An Aspect-Oriented Approach to Securing Distributed Systems

Henner Jakob, Nicolas Loriant, Charles Consel

HAL Id: inria-00455398

https://inria.hal.science/inria-00455398

Submitted on 10 Feb 2010

An Aspect-Oriented Approach
to Securing Distributed Systems

Henner Jakob
INRIA
henner.jakob@inria.fr

Nicolas Loriant
INRIA
nicolas.loriant@inria.fr

Charles Consel
INRIA / LaBRI
charles.consel@inria.fr

ABSTRACT

The increasing size and complexity of distributed systems create a need to raise the level of abstraction for their development. This need becomes critical for pervasive computing where non-functional properties, such as security, must be guaranteed. Architecture description languages (ADLs) propose a promising approach to coping with the size and complexity of pervasive computing systems. A system is defined by a high-level description that may be used to produce a programming framework. However, non-functional properties are not specifically addressed by existing ADL works. To address this issue aspect-oriented programming is a well-proven technique to properly modularize non-functional concerns that can be dealt with by weaving dedicated code into a program.

In this paper, we present DiaAspect, an aspect-oriented language for an ADL. ADLs are a key to our approach because they expose features enabling an accurate coordination of aspects. We demonstrate the expressiveness of DiaAspect with two examples of security policies in pervasive computing. We also show how, combining the knowledge of the architecture description with aspect code, improves aspect weaving in the implementation code.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Design, Languages

Keywords

Distributed systems, Security, Pervasive computing systems, Aspect-oriented programming

1. INTRODUCTION

Computing platforms are rapidly shifting from desktops to environments populated with networked devices. These distributed systems consist of an ever larger range of entities that offer computing power, extensive functionalities and network capabilities. The advent of these systems provides services to users at anytime and everywhere, opening up a host of application areas, ranging from home automation to assisted living.

Non-functional properties like security typically impact every aspect of a system. To cope with these concerns, one must integrate them at an early stage of the system development, as is proposed by an ADL. An ADL allows to precisely and formally define a software architecture in terms of the elements of the system, their behavior, their externally visible properties and the relationships among them [2]. However, non-functional properties have received limited attention in the ADL community in that, only specific concerns have been addressed. This situation leads architects to decorate an architecture description with non-functional concerns, obfuscating the entire design.

Some middlewares provide support for non-functional concerns. For example, in the Enterprise Java Bean (EJB) component model [13], EJB containers provide support for security through encryption and authentication. Such component models are either dependent on a specific middleware or provide little development support, if any.

Aspect-oriented programming (AOP) [7] is a software engineering approach to combining non-functional concerns with software design. AOP provides techniques and tools to systematically represent, modularize and compose concerns that are crosscutting an entire system. While the integration of AOP in programming languages has been heavily studied, its usage at the architecture level is still to be explored.

This paper.

This paper presents an approach to defining non-functional concerns alongside the description of a pervasive computing system. These concerns are automatically mapped into the system implementation, using an aspect-oriented approach. Our contributions are as follows.

- We propose to extend our lightweight ADL, named DiaSpec [5], with annotations to express non-functional concerns, as a pervasive system architecture is being defined. A dedicated programming framework is generated from an architecture description. Additionally, non-functional annotations are mapped into aspect-oriented declarations. These declarations are used by

an aspect-oriented engine to inject code into the generated framework to enforce declared non-functional concerns.

- To enable aspects to be declared at the architecture level, we have developed an aspect-oriented language for ADLs, named DiaAspect. Our language provides support to manipulate the common set of architecture abstractions found in existing ADLs. This support takes the form of a join point model on top of which architectural crosscutting concerns are to be expressed. Because our model covers features common to most ADLs, we believe it is widely applicable.

- We describe how aspects at the architecture level allow powerful aspect weaving in existing middleware. This situation is due to the fact that aspect weaving is not performed on an arbitrary program, but on an automatically generated framework, whose structure is known. As a result, the aspect weaver may select the optimal join point projection depending on the architecture specification and the aspect code.

- We validate our approach by securing the communication between components and enforcing access control. First, we show how to express the distribution of certificates to components of a Remote Method Invocation (RMI) distributed system to secure the communication with the secure socket layer (SSL) protocol. Second, we illustrate how to easily enforce access control lists on the interfaces of the components of a system.

Outline.
The rest of this paper is organized as follows. Section 2 presents notions and concepts of aspect-oriented software development (AOSD) and ADL. It also introduces DiaSpec, our lightweight ADL on which we illustrate our approach. Section 3 introduces two security examples used throughout the paper to illustrate our approach. In Section 4, we describe DiaAspect, our dedicated aspect-oriented language for the DiaSpec ADL. In Section 5, we illustrate our language on the two examples introduced. In Section 6, we present our implementation. We discuss related works and conclude in Section 7 and Section 8.

2. BACKGROUND

Let us now introduce a few notions that are relevant to the rest of the paper. We present some ADL concepts that provide a basis to identify what is required to express aspect-oriented programming for ADLs. These requirements are used to define an aspect-oriented language dedicated to ADLs, and more specifically to DiaSpec. We describe how DiaSpec descriptions are used by a programming framework generator, named DiaGen, and the distributed system-based runtime support for DiaSpec descriptions, named DiaEnv. Finally, we introduce requirements for AOSD at the architecture level.

2.1 Architecture Description Languages

"A software architecture for a system is the structure or structures of the system, which consist of elements, their externally visible properties, and the relationships among
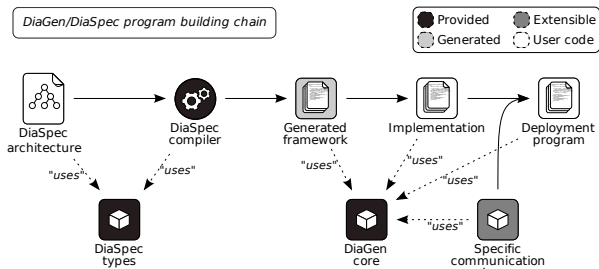

Figure 1: The DiaSpec development process.

them" [2]. Because a software system may not be completely represented as a single structure, it is usually modeled with multiple views.

ADLs allow architects to represent and specify systems in both human and machine readable languages. A precise semantics increases the coordination of multiple participants while providing tools to perform verifications on various properties and generate code and test cases.

The vast majority of ADLs use a component-and-connector idiom to represent the software system structure. Basic connectors, such as synchronous Remote Procedure Calls (RPC) and asynchronous messages, may be used to develop new connectors. In addition, ADLs may sometimes feature code generation tools and runtime support for the deployment of distributed systems. The mandatory feature, to be supported by a middleware, is a service discovery mechanism, allowing the dynamic binding of components.

2.1.1 DiaSpec

We now give a brief overview of our lightweight ADL to provide some background for our aspect-oriented language, DiaAspect. Notice that DiaSpec and DiaAspect are general enough to be introduced on top of most ADLs.

DiaSpec is an ADL that is part of a generative approach to supporting the implementation of pervasive computing systems. As illustrated in Figure 1, it features the DiaGen compiler that takes a DiaSpec specification and produces a typed programming framework. This framework closely guides the implementation of the software components. DiaGen performs various verifications of both the architecture specification and the component implementations. It partially generates a software layer, DiaEnv, to support runtime execution. In addition to performing runtime consistency checks, DiaEnv abstracts over the communication layer, allowing to transparently deploy a system implementation over multiple middlewares.[1]

DiaSpec models software architectures in terms of components and connectors. As illustrated in Figure 2, it is a dedicated ADL, where component interactions are included in component declarations. It differs from most ADLs where component interfaces are defined with ports. Nevertheless, this difference has no implication on our work. DiaSpec features three built-in connectors: commands, events and sessions.

- A command connector amounts to a synchronous RPC, allowing a one-to-one interaction.

---

[1]Currently, DiaEnv supports Web Services, RMI, local and SIP deployments.

```
component Device(String location) {}

component Sensor extends Device {}

component Actuator extends Device {}

component TemperatureSensor extends Sensor {
  provides event Temperature
                    to AirConditioning;
  provides event Temperature to FireManager;
}

component Fan extends Actuator {
  provides command Speed to AirConditioning;
  provides command OnOff to FireManager;
}

component FireManager {
  requires command OnOff from AirConditioning;
  requires command OnOff from Fan;
  requires event Temperature
                from TemperatureSensor;
}

component AirConditioning {
  provides command OnOff to FireManager;
  requires command Speed from Fan;
  requires event Temperature
                from TemperatureSensor;
}
icommand Speed {
  void setSpeed(int speed);
}

icommand OnOff {
  void on();
  void off();
}
```

Figure 2: A DiaSpec example specification

- An event connector corresponds to the well-known publish/subscribe paradigm, in which a publisher sends events to receivers registered for the corresponding type.

- A session connector defines an offer/answer session negotiation protocol [12], followed by a media session.

In addition to component inheritance, a component declaration includes attributes that enable component instances to be distinguished. The example in Figure 2 specifies a simple air conditioning system in which four components must be implemented: `TemperatureSensor`, `Fan`, `FireManager` and `AirConditioning`. The first two extend `Device` and thus inherit the attribute `location`. This attribute is used to refer to specific instances of components.

DiaGen and DiaEnv provide developers with a service discovery mechanism that filters component instances according to their type and their attribute values. A dedicated implementation of this mechanism is generated in conformance with the DiaSpec description. It corresponds to a typed version of existing service discovery mechanisms.

2.2 Aspect-Oriented Programming

The decomposition of software into small, meaningful, manageable and comprehensible parts has been a core idiom of software engineering for decades. A proper separation of concerns promotes reusability, traceability, adaptation, and comprehensibility. But, the relevance of concerns may vary with respect to roles: architects, developers and administrators. It may also vary depending on the stage of the software life cycle. Moreover, concerns may be constrained by the implementation language being used. For example, the object-oriented paradigm drives the decomposition of data structures into classes.

Despite the numerous software engineering approaches to system decomposition (*e.g.* libraries, modules, components, *etc.*), achieving a proper decomposition, where every concern is correctly modularized, is not possible in practice [14]. Some concerns are then spread out and mixed; these are said to be crosscutting the decomposition of the system.

AOSD is a software engineering approach that focuses on the identification and representation of crosscutting concerns, and their modularization in separate units, as well as their automated composition into a complete system.

AOP [7] is a language independent paradigm, where aspects encapsulate crosscutting concerns. An aspect associates an advice, the actual code of the concern, with pointcuts that refer to the regions in the base program where the advice is to be applied. An aspect weaver then realizes the coordination of the aspects with the base program, either statically, *e.g.* through static code inlining, or dynamically, *e.g.* using the host language's reflection mechanism.

While a significant body of work has focused on the proper decomposition of system architectures [1], it does not explicitly focus on crosscutting concerns. Properties such as security and QoS are inherently crosscutting. Moreover, these properties must be explicitly specified at design time to reason about them.

3. ILLUSTRATING EXAMPLES

Security is a key challenge in the development of pervasive computing systems. Even if most middlewares provide support for the implementation of security policies (*e.g.* encryption and authentication) developers must properly design systems to guarantee a correct implementation of these policies. Security typically impacts every aspect of a system. As a result, it must be dealt with at an early stage of the application design.

This section introduces two examples of non-functional security concerns that crosscut a system architecture. We revisit these examples in Section 5 to illustrate the expressiveness and effectiveness of our approach.

3.1 Managing Certificates for SSL communication

Authentication, confidentiality and integrity are key objectives for the security of pervasive computing systems. Their effective implementation depends on numerous factors, such as the network type, the level of trust between users, *etc.*

Let us consider the example of the RPC. It is a well-known mechanism to perform remote computations and to exchange messages in distributed systems. RMI is the Java application programming interface that performs the object-oriented equivalent of an RPC. While RMI offers a simple programming interface, it provides no security guarantee: RMI is built on top of the Java remote method protocol, which exchanges serialized Java objects in clear.

In this situation, good practices to achieve a secure design require the use of a secure channel. SSL is a wide spread protocol used to secure transmission in distributed systems. It performs authentication and encryption. Operationally, SSL requires participants to store certificates of trusted entities. Fortunately, the RMI API provides support for RMI over
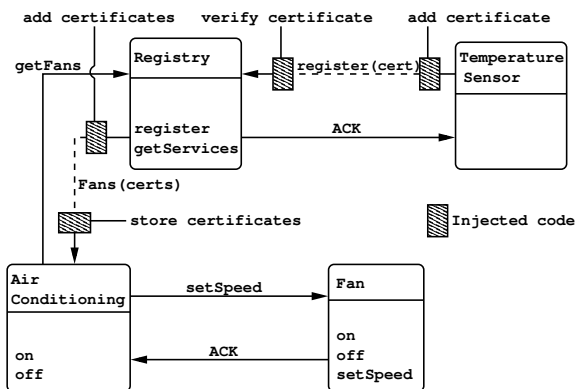
Figure 3: Example of the distribution of certificates

Figure 4: Enforcement of Access Control Lists

SSL. In practice, developers pass a reference to a *TrustStore* containing certificates of trusted entities to the RMI API that transparently performs authentication and encryption.

This approach is suitable for the client/server model, where every client has to hold the server certificate and the server to eventually have all client certificates. However, its application to multi agent systems, where distributed entities *(dis)*appear and *(un)*register dynamically, requires additional code to manage certificates.

In the example depicted in Figure 3, three components `AirConditioning`, `Fan` and `TemperatureSensor` are declared. The component `Fan` provides the interface `setSpeed` required by the component `AirConditioning` to regulate the temperature. Upon instantiation, instances of a component must register to the `Registry` and pass their certificates. In this example, a new instance of `TemperatureSensor` registers to the server. The `Registry` must then verify the certificate before the service can be successfully registered.

Whenever an instance of `AirConditioning` wants to invoke `setSpeed`, it must first obtain an instance of a `Fan` from the service discovery service. The `Registry` then returns proxies on instances of `Fans` with their certificates. `AirConditioning` may then communicate with any of these `Fans` via SSL.

As illustrated in Figure 3, the management of certificates for SSL communications typically crosscuts multiple aspects of a distributed system, *e.g.* registration and discovery, in both the component code and the built-in services (DiaEnv).

3.2 Enforcing Access Control Lists

The use of SSL connections combined with signed certificates allows the middleware to enforce authentication and encryption of communications in a distributed system. Nevertheless, the security provided by SSL communication is coarse grained. In a pervasive computing system, every entity must verify each access at a fine-grained level. The fact that an entity is authenticated does not mean that it has access to all resources.

For example in Figure 4, the `AirConditioning` may change the speed of a `Fan` by using the `setSpeed` method. This is a valid action and thus allowed by the `AuthorizationEnforcer`. On the contrary, the use of the `on` or `off` method should be denied. These methods should only be available for the `FireManager` (Figure 2, line 15). Even if DiaGen generates a programming framework for the `AirCondition-`
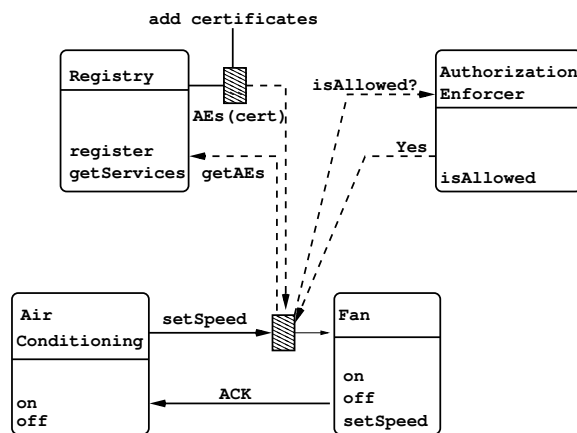
ing that only exposes a limited view of the `Fan` (where only used interfaces are accessible), a malicious developer might escape the programming framework and craft a request to access every interface exposed by the `Fan` component.

To forbid such uncontrolled access, the system must enforce access control lists upon requests on provided operations. As shown in Figure 4, the access list enforcement logic should be externalized from components. That is, on reception of requests, the receiver must query the access controller to verify that the caller is allowed to request a particular action from the callee.

Again, the implementation of the access control enforcement impacts multiple regions of the architecture; that is, every interface entry points.

4. THE DIAASPECT LANGUAGE

This section presents DiaAspect, an aspect-oriented language for the DiaSpec ADL. It first describes the join point model. A join point refers to a region in the DiaSpec architecture description and/or DiaSpec generated programming framework, where aspect code is injected.

Afterwards, we present our pointcut language. Pointcuts represent the occurrence of one or more join points. A pointcut is a predicate; it may or may not match the current join point. In addition, a pointcut may expose information specific to the underlying join points at runtime.

Finally, we introduce the DiaAspect language.

4.1 The Join Point Model

The DiaAspect join point model defines the set of events of interest in a system architecture description and its associated generated programming framework. AOP events are represented as messages in DiaEnv. For the sake of conciseness, only messages of interest between components of a DiaSpec architecture specification are listed here. Each listed message corresponds to two distinct join points in our model: one at the message emitter and one at the receiver. Figure 5 illustrate our model.

Components Registration.

The `register`, *respectively* `unregister`, message occurs when a component instance notifies a registry of the arrival of a component instance, *respectively* departure, in/from the system. Both the `register` and `unregister` messages have
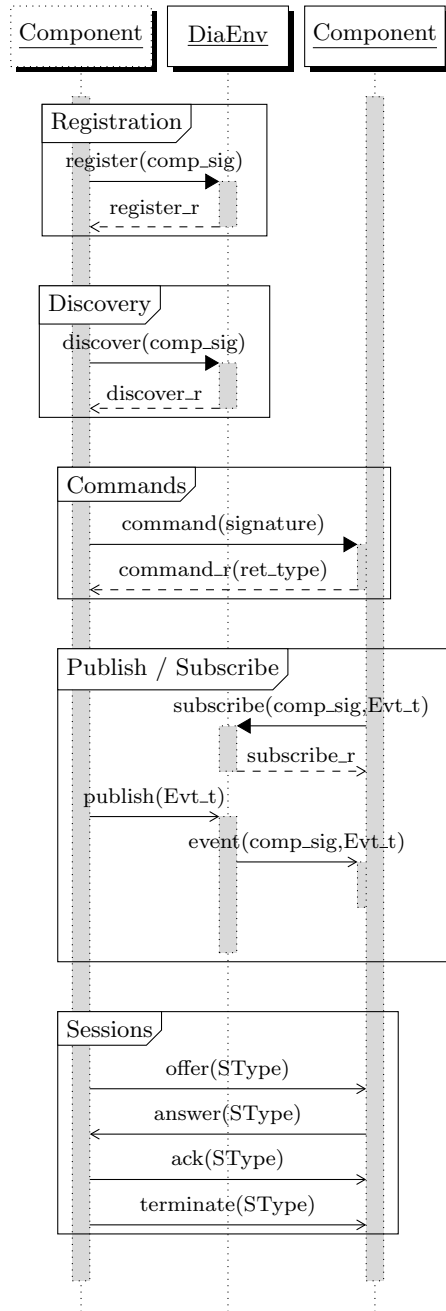
Figure 5: DiaAspect Join Point Model

one argument, the signature of the component arriving/leaving. Our model distinguishes the component issuing the (un)register message from the actual component that is arriving/leaving. The `register_r`, *respectively* `unregister_r`, message corresponds to the acknowledgment from the DiaSpec registry to a component, after a `register`, *respectively* `unregister`, message. These messages have no argument.

Component Discovery.

The `discover` message occurs when a component instance queries a registry for a component. A discovery is parameterized with the partial signature of the component to search for, that is, the component type and the attributes on which to apply a filter. On reply, the DiaSpec registry emits a `discover_r` message containing an array of components.

Commands.

The `command` message matches the call or the reception of a command. The corresponding join point takes the following arguments: the method's name, the argument's types and the return type. The `command_r` join point matches command reply and has a single argument, the return type of the method.

Event Subscription.

The `subscribe` message matches the subscription of a component instance to an event queue. Subscriptions have two arguments, the signature of the publisher and the type of events the subscribers are interested in. Note that the publisher (the parameter) is not the same component as the receiver of the subscription message. Indeed, in DiaSpec, a subscriber subscribes to the DiaEnv, not directly to the publisher. The response corresponds to the `subscribe_r` message that has no argument.

Events Publishing and Reception.

The `publish` message occurs when a component publishes an event. It is parameterized with respect to the event type and received by a registry. Consequently, the registry emits an `event` message to each subscriber. That message has two parameters, the signature of the publisher and the event type.

Sessions.

There are four kinds of `session` messages exchanged during the life-cycle of a session: `offer`, `answer`, `ack` and `terminate`. Each message has one parameter: the type of the session. That is, the type of the data that has been negotiated during the session establishment process.

4.2 The Aspect Language

Figure 6 presents the partial BNF of the DiaAspect language. We chose a syntax similar to AspectJ, an aspect-oriented system for Java. The benefits of reusing an existing syntax are well-known from both a user and an implementer perspective. The latter perspective is illustrated in Section 6 where AspectJ is shown to greatly simplify the implementation of our language.

Our language associates pointcuts with advice written in Java. An advice holds the implementation of crosscutting concerns and pointcuts select join points where an advice is to be executed. We first present our pointcut language.

Then, we introduce the runtime API supporting advice implementation with the information relative to join points matched at runtime.

4.2.1 The Pointcut Language

In the aspect-oriented paradigm, pointcuts act as join point selectors. In addition, to capture the occurrence of one or more join points, pointcuts may expose runtime information specific to these join points. The DiaAspect language proposes six kinds of pointcuts for which we distinguish two categories:

- Architectural pointcuts. These directly relate to artifacts defined by the architect in the system specification. That is, component declaration and component relationships (connectors).

- Built-in pointcuts. These correspond to built-in services provided by the DiaSpec runtime. That is, component registration, *etc.*

The following lists the pointcut featured in DiaAspect. We previously stated that a join point may catch the emission and/or reception of messages between DiaSpec components. Accordingly, pointcuts in DiaAspect distinguish join points on the emitter and/or the receiver side. If a pointcut is preceded by the `send`, *respectively* `recv`, keyword, it matches only the join point on the emitter, *respectively* receiver, side. If no keyword precedes, join points on both sides are matched.

register The `register` pointcut (Figure 6, line 27) matches the occurrence of `register` join points and the corresponding `register_r` join points, where the registered component matches the signature given as a parameter. For example, the pointcut `recv register (Service (..))` matches every `register` join points received, where the component instance extends the `Service` component.

unregister The `unregister` pointcut (Figure 6, line 29) captures the occurrence of `unregister` join points and the corresponding `unregister_r` join points, where the registered component matches the signature given as a parameter. For example, the pointcut `send unregister (* (..))` matches every `unregister` join point.

discover The `discover` pointcut (Figure 6, line 31) matches the `discover` and corresponding `discover_r` join points given a partial component signature (that is, the type of the component requested and the attributes on which the results are filtered). For example, the pointcut `recv discover (WebCam (Location loc == "Room B", ..))` matches the reception of every discovery operation for components of type `WebCam` with a specific location attribute.

command The `command` pointcut (Figure 6, line 35) captures all `command` and their respective `command_r` join points for a given method signature. For example, the pointcut `command (* OnOff.*(..))` catches every call of commands defined in the `OnOff` interface.

subscribe The `subscribe` pointcut (Figure 6, line 33) intercepts all `subscribe` calls and their respective `subscribe_r` joinpoints for a given component signature

and event type. For example, the pointcut `recv subscribe (* (..), Time)` matches every subscription to an event of type `Time`, regardless of the publisher.

publish The `publish` pointcut (Figure 6 line 37) matches the occurrence of `publish` join points for a given event type. For example, the pointcut `publish (Alert)` matches every published `Alert` event.

event The `event` pointcut (Figure 6 line 38) catches the occurrence of `event` join points for a given event type and a publisher signature. For example, the pointcut `send event (Sensor (..), *)` matches the sending of any kind of `event` following the publication by a publisher extending `Sensor`.

session The `session` pointcuts (`offer`, `accept`, `ack` and `terminate`) (Figure 6, line 38) match the occurrence of session join points matching a session type. For example, the pointcut `recv session:terminate(*)` catches the reception of every session termination.

Pointcuts select join points that correspond to messages exchanged between component instances. Pointcuts filter join points on their types and the values of their arguments. In our language, an aspect may further filter the messages of interest by specifying additional clauses alongside pointcuts.

from and **to** As stated before, join points selected by pointcuts relate to messages exchanged between DiaSpec components. Hence, pointcut arguments correspond to the specific content of these messages. In addition to filtering join points according to message type and content, one can further restrict the collected join points by using the `from` and `to` clauses. Given a component signature, the `from`, *respectively* `to`, clause (Figure 6 line 23) restricts collected join points to those emitted, *respectively* received, by components matching the signature in that clause. In `pointcut && from(Service-(Location *, ..))`, join point matching is limited to those emitted by components extending Service and with at least one attribute of type Location.

if To allow aspect developers to further specify the region where to trigger advice, DiaAspect features an `if` clause similar to the one of AspectJ. The advice only executes if the given expression holds true. That expression must be a Java boolean expression that may refer to the DiaAspect runtime API and variables bound in the pointcut definition.

4.2.2 The Runtime API

An advice in DiaAspect is developed in Java. To support developers in writing advice, DiaAspect features a runtime API that exposes similar features to those in AspectJ.

The `proceed` method is similar to the one in AspectJ. It is a virtual method that has the same signature as the pointcut matched, and is used in `around` aspects. Its execution evaluates the join point matched by the aspect. For example, inside an aspect on the sending of a `register` message, the execution of the `proceed` method in the advice resumes the execution of the `register`.

As in AspectJ, DiaAspect also provides an advice-visible variable `thisJoinPoint`. It exposes reflective information about the join point that triggered the advice. We extended

```
aspect around recv discover (* (..))
&& to (registry) {
  RemoteServiceInfo[] rsis = proceed();
    for (RemoteServiceInfo rsi: rsis) {
      rsi.setCert(registry.fetchCert());
    }
  returns rsis;
}

aspect around send discover (* (..)
&& from (service) {
  Proxy[] proxies = (Proxy[]) proceed();
    for (Proxy p: proxies) {
      service.storeCert(
        p.getRemoteServiceInfo().getCert());
    }
  return proxies;
}
```

Figure 7: DiaAspect code managing certificates in a RMI with SSL distributed system (excerpt)

the `thisJoinPoint` to expose the architecture specific information about the current join point in addition to Java semantic information about it. For example, in the case of an aspect on a command call, `thisJoinPoint.getCaller-Signature()` returns the signature of the caller component.

Because aspects may be developed independently of the DiaSpec implementation code, developers can not benefit from the support of a generated programming framework when writing advice. Still, to allow developers to publish events and execute commands, we expose the `Processor` API, that acts as a front-end to the generated framework. Hence, developers may write an advice that interacts with DiaSpec components, while still benefiting from runtime coherency checks provided by the framework.

5. EXAMPLES OF CROSSCUTTING CONCERNS REVISITED WITH DIAASPECT

This section revisits the two concerns discussed in Section 3 to illustrate DiaAspect and to show that it allows such concerns to be concisely modularized at the architecture level. We first return to the management of certificates to implement SSL communications. Afterwards, we revisit the enforcement of access control lists.

5.1 Managing Certificates for SSL Communications

Figure 7 presents the DiaAspect code that implements the distribution of certificates for SSL encrypted communications. The first aspect augments the behavior of the DiaEnv registry on service discovery to pass the certificates of the discovered services to the requesting component. The second aspect intercepts discoveries to store those certificates locally.

5.2 Enforcing Access Control Lists

Figure 8 contains the DiaAspect code to enforce access control lists on interfaces declared in a DiaSpec architecture. It contains a single aspect that intercepts any receiving DiaSpec command call (line 1). We assume that a component `AuthorizationEnforcer` performs the actual enforcement of an access rule. As a result, the pointcut must not match when the receiver is an `AuthorizationEnforcer` to avoid an infinite recursion (line 2).

```
aspect around recv call (* *::* (..))
    && to (!AuthorizationEnforcer()) {
AuthorizationEnforcerProxy ae =
  getAuthorizationEnforcer();
  if (ae != null
      && ae.isAllowed(thisJoinPoint))
  { return proceed(); }
  throw new
    DiaGenSecurityException(thisJoinPoint);
}
```

Figure 8: DiaAspect code enforcing access control lists (excerpt)

On the first invocation, the advice code (lines 2 to 10) obtains an `AuthorizationEnforcer` instance using service discovery through the DiaAspect runtime API. The advice calls the `isAllowed` method on the `AuthorizationEnforcer` (line 5). Depending on the result, the advice either throws a `DiaGenSecurityException`, notifying the caller of the failure, or proceeds with the original command call.

6. IMPLEMENTATION

Weaving is the process of coordinating the aspect code with non-aspect code, *i.e.* the base program. In our approach, the aspect code refers to the architecture description. However, instead of weaving the architecture description, we inject aspects in the implementation code: the programming framework generated by DiaGen and DiaEnv. This approach exploits an architecture description to adapt the weaving process to a given generated implementation support.

This section first describes the structure and organization of the Java framework generated for a DiaSpec specification. Then, we present how DiaAspect aspects are translated into AspectJ aspects, that are woven into the implementation code.

6.1 The DiaSpec Generated Framework Organization

Given a software architecture declaration, the DiaGen compiler generates a typed framework on which to develop the distributed application.

For each command interface declaration, DiaGen includes a Java interface defining those commands. Similarly, each type of events, a component may subscribe to, creates a Java interface defining a listener to that event. For each component definition, DiaGen includes an abstract class whose implementation must provide a method definition for the provided commands and event listeners. In addition, for each couple of components sharing provided/required connectors, DiaGen generates a proxy interface of the providing component that exposes a limited view of its functionalities to the requiring component. Proxies only provide the methods of the connected interfaces. For example, given the command interface `OnOff` defined in Figure 2, DiaGen generates a Java interface (Figure 9) that must be declared by both abstract classes `Service` and `ServiceProxy`.

To ensure implementation independence from a specific middleware, DiaGen and DiaEnv introduce an abstract layer, as depicted in Figure 10. This abstraction layer makes a DiaSpec implementation and its corresponding generated framework completely reusable to different middlewares. As

```
diaaspect::= (aspect_def | pointcut_def)*;

aspect_def::=
    'aspect' ID ('before'|'after'|'around') '(' signature ')'
    ('returns' type)? throw? ':' pointcut_ref advice;

pointcut_ref::= ID | pointcut_def;

advice::= '{' /* Java code + thisJoinPoint + proceed */ '}'

pointcut_def::= 'pointcut' 'ID '(' signature ')' ':' pointcut ';';

pointcut::= ('send' | 'recv')?
    (
        register
        | unregister
        | discover
        | subscribe
        | command
        | publish
        | event
        | session
    ) (&& from(component_signature)? (&& to(component_signature)?
    (&& if (expression))?
    ;

register::= 'register' '(' component_signature ')';

unregister::= 'unregister' '(' component_signature ')';

discover::= 'discover' '(' component_signature ')';

subscribe::= 'subscribe' '(' component_signature ',' event_type ')'

command::= 'command' '(' command_signature ')';

publish::= 'publish' '(' event_type ')';
event::= 'event' '(' component_signature ',' event_type ')';
session::= 'session' ':'
    ('*' | 'offer' | 'answer' | 'ack' | 'terminate')
    '(' session_type ')';

component_signature::= pattern '(' (arg_sig (',' arg_sig)*)? ')';
arg_sig::= (pattern pattern) | '..'; /* AspectJ like */
command_signature::=
    type pattern '.' pattern '(' (arg_sig (',' arg_sig)*)? ')';
event_type::= pattern;
session_type::= pattern;

pattern::= /* AspectJ like string pattern */
```

Figure 6: DiaAspect BNF

```
public interface OnOff {
    void on () throws DiaGenRemoteException;
    void off() throws DiaGenRemoteException;
}
```

Figure 9: The Java interface generated for the OnOff command interface declared in Figure 2

of today, DiaSpec supports local, RMI, Web Services and SIP technologies, in addition to our simulation environment DiaSim [4].

Given this structure, a received message first passes through the specific communication layer that formats it in a unified form before passing it to the core layer. The core layer unmarshals the message content to extract the sender information, the message type, (*e.g.* command, command return, event, *etc*), and its content. The extracted information is passed to the generated framework, which in turn dispatches it to the appropriate component proxy and finally to the
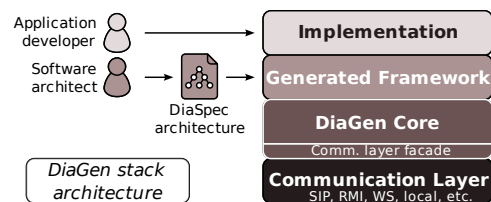

Figure 10: DiaSpec/DiaGen generated code structure.

component implementation. Similarly, command calls and event publishing follow the inverse path. One can note that this particular implementation offers multiple regions to intercept similar join points. The next section shows how to benefit from this strategy to optimize aspect weaving.

6.2 DiaAspect Aspects Weaving

The DiaAspect language describes aspects that coordinate join point regions. These join points refer to artifacts, *i.e.* components, connectors and built-in services, which are defined in a DiaSpec architecture description and its runtime environment (Section 4). An advice is defined purely in Java (exception made of the `proceed` keyword). The compilation and weaving process must connect the architecture pointcuts with the generated framework and the implementation code. To do so, DiaAspect aspects are translated into AspectJ code that is woven into the generated code and DiaEnv.

Translating DiaAspect aspects into AspectJ code amounts to projecting the DiaAspect join point model into the programming framework generated by DiaGen. DiaAspect pointcuts are translated into AspectJ method call pointcuts on the methods of the Java interfaces generated by DiaGen. For example, the DiaAspect pointcut

```
command (* OnOff::* (..))
```

intercepts any command declared in the command interface `OnOff` and is translated into the following AspectJ pointcut,

```
call(* spec.package.name.interfaces.OnOff.*
(..)).
```

The translation of the DiaAspect `from` and `to` clauses depends on the pointcut type, *i.e.* `send` or `recv` pointcuts. In the case of `send` type pointcuts, for example in

```
send command(* *::* (..))  && from(Manager())
&& to(Service)
```

the `from` clause is translated to limit AspectJ pointcut matching, to callers extending the abstract Java class `Manager`. This is done by using the following AspectJ code

```
this(spec.package.name.components.Manager)
```

and the following the invocation

```
this(spec.package.name.proxies.ServiceProxy)
```

This results in restricting the object on which a method is called to `ServiceProxy` classes.

In the case of a `recv`, the pointcut is limited to a caller of type proxy. In addition, the object on which the method is called must extend the corresponding abstract class.

Figure 11 presents the AspectJ code generated for the DiaAspect code from the example on enforcing access control lists. The DiaAspect aspect in Figure 8 is translated into an AspectJ aspect. That aspect is woven around the execution of the `commandReceived` method that is called whenever a DiaSpec command call is received by a component instance providing that command. The pointcut limits the matching to classes, extending the `Service` class (the base class for every component) and excluding components of type `AuthorizationEnforcer`. The advice code is left unchanged.

The generated framework is composed of multiple layers. Depending on the aspect code, it is possible to modify the

```
Object around() throws DiaGenException:
call(Object commandReceived(
  RemoteServiceInfo, String, Object...)
throws DiaGenException)
&& target(Service+)
&& target(!AuthorizationEnforcer+){
  AuthorizationEnforcerProxy ae =
    getAuthorizationEnforcer();
  if (ae[0] != null
      && ae.isAllowed(thisJoinPoint)) {
    return proceed(rsi, callee, method);
  }
  throw
    new DiaGenSecurityException(thisJoinPoint);
}
```

Figure 11: AspectJ code generated for the DiaAspect aspect from the example on enforcing access control lists (excerpt)

aspect projection (weaving) to shortcut these layers. This is done to optimize performance of the applications. For example, consider a DiaAspect aspect intercepting all receiving calls for a given command. Instead of weaving the aspect in the generated framework layer, we can inject it directly into the communication layer (*i.e.* RMI, Web Service, *etc*). When the advice code does not call the `proceed` method, this optimization avoids unmarshalling the call up to the application layer. Similarly, weaving can be specialized depending on the specific communication layer and of the aspect code.

7. RELATED WORK

This section briefly reviews approaches related to the modeling of crosscutting concerns in architecture design and in pervasive computing.

Multi-dimensional separation of concerns [14] shows how the software artifacts, corresponding to different concerns (*a.k.a.* hyperslices), can be merged to generate a full application. This is the approach chosen by subject-oriented programming [3], where hyperslices are pieces of code (*e.g.* partial class hierarchies). AOP [7] is quite similar, but it is asymmetric: it considers the structure of a base program and it provides pointcut languages to specify where another code crosscuts the base program and the corresponding pieces should be woven.

DAOP-ADL [10] is an XML-based architecture description language that integrates aspects as first class entities of architecture description. The interconnection of aspects with components in DAOP-ADL is specified as evaluation rules on the interfaces of the architecture. Similarly, Pessemier *et* al. integrate aspect entities in the Fractal ADL, by extending the component membrane to support aspect weaving [9]. PRISMA [8] is an ADL that integrates an aspect-oriented approach directly in the component-and-connector approach: aspects are modeled as components, allowing direct reuse of consistency checks and code generation tools. In comparison to these works, our aspect language DiaAspect is not limited to the artifacts of the ADL but also allows aspects to coordinate with the built-in services, like component registration and discovery provided by the generated framework.

Ren and Taylor present an extension of xADL, a XML ADL, for modeling security at the architecture level [11].

Their extension permits architects to control component instantiation, interface access, and data flow by annotating the components and connectors. Vigil [6] is a pervasive computing middleware that enforces a trust-based security policy. To this end, it exposes services such as a certificate controller, a role assignment manager and a security agent. It also augments communication stubs with policy enforcement code. In comparison to this work, our approach allows to inject the necessary code to enforce such policies without modifying the system architecture. Moreover, by specifying these changes at the architecture level, our implementation is reusable with multiple middleware.

8. CONCLUSION AND FUTURE WORK

This paper presented our approach to expressing non-functional properties of system architectures using an aspect-oriented approach. The specification of these properties, *e.g.* security and QoS, impacts every aspect of a software system and cannot be properly expressed in the traditional component-and-connector idiom.

To overcome this limitation, we proposed DiaAspect, an aspect-oriented language dedicated to distributed ADLs and their runtime support. We developed DiaAspect on top of both the component-and-connector idiom, and common runtime services provided by relevant ADLs. We showed that DiaAspect is expressive enough to implement concrete solutions on two widespread security problems: the distribution of certificates to encrypt communication and the enforcement of access control lists in distributed systems.

We presented the implementation of the DiaAspect aspect weaver. It injects aspect code into the programming framework generated for DiaSpec specifications. We implemented our weaving process by translating DiaAspect code into aspects written in AspectJ, a well-known aspect-oriented system for the Java programming language. We also demonstrated how our translation scheme allows to modify the DiaAspect pointcuts projection to optimize the woven code according to an architecture specification and the structure of generated frameworks.

In the future, we plan to extend the DiaAspect pointcut model to capture new join points; currently, our pointcut language only captures join points related to the relationships between components or to the runtime built-in services. A number of join points would be of particular interest: component declaration, component instantiation, connector declaration, *etc.*

Also, we plan to integrate features similar to AspectJ inter-type declaration, for the attributes of DiaSpec components and their connectors. Another line of work we intend to explore is to define a structural and runtime model of aspect injection in system specification without introducing new artifacts in the ADL. This would allow us to transparently reuse static and runtime consistency checks performed by the DiaSpec tool suite.

9. REFERENCES

[1] J. Araujo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdoğan. Early aspects: The current landscape. Technical report, Lancaster University, 2005.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 1997.

[3] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *OOPSLA '93: Proceedings of the 8th annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM.

[4] W. Jouve, J. Bruneau, and C. Consel. DiaSim: A parameterized simulator for pervasive computing applications (demo). In *PERCOM'09: Proceedings of the 7th IEEE Conference on Pervasive Computing and Communications*, Galveston, Texas, USA, 2009. Demo.

[5] W. Jouve, N. Palix, C. Consel, and P. Kadionik. A SIP-based programming framework for advanced telephony applications. In *IPTComm'08: Proceedings of the 2nd Conference on Principles, Systems and Applications of IP Telecommunications*, Heidelberg, Germany, 2008. LNCS. Best Student Paper Award.

[6] L. Kagal, T. Finin, and A. Joshi. Trust-based security in pervasive computing environments. *Computer*, 34(12):154–157, 2001.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[8] J. Pérez, N. Ali, J. A. Carsí, I. Ramos, B. Álvarez, P. Sanchez, and J. A. Pastor. Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. *Inf. Softw. Technol.*, 50(9-10):969–990, 2008.

[9] N. Pessemier, L. Seinturier, and L. Duchien. Components, ADL & AOP: Towards a common approach. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE) at ECOOP'04*, 2004.

[10] 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 2nd international conference on Generative programming and component engineering*, pages 118–137, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[11] J. Ren and R. N. Taylor. A secure software architecture description language. In *Workshop on Software Security Assurance Tools, Techniques, and Metrics*, 2005.

[12] J. Rosenberg and H. Schulzrinne. An offer/answer model with session description protocol (SDP). RFC 3264 (Proposed Standard), 2002.

[13] Sun Microsystems. Entreprise Java beans specification, 2007.

[14] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.