History-based aspect weaving for WS-BPEL using Padus

Mathieu Braem and Dimitri Gheysels
System and Software Engineering Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{mbraem,dgheysel}@vub.ac.be

Abstract

Workflow languages provide a natural way to express business processes, and therefore they are preferred over general-purpose programming languages to specify such processes. However, current workflow languages offer no support for separating crosscutting concerns which results in workflows that are hard to maintain and evolve. Recent research introduces aspect-oriented extensions to these languages, but many advanced features of aspect-oriented programming technology are still unavailable for workflow languages. In this paper we present the implementation of one such advanced feature: "stateful aspects". We introduce a high-level, logic-based pointcut language to express aspect activation depending on past and current state of the workflow execution. We propose a weaving strategy based on finite state automata in order to transparently weave history tracking code in the workflow. Our approach is implemented as an extension of the Padus AOP language for WS-BPEL.

1. Introduction

The Service Oriented Architecture (SOA) promotes flexible, transparent and maintainable distributed software systems by treating the collaborating software components as independent services. One of the success stories of SOAs are the web services [3]. Web services expose an interface to a software system on a network, by means of standardized protocols. These services work together and communicate information to each other to implement some process or task. The specification of these collaborations was originally expressed in general purpose languages, but it became clear that that dedicated workflow languages are more suited to express these compositions.

The Business Process Execution Language for Web Services (WS-BPEL) [4] is such a workflow language for web services, and can be considered as the de-facto standard for expressing web service compositions.

However, it has been shown that these compositions suffer from crosscutting concerns that are not easily separated from the main process description [5]. This problem manifests itself by code that occurs scattered in the process and entangled with the code of other concerns. This crosscutting adds complexity and makes it difficult to add, modify and remove such concerns.

An accepted solution to deal with this problem in object-oriented programming is by applying techniques from aspect-oriented programming (AOP) [12]. And while initial research on AOP has concentrated on this area of software development, research has already extended the application domain to that of web services [7, 9, 17]. Limiting the scope to aspect-oriented language extensions for WS-BPEL we find two notable approaches: AO4BPEL [7] and Padus [6]. Padus is an aspect-oriented extension to WS-BPEL that employs source code transformation and features a logic based pointcut language.

The support for advanced aspect-oriented techniques in these approaches is however quite basic. Some of the current research topics in AOSD are left untouched. One of these topics is that of "stateful aspects" [11, 15]. This technique allows the activation condition of aspects to be based on the current and past state of the program execution. This is an issue that is frequently encountered in business processes. Consider the example where some additional behavior must be invoked after a particular sequence of events has been executed. In order to match this sequence, the history of the program state must be traced. This is an extra concern in its own and without language support, will be tangled with the base process and the concern that depends on this history tracking.

In this paper we present our solution for expressing crosscutting concerns that depend on the history of the program state. We extend the aspect language Padus by introducing new predicates that describe protocols in the pointcut language. We also modify the weaving engine to generate code that traces the current state of the process execu-

tion.

The remainder of this paper is organized as follows. Section 2 further details stateful aspects and shows two approaches that implement this concept. Section 3 then shows the new language constructs introduced in the Padus language to support such a solution and describes its implementation in the Padus weaver. Some related work is presented in section 4. Finally, section 6 closes the paper with our conclusions and provides some directions for future work on this subject.

2. Stateful aspects

Aspects introduce new behavior in the execution of the program, when a certain condition in the base program is met. In the traditional sense, this means that when the execution flow of the program meets a certain point, described by the aspect, the additional behavior is to be executed. The expression that denotes which points in the base program are to be selected is called a *pointcut*. Those points that are exposed by the aspect model and can be described by pointcuts are called *joinpoints*.

However, these pointcuts are usually restricted to describing joinpoints and conditions that relate to the current action. There is also a technique that allow aspects to depend on the history of the program execution. Aspects that depend on this past state of the program execution are called stateful aspects. This technique is available in the JAsCo aspect language, and has been recently introduced in an extension to AspectJ, the so-called tracematches. Other approaches that allow state-based joinpoint selection are not discussed here (e.g. [13]).

2.1 Stateful aspects in JAsCo

JAsCo [14] aims to combine techniques from AOP and component-based software development. Since protocols are frequently encountered in these components, JAsCo has been extended to support stateful aspects [15]. The extension consists of a way to specify protocols as a pointcut description. Transitions in the protocol correspond to particular runtime events and are determined by a standard pointcut. When the pointcut is matched at run-time, the transition is fired. JAsCo allows the programmer to attach advice to all transitions in the protocol and the advice will be executed as soon as that part of the protocol has been reached.

Extra features that are supported by the stateful aspects in JAsCo are *strict* protocols, which only match if no other joinpoints can be matched between the activations of the transitions in the protocol. Protocols can also be defined in terms of the complement of a number of transitions. Which means that the advice will be executed, if the protocol is not matched. Thirdly, extra conditions can be set on the firing of a transition by means of the implementation of an `isApplicable` method.

2.2 Tracematches

Tracematches [1] is an extension of the abc framework [2] for defining stateful aspects. This extension uses regular expressions to define the protocol of a stateful aspect. When the regular expression is matched, advice is executed. It is however not possible to execute advice at the intermediate points of the protocol. One of the compelling features of tracematches is that it allows to keep traces based on variable bindings. For a single regular expression, different traces are kept depending on the different variable bindings. When a new joinpoint is encountered, only the trace that corresponds to variables of the joinpoint is updated. As such, tracematches allows several instances of each protocol depending on the variables captured by the protocol.

Furthermore, tracematches allows to define the symbols (i.e. joinpoints) that are relevant for matching the regular expression. Symbols that are not defined are simply ignored. This allows to define strict or non-strict aspects as is possible in JAsCo.

2.3 Example in workflow

To illustrate how this is relevant to workflow languages, consider the following example. An international travel agent has a head brach office that offers a service for obtaining information for traveling destinations and registering bookings. This service is implemented by means of a workflow that performs several steps. Various methods of transportation to the destination are looked up, books with information on the region are searched for, a weather service is queried to show what temperatures can be expected.

However, the use of all these external services cost the traveling agent quite a bit of resources and they want to urge their branches to not rely on this too much. They will charge for each service used if in the end, no travel package is registered with them. If the branches do book a travel package, they get a bonus which is to be settled later.

The accounting of the services used and how much is to be charged is not part of the main functionality of this workflow, and should be separated in an aspect. We can implement this with an aspect that selects all the invocations of the external services and adds a cost to the bill for the current branch, executing the service. When the travel package is eventually booked, this bill is waved and a bonus is registered. A way of describing this protocol is shown in a following section.

```
<pointcut name="invocations(Jp, Operation)"
          pointcut="invoking(Jp, 'booking',
                    'bookingPT', Operation)" />
```

Listing 1. Padus pointcut definition

3. Introducing stateful aspects for Padus

To implement stateful aspects for WS-BPEL, we extend the aspect language Padus. Padus is an aspect-oriented extension to WS-BPEL that employs static source code transformation as a weaving strategy. The weaver introduces the advice and the activation conditions, and this resulting process can be executed on a standard WS-BPEL execution engine. However, as a result on this, we cannot rely on the execution engine to provide information of the history of the process. More on the Padus weaver can be found in [6]. The following section further explains the basic language constructs of Padus, and we follow with the extension we propose to support stateful aspects.

3.1 The Padus aspect language

The joinpoint model of Padus exposes every activity in a WS-BPEL process. Those joinpoints vary from structural joinpoints such as for instance sequencing and looping activities, to behavioral joinpoints such as e.g. invoking activities and assigning activities.

Pointcuts that select these joinpoints are expressed in a logic-based language. Each type of joinpoint has a corresponding predicate. The parameters to these predicates can restrict the number of selected joinpoints, or can expose context information that is available at those points in the execution of the workflow. The code in listing 1 shows a named pointcut definition that selects all invocations of services on the partnerlink `booking` with the `bookingPT` port type. The name of the operation that is executed is made available in the `Operation` variable.

Padus also has a deployment construct, which details what aspects are to be woven with which processes and which aspects take precedence when multiple aspects can activate on the same joinpoint.

3.2 Language constructs

In order to be able to declaratively describe protocols in Padus, some new language constructs and keywords are needed. Our extension describes a protocol with a regular language.

Regular expressions are often used to define a regular language because it provides a concise way for writing the grammar of the language. Three basic operations and an alphabet of symbols is needed for writing regular expressions. The operations are: concatenation, union and Kleene's star. The set of defined pointcuts in the aspects can be considered as the alphabet of symbols. By writing down a regular expression with the standard operations and this alphabet, a particular protocol is defined. To keep the description of protocols in Padus as simple as possible, only these three basic operations are supported.

An extra predicate "sequence" is added to the pointcut language. The sequence predicate takes one parameter that describes a protocol by means of three other predicates. For concatenations "concat"; for unions "choice", and for the Kleene's star operation "loop". Each of these predicates again takes a regular expression denoting a (sub) protocol.

Listing 2 shows the definition of a pointcut that is based on the history of the process execution. The first three lines of this example define named pointcut definitions, which will be reused in the last line. This last pointcut describes a the actual protocol. Here we want to match on a number of activities that occur in order (`concat` keyword). First a login occurs, followed by a number of external service invocation (`loop` keyword), followed by the invocation of the booking service.

The advice language is not changed. It is still possible to attach advice to any joinpoint selected by the pointcut. In the example in listing 2, we can execute advice at `Jp1` and at `Jp2`. Because it is unknown at the intermediate joinpoints whether the full protocol will be matched, advice at these points is executed if the partial protocol matches up to this point.

3.3 Finite state automata

One of the key issues for implementing stateful aspects, is that the history of the execution of events has to be tracked. Each execution of an event can be seen as a change of the current state of the application. Matching a protocol during the execution of a process means that the right state changes occur in the right order. The aspect is executed, only when the protocol matches the history of state changes in the process.

A protocol is expressed by means of a regular expression. To implement protocol matching, we translate this regular expression to a finite state automaton (FSA). We do this, because with FSAs we can check efficiently whether a given string is generated by a particular regular language.

Informally, a FSA can be defined as graph in which each node represents a state, and of which the set of labeled transitions are defined to move from one state to another. There is also a designated start-state and a set of end-states. A transition is fired if its label matches with the first symbol of the string.

By reading a string symbol by symbol, and check if the transitions can be followed in such a way that the end state is

```
<pointcut name="externalService(Jp)" pointcut="invoking(Jp, Partnerlink, Porttype, Operation)" />
<pointcut name="doBooking(Jp)" pointcut="invoking(Jp, 'booking' , 'bookingPT', 'doBooking')" />
<pointcut name="externalNotBooking(Jp)" pointcut="externalService(Jp), not(doBooking(Jp))" />

<!-- advice can be attached to Jp1 and to Jp2 -->
<pointcut name="aProtocol(Jp1, Jp2)"
          pointcut="sequence(concat(login(Jp), loop(externalNotBooking(Jp1)), doBooking(Jp2)))" />
```

Listing 2. Protocol definition

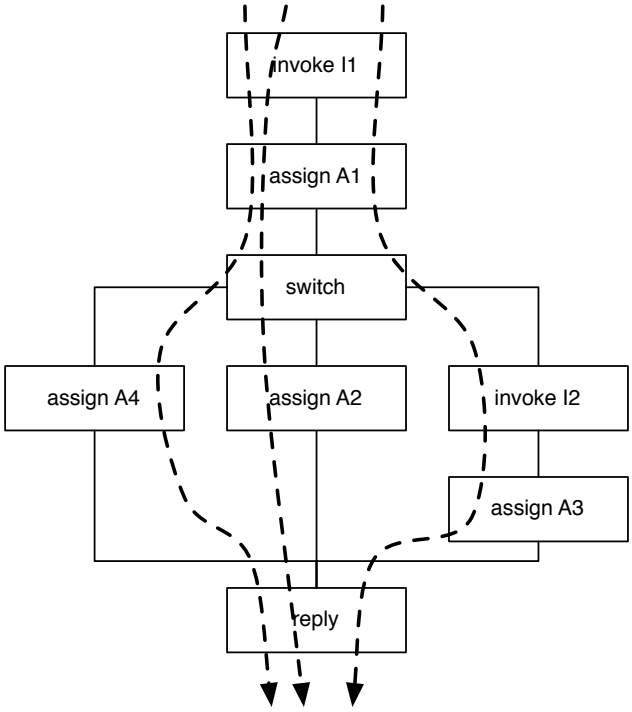
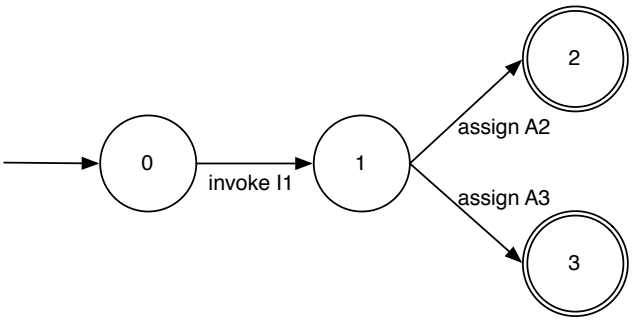


Figure 1. Obtaining a string from a process execution




```
concat(['invoke I1',
choice(['assign A2', 'Assign A3'])])
```

Figure 2. Finite state automaton for a given protocol

reached at the end of the string, we can say that the string is recognized by the FSA. It also satisfies the regular language the automaton corresponds to.

The same principle can be applied in order to match protocols [11]. The protocol specification, which is expressed in a regular language, is translated to a FSA. The string that has to be validated by this automaton is a particular execution flow of the process. If one of those flows is accepted by the automaton, the protocol is matched and the corresponding advice can be executed.

Figure 2 shows the FSA that represents the protocol `concat(['invoke I1', choice(['assign A2', 'Assign A3'])])`. In this protoc specification, the keyword `concat` denotes that the following activies should occur in order, and the keyword `choice` denotes that either of its arguments will be accepted. As a result, this FSA accepts the strings 'invoke I1 assignA2' and 'invoke I1 assign A3'. It would not accept the string 'assign A2 invoke I1'. Section 3.5 contains a discussion how we can match a sequence of events, irregardless of the order they are encountered in.

How such a string is obtained from a particular execution of a WS-BPEL process is shown in figure 1. The dotted lines represent a particular execution flow. The flow on the left generates a string "'invoke I1' 'assign A1' assign A4' ", while the flow in the middle of the figure generates

the string "'invoke I1' 'assign A1' 'assign A2' ". The flow on the right generates the string "'invoke I1' 'assign A1' 'invoke I2' 'assign A3' ".

If such a string is obtained and is matched by the FSA, we know that the specified protocol has been matched, and the advice specified in the aspect may be applied. It is clear that the flow on the left does not match the given protocol, while the one in the middle does.

By default, this implementation accepts protocols in a non-strict way. This means that other activities may be executed in the meantime which are not part of the protocol specification. This is sensible because otherwise every activity, which is possibly executed during the protocol but makes no part of it, should be added to the protocol specification. This could be very cumbersome, due to the rich joinpoint model of Padus. Every activity is represented in this joinpoint model, even if it doesn't provide any real behavior. For instance, the "sequence" activity on its own does not provide any logic, but is available in the joinpoint model and can be described with the pointcut language.

We see this in figure 1. The flow on the right contains the execution of a activity during the protocol ('invoke I2'). In the non-strict sense, this string is still valid for the automaton and as such will trigger the aspect.

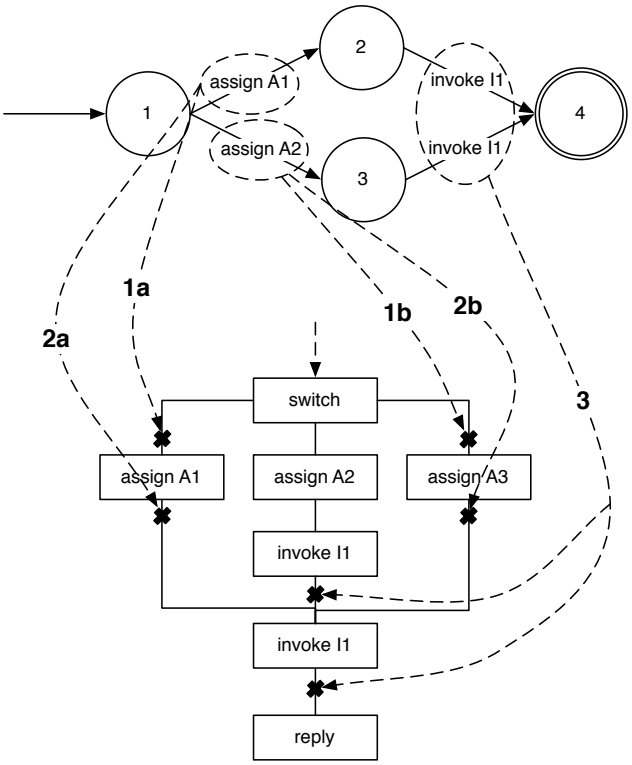


Figure 3. Tracking the FSA state

3.4 Weaving history tracking code

As indicated, finite-state automata come in handy for implementing stateful aspects. However, the FSAs only describe the protocol. The history of the execution of the process has to be checked against this FSA. This section shows how the history of the execution of the application is traced. Some instrumentation code is needed for this and is woven by the weaver at weave-time.

The logic which traces the process history is added by injecting small pieces of advice into the process. How the FSA is traversed is controlled by these advices which perform some bookkeeping on the state of the process by employing process variables to store this state information. These advice also include the advice code if it is to be executed at that time in the process execution. Because a transition of the FSA is fired when the corresponding pointcut is executed, all these instrumentation advice blocks are added after that pointcut. When the current state of the process is an end-state of the FSA, the aspect advice is executed. An extra variable is also woven into the process to store the current state of the FSA.

The protocol tracing advices are generated as ordinary "switch" activities. Figure 3 illustrates how these "switch" activities are formed and where they are placed in the process. Listing 3 shows the tracking advice that is inserted. At the very beginning of the protocol matching, the state-variable which holds the current state of the FSA is initially set to the start-state of the FSA, which is always `[0]`. This initialization is woven in by a before advice, at each point where the protocol might start. In other words, before each transition which starts from the start-state (arrows 1a and 1b on figure 3, fragment 1a/1b in listing 3).

Furthermore, after the pointcuts that represent transitions, advice is woven to update the state of the FSA (arrows and code fragments 2a and 2b). When more than one transitions exist with the same label, a separate branch is made in the "switch" activity for every such transition. The state is updated in theses branches according to the transition. Similar tracing advice is generated for the transitions which do lead to an end-state, but the original aspect advice is injected directly into the branches. Arrow 3 on figure 3 illustrates this. Two transitions exist with the same label 'invoke I1'. Consider therefore the two branches in the tracing advice which changes the state.

This implementation of stateful aspects in Padus offers two ways for attaching advices to a protocol specification. One way is by specifying that behavior should be executed after the whole protocol is matched with the execution flow of the process. The other possibility is to execute behavior while matching the protocol. In other words, consider the protocol encoded as a FSA, advice is then attached to a transition. Weaving advice for this situation is similar to weaving advice after a whole protocol is matched. Instead

```
<assign> 1a / 1b
  <copy><from expression="'[1]'" />
        <to variable="protocol" /></copy>
</assign>

<switch> 2a / 2b
  <case condition='protocol = [1]'>
    ... set protocol to [2] (or [3]) ...
  </case>
</switch>

<switch> 3
  <case condition='protocol = [2]'>
    ... set protocol to [1] ...
    ... execute aspect advice ...
  </case>
  <case condition='protocol = [3]'>
    ... set protocol to [4] ...
    ... execute aspect advice ...
  </case>
</switch>
```

Listing 3. FSA updating advice

```
<pointcut name="parallelProtocol(Jp1, Jp2, Jp3)"
          pointcut="sequence(concat([getInformation(Jp1),
                        parallel([bookHotel(Jp2),
                                  bookFlight(Jp3)])]))" />
```

Listing 4. Pointcut description inconsiderate about the order of certain events

of injecting the aspect advice in the tracing advices for transitions leading to an end-state, the aspect behavior should now be added to the tracing advice of the corresponding transition. The advice should only be executed if the protocol has matched up to this point. We can safely assume that this is the case, since the advice is inserted right at the point where the state of the FSA is updated.

3.5 Shuffled protocol

WS-BPEL, and workflow languages in general, have the capability of executing parts of the process concurrently. For WS-BPEL, the "flow" activity handles the concurrent execution of activities and shields the developer from further details. Multiple sequences of activities can be specified in separate branches in this flow activity, and are then executed concurrently. The WS-BPEL execution engine translates the flow activity into a set of separate threads, one thread for each branch of the split.

Irregarding the underlaying reason, we can not reasonably predict the order in which parallel activities will be executed. And this leads to a complication with regard to expressing protocols for stateful aspects. Often it not important in which order certain events are encountered, but we cannot simply write this down with the available language primitives. Consider for instance the protocol 'A B C', but the order in which B and C are encountered is not important. We want to accept the executions traces 'A B C' and 'A C B'. In order to achieve this, we add a new predicate "parallel" to the language extension for Padus. This predicate takes as its single parameter a list of pointcuts which have to be encountered, but not necessarily in that order.

To implement this, we combine the automata that represent the pointcuts given to the parallel predicate, and construct the "shuffle product" (as shown in [18]). This shuffle product is obtained by sparsely inserting one FSA into the other, and thus obtaining a combined automaton that keeps track of all possible combinations of states. Figure 4 shows how a new FSA is obtained from the shuffle product of two other automata. Listing 4 shows the definition of such a protocol. The pointcut in this examples describes a sequence that should be matched when `getInformation` is followed by both `bookFlight` and `bookHotel`. However, the order in which these booking services are encountered is not relevant. In other words, this pointcut matches when `getInformation` is followed by `bookHotel` and then `bookFlight`, or when `getInformation` is followed by `bookFlight` and then `bookHotel`.

Applying the shuffle product leads to a combinatorial explosion of states in the resulting automaton. The weaver has to take these additional states into account and include them in the "switch" activities that are formed at the selected joinpoints. The process in itself has not become more complex, but now includes much more tracing code.

4. Related work

Two approaches that implement stateful aspects are already discussed in section 2, and we will not repeat this here. As far as we know, no other approaches support stateful aspects in workflow languages. We do however acknowledge the work on introducing AOP in workflow languages in the following approaches.

The *Web Services Management Layer (WSML)* [8] uses aspects implemented in JAsCo to capture client-side web service management concerns such as billing, transactions, selection and caching. Compositions of web services are handled by traditional approaches such as WS-BPEL. This management layer employs aspects to store communication details and conversational context of the web services [16]. The WSML uses custom JAsCo aspect factories to define a instantiation strategy that ties stateful aspects to a specific conversation.

AO4BPEL [7] is an aspect-oriented extension to WS-BPEL that allows for more modular and dynamically adaptable web service compositions. Each WS-BPEL activity is a potential join point. In contrast to Padus, AO4BPEL uses the lower-level XPath pointcut language. As a consequence,

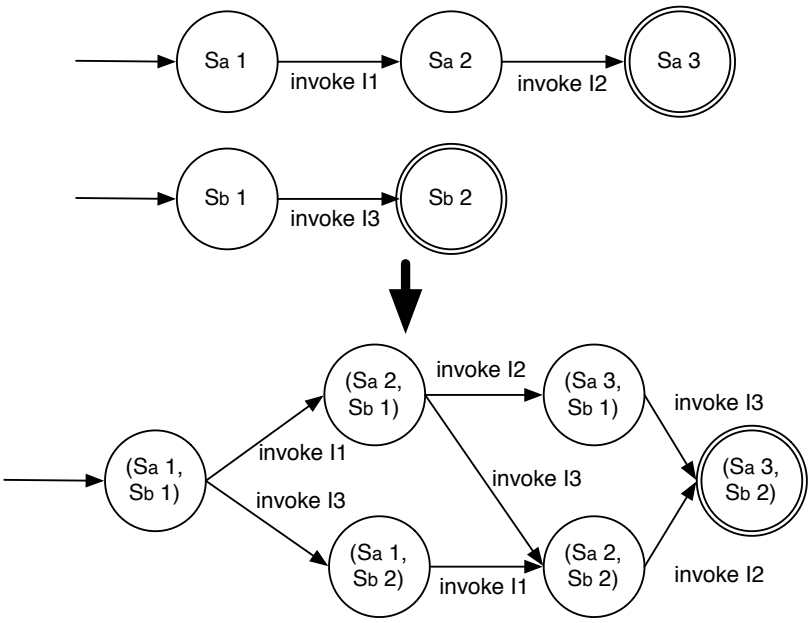


Figure 4. Computing a shuffled state machine

pointcuts are low-level and refer directly to paths in the document tree, which limits their reusability and makes them fragile with respect to evolution of the base process. Furthermore, the approach does not support an explicit aspect deployment construct nor does it allow aspect reuse. While AO4BPEL allows for aspect addition and removal while processes are running, supporting this requires a custom-made WS-BPEL engine.

Courbis and Finkelstein [10] present an aspect-oriented language extension very similar to AO4BPEL. They also use XPath as a pointcut language and use a custom WS-BPEL engine for allowing dynamic aspect addition and removal. In contrast to AO4BPEL and Padus, however, the advice language is Java.

Further discussion on the approach that is followed in Padus and on the logic pointcut language it features is available in previous work [6].

5. Discussion

About the performance of the stateful aspect extension in Padus, it can be easily stated that the protocol weaving is not as optimal as possible. Both the performance of the weaving process and the run-time performance are discussed. First, we take a closer look at the performance of the weaving process itself.

By generating a deterministic finite-state automaton from the protocol specification, many efforts are made to make the weaving process as efficient as possible. With a deterministic FSA, no backtracking is needed during the weaving of the protocol in the process which makes it more performant. However, it is very time consuming to compute a shuffled machine. Each argument to the "parallel" predicate requires the generation of a separate deterministic FSA from which the shuffled machine is generated afterwards.

Another issue is that some instrumentation advice that is added to the process will never be executed, because it is in a branch of the process that is not reached. This can be optimized by performing a graph-analysis on the protocol to exclude the weaving of unreachable and unexecutable protocol trace advices in the process.

Padus is implemented as a static weaver. The idea behind this decision was on one side to maintain efficient runtime performance, and on the other side to deliver woven WS-BPEL processes that can be executed on standard execution engines. With stateful aspects however, many pieces of instrumentation code that trace the process state are woven in, and this poses some additional runtime overhead. Currently these advices are always inserted, even when the advice is in an unreachable branch of the process. A smarter weaver could anticipate such cases and avoid weaving in these locations. The advice is also always executed. In the occasion that a protocol can no longer match against the state of the process, the tracing code is still executed. A dynamic weaver can in this case temporarily disable the aspect, so the state of the process no longer needs to be traced.

The fact that Padus is implemented with a static weaver also imposes some run-time performance deficiencies. During weave-time, many instrumentation pieces of code are injected into the process and generate run-time overhead.

Even when it is not possible to match the protocol during the execution, these protocol trace advices are still executed. A dynamic weaver would avoid this problem and only insert the trace advice when needed.

6. Conclusions and future work

In this paper we introduce history-based aspects to the existing language Padus. We added a number of new predicates to the Padus pointcut language, and also modified the weaver to include history tracking in the process. The pointcut that describes the protocol is translated in a finite-state automaton. The state of this FSA is tracked in the process execution, and when it reaches its final state, the advice is executed.

This proof-of-concept implementation is a first step for further research in this particular domain of aspect-oriented programming and workflow languages. In future work, we plan to investigate support for strict protocols. These protocols do not match an execution trace when additional joinpoints are encountered between those that are defined in the protocol. This can prove to be a problem, because of the rich joinpoint model of Padus. Tracematches provide an interesting clue to only take those joinpoints in account that are described, but not included in the protocol definition [2].

Likewise, we can include support for protocol complements, where advice is to be executed when a protocol is not matched (can not be matched). These two features are available in the stateful aspect extension of JAsCo and have been proven to be very useful.

Regular expressions are used for the specification of protocols because they are rather easy to understand, but they restrict the way how protocols can be described. For example, with regular expressions it is not possible to describe recursive protocols. This issue can be solved by using a context-free language instead of a regular language. Whether these advanced languages and which specific constructs of theirs are required is also topic for future research.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 345–364. ACM Press, 2005.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: The AspectBench compiler for AspectJ. In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 10–16. Springer, Sept. 2005.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, editors. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, Heidelberg, Germany, 2004.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, version 1.1, May 2003.

[5] A. Arsanjani, B. Hailpern, J. Martin, and P. Tarr. Web services: Promises and compromises. *Queue*, 1(1):48–58, 2003.

[6] M. Braem, K. Verlaenen, N. Joncheere, W. Vanderperren, R. Van Der Straeten, E. Truyen, W. Joosen, and V. Jonckers. Isolating process-level concerns using Padus, 2006. Accepted at the 4th International Conference on Business Process Management (BPM 2006).

[7] A. Charfi and M. Mezini. Aspect-oriented web service composition with AO4BPEL. In L.-J. Zhang, editor, *Proceedings of the 2nd European Conference on Web Services (ECOWS 2004)*, pages 168–182, Erfurt, Germany, Sept. 2004. Springer-Verlag.

[8] M. A. Cibrán, B. Verheecke, and V. Jonckers. Aspect-oriented programming for dynamic web service monitoring and selection. In L.-J. Zhang, editor, *Proceedings of the 2nd European Conference on Web Services (ECOWS 2004)*, Erfurt, Germany, Sept. 2004. Springer-Verlag.

[9] T. Cottenier and T. Elrad. Dynamic and decentralized service composition with Contextual Aspect-Sensitive Services. In *Proceedings of the 1st International Conference on Web Information Systems and Technologies (WEBIST 2005)*, pages 56–63, Miami, FL, USA, May 2005.

[10] C. Courbis and A. Finkelstein. Towards aspect weaving applications. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 69–77, New York, 2005. ACM Press.

[11] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 141–150. ACM Press, Mar. 2004.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[13] D. Stein, S. Hanenberg, and R. Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2006. ACM Press.

[14] D. Suvée and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 21–29. ACM Press, Mar. 2003.

[15] W. Vanderperren, D. Suvée, M. A. Cibrán, and B. De Fraine. Stateful aspects in JAsCo. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Proc. 4th Int'l Workshop on Software Composition (SC 2005) , Revised Selected Papers*, volume 3628 of *lncs*, pages 167–181, Edinburgh, UK, 2005. Springer-Verlag.

[16] B. Verheecke and V. Jonckers. Stateful aspects for conversational messaging with stateful web services. In *NWESP '05: Proceedings of the International Conference on Next Generation Web Services Practices*, page 363, Washington, DC, USA, 2005. IEEE Computer Society.

[17] B. Verheecke, W. Vanderperren, and V. Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1):42–50, Jan. 2006.

[18] A. Wombacher, P. Fankhauser, and E. J. Neuhold. Transforming bpel into annotated deterministic finite state automata for service discovery. In *ICWS*, pages 316–323. IEEE Computer Society, 2004.