# Modularization of distributed web services using Aspects With Explicit Distribution (AWED)*

Luis Daniel Benavides Navarro[1], Mario Südholt[1]
Wim Vanderperren[2], and Bart Verheecke[2]

[1] OBASCO project; EMN - INRIA, LINA
École des Mines de Nantes, 4 rue Alfred Kastler, 44307 Nantes cedex 3, France
{lbenavid,sudholt}@emn.fr
[2] System and Software Engineering Lab
Vrije Universiteit Brussel, Pleinlaan 2 1050 Brussels, Belgium
{wvdperre,bverheec}@ssel.vub.ac.be

**Abstract.** With the adoption of Web services technology to realize Service Oriented Architectures, the need arises for more flexible and dynamic technologies for the just-in-time integration and composition of services. As the runtime integration, selection and management of services involves a variety of crosscutting concerns, such as error handling, service monitoring, and QoS enforcements, Aspect Oriented Programming (AOP) is useful to modularize such concerns.

In this paper we investigate aspect-oriented support for crosscutting concerns of distributed management of web service compositions. We propose to use a distributed AOP approach, Aspects with Explicit Distribution (AWED), to modularize such concerns in a distributed variant of the Web Services Management Layer (WSML). Concretely, we present three contributions. First, we present an extension of the WSML to distributed compositions. Second, we present two extensions to AWED's aspect language which are useful for the modularization of crosscutting concerns of web services: support for chains of (a)synchronous remote advice that communicate through futures, and support for different modes of parameter passing between remote pointcuts and advice. Third, we illustrate our approach by investigating error handling in distributed web compositions.

## 1 Introduction

Because of their platform and hardware independent nature, web services have become popular to realize Service Oriented Architectures (SOA) [31]. It is well known that low-level infrastructure-related code frequently abounds in the implementation of SOAs based on web services [29, 10]. For instance, exception handling code has often to be added around service invocations in order to log

* This work has been supported by AOSD-Europe, the European Network of Excellence in AOSD (http://www.aosd-europe.net).

failures and trigger backup services. The resulting code frequently crosscuts the business code required by the services.

Aspect Oriented Programming (AOP) [3, 17] is useful to modularize such crosscutting concerns of web service based applications. Several previous approaches have investigated this claim in the context of web service infrastructures relying on centralized service composition (as is common in BPEL-based approaches). We have, in particular, proposed the Web Services Management Layer (WMSL) [28] that focuses on achieving dynamic and flexible client-side integration and management of services in the clients. Concretely, the WSML deals with redirecting client requests to services while taking into account QoS selection policies, service composition and client-side management concerns such as monitoring, logging, security and caching.

Recently, the first approaches for decentralized management of web service compositions have been put forward, in particular, to avoid performance bottlenecks arising from centralized web service composition (see, *e.g.*, [8]). However, the handling of crosscutting concerns in such decentralized web service management systems is an open issue. In this paper we investigate the modularization of crosscutting concerns of distributed web service management. We propose an infrastructure enabling distribution of web service compositions based on a variant of the WSML where crosscutting concerns are modularized using a recently proposed aspect language for distributed applications, Aspects with Explicit Distribution (AWED). AWED allows distribution-related crosscutting concerns to be modularized in terms of sequences of remote execution events, advice that executes on remote hosts, and aspects providing for a distributed notion of state.

Here, we present three contributions. First, we present a model extending the WSML to support distributed web service compositions. Second, we extend the AWED language by two new features: chains of advice which may integrate synchronously and asynchronously executed remote advice, and a richer model of parameter passing between remote pointcuts and remote advice. Third, we illustrate how crosscutting concerns of distributed web service compositions may be better modularized than with previous approaches by implementing a non-trivial distributed error handling strategy using our approach.

The paper is structured as follows. Section 2 provides further motivation of crosscutting concerns in distributed web service compositions. In Sect. 3 we introduce our model of the distributed WSML. The AWED language for aspects with explicit distribution and an overview of the implementation of our system is presented in Sect. 4. Section 5 presents how error handling for distributed web services can be implemented using our approach. Related work is discussed in Sect. 6. Section 7 gives a conclusion and presents future work.

## 2 Motivating example: travel agent application

In order to motivate our approach, we first present an example motivating distributed WS infrastructures and why crosscutting is a major issue in such a

setting. We then give an overview of the distributed web service platform we base our extension on, the distributed Web Services Management Layer.

As a motivating example for our approach, let us first discuss how a travel agent application can benefit from a distributed web service infrastructure and why crosscutting concerns are a particular hard problem for such an application.

A travel agent application is a typical example of a SOA used by customers to, for instance, book an online holiday. For this purpose, the travel agent needs to communicate with a wide variety of services to obtain hotel and flight information and to arrange bookings. In a centralized fashion, this SOA would implement a business process where for instance, first the flight is booked, and based on these results, a hotel is booked at the destination city for the corresponding period. In the distributed approach, the different parts of the WS composition become decentralized and each node deals with a particular subset of the business process at different locations within the distributed system. The nodes communicate directly with each other to transfer data and control, instead of relying on a central coordinator. A distributed implementation of this setting is obviously attractive because, *e.g.*, it allows concurrent execution of independent parts of the composition. However, distributed WS infrastructures must frequently be able to dynamically accommodate changes to the current WS composition. Let us consider three typical scenarios for travel agent applications:

- *Error handling.* If both the hotel booking and flight booking sub processes are handled concurrently by two WSML instances, and one process results in a failure, the other process can be rolled back, too. More generally, a failure situation may require the termination of some executing parts of the composition and rollbacks at a large number of nodes.
- *Performance optimization.* SOAs are frequently performance-critical. One way to measure performance consists in setting up measurement points as part of the composition for each subprocess, network communication and the involved services. Based on this, bottlenecks can be analyzed and actions undertaken. For instance, calls to slow services can be distributed over multiple semantically equivalent services, network traffic on congested networks can be optimized by installing caching functionality and service invocations that take a long time can be executed in advance.
- *Evolution of business requirements.* A change to the business requirements, for instance, relaxation of the rules for booking travel tickets (an operator may start offering the possibility of later reservations for premium customers), frequently call for adaptations of web services compositions.

Each of these three require communication between many nodes. Furthermore, most of these are difficult to anticipate because they often depend on the specific composition at hand. In a context where new compositions may be developed, anticipation would be next to impossible. A dynamic distributed WS infrastructure therefore seems most appropriate for them.

Note that all three scenarios include crosscutting behavior. Error handling [18] is a typical crosscutting concern whose implementation requires modifications at a large number of places in the code that partially depend on the node where an

error occurs and where the corresponding error handlers are executed. (see [8] for more information on error handling in distributed WS infrastructures and an approach to deal with part of this problem). Similarly, the monitoring and performance optimization scenario also requires modifications to many code locations on many nodes. Furthermore, optimization actions may vary much from node to node. Finally, note that even business requirements are frequently crosscutting if a legacy code base has to be adapted (see [6] of a detailed discussion of this issue in the context of replicated transactional caches).

This paper presents a distributed WS infrastructure supporting such dynamic adaptations and, in particular, the modularization of such crosscutting concerns.

## 3 Overview of distributed WSML

The Web Services Management Layer (WSML) is an architectural framework for the mediation of Web services in client applications. The WSML is placed in between the client and the web services management infrastructure. All service related code is extracted from the clients and placed in the WSML, where each service related concern is enforced transparently for both the client and the services.

Concretely, the WSML is able to redirect client requests to functionally compatible services, while dealing with possible mismatches and incompatibility. If necessary, multiple services can be composed so that they can cooperate to deal with a given request. Additionally, the WSML enforces a more advanced selection mechanism that contemplates service criteria based on non-functional Quality-of-Service properties of services and deals with other client-side management concerns such as monitoring, logging, security and caching.

The original WSML [28] provided a web service composition model based on a centralized coordinator, similar to common WS-BPEL engines. We have extended this model to a decentralized setup as depicted in Fig. 1. There, two hosts that are part of the composition are equipped with a separate instance of the WSML, which takes care of all service related coordination with third-party services (which are shown in the figure on the right-hand side). The WSML instances communicate with each other to transfer data and control, while not relying on a central coordinator. This approach differs from the one described in [8] where existing composition descriptions are partitioned and statically deployed on multiple WS-BPEL engines. With the WSML, the composition is realized by multiple aspects that are combined together, possibly at runtime. By changing or adding aspects, the composition is dynamically adapted. This runtime flexibility is needed to accommodate the composition of unanticipated changes in the third-party service environment and possible business requirements imposed by the SOA.

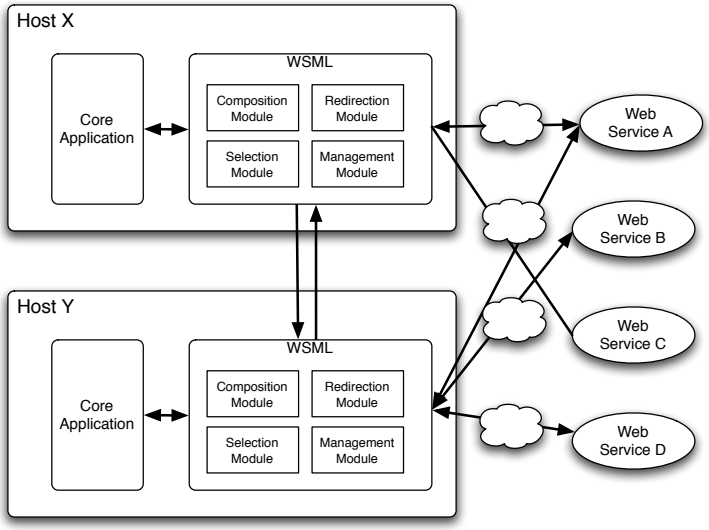


**Fig. 1.** Overview of distributed Web Services Management Layer.

## 4 The AWED language

Modularization of crosscutting concerns for web services using an aspect language (*i.e.*, in terms of pointcut, advice and aspect abstractions), suggests support for the following issues: (i) a notion of remote pointcuts allowing to capture relationships between execution events occurring on different hosts, (ii) a notion of groups of hosts which can be referred to in pointcuts and manipulated in advice, (iii) execution of advice on different hosts in an asynchronous or synchronous way and (iv) flexible deployment, instantiation, and state sharing models for distributed aspects.

AWED provides such support through three key concepts at the language level. First, *remote pointcuts*, which enable matching of join points on remote hosts and include remote calls and remote cflow constructs (*i.e.*, matching of nested calls over different machines). As an extension of previous approaches AWED supports remote regular sequences which smoothly integrate with JAsCo's stateful aspects [27] but also include features of other recent approaches for (non-distributed) regular sequence pointcuts [13–15, 4]. Second, support for *distributed advice*: advice can be executed in an asynchronous or synchronous fashion on remote hosts and pointcuts can predicate on where advice is executed. Third, *distributed aspects*, which enable aspects to be configured using different deployment and instantiation options. Furthermore, aspect state can be shared flexibly among aspect instances on the one hand, as well as among sequence instances which are part of an aspect on the other hand.

In this paper, a subset of the AWED language is presented. In particular, we focus on two novel features of AWED with respect to our previous work [5],

namely advice chains involving synchronous as well as asynchonous advice with futures, and parameter passing by reference.

AWED's syntax is shown in Fig. 2 using EBNF formalism (*i.e.*, square brackets express optionality; parentheses denote multiple occurrences, possibly none; terminal parentheses are enclosed in apostrophes).

### 4.1 Pointcuts

Pointcuts (which are generated by the non-terminal Pc) are basically built from **call** constructors (**execution** allows to denote the execution of the method body), field getters and setters, nested calls (**cflow**) and sequences of calls (non-terminal Seq).

AWED employs a model where, upon occurrence of a join point, pointcuts are evaluated on all hosts where the corresponding aspects are deployed. Pointcuts may then contain conditions about (groups of) hosts where join points originate (term **host**(Group)), *i.e.*, where calls or field accesses occur. Furthermore, pointcuts may be defined in terms of where advice is executed (term **on**(Group)). Advice execution predicates may further specify a class implementing a selection strategy (using the term **on**(Group, Select)) which may act as an additional filter or define an order in which the advice is executed on the different hosts. Groups are sets of hosts which may be constructed using the host specifications **localhost**, **jphost** and `adr:port`, which respectively denote the host where a pointcut matches, the host where the corresponding join point occurred and any specific host. Alternatively, groups may be referred to by name. (Named groups are managed dynamically within advice by adding and removing the host which an aspect is located on, see Sec. 4.3 below.)

Finally, pointcut definitions may extract information about the executing object (**target**) and arguments (**args**), and may test for equality of expressions (**eq**), the satisfaction of general conditions (**if**), and whether the pointcut lexically belongs to a given type (**within**). Pointcuts may also be combined using common logical operators.

As a first example, the simple pointcut shown in Fig. 3 could be part of a distributed monitoring aspect. The pointcut matches calls to the bookHotel method that originate from the host that has the specified address. Advice attached to this pointcut will be executed on any host where the aspect is deployed (possibly multiple ones) as there is no restriction on the advice execution host.

In Fig. 4, the pointcut restricts the execution hosts to be different from the host where the joinpoint occurred. Here, the pointcut matches calls to book hotel operations on any type by using wildcards. The advice is executed on hosts other than the joinpoint host and binds the corresponding arguments. Note that in this case the clause !**host**(**localhost**) could replace !**on**(**jphost**) to achieve exactly the same effect of matching non-local joinpoints.

AWED also supports the concept of stateful pointcuts [14] through sequences derived by the non-terminal Seq. A stateful pointcut triggers on a specified sequence of events instead of a single event. For more information about sequences, we refer to [5].

// Pointcuts

| | | |
|---|---|---|
| Pc | ::= | **call**(MSig) \| **execution**(MSig) |
| | \| | **get**(FSig) \| **set**(FSig) |
| | \| | **cflow**(Pc) \| Seq |
| | \| | **host**(Group) \| **on**(Group[, Select]) |
| | \| | **target**({Type}) \| **args**({Arg}) |
| | \| | **eq**(JExp, JExp) \| **if**(JExp) |
| | \| | **within**(Type) |
| | \| | **passbyval**({Id}) |
| | \| | Pc ‖ Pc \| Pc && Pc \| !Pc |
| Seq | ::= | [Id:] **seq**({Step}) \| **step**(Id,Id) |
| Step | ::= | [Id:] Pc [→Target ] |
| Target | ::= | Id \| Id ‖ Target |
| Group | ::= | { Hosts } |
| Hosts | ::= | **localhost** \| **jphost** \| "Ip:Port" |
| | \| | GroupId |
| GroupId | ::= | String |
| Select | ::= | JClass |

// Advice

| | | |
|---|---|---|
| Ad | ::= | [**asyncex**] Pos({Par}) : PcAppl '{' {Body} '}' |
| Pos | ::= | **before** \| **after** \| **around** |
| PcAppl | ::= | Id({Par}) |
| Body | ::= | JStmt \| **proceed**({Arg}) \| **localproceed**({Arg}) |
| | \| | **addGroup**(Group) \| **removeGroup**(Group) |

// Aspects

| | | |
|---|---|---|
| Asp | ::= | [Depl] [Inst] [Shar] **aspect** Id '{' {Decl} '}' |
| Depl | ::= | **single** \| **all** |
| Inst | ::= | **perthread** \| **perobject** \| **perclass** |
| | \| | **perbinding** |
| Shar | ::= | **local** \| **global** \| **inst** \| **group**(Group) |
| Decl | ::= | [Shar] JVarD \| PcDecl \| Ad |
| PcDecl | ::= | **pointcut** Id({Par}) : Pc |

// Standard rules (intensionally defined)

| | | |
|---|---|---|
| MSig, FSig | ::= | // method, field signatures (AspectJ-style) |
| Type | ::= | // type expressions |
| Arg,Par | ::= | // argument, parameter expressions (AspectJ-style) |
| Id | ::= | // identifier |
| Ip,Port | ::= | // integer expressions |
| JClass | ::= | // Java class name |
| JExp | ::= | // Java expressions |
| JStmt | ::= | // Java statement |
| JVarD | ::= | // Java variable declaration |

**Fig. 2.** AWED language

```
1    call(void bookHotel()) && host("134.184.2.3:1432")
```

**Fig. 3.** Simple pointcut limiting the joinpoint host.

```
1 hotelReservation(Hotel hotel, Period period):
2     call(* *.bookHotel(Hotel, Period))
3     && !on(jphost) && args(hotel,period)
```

**Fig. 4.** Pointcut limiting the execution host.

### 4.2 Parameter passing

Sharing of remote object information is an inherent need in distributed applications. AWED's new proposal includes a mechanism to manage explicitly how parameters of a given joinpoint are distributed.

The pointcut language model allows joinpoint information like parameters, caller object and target object to be bound to specific variables in pointcut or advice definitions. The model also allows those variables to be explicitly distributed by value or by reference (the latter is the default behavior). The first option, by value, creates a copy of the object in the hosts where remote advice are executed, binding the new value to the formal parameter used in the advice body. The second option, by reference, creates a remote reference to the original object. Once the remote reference or the copy are bound, they can be treated as local objects without any distinction in the advice body.

Figure 5 shows an example of the usage of the pass by value behavior. The pointcut definition has four variables `i, y, v` and `c`. The parameters of the matched method, `foo`, and the target object of that method are bound to the pointcut's variables. The pointcut **passbyval** is used to annotate the variables `i` and `c` to be passed by value.

```
1 pointcut myDef(Integer i, Object y, Vector v, MyObject c):
2     call(* foo(Integer,Object, Vector, String)) &&
3     args(i, y, v, String) && target(c) &&
4     passbyval(i, c);
```

**Fig. 5.** Parameter passing example using a pointcut definition

As usual, by value passing has to be used with care, in particular, because it implies a copy of the whole object graph below the object that is passed by copy. AWED allows all objects to be referenced remotely, copied and distributed. Reflective information queried through the `thisJoinPoint` keyword is always passed by reference.

The remote referencing model of AWED is fully transparent with respect to the object model. This means that there is no distinction between the remotely

referenced objects and the locally referenced objects. However, it is important to note that the language provides a richer and finer grained model for parameter passing in the pointcut definition language than the one provided in direct method invocation over objects. When a method is invoked directly in a remote referenced object, the parameters are passed by value as with normal Java method invocations. There is no language support to specify pass by reference behavior (although a work-around using a proxy object is possible). This is motivated by the fact that we aim to stay as close as possible to Java and changing the method invocation syntax and semantics would be a drastic measure.

### 4.3 Advice

Advice (non-terminal Ad) is mostly defined as in AspectJ: it specifies the position (Pos) where it is applied relative to the matched join point, a pointcut (PcAppl) which triggers the advice, a body (Body) constructed from Java statements, and the special statement **proceed** (which enables the original call to be executed).

In an environment where advice may be executed on other hosts (which is possible in AWED using the **on** pointcut specifier), the question of synchronization of advice execution with respect to the base application and other aspects arises. AWED proposes one unified model for (local and remote) advice execution: all advice (including remote ones) are triggered by one controller. This means that there is only one advice chain per joinpoint. The host where the joinpoint occurs is responsible for managing the advice chain. As such, there is a well-defined precedence as defined by the AspectJ precedence rules (e.g. **declare precedence**), even for advice executed on remote hosts.

Per default, advice executes synchronously to the base application, meaning that the application waits until completion of the advice in order to proceed to the original behavior or to execute the next advice in the chain. Both remote and local advice conform to this general model. The programmer may also choose to execute an advice asynchronously with respect to the application on the joinpoint host by marking the advice with the **asyncex** keyword. This means that the base application proceeds with the original behavior or executes the next advice in the chain while the asynchronous advice is executing. Of course, asynchronous advice are still treated in the same advice chain and thus are only executed when the previous synchronous advice are finished or when previous asynchronous advice are started.

Multiple around advice applying at a joint point is executed as usual in a nested fashion as part of a chain controlled by the invocation of **proceed** (see figure 6). Such advice is expected to return a value that can be processed by the previous advice that invoked proceed or by the original application in case of the first advice. In case of asynchronous advice, this value is possibly not yet computed, so the invocation of such an advice returns a future[3] object [22]

[3] A future object is an object that represents the result of an asynchronous computation, the actual value of the computation is bound to the object once the concurrent computation is finished.

that synchronizes with the remote advice in case the object is claimed. The future object is implicit, as such the advice caller can safely treat the return value as a real value. The AWED infrastructure and run-time weaver take care of generating a transparent future and claiming it whenever its value might be accessed. It is also possible to make the future explicit by casting it to a standard Java Future object[4] when useful. This way, the invoking advice may manage its behavior depending on the availability of the result.

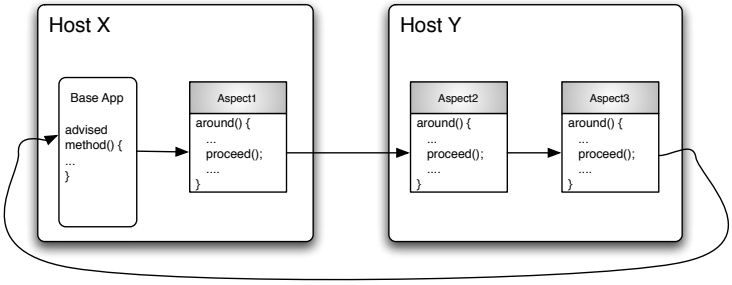


**Fig. 6.** Around advice chaining in AWED. Advice applicable to the same joinpoint execute in a single advice chain, regardless of execution host.

AWED introduces one general model for (a)synchronous distributed advice. The semantics of AWED remains backward compatible with AspectJ. The semantics of advice is also independent of whether the joinpoint host is different or the same as the execution host. This is in contrast to the previous version of AWED [5], where advice was treated differently because every host had its own advice chain. In that model, advice always executes asynchronously to advice on different hosts, impeding, for instance, remote authentication that blocks all other advice and the original behavior until authentication has been successful. The new general model of AWED is able to support such behavior as illustrated by the code fragment of figure 7. The authentication advice is guaranteed to always execute before other advice that applies to that joinpoint such as, for instance, a billing advice [12]. The billing advice will not be executed when the authentication fails because the authentication advice does not invoke **proceed** in that case.

The advice body has one important additional keyword in comparison to AspectJ: **localproceed**. The invocation of **localproceed** makes sure that the original behavior (*i.e.*, the joinpoint) is executed on the host where the advice occurs instead of on the host where the original behavior originated from (*i.e.*, the joinpoint host). As an example, consider the aspect shown in Fig. 8 that implements the distribution concern. It is well-known that distribution can be seen as a crosscutting concern that can be modularized using aspects (see, *e.g.*, [24]).

[4] Java supports futures since version 1.5 through the `java.util.concurrent` API.

```
around(User user,Object ttarget): toAuthenticate(user,ttarget) {
    if(userMayAccess(user,ttarget))
       proceed();
    else throw new AccessSecurityException(user,ttarget);
}
```

**Fig. 7.** Simple authentication around advice

The `distribution` pointcut selects all calls to `Facade` methods on the client and makes sure that the accompanying advice is only executed at the server side. By employing the negation of the `host` designator, calls on the server side will not match the pointcut themselves. The redirection behavior is encapsulated in a synchronous around advice. As the around advice gets executed on the server host, the `getInstance` method of the `Facade` class will retrieve an instance which is local to the server host (this could be generalized in order not to rely on a single object.) The **localproceed** expression makes sure to invoke the original behavior on the server host instead of on the joinpoint host. An interesting variation of the distribution concern would be to mark the advice asynchronous. The result is that the original sequential application is not only distributed but also parallelized. In such a case a future object is immediately returned to the base application and the advice is executed in parallel with the base program. The base program and the advice get synchronized once the actual value of the computation is claimed through the future.

```
pointcut distribution(Facade f):
  target(f) && call(* *(..)) &&
    && !host("Serveripadr:port") && on("Serveripadr:port");

syncex Object around(Facade f): distribution(f) {
  return localproceed(Facade.getInstance()); }
```

**Fig. 8.** Distribution as an aspect

Finally, note that AWED enables advice to manage named groups of hosts: **addGroup** adds the current host to a given group, **removeGroup** allows to remove the current host from a group.

### 4.4 Aspects

Aspects (non-terminal Asp) group a set of fields as well as pointcut and advice declarations. Aspects may be dynamically deployed (Depl) on all hosts (term **all**) or only the local one (term **single**).

Furthermore, aspects support four instantiation modes (Inst): similar to several other aspect languages, aspects may be instantiated per thread, per object, or per class.

### 4.5 Implementation

*New language features.* The AWED language is implemented on top of the dynamic AOP framework JAsCo [25]. JAsCo has two main advantages over other approaches: its dynamism and run-time performance. JAsCo allows adding and removing aspects while an application is running through its novel run-time weaver. The performance cost of advice execution is minimal and similar to a regular Java method execution. The AWED DJAsCo extension has been made publicly available as a part of the regular JAsCo distribution [16].

We have revised and extended DJAsCo to reflect the improved AWED language semantics as presented in this paper. More concretely, we have implemented support for the new advice chaining strategy including transparant futures. Furthermore, the pass by reference strategy for parameters has been added and installed as default. An interesting fact of the extension is that some parts of the language are themselves implemented by making use of basic AWED aspects. As such, we are able to tackle crosscutting concerns in our own implementation. Due to space constraints, we cannot provide more details about the implementation. For an in-depth discussion of the AWED implementation we refer the interested reader to a companion technical report [6].

*Integration with WSML.* The Web Services Management Layer can be easily implemented on top of AWED because it offers all the features of the original AOP platform used by WSML (JAsCo). The WSML generates AWED aspects that realize a distributed composition based on higher level templates. By exploiting AWED's distributed capabilities, the WSML is able to solve distribution specific crosscutting concerns. The following section discusses an interesting example of such a concern, namely error handling.

## 5 Error handling revisited

In Sect. 2 we have presented three scenarios motivating the need for aspect-oriented support for distributed WS compositions. In this section we show how to use the AWED language in order to concisely implement such scenarios.

Let us reconsider the problem of error handling in the context of distributed WS compositions. We only consider a model in which errors cause the current composition to be aborted (The same assumption is common in other approaches to distributed WS composition [8]). Error handling requires errors to be identified on all nodes taking part in a WS composition and two corrective actions to be applied when an error occurs: first, active components of the composition have to be terminated; second, effects performed by the composition before the error occurred have to be rolled back. (This obviously requires WS actions to be reversable, a property we assume here.)

Fig. 9 shows how such an error handling strategy for distributed WS compositions can be implemented using AWED. The first two advice (lines 3–11) monitor undo information for two basic web service actions (represented by calls to `store` and `invoke`) to a set of monitors. The monitoring of composition errors

```
// ---- 2 example advice for monitoring purposes-----------------

// monitor database operations within the composition group
asyncex after(): call(store(key, val)) && host(CompGroup) {
  send2Monitors(oldVal(key));
}

// monitor service invocations
asyncex after(): call(invoke(target, service)) && host(CompGroup) {
  sendUndoInfo2Monitors(thisCompositionNode, INVOKE, target, service);
}



// ---- error handling ------------------------------------------

// composition errors:
// call to webServiceError() after service initialization
pointcut compositionError:
  seq(initCompositionMonitors(CorrelationIdMonitors) && host(Monitor),
      initCompositionNode(CorrelationIdNode)
        && host(CompGroup) && eq(CorrelationIdMonitors, CorrelationIdNode),
      webServiceError() && host(CompGroup)
     )

// error: abort local service
asyncex after(): compositionError {
  abortComposition();
}

// service abortion: register abortion with monitor
asyncex after(): call(abortComposition()) &&
jphost(CompGroup) && on(Monitor) {
  registerAbortion(joinPointHost());
}

// service abortion registered: terminate blocked services
// synchronized to perform undo on terminated services later
after: call(aborted(_, _, _)) && on(CompGroup, blockedOnInput) {
  terminateService();
}

// service abortion registered:
// undo previous actions of terminated actions
asyncex after: call(aborted(_, undoInfo, compositionHandle)) &&
               passbyval(undoInfo) &&
               on(CompGroup, terminated) {
  undoService();
}
```

**Fig. 9.** Error handling using AWED

is defined in a distributed fashion by means of the pointcut `compositionError` (line 19). This pointcut defines a composition error as a call to `webServiceError` occurring at a host that is part of the host group `CompGroup` and correlated to the relevant set of monitors (using the equality test `eq(CorrelationIdMonitors, CorrelationIdNode)`).

The last four advice define the error handling strategy: First, on occurrence of a composition error the computation at the current node is aborted (line 28). Second, the abortion has to be registered with the monitor (line 34). Third, other still active components (which have to be blocked and waiting for input) have to be terminated (line 40). To this end, the pointcut matches a call to `abort` on the monitor, which provides access to the undo information and a handle to the current WS composition that is used in case local undos depend on the undos of other activities. Fourth, all terminated activities are undone (line 48).

This example presents the major features of AWED's features for explicit distribution:

- The *pointcuts* are triggered on different hosts and the pointcut `compositionError` line 19) consists of a *sequence* of remote events.
- Steps 2–4 (lines 31–49) use AWED's *remote execution feature*: In step 2, the call `abortComposition` is executed on a host of group `CompGroup`, while the triggering action `registerAbortion` occurs in the monitors.
- Steps 2 and 3 (lines 31–41) illustrates the first new language feature introduced in this work, a *mixed synchronous/asynchronous advice chain*: both advice are triggered on the monitor. In Step 2, termination is triggered synchronously in order to ensure that all activities are terminated before rollbacks are executed. The rollbacks (step 3), however, may be executed concurrently. Note that the body of the advice of steps 2 and 3 are executed on different but overlapping sets of hosts (in fact the set of hosts identified by the predicate `terminated` strictly includes the set identified by `blockedOnInput`).
- Step 4 (lines 43–49) illustrates the use of the new *parameter passing* feature. The (immutable) local undo information is passed by value. Dependencies to other nodes may appear dynamically depending on how their respective undos evolve. A handle to the composition information stored in the monitors is therefore passed by reference, so that those dependencies can be updated dynamically via the monitors.

Finally, note that this application provide clear evidence that we can better handle typical crosscutting issues in distributed WS compositions. We improve, in particular, on the recent work by Chafle et al. [8], who propose a fully centralized monitor and do not accommodate concurrent error handling actions. Our solution improves on these two issues. First, monitoring is done in a distributed fashion in Fig. 9 thus reducing the danger of bottleneck of a single monitor. Second, our implementation directly exploits parallelism in the error handling strategy by the use of asynchronously executed advice that are synchronized only if necessary. Such synchronization occurs in the above example, if the remote reference `compositionHandle` is accessed in step 4 (lines 43–49).

## 6 Related work

Since we have proposed a new model for distributed web service composition using AOP, four kinds of related work are presented in the following: approaches using AOP in the context of de/centralized web service composition, distributed web service composition infrastructures, and approaches for AOP in distributed systems.

*AOP and decentralized web service composition.* There are only very few approaches applying AO techniques to distributed web service composition. A notable exception is Aspect-Sensitive Services (CASS) [11], which provides a distributed aspect platform that targets the encapsulation of coordination, activity lifecycle and context propagation concerns in service-oriented environments. In contrast to our approach is does not support features for explicit distribution, especially asynchronous advice chaining, as we do.

*AOP and centralized web service composition.* Some more recent AOP approaches are explicitly targeted at Web services. With Padus [7] and Ao4BPEL [9], aspects can be (un)plugged into BPEL composition processes. Since BPEL processes consist of a set of activities, joinpoints in Padus and AO4BPEL are well-defined points in the execution of the processes: each BPEL activity is a possible joinpoint. The attributes of a business process or certain activity can be used as predicates to choose relevant joinpoints. BPEL, Padus and AO4BPEL differ from our approach as they realize centralized compositions.

Singh et al. [23] present a software architecture for web services: Aspect-Oriented Web Services (AOWS). It is targeted at describing crosscutting concerns between web services to give more complete description of Web services, supporting richer dynamic discovery and seamless integration. An implementation is made on the .NET platform and all AOWS subsystems and their relationships have been formally modelled. While aiming to achieve similar goals as the WSML, AOWS does not support third-party independent services as services need to be modelled in an AOWSDL language, and registered in a dedicated AOUDDI registry.Multiple services are bundled in a centralized fashion in an AOComposite. The aspectual features of the AOWS framework are used to provide more efficient and effective dynamic description, discovery and integration. Similar as in our approach, service related code is extracted from the client, and the client only needs to communicate with the AOconnectors.

*Decentralized web service composition.* A few approaches have recently been put forward for decentralized web service composition that do not employ AOP techniques. Most notably, Chafle et al. [8] propose techniques for the partitioning of web service compositions and error handling mechanisms for distributed web service compositions. As presented in Sect. 5 our approach allows the concise definition of more general error handling strategies than they propose. Furthermore, their partitioning technique results in static overall architecture which precludes dynamic changes to the composition.

*AOP for distributed applications.* There are only very few AOP languages and system that support features for explicit distribution (notable exceptions being D [30], JAC [20] and DJCutter [19]). These systems could potentially be used to modularize crosscutting functionalities in web services. However, the distributed features of all of these systems are much more restricted than those of AWED. None of these systems support asynchronous advice execution. Recently, ReflexD [26] has been proposed as a kernel supporting a range of distributed AO languages. ReflexD provides a meta-object protocol like interface to features for the implementation of distributed pointcuts, advice and bindings. As is, *i.e.*, without language support, ReflexD is not suitable for the concise definition of crosscutting concerns in web services. Furthermore, it is an open question whether the system can be reasonably integrated with existing web service infrastructures, as we show here for AWED and distributed WSML.

Most approaches that can be used currently to apply aspects dynamically to distributed applications (*e.g.*, JBoss AOP [1], Spring AOP [2]) as well as many academic approaches (see DAOP [21] to name just one), essentially allow non-distributed aspects to manipulate applications implemented using an existing framework for distribution, such as industrial component platforms or other middlewares. These approaches do not allow to express web service related crosscutting functionalities concisely because all communication and coordination issues that are handled using AWED by remote advice execution, would have to be implemented manually using the corresponding middleware libraries.

## 7 Conclusion

In this paper we have investigated the problem of crosscutting concerns of distributed composition of web services. We have presented an approach integrating a new model for such compositions, distributed WSML, with the AWED language for aspect-oriented programming of distributed applications to modularize crosscutting concerns of web services compositions. We have introduced two language extensions for AWED: chains of mixed a/synchronous executed remote advices and an extended model of parameter passing between remote pointcuts and advice. Finally, we have given evidence that these extensions allow the concise modularization of crosscutting concerns for the case of an error handling concern.

This work paves the way for different leads of future work. Most importantly, previous work on AOP has shown that domain-specific abstractions are often useful. In the context of web services a predefined set of composition operators should be useful as well as for the more declarative definition of web service compositions as for reasoning about the correctness of compositions. Second, in this paper we mainly applied AWED's features for remote pointcuts and advice; the application to web services of some other features, especially for distributed state, remains to be done.

## References


1. *JBoss AOP*. http://labs.jboss.com/portal/jbossaop.
2. *Spring AOP*. http://www.springframework.org/.
3. M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004.
4. 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, New York, NY, USA, 2005. ACM Press.
5. L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM Press.
6. L. D. Benavides Navarro, M. Südholt, W. Vanderperren, and B. Verheecke. Modularization of distributed web services using AWED. Research report, INRIA, June 2006.
7. 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. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006)*, Vienna, Austria, Sept. 2006. Springer-Verlag. (to appear).
8. G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM Press.
9. A. Charfi and M. Mezini. Aspect-oriented web service composition with AO4BPEL. In L.-J. Zhang, editor, *ECOWS*, volume 3250 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2004.
10. A. Colyer and A. Clement. Large-scale aosd for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65, New York, NY, USA, 2004. ACM Press.
11. T. Cottenier and T. Elrad. Dynamic and decentralized service composition: With contextual aspect-sensitive services. In J. Cordeiro, V. Pedrosa, B. Encarnação, and J. Filipe, editors, *WEBIST 2005, Proceedings of the First International Conference on Web Information Systems and Technologies, Miami, USA, May 26-28, 2005*, pages 56–63. INSTICC Press, 2005.
12. M. D'Hondt and V. Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 132–140. ACM Press, Mar. 2004.
13. R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of GPCE'02*, volume 2487 of *LNCS*, pages 173–188. Springer-Verlag, Oct. 2002.
14. R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of AOSD'04*, pages 141–150. ACM Press, Mar. 2004.
15. R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. In *AOSD '05: Proceedings of the 4th international conference on*

*Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.
16. JAsCo. *JAsCo website*. http://ssel.vub.ac.be/jasco/.
17. 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.
18. M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc. of the 22nd Int. Conf. on Software Engineering (ICSE'00)*, pages 418–427, New York, NY, USA, 2000. ACM Press.
19. M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed aop. In *Proc. of the 3rd Int. Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 7–15, New York, NY, USA, 2004. ACM Press.
20. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Reflection*, LNCS 2192, pages 1–24. Springer, 2001.
21. M. Pinto, L. Fuentes, and J. M. Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 118–137, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
22. J. Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
23. S. Singh, J. Grundy, J. Hosking, and J. Sun. An architecture for developing aspect-oriented web services. In *Proceedings of the third European Conference on Web Services (ECOWS)*, pages 72–82, Vxjo, Sweden, 2005. IEEE Computer Society.
24. S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proc. of OOPSLA'02*, pages 174–190. ACM Press, 2002.
25. D. Suvée and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29. ACM Press, Mar. 2003.
26. É. Tanter and R. Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, 2006. Springer-Verlag.
27. W. Vanderperren, D. Suvée, M. A. Cibrán, and B. D. Fraine. Stateful aspects in jasco. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, *Software Composition*, LNCS 3628, pages 167–181. Springer, 2005.
28. B. Verheecke, M. A. Cibran, W. Vanderperren, D. Suvee, and V. Jonckers. AOP for dynamic configuration and management of web services. *International Journal of Web Services Research (JWSR)*, 3(1):25–41, July 2004.
29. B. Verheecke, W. Vanderperren, and V. Jonckers. Unraveling crosscutting concerns in web services middleware. *IEEE Software*, 23(1):42–50, Jan. 2006.
30. C. Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.
31. O. Zimmermann, V. Doubrovski, J. Grundler, and K. Hogg. Service-oriented architecture and business process choreography in an order management scenario: rationale, concepts, lessons learned. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 301–312, New York, NY, USA, 2005. ACM Press.