Stateful Aspects in JAsCo

Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
{wvdperre, dsuvee, mcibran, bdefrain}@vub.ac.be
http://ssel.vub.ac.be

Abstract. Aspects that trigger on a sequence of join points instead of on a single join point are not explicitly supported in current Aspect-Oriented approaches. Explicit protocols are however frequently employed in Component-Based Software Development and business processes and are as such valid targets for aspect application. In this paper, we propose an extension of the JAsCo aspect-oriented programming language for declaratively specifying a protocol fragment. The proposed pointcut language is equivalent to a finite state machine. Advices can be attached to every transition specified in the pointcut protocol. Furthermore, the complement of a protocol can also be used for triggering aspects. The JAsCo tools support the stateful aspects language and implement it very efficiently by employing the JAsCo run-time weaver. As a validation of the approach, we present a case study in the context of reaction business rules.

1 Introduction

Aspect-Oriented Programming (AOP) [15] is a recent software programming paradigm that aims at providing a better separation of concerns. At its root is the observation that some concerns cannot be cleanly modularized using traditional abstraction mechanisms such as class hierarchies. These so-called *crosscutting* concerns will therefore inevitably appear scattered across different modules of the system, making them difficult to comprehend and maintain. Typical examples of such concerns are tracing, synchronization and transaction management.

In order to enable a clean modularization of crosscutting concerns, AOP techniques such as AspectJ [16] introduce the concept of an *aspect*, in addition to the use of regular classes. An aspect defines a set of *join points* in the target application where *advices* alter the regular execution. The set of joint points is declaratively specified through a *pointcut*. The aspect logic is then automatically *woven* into the target application.

Although AOP research originally focused on a model where aspects are invoked on static locations in the compile-time structure of the program, it was early on argued that the applicability of certain so-called *jumping* aspects [3] can only be expressed in terms of dynamic conditions. Most of the current approaches therefore feature a dynamic join point model, i.e. a model where the join points are run-time events of the program execution. As such, it becomes possible to invoke aspect behavior based on run-time types, call-stack context (e.g. AspectJ's `cflow()` pointcut), dynamically evaluated expressions,...

Describing the applicability of aspects in terms of a *sequence* or *protocol* of run-time events however, is generally not supported. With the exception of the `cflow()` pointcut, the pointcuts of current mainstream AOP languages cannot refer to the history of previously matched pointcuts in their specification. In order to trigger an aspect on a protocol sequence of join points, one is obliged to program code for maintaining a state regarding the occurrence of relevant join points, as such implementing the protocol by hand. Not only is this a cumbersome task, but it is also conceptually undesirable, because it involves mixing the aspect-applicability control-mechanism with the advice code itself.

Explicit protocols are nevertheless frequently encountered in a wide range of fields such as Component-Based Software Development [25,10], data communications [19] and business processes [1]. We therefore believe that protocols are valid targets for aspect application, and argue that it is desirable to support them in the pointcut language itself; delegating the actual control-mechanism implementation to the weaver. This paper proposes an extension of the JAsCo [18] programming language for *stateful aspects* that can declaratively specify a protocol of expected pointcuts.

The paper is structured as follows. Section 2 introduces the JAsCo language and illustrates the need for explicit support of protocols. The JAsCo extension for supporting stateful aspects is discussed in section 3. Section 4 focuses on the implementation details of our approach, while section 5 validates it by presenting a case study. Finally, we discuss related work in section 6 and end up with conclusions in section 7.

2 Introduction to JAsCo

The JAsCo [18] AOP approach is an aspect-oriented extension for Java that allows for a clean modularization of crosscutting concerns. The JAsCo language stays as close as possible to the original Java syntax and concepts and introduces two new entities, namely *Aspect Beans* and *Connectors*. An aspect bean is an extended version of a regular Java Bean that allows describing crosscutting concerns independently of concrete component types and APIs. JAsCo connectors on the other hand are used for deploying one or more reusable aspect beans within a concrete component context and provide support for describing their mutual interactions.

A typical example of a crosscutting concern is the run-time checking of timing contracts [21]. Instead of inserting the logic behind these contracts at various places within the base code, one can modularize this behavior into a single entity by employing a JAsCo aspect bean. Figure 1 illustrates the implementation of the dynamic timer aspect bean.

Typically, an aspect bean contains one or more hook definitions that implement the crosscutting behavior and usually a number of ordinary Java class members which are shared among all hooks. The `DynamicTimer` aspect bean of figure 1 describes a `TimeStamp` hook (lines 18-29) and the notification system for its listeners (lines 3-16). The `TimeStamp` hook is responsible for capturing

```
1  class DynamicTimer {
2
3    private Vector<TimeListener> listeners = new Vector<TimeListener>();
4    private long timestampbefore, timestampafter;
5
6    void addTimeListener(TimeListener aListener) {
7      listeners.add(aListener);
8    }
9    void removeTimeListener(TimeListener aListener) {
10     listeners.remove(aListener);
11   }
12   void notifyTimeListeners(Method method, long time) {
13      for (TimeListener listener : listeners) {
14        listener.timeStampTaken(method,time);
15      }
16   }
17
18   hook TimeStamp {
19     TimeStamp(timedmethod(..args)) {
20       execute(timedmethod);
21     }
22     before {
23       timestampbefore = System.currentTimeMillis();
24     }
25     after {
26       timestampafter = System.currentTimeMillis();
27       notifyListeners(thisJointPoint,timestampafter-timestampbefore);
28     }
29   }
30 }
```

Fig. 1. The JAsCo-aspect for dynamic timing

a timestamp and notifying its listeners whenever some functionality of a component is executed. To this end, the `TimeStamp` hook describes a constructor (lines 19-21), which specifies in an abstract way when the normal execution of a method should be interrupted in order to trigger the aspect behavior. Each constructor receives several abstract method parameters as inputs which are bound to one or more concrete method signatures whenever the hook is explicitly deployed using a connector. The constructor body (line 20) outlines when the behavior of the hook should be triggered. In case of the `TimeStamp` hook, the crosscutting behavior is performed whenever one of the methods bound to the abstract method parameter `timedmethod` is executed. The advices of a hook, namely `before`, `replace`, `after`, `after throwing` and `after returning`, are employed for specifying the various actions a hook needs to perform whenever its behavior is triggered. The implementation of an advice is able to refer to the arguments of the abstract method parameters and accesses the currently visited joinpoint and its reflective information by employing the `thisJoinPoint` keyword. In the aspect bean of figure 1, the before advice describes that a timestamp should be taken prior to the execution of `timedmethod` (lines 22-24). The after advice calculates the time that was required to execute the `timedmethod` method and announces it to all registered listeners (lines 25-28)[1].

[1] Notice that the aspect bean implements a very simplistic timestamping mechanism in order to keep the example simple and easy to understand. The provided implementation is not thread-safe and does not work for recursive methods.

```
1  connector TimeConnector {
2    DynamicTimer.TimeStamp timer =
3      new DynamicTimer.TimeStamp (void ComponentX.a());
4    timer.before();
5    timer.after();
6  }
```

Fig. 2. The JAsCo connector for dynamic timing

Abstract and reusable aspect beans are deployed onto a concrete component context by making use of connectors. Each connector allows to explicitly instantiate and initialize one or more logically related hooks. Figure 2 illustrates a connector that instantiates the `TimeStamp` hook of Figure 1 onto the `a()` method of the `ComponentX` component (lines 2-3). As a result, the abstract method parameter `timedmethod` of the `TimeStamp` hook constructor is bound to this given method. Additionally, it is specified that the `before` and `after` advices need to be executed whenever a join point of the newly instantiated hook `timer` is encountered (lines 4-5). To sum up, this connector specifies that whenever the `a()` method of the `ComponentX` component is executed, a timestamp is taken and the corresponding listeners are notified afterwards in order to verify whether the `a()` method satisfies the specified timing contracts.

The `DynamicTimer` aspect bean is suitable to perform time contract verification on a single method. It can however not be reused to verify timing contracts on a full component protocol. Imagine one wants to check whether a particular component protocol `methodA-methodB-methodC` is executed within a predefined time period. These methods can occur in any sequence, so it does not suffice to just add a before advice to methodA and an after advice to methodC in order to time this protocol. One has to explicitly keep track of the protocol. The `ProtocolDynamicTimer` aspect bean, illustrated in figure 3, presents an ad-hoc solution to implement the time contract verification of this particular protocol. It extends the basic `DynamicTimer` aspect bean, this way inheriting the listener notification system. For each step in the protocol, a dedicated hook is introduced, which implements the associated actions. When `methodA` of hook `ProtocolMethodA` is executed (lines 7-9), a timestamp is taken (line 11) and the boolean value `methodAExecuted` associated with that step of the protocol is set (line 12). This `before` advice is only performed the very first time protocol `methodA-methodB-methodC` is executed. This behavior is enforced through the `isApplicable` method (line 14). This additional JAsCo language construct is a method that allows to describe a run-time condition which ensures that the advices of an aspect bean are only performed when its body evaluates to true (similar to the `if` pointcut designator in AspectJ). By employing the `isApplicable` method, the advices associated with hook `ProtocolMethodC` will not be executed as long as the boolean value associated with `ProtocolMethodB` is not set to true (line 36). As such, the three hooks defined within the `ProtocolDynamicTimer` aspect bean ensure that the associated listeners are only notified when a full `methodA-methodB-methodC` protocol is encountered.

Although the aspect bean illustrated in Figure 3 can be used to verify whether the particular component protocol is performed within a specified time period,

```
class ProtocolDynamicTimer extends DynamicTimer {

  boolean methodaexecuted, methodbexecuted = false;

  hook ProtocolMethodA {
    ProtocolMethodA(methodA(..args)) {
      execute(methodA);
    }
    before {
      timestampbefore = System.currentTimeMillis();
      methodaexecuted = true;
    }
    isApplicable() { return !methodaexecuted; }
  }

  hook ProtocolMethodB {
    ProtocolMethodB(methodB(..args)) {
      execute(methodB);
    }
    before {
      methodbexecuted = true;
    }
    isApplicable() { return methodaexecuted; }
  }

  hook ProtocolMethodC {
    ProtocolMethodC(methodC(..args)) {
      execute(methodC);
    }
    after {
      timestampafter = System.currentTimeMillis();
      notifyListeners(method,timestampafter-timestampbefore);
      methodaexecuted = false; methodbexecuted = false;
    }
    isApplicable() { return methodbexecuted; }
  }

}
```

Fig. 3. The JAsCo-aspect for dynamic timing of a component protocol

it requires to explicitly capture each possible state of the protocol in a separate hook. As such, one is obliged to describe and implement the full protocol by hand, as a protocol sequence of join points is not explicitly supported using regular JAsCo and other state-of-the-art aspect-oriented approaches. This also involves tangling the description of the applicability of the aspect with its behavior. In the next section, an extension to the JAsCo language is proposed, which allows to declaratively specify a protocol of expected pointcuts. Advices can be attached to each step of the pointcut protocol, allowing to describe the time contract verification of a component protocol in a more declarative way.

3 Stateful Aspects Language

Mainstream aspect-oriented approaches rarely support protocol history conditions. In many cases, it is only possible to refer to previous join points when they still have an activation record on the stack (i.e. using the `cflow()` keyword in AspectJ). In order solve this limitation, Douence et al. [6,7] propose a formal model for aspects with general protocol based triggering conditions, named *stateful aspects*. In this section, we illustrate how the JAsCo language is extended with stateful pointcut expressions, based on this formal model.

```
1  class ProtocolDynamicTimer extends DynamicTimer {
2
3    hook StatefulProtocolTimer {
4
5      long timestamp;
6
7      StatefulProtocolTimer(methodA(..args),methodB(..args),methodC(..args)) {
8        ATrans: execute(methodA) > BTrans;
9        BTrans: execute(methodB) > CTrans;
10       CTrans: execute(methodC) > ATrans;
11     }
12
13     before ATrans() {
14       timestamp=System.currentTimeMillis();
15     }
16     after CTrans() {
17       long resultingtime = System.currentTimeMillis();
18       notifyListeners(calledmethod,resultingtime-timestamp);
19     }
20
21   }
22 }
```

Fig. 4. The JAsCo stateful aspect for dynamically checking a timing contract of a component protocol

In figure 3, an ad-hoc solution was presented for implementing time contract verification of a protocol `methodA-methodB-methodC`. Figure 4 illustrates how the same protocol can be declaratively described by making use of the JAsCo stateful aspect language. The constructor of the hook `StatefulProtocolTimer` (line 7-11) describes a protocol-based pointcut expression. Every line in the constructor defines a new transition within the protocol. Each transition is labeled with a name (e.g. `ATrans`), defines a JAsCo compatible pointcut expression (e.g. `execute(methodA)`) and specifies one or more destination transitions that are matched after the current transition is fired. A transition fires when its pointcut expression evaluates to true. For example, the `ATrans` transition only fires whenever the concrete method(s) bound to the abstract method parameter `methodA` are executed. In that case, transition `BTrans` is activated and will be evaluated for the subsequent join points encountered in the application.

A stateful aspect always starts by evaluating the first defined transition. As a result, a protocol `methodA-methodB-methodC` is described. In between the fired transitions, other join points can also be encountered. As such, a sequence of events `methodA-methodX-methodC-methodB-methodC` is also a valid instance for the defined protocol and will trigger the associated transitions. Notice that the JAsCo stateful aspect pointcut does not have to specify the full protocol of the application; a protocol fragment is sufficient.

On every transition defined in the stateful constructor, advices can be attached which are executed whenever the transition is fired. For example, the `before ATrans` advice (line 13-15) is only triggered whenever the transition

`ATrans` is fired. In other words, the advice is executed whenever the concrete method(s) bound to the abstract method parameter `methodA` are executed in that state of the stateful aspect. To sum up, the `StatefulProtocolTimer` hook will take a timestamp before the protocol `methodA-methodB-methodC` is executed and will notify all interested listeners after the full protocol is performed.

3.1 Advanced Language Features

In addition to attaching advices on each transition separately, it is also possible to describe global advices that are triggered for all fired transitions. In this case, the advice is specified as usual, but the transition label is omitted. It is also possible to attach a specific `isApplicable` method to a particular transition in the protocol. As such, the transition will only be fired when both the pointcut expression and the `isApplicable` condition evaluate to true. Likewise to advices, a global `isApplicable` condition can be specified which is applied to all transitions. In that case, transitions are only fired when they satisfy their pointcut expression and both the global and local `isApplicable` conditions. The following code fragment shows both a global and local `isApplicable` condition.

```
isApplicable() { //global condition for all transitions
  // returns true if advices should be executed
}
isApplicable XTrans() { //local condition only relevant for the transition XTrans
  // returns true if advices should be executed for the XTrans transition
}
```

The JAsCo stateful aspects constructor can also specify multiple destination transitions for a given transition. The syntax is illustrated in the code fragment below. After firing the `XTrans` transition, both the `YTrans` and `QTrans` transitions are evaluated for subsequent encountered join points (line 2). Note that the destination transitions are evaluated in the sequence defined in the destination expression. As such, when both the `YTrans` and `QTrans` transitions are applicable for a given join point, only the `YTrans` transition will be fired and only the `YTrans` destination transitions will be evaluated for subsequent encountered join points. This allows to keep the protocol deterministic and efficient to execute. It is also possible to omit a destination transition for a certain transition. In that case, when the transition fires, no more transitions need to be evaluated and the aspect *dies*. This concept is illustrated by the `QTrans` transition (line 3). Also notice that this transition describes a more involved pointcut designator using the `cflow` keyword.

In case the stateful aspect requires to start by evaluating more than one transition, the `start` keyword can be employed. This keyword is followed by a list of starting transitions for matching join points when the aspect is deployed. Multiple start transitions are specified similarly to multiple destination transitions, by using `||` as delimiters. When no start transition is specified, the first defined transition is used as the starting one.

```
1  start > XTrans || QTrans; //starting with two transitions
2  XTrans: execute(methodA) > YTrans || QTrans; //two destination transitions
3  QTrans: execute(methodB) && !cflow(methodC); //no destination transition
4  YTrans: execute(methodC) > YTrans;
```

The syntax proposed in the previous paragraphs provides a way for specifying powerful protocols but might be tedious in case of simple protocols. Therefore JAsCo also supports a simpler syntax for protocols that do not require multiple destination transitions for a given transition. The following code fragment shows a constructor that is equivalent to the constructor of figure 4. Labeling transitions is still possible in order to be able to attach local advices to specific transitions. Notice that the label `start` automatically refers to the first transition.

```
StatefulProtocolTimer(methodA(..args),methodB(..args),methodC(..args)) {
  ATrans: execute(methodA) > execute(methodB) > CTrans: execute(methodC) > start;
}
```

Normally, aspects are instantiated explicitly in a connector and this instance is used for all encountered join points. In case of protocol checking stateful aspects, it is sometimes desirable to have a unique instance of the stateful aspect for every execution thread in the application as every thread is typically related to a different interaction. JAsCo allows automatically instantiating multiple instances for a single hook instantiation expression by using specialized keywords in front of the instantiation expression in the connector. The following keywords are supported: `perobject`, `perclass`, `permethod`, `perall`, `percflow` and `perthread`. Thus, in order to obtain a unique aspect instance per execution thread, the `perthread` keyword can be used. The JAsCo run-time system will automatically manage the aspect instances for every thread. This is illustrated by the following code fragment:

```
static connector PerThreadConnector {
  perthread ProtocolDynamicTimer.StatefulProtocolTimer timer =
    new ProtocolDynamicTimer.StatefulProtocolTimer(void ComponentX.a(),
      void ComponentX.b(), void ComponentX.c());
}
```

3.2 Protocol Complement

The JAsCo stateful aspects language currently supports triggering aspects on a protocol fragment. However, the opposite, namely triggering aspects on every join point besides the defined protocol, can also be useful in many cases. For example, Farias et al. [10] identify that in the context of checking security policies of Enterprise Java Beans (EJBs), explicit protocols are necessary. The current EJB specification only allows to describe that particular methods need to adhere to a given security policy. It is however often required that only the protocol `methodA-methodB-methodC` on a given component X is allowed for certain users. Farias et al. propose a formal model to specify the allowed protocol of a component, based on finite state machines [12]. As already identified in literature [23], security concerns are typical examples of crosscutting concerns and thus good targets for AOP.

JAsCo supports triggering aspects on the opposite of a protocol using the `complement` keyword. Figure 5 illustrates a contract checking aspect that makes sure that all invocations on a certain component, besides the defined protocol, are blocked. The stateful aspect defines the same protocol as the one in

```
1  class ProtocolChecker {
2
3    hook StatefulProtocolCheck {
4
5      StatefulProtocolCheck(methodA(..arg),methodB(..arg),methodC(..arg),methodsContext(..arg)) {
6        complement[execute(methodsContext)]:
7          ATrans: execute(methodA) > BTrans;
8          BTrans: execute(methodB) > CTrans;
9          CTrans: execute(methodC) > ATrans;
10     }
11
12     replace complement() {
13       throw new SecurityException(
14         "This protocol on component"+thisJoinPoint.getClassName()+" is not allowed!");
15     }
16   }
17 }
```

Fig. 5. The JAsCo stateful aspect for checking a security contract

figure 4, namely a protocol `methodA-methodB-methodC`. The additional complement definition states that the aspect is interested in the complement of the protocol. The complement expression is also able to specify a JAsCo compatible pointcut designator in order to limit the complement to a certain set of join points. This is often required in the context of checking security contracts of a component. It is not very useful to trigger the aspect on every possible protocol fragment outside the given protocol, because this would also include methods on other components. Therefore, JAsCo allows to limit the set of join points to which the complement of the protocol is computed. For example, the `StatefulProtocolCheck`, illustrated in figure 5, defines that the complement is only triggered when methods bound to `methodsContext` (line 5) are executed and not exactly following the defined protocol. Advices can be attached to the complement of a protocol by specifying the `complement` keyword after the advice name. In this case, a `replace complement` advice is specified (lines 12-15) that replaces the original behavior and throws a security exception instead. Moreover, it is still possible to attach advices to transitions in the allowed protocol by using the syntax introduced before.

Figure 6 illustrates the connector that deploys the security protocol checking aspect bean onto the `a-b-c` protocol of the `componentX` component (lines 3-4). The `methodsContext` abstract method parameter of the aspect bean is bound to all methods of `componentX`. As a result, whenever a method is executed on `componentX` that falls outside the defined protocol `a-b-c`, like for example `a-b-d-c`, a security exception is thrown.

```
1  static connector SecurityConnector {
2
3    ProtocolChecker.StatefulProtocolCheck checker = new ProtocolChecker.StatefulProtocolCheck(
4      void componentX.a(), void componentX.b(), void componentX.c(), void componentX.*());
5  }
```

Fig. 6. Connector for deploying the security protocol contract checking aspect

4 Implementation Discussion

The JAsCo stateful aspects language is equivalent to a Deterministic Final Automaton (DFA) [12] because every expression defines one DFA transition, two DFA states and possibly several connection DFA transitions for the destinations. Therefore, the JAsCo compiler compiles a stateful aspect constructor to a DFA that is interpreted at run-time. Every transition of a DFA contains a representation of the pointcut definition and possibly an `isApplicable` condition. When a join point is encountered, the outgoing transitions of the current state are evaluated with the given join point and when a match is encountered, the state machine moves to the destination state. When this event occurs, all associated advices are executed. Because of this implementation strategy, a stateful aspect can be executed very efficiently. It suffices to check only the transitions of the current state, as JAsCo stateful aspect protocols are regular and can be interpreted by a regular DFA. When non-regular protocols are allowed, a history of all relevant encountered events should be maintained, which is very expensive.

A naive approach for integrating the stateful aspect would be weaving it at all possible join points defined within the protocol. However, this induces a performance overhead at all these join points, while the stateful aspect is only interested in a limited set of join points corresponding to the subsequent transitions that are to be evaluated. JAsCo is a dynamic AOP language and features a genuine run-time weaver that is able to weave and unweave aspects at run-time [20]. The run-time performance of JAsCo is even able to compete with AspectJ's, which is a statically woven AOP approach [13]. Because of this run-time weaver, the JAsCo stateful aspect language induces only a minimal performance overhead. The JAsCo run-time weaver only weaves the stateful aspect at those join points where the aspect is currently interested in. When a transition is fired, the weaver unweaves the aspect at the join points associated with the current transition and weaves it back in at the join points relevant for the subsequent transitions. As such, a real *jumping aspect* is realized. Notice that when the aspect *dies* because no subsequent transitions are defined, it is completely unwoven. As a result, no performance overhead for the aspect is endured any longer.

The weaving process itself does however also require a significant overhead. Therefore, when a given protocol is encountered many times in a short time interval, it might be more efficient to weave the aspect at all possible join points of the protocol instead of weaving and unweaving it on-the-fly. This can be configured in JAsCo by using the novel Java 1.5 annotations (meta-data). When the `@jasco.runtime.aspect.WeaveAll` annotation is supplied to the hook, as illustrated by the code fragment below, the run-time weaver weaves the aspect at all join points and never unweaves it unless the aspect itself is manually removed or dies.

```
@jasco.runtime.aspect.WeaveAll
hook StatefulHook { ...
```

A proof-of-concept implementation for the JAsCo stateful aspect extension is made available through the regular JAsCo distribution [14].

5 Case Study

To illustrate the usefulness of stateful aspects, an example of reaction business rules is presented. Business rules are volatile and tend to change faster than the core application functionality. Therefore, and due to the crosscutting nature of their integration with the core application [17,5,11], it is recommended to keep them decoupled from the rest of the application. In previous work [4], aspects were successfully employed for integrating business rules that are triggered at single events that denote dynamic points within the core application. In this case study however, business rules that depend on complex behavioral states of the system are considered.

The presented case study is based on an application used in the context of the ADAPSIS[2] (Adaptation of IP Services based on Profiles) project [24]. One of the research topics of this project is the development of an approach that allows to mine end-user profiles, building on existing component-based applications. The main problem is that these applications are generally not designed to allow end-user profile mining. As a concrete research artifact, an e-commerce application is developed that allows customers to buy different products online, such as books, music and movies. This application is an adaptive web application as it incorporates data mining strategies in order to analyze the purchasing behavior of the customers. The Data4s data mining engine is employed to react according to the customer's behavior in an intelligent way. AOP is used to capture all relevant data for the data mining engine and influencing the behavior of the original application depending on the mined information and the user profile. For gathering application events, it is often required to keep track of the history of the user's interactions with the web application. When regular aspects are employed, this historical information results hard-coded in the aspects themselves, making them difficult to comprehend and evolve. With the advent of JAsCo stateful aspects, this problem can be tackled in a more declarative way.

Consider for instance a set of business rules that define a categorization of the customers, depending on their susceptibility to promotions. These rules allow understanding the interests of the different customers to better accommodate to their needs and goals. Promotions can be viewed and purchased from different product-specific web pages as well as from the home page of the shop. Normally, different customers react to the advertisement of promotions in different ways. Some customers for instance tend to browse to the promotions as a first reaction when accessing the shop's home page. Others might prefer to browse the available books, music or videos first and later on decide whether to choose a promotion. The following business rules determine the different degrees of sensitivity to promotions:

1. *High promotion-prone customer*: The customer browses the promotions as his/her first action

[2] ADAPSIS is partly funded by the IWT, Flanders (Belgium), partners are the University of Brussels (VUB), Alcatel Belgium and Data4s Future Technologies.

```
1  class MediumPromotionProneCustomerAspect {
2
3    hook MediumPromotionProneCustomerHook {
4
5      MediumPromotionProneCustomerHook(
6        browseProducts(Category category),accessPromotions(CustomerID customer)) {
7          start > browseProdTrans;
8          browseProdTrans : execute(browseProducts) > browseProdTrans || browsePromTrans;
9          browsePromTrans : execute(accessPromotions);
10     }
11
12     after browsePromTrans () {
13       CustomerManager.classifyCustomer(customer, MediumPromotionProneCustomer);
14     }
15 }
```

Fig. 7. Medium promotion-prone customer business rule

2. *Medium promotion-prone customer*: The customer browses first either the books, the music or the videos and only then he/she accesses the promotions
3. *Low promotion-prone customer*: The customer first browses the books and the music and the videos and then he/she accesses the promotions
4. *Promotion-insensitive customer*: The customer never follows the promotions links

In this example a single action, such as the browsing of the promotions, cannot be analyzed in isolation. In order to classify a customer in different categories, it is required to know which actions the customer already performed. In order to keep track of the customer history and trigger the customer classification, stateful aspects are employed. Figure 7 illustrates a stateful aspect implementing the medium promotion-prone customer business rule. In this example, the customer starts by browsing a catalog of products (line 8) and checks out the promotions afterwards (line 9). Figure 8 illustrates how the medium promotion-prone customer business rule is deployed within the e-commerce application.

Stateful aspects can also be employed to implement more complex patterns of behavior such as the following action sequence which could be performed by a customer:

```
getPromotions > browseBooks > browseCDs > browseVideos > getPromotions
```

Without stateful aspects, it would not be straightforward to detect this path of execution. We can imagine making the distinction between the place in the system where the promotions and the products are retrieved. However, this change would not be sufficient to distinguish the contextual difference between the first

```
static connector MediumPromotionProneCustomerConnector{

  MediumPromotionProneCustomerHook hook1 =
    new MediumPromotionProneCustomerHook(
      * ProductManager.browse*(*), * PromotionManager.getPromotions(CustomerID));

}
```

Fig. 8. Connector for deploying the medium promotion-prone customer business rule

and second invocation of the `getPromotions` and would imply tangling of code to manually keep track of the states. Stateful aspects avoid this problem, allowing the expression of complex contextual scenarios in a natural and non-invasive way.

6 Related Work

Douence et al. propose a model for supporting stateful aspects [6,7,8] as a part of their formal aspect model. The advantage of this formal model is that it allows to automatically deduce possible malicious interactions between aspects. Furthermore, the model supports composition of stateful aspects using well-defined composition operators. A proof of concept implementation of this model is also available [9]. This implementation is however based on static program transformations and as such it requires to advice all possible join points defined within the protocol. The JAsCo implementation improves on this because only a subset of the join points are actually advised.

Walker et al. introduce *declarative event patterns* (DEPs) [22] as a means to specify protocols as patterns of multiple events. They augment AspectJ aspects with special DEP constructs (called *tracecuts*) that can be advised similarly to pointcuts. Their approach is based on context-free grammars, and involves a transformation of the DEP constructs into standard AspectJ aspects containing an event parser, similar to the transformation realized by parser generators in compiler technology. While DEPs can recognize properly nested events and thus possess an even higher degree of declarative expressibility than the JAsCo approach, they only provide for the ability to attach advice code to entire protocols. Separate transitions of the protocol cannot be advised, and several overlapping protocols (realized through several independent event parsers) would have to be employed to mimic this possibility of JAsCo. Furthermore, the fact that DEPs lose their identity in a preprocessing step that reduces them to standard aspects, rules out the possibility for optimizations by a weaver that analyzes the feasible transitions of the protocol. Also, there are some unresolved issues in the current implementation of DEPs regarding optimal conservation of relevant execution traces.

7 Conclusions

In this paper we introduce an extension of the JAsCo language that enables triggering aspects on a sequence of join points. The JAsCo stateful aspects extension allows to declaratively specify a regular protocol. Advices can be attached to each transition in the protocol. JAsCo also allows to trigger aspects on the complement of a protocol given a set of join points. Because of the declarative specification, the stateful aspect is easier to understand and evolve than a manual implementation using singular join points. In addition, the declarative specification allows to optimize the execution of the stateful aspect. By employing dynamic AOP, the stateful aspect behavior is only woven at those join points the aspect is currently interested in.

A limitation of the current approach is that JAsCo stateful aspects can only specify regular protocols. Protocols that require a non-regular language (like for example `n times A; B; n times A`, where `n` can be a different number in every occurance of the protocol), cannot be represented. The advantage of keeping the protocols regular is that they can be efficiently evaluated using a DFA. A naive implementation of a non-regular protocol would require to keep the complete history of all encountered join points in memory, which is not very practical. In literature, several domain-specific optimization techniques for interpreting non-regular languages have been proposed [2]. Extending the JAsCo stateful aspects language to non-regular protocols while still allowing an efficient implementation is subject for future work.

Acknowledgements

Davy Suvée and Bruno De Fraine are supported by a doctoral scholarship from the Institute for the promotion of Innovation by Science and Technology in Flanders in the Industry (IWT).

References

1. T. Andrews et al. Business Process Execution Language for Web Services Specification, May 2003. http://www.ibm.com/developerworks/library/ws-bpel/.
2. J. Aycock and N. Horspool. Schrodinger's token. *Software Practice and Experience*, 31(8), 2001.
3. J. Brichau, W. De Meuter, and K. De Volder. Jumping Aspects. In *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.
4. M. A. Cibrán, M. D'Hondt, and V. Jonckers. Aspect-Oriented Programming for Connecting Business Rules. In *Proceedings of BIS International Conference*, Colorado Springs, USA, June 2003.
5. C. Date. *What not How: The Business Rules Approach to Application Development.* Addison Wesley, 1st edition, 2000.
6. R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, Pittsburgh, USA, October 2002.
7. R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of the 3th International Conference on Aspect-Oriented Software Development*, Lancaster, UK, March 2004.
8. R. Douence, P. Fradet, and M. Südholt. Trace-based Aspects. *Aspect-Oriented Software Development*, September 2004.
9. R. Douence and M. Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.
10. A. Farias and M. Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *Distributed Objects and Applications 2002*, Irvine, USA, October 2002.
11. B. Von Halle. *Business Rules Applied.* Wiley, 1st edition, 2001.

12. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory.* Addison Wesley, 2st edition, 2001.
13. JAsCo. *JAsCo Run-Time Weaver.* http://ssel.vub.ac.be/jasco/documentation:ruw.
14. JAsCo. *JAsCo website.* http://ssel.vub.ac.be/jasco/.
15. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and G.W. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, Budapest, Hungary, June 2001.
16. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
17. R. G. Ross. *Principles of the Business Rule Approach.* Addison Wesley, 1st edition, 2003.
18. D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, USA, March 2003.
19. A. S. Tanenbaum. *Computer Networks.* Prentice Hall Professional Technical Reference, 4th edition, 2002.
20. W. Vanderperren and D. Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *Proceedings of Dynamic Aspects Workshop*, Lancaster, UK, March 2004.
21. W. Vanderperren, D. Suvée, and V. Jonckers. Combining AOSD and CBSD in PacoSuite through Invasive Composition Adapters and JAsCo. In *Proceedings of Node 2003 International Conference*, Erfurt, Germany, September 2003.
22. R.J. Walker and K. Viggers. Implementing Protocols via Declarative Event Patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, USA, November 2004.
23. B. De Win, B. Vanhaute, and B. De Decker. How aspect-oriented programming can help to build secure software. *Informatica*, 26(2), 2002.
24. B. Wydaeghe, W. Vanderperren, T. Pijpons, and F. Westerhuis. Adapsis: Adaptation of IP Services Based on Profiles. In *SSEL Technical Report*, May 2002.
25. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.