas. As the `ChristmasBR`-rule extends the discount aspect bean, it also inherits the `BRPriceDiscount`-hook.

3.3 Connectors

Connectors are used for instantiating one or more logically related hooks with a concrete deployement context (method or event signatures) and for specifying advanced aspect-combinations. Connectors make it possible to deploy generic business rules in a specific context. Imagine our application implements a `checkOut`-method that iterates over all purchased products and returns the total price. Figure 3 illustrates the `ChristmasDiscountDeployment` connector that deploys the `ChristmasBR`-rule upon this `checkOut`-method.

```
1   connector ChristmasDiscountDeployment {
2
3     ChristmasBR.BRPriceDiscountHook discount =
4       new ChristmasBR.BRPriceDiscountHook(Float CheckOut.CheckOut(Customer));
5
6     discount.setDiscount(new Float(0.05));
7     discount.replace();
8
9 }
```

Figure 3: Deployment of the Christmas business rule

The connector of Figure 3 initializes the `BRPriceDiscount`-hook with the `checkOut`-method defined in the `CheckOut`-class (lines 3 to 4). After initializing this hook, the `ChristmasDiscountDeployment` connector specifies the exact discount (line 6) and the execution of the replace behaviour method (line 7). Consequently, the deployment of this connector has the following implication: apply a discount of 5% on the total price when a customer checks out during the Christmas period.

4. JASCO FOR BUSINESS RULES

4.1 Explicit Connectors

As mentioned before, one of the main advantages of the use of JAsCo is the separation and encapsulation of the deployment details in a new connector construct. For achieving the decoupling of the business rule link, the abstract logic for the application of a business rule is specified by using a generic hook defined in the aspect bean. This way, the crosscutting code remains independent from the details of the concrete deployments and is encapsulated in the connectors.

The example illustrated in Figure 3, specifies the deployment of the application logic of the ChristmasBR whenever the checkout method is executed. Suppose that the business requirements change and the ChristmasBR should be applied only on a specialized customer such as an employee of the firm. This requirement can easily be achieved by specifying another connector that instantiates the same BRPriceDiscountHook, providing the Employee-Customer as a parameter for the checkout method. This way, the specification of this new deployment is encapsulated in the new connector without affecting the previous abstract definition.

```
class FrequentCustomer{

  public Boolean checkFrequentCustomerCondition(Customer c){
    // returns whether customer c is frequent or not
  }

  hook FrequentCustomerHook {

    FrequentCustomerHook(Float method(Customer c)) {
      execute(method);
    }

    isApplicable(){
      return checkFrequentCustomerCondition(c);
    }

    after(){
      FrequentCustomers.addFrequentCustomer(c);
    }
  }
}
```

Figure 4: The FrequentCustomer aspect

Another advantage of having explicit connectors is the possibility to group together the deployment details of logically related business rules. This advantage is illustrated by introducing the following example. Suppose customers must be classified by considering them as frequent or not frequent. To achieve this, a new business rule is specified: if the customer purchased more than 10 items, then the customer becomes frequent. Figure 4 shows the implementation of this business rule as an aspect bean.

Now consider a new business rule for the price personalization that makes use of this new concept of customer frequency: if the customer is frequent, then apply a 5% discount. The aspect bean FrequentCustomerBR that implements this rule (Figure 5) extends the BRPriceDiscount aspect bean as it is a rule for price personalization. In Figure 6, the FrequentCustomers class is introduced for holding the frequent customer information that is shared among the two business rules.

```
class FrequentCustomerBR extends BRPriceDiscount{

  public boolean discountCondition(Customer customer){
    return FrequentCustomers.isFrequentCustomer(customer);
  }
}
```

Figure 5: The FrequentCustomer business rule

```
class FrequentCustomers {

  private static Vector customers = new Vector();

  public static void addFrequentCustomer(Customer c){
    customers.add(c);
  }

  public static boolean isFrequentCustomer(Customer c) {
     return customers.contains(c);
  }

}
```

Figure 6: The FrequentCustomer-class

Both rules are logically related, because they specify business considerations about customer frequency. As a result, only one connector is defined to gather the concrete information about the deployment of both rules. Another advantage of having the deployment information in the same connector is that the order in which the application of the rules should be triggered can be controlled by explicitly invoking the application of the rules in the desired order (lines 10 to 11). Figure 7 illustrates the implementation of the connector for the deployment of both rules.

```
1   connector FrequentCustomerDiscountDeployment{
2
3     FrequentCustomerBR.BRPriceDiscountHook discount =
4       new FrequentCustomerBR.BRPriceDiscountHook(Float CheckOut.CheckOut(Customer));
5     discount.setDiscount(0.05);
6
7     FrequentCustomer.FrequentCustomerHook frequent =
8       new FrequentCustomerBR.FrequentCustomerHook(Float CheckOut.CheckOut(Customer));
9
10    discount.replace();
11    frequent.after();
12  }
```

Figure 7: The deployment of the Frequent Customer business rule

4.2 Precedence and Combination Strategies

When several business rules are deployed within a single software system, it is possible that these rules influence each other's execution. This problem is a well-known issue in AOP, and is identified as the *feature interaction problem* [26]. To solve this problem, the JAsCo language provides a powerful, reusable and extensive system for specifying the precedence and the combination of aspects.

```
connector ChristmasFrequentCustomerDiscountDeployment {

  ChristmasBR.BRPriceDiscountHook christmasDiscount =
    new ChristmasBR.BRPriceDiscountHook(Float CheckOut.CheckOut(Customer));
  christmasDiscount.setDiscount2(0.10);

  FrequentCustomerBR.BRPriceDiscountHook freqDiscount=
    new FrequentCustomerBR.BRPriceDiscountHook(Float CheckOut.CheckOut(Customer));
  freqDiscount.setDiscount(0.05);

  FrequentCustomerBR.FrequentCustomerHook freqChecker=
    new FrequentCustomerBR.FrequentCustomerHook(Float CheckOut.CheckOut(Customer));

  christmasDiscount.replace();
  freqDiscount.replace();
  freqChecker.after();

}
```

Figure 8: Additive deployment of discount business rules

4.2.1 Precedence Strategies

The JAsCo language allows arranging the execution of a set of business rules, by explicitly specifying the desired sequence in the connector. Whenever two or more hooks interfere, the order in which their behaviour must be executed is derived from the connector. Figure 8 illustrates the deployment of a business strategy where the Christmas discount is given prior to the frequent customer discount (lines 14 to 16).

4.2.2 Combination Strategies

Being able to specify the sequence in which the various business rules are executed is in many cases not sufficient. Some business strategies require more advanced techniques to specify the combination of the various business rules that are deployed within the system. In the previous section for instance, an additive discount strategy is employed. However, the business policy could specify that only one discount is offered for a given product: if somebody buys an item during the Christmas period, the frequent customer discount is not applicable. The JAsCo language provides a solution to be able to specify this kind of advanced aspect-combinations, by providing a mechanism called combination strategies. A combination strategy acts like a kind of filter that validates the list of applicable hooks, which are obtained at run-time. Each specific combination strategy implements the `CombinationStrategy`-interface introduced in Figure 9. The interface itself only specifies the `validateCombinations`-method, which is used to describe the specific logic of a combination strategy. This mechanism of combination strategies allows maximum flexibility, as user-defined relationships between the various aspects can be implemented.

```
public interface CombinationStrategy {

  public HookList validateCombinations(HookList aHookList);

}
```

Figure 9: The CombinationStrategy interface

The exclude combination strategy illustrated in Figure 10 specifies a combination strategy where the behaviour of hook *B* cannot be executed whenever hook *A* is encountered. This combination strategy can be used to specify the relationship between the Christmas and the frequent customer discount business rules.

```
class ExcludeCombinationStrategy implements CombinationStrategy {

  private Object A;
  private Object B;

  public ExcludeCombinationStrategy(Object hookA,hookB){
    A = hookA;
    B = hookB;
  }

  public HookList validateCombinations(HookList aHookList){
    if (aHookList.contains(A))
      aHookList.remove(B);
    return aHookList;
  }
}
```

Figure 10: The Exclude CombinationStrategy

The connector illustrated in Figure 11 deploys the Christmas and the frequent customer discount business rule. Both business rules are initialized with a specific context, and the execution of their behaviour methods is specified. The connector however also specifies an exclude combination strategy between both business rules (lines 18 to 19). As a result, whenever the Christmas discount is applied, the behaviour of the frequent customer business rule is ignored.

```
connector ChristmasFrequentCustomerDiscountDeployment {

  ChristmasBR.BRPriceDiscountHook christmasDiscount =
    new ChristmasBR.BRPriceDiscountHook(Float CheckOut.CheckOut(Customer));
  christmasDiscount.setDiscount2(0.10);

  FrequentCustomerBR.BRPriceDiscountHook freqDiscount=
    new FrequentCustomerBR.BRPriceDiscountHook(Float CheckOut.CheckOut(Customer));
  freqDiscount.setDiscount(0.05);

  FrequentCustomerBR.FrequentCustomerHook freqChecker=
    new FrequentCustomerBR.FrequentCustomerHook(Float CheckOut.CheckOut(Customer));

  christmasDiscount.replace();
  freqDiscount.replace();
  freqChecker.after();

  ExcludeCombinationStrategy strategy = new ExcludeCombinationStrategy(chDiscount,frDiscount);
  addCombinationStrategy(strategy);
}
```

Figure 11: Exclusive deployment of discount business rules

4.3 Controlled Instantiation, Initialisation and Execution of Aspects

Most aspect-oriented technologies do not allow sophisticated control for instantiating, initializing and executing aspects, as this is done implicitly when the aspect is woven into the core of the base application. The JAsCo system improves upon these techniques, as the instantiation of an aspect with a specific context is described explicitly in the connector. As a result, every instantiated aspect can be accessed as being a first class entity. This allows initializing each aspect instance with some specific properties. Considering the business rules environment, this is a vital contribution, as it allows fine-tuning general-purpose business rules to conform to the specific business requirements. Also, the execution of the behaviour of the business rules is specified explicitly in the connector, allowing even more fine-grained control.

4.4 Dynamic Reconfiguration of Business rules

Business rules tend to evolve continuously in comparison to the core functionality of the system. Some business rules, such as the Christmas discount rule introduced in Section 3, are only obligatory during a certain period of the year. Other business rules need to be adapted regularly to be able to fulfil new business requirements. Consequently, it should be possible to add, edit and remove business rules at run-time. Current AOP technologies however, do not allow easy management of business rules, as the deployment of an aspect within the system is rather static. This is mainly because an aspect loses its identity when it is woven into the base-application. JAsCo solves this issue, by also providing a run-time separation between the aspects and the base implementation of the system. This way, JAsCo aspects remain first class entities when they are deployed and their logic is not weld together with the base functionality of the application. This property of the JAsCo system is a valuable concept in the business rules environment, as this run-time separation, together with the new component model, allows dynamic reconfiguration of business rules, without the need to shut down business-critical applications.

5. RELATED WORK

Our work is an original combination of two areas, more specifically aspect-oriented programming and separating rule-based knowledge in object-oriented software applications. We have contributed previously to this line of research. First of all, in [6] and [7] we discuss a similar experiment with AspectJ, whereas in [8] we review different features of several existing AOP approaches for addressing the issues of linking business rules. Secondly, we developed aspect-oriented techniques for encapsulating the rule integration code in the context of *hybrid systems*, which combine an object-oriented language and a full-fledged rule-based language for representing rules [14] and [15]. These aspect-oriented techniques are based on HyperJ and AspectJ. In these papers we do not consider advanced aspect-oriented features as described in this paper, but we have to deal with the additional challenge of combining two languages of different programming paradigms. To our knowledge, there have been no other efforts that apply aspect-oriented programming to improve the separation of rules from object-oriented software.

Furthermore, we consider related work in the areas separately. First of all, the main body of this paper provides a thorough overview of the relevant work in the field of aspect-oriented programming. Secondly, there exist many technologies that represent rules explicitly and separately from core functionality in object-oriented software applications. We observe that they take radically different approaches:

- ✓ Rule-based knowledge can be represented separately in the object-oriented programming language itself. An extension to this approach is representing rule-based knowledge explicitly using object-oriented design patterns, such as the *Rule Object Pattern* [1], *Patterns for Personalisation* [27] and *Rule Patterns* [20].
- ✓ Other approaches focus on *externalising* explicit rules, such as *Business Rule Beans*, which store rules as XML fragments [29].
- ✓ There are dozens of both commercial and academic hybrid systems, which support explicit and separate representation of rules in a rule-based language. Due to space limitation we do not list them here. The results of a survey of hybrid systems are presented in [14]. A few examples are *OPSJ* [16], *JRules* [19], *SOUL* [24] and *Jinni* [18].

However, since none of these approaches support the encapsulation of tangled or crosscutting code, we find that they are not able to separate the rule integration code fully.

Note that we do not consider information systems, although some database management systems offer support for business rules. The reason is that they implement a *data-change-oriented* approach, activating rules when data changes. However, when rules are not bound to a particular object or data but are "free-floating", a *service-oriented* approach is warranted [34]. Moreover, even C. J. Date states that not all rules can be implemented in the database layer, but have to be considered in the application layer [11].

Aspect-Oriented approaches are originally conceived with low-level implementation aspects in mind, such as synchronization, error handling and logging. However, Tarr et al. also apply their idea of *Multi-Dimensional Separation of Concerns* and their tool HyperJ [25][33] to other kinds of concerns like business rules. In their approach, business rules can be encapsulated in different hyperslices, which are their modularization mechanism for crosscutting concerns. Hyperslices are loosely coupled with the base model, which implies that the business rules they encapsulate are reusable in different contexts. In this approach it is possible to specify a separate module (hypermodule) to encapsulate the details of how the business rules are linked to the core application. However, not much support for hyperslice relations is provided, limiting the combination of business rules. Moreover, mapping concerns is done statically, by matching structural units present in different hyperslices. This characteristic does not allow the connection of business rules to core application events that depend on run-time properties, one of the desired requirements we pursue.

6. CONCLUSIONS

The main goal of this research consists of realizing independent, reusable and manageable business rules at the implementation level of object-oriented software applications. In order to achieve this we propose to use aspect-oriented ideas to link the business rules to the core application. In a previous attempt, we used AspectJ as a concrete AOP technology and identified several problems. In this paper, we show that JAsCo is able to improve on AspectJ for representing business rules on several essential points. First of all, JAsCo allows specifying reusable business rules that can be instantiated to fit the application at hand. Secondly, the connector concept of JAsCo allows controlling the instantiation and initialization of the business rules. An additional advantage of the connector is that it allows specifying and managing more advanced and fine-grained business rule combinations than in AspectJ. Last but not least, JAsCo allows run-time application and removal of business rules which is an essential property in this context. On the other hand, some considerations need to be taken into account. JAsCo is a rather new AOP language whereas AspectJ is already mature and applied to large industrial case studies. In addition, AspectJ offers more advanced join point expressions than JAsCo.

This paper takes an important step in bridging the gap between business rule specification and implementation. The use of AOP and in particular JAsCo enables us to maintain the modularity conceived at the conceptual level to the implementation level. However, in this work the representation of the business rules themselves changes from a conceptual *if-then* format to objects. The reason is that this allows us to concentrate on the business rule link and it also facilitates the use of existing AOP approaches such as AspectJ and JAsCo since they extend Java. A continuation of this work is to consider a more suitable representation for business rules, such as a rule-based language, in order to minimize transition from specification to implementation even more.

7. REFERENCES

[1] A. Arsanjani, Rule object 2001: A pattern language for adaptive and scalable business rule construction.

[2] A. Arsanjani and J. Alpigini, Using grammar-oriented object design to seamlessly map business models to component-based software architectures, Int. Symposium of Modelling and Simulation, Pittsburgh, USA, pp. 186-191, 2001

[3] Aspect-Oriented Software Development. http://www.aosd.net/

[4] L. Bergmans and M. Aksit, Composing Crosscutting Concerns Using Composition Filters, Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001

[5] The Business Rules Group, Defining Business Rules: What Are They Really?, http://www.businessrulesgroup.org/, July 2000

[6] M. A. Cibrán, Using Aspect-Oriented Programming for Connecting and Configuring Decoupled Business Rules in Object-Oriented Applications, Master Thesis, Vrije Universiteit Brussel, Belgium, 2002

[7] M. A. Cibrán, M. D'Hondt and V. Jonckers, Aspect-Oriented Programming for Connecting Business Rules, Proceedings of the 6th International Conference on Business Information Systems, Colorado Springs, USA, 2003

[8] M. A. Cibrán, D. Suvee, M. D'Hondt, W. Vanderperren and V. Jonckers, Integrating Rules with Object-Oriented Software Applications using Aspect-Oriented Programming, Proceedings of the Argentine Symposium on Software (ASSE 2004). Córdoba, Argentina (to appear)

[9] Communications of the ACM, Aspect-Oriented Software Development, October 2001

[10] Communications of the ACM, The Adaptive Web, June 2002

[11] C. Date, What not How: The Business Rules Approach to Application Development, Addison-Wesley, 2000

[12] M. D'Hondt, W. De Meuter and R. Wuyts, Using Reflective Logic Programming to Describe Domain Knowledge as an Aspect, Proceedings of the first Symposium on Generative and Component-Based Software Engineering, Erfurt, Germany, 1999

[13] M. D'Hondt and T. D'Hondt, Is Domain Knowledge an Aspect?, 13th European Conference on Object-Oriented Programming, Workshop on Aspect-Oriented Programming, Lisbon, Portugal, 1999.

[14] M. D'Hond, K. Gybels and V. Jonckers, Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis, Proceedings of the ACM Symposium of Applied Computing, Nicosia, Cyprus, 2004

[15] M. D'Hondt and V. Jonckers, Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge. Proceedings of the third International Conference on Aspect-Oriented Software Development, ACM, March 2004

[16] C. L. Forgy, OPSJ 4.1 Manual, Production Systems Technologies Inc., 2001

[17] B. N. Grosof, Y. Labrou and H. Y. Chan, A declarative approach to business rules in contracts: courteous logic programs in XML, Proceedings of first ACM conf. on Electronic commerce, pp. 68-77, ACM Press, 1999

[18] Jinni, Jinni 2004 Prolog Compiler: A High Performance Java and .NET-based Prolog for Object and Agent-Oriented Internet Programming, User Guide. BinNet Corp., 2003

[19] JRules, JRules 4.0 Technical White Paper, ILOG, 2002

[20] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen, From rules to rule patterns, International Conference on Advanced Information Systems Engineering, pp. 99-115, 1996

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm and W. G. Griswold, An overview of AspectJ, Proceedings of 15th European Conference on Object-Oriented Programming, volume 2072 of Lecture Notes in Computer Science, pp. 327-353, Berlin, Heidelberg and New York, Springer-Verlag

[22] K. Lieberherr, D. Lorenz and M. Mezini, Programming with Aspectual Components, Technical Report, NU-CCS-99-01, March 1999

[23] K. Lieberherr, D. Orleans, J. Ovlinger, Aspect-oriented programming with adaptive methods, Communications of the ACM, Vol. 44, No. 10, pp. 39-41, October 2001

[24] K. Mens, I. Michiels and R. Wuyts, Supporting Software Development through Declaratively Codified Programming Patterns, Proceedings of SEKE, Buenos Aires, Argentina, 2001

[25] H. Ossher and P. Tarr, Using multidimensional separation of concerns to (re)shape evolving software, Communications of the ACM, 44(10):43-50, October 2001

[26] E. Pulvermueller, A. Speck, M. D'Hondt, W. De Meuter and J. O. Coplien, Report from the ECOOP'2001 Workshop on Feature Interaction in Composed Systems, Springer-Verlag

[27] G. Rossi, A. Fortier, J. Cappi and D. Schwabe, Seamless personalization of e-commerce applications, 2nd International Workshop on Conceptual Modeling Approaches for e-Business, 20th International Conference on Conceptual Modeling, Yokohama, Japan, 2001
[28] G. Rossi, D. Schwabe and R. Guimaraes, Designing personalized web applications, World Wide Web, pp. 275-284, 2001
[29] I. Rouvellou, L. Degenaro, K. Rasmus, D. Ehnebuske and B. McKee, Extending business objects with business rules, TOOLS Europe 2000, St-Malo, France, pp. 238-249, 2000
[30] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. Van de Velde and Wielinga B., Knowledge Engineering and Management: The CommonKADS Methodology, MIT Press, 2000
[31] C. Szyperski, Component software: Beyond Object-oriented programming. Addison-Wesley, 1998
[32] D. Suvée and W. Vanderperren, JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development, Proceedings of 2nd International Conference on Aspect-Oriented Software Development, Boston, USA, 2003
[33] P. Tarr, H. Ossher, W. Harrison and S. Sutton, N degrees of separation: Multi-dimensional separation of concerns, ICSE 1999, pp. 107-119, IEEE Computer Society Press / ACM Press
[34] B. von Halle. Business Rules Applied. Wiley, 2001