Use Case Level Pointcuts

Jonathan Sillito, Christopher Dutchyn,
Andrew David Eisenberg, and Kris De Volder

Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
{sillito,cdutchyn,ade,kdvolder}@cs.ubc.ca

Abstract. Software developers create a variety of artifacts that model the behaviour of applications at different levels of abstraction; e.g. use cases, sequence diagrams, and source code. Aspect-oriented programming languages, such as AspectJ, support the modularization of crosscutting concerns at the source code level. However, crosscutting concerns also arise in other behavioural models of software systems. We provide a new aspect language, AspectU, which supports modularization of crosscutting concerns in the use-case model. Further, we provide a prototype tool that partially translates AspectU aspects into AspectJ aspects. To facilitate this translation we introduce a third aspect language, AspectSD, which targets the sequence-diagram model. AspectU together with our translation tool allows developers to express advice using use case level concepts while still affecting the runtime behaviour of a system, yielding a natural and intensional expression of some concerns.

1 Introduction

When limited to a hierarchical decomposition of a system as supported by object-oriented languages, some concerns of interest cannot be cleanly modularized. Instead, the implementation of the concern is scattered across multiple modules, tangled with the primary concerns of these modules. Aspect-Oriented Software Development (AOSD) has focused on improving the modularity of these *crosscutting concerns*. One prominent AOSD tool is *AspectJ* [9], a general-purpose aspect-oriented programming language. It extends the *Java*[1] programming language with join points, pointcuts, and advice [11]. These additions enable aspects to be written that gather the otherwise scattered and tangled source code into one location.

We believe crosscutting concerns exist in other behavioural models as well, including the use-case and sequence-diagram models. Existing aspect-oriented programming languages operate at the level of implementation and support pointcuts reflecting programming language constructs such as classes, methods, fields, and objects. When limited to such constructs, it can be difficult to understand the effects of the pointcuts in terms of other behavioural models.

[1] Java is a trademark of Sun Microsystems, Inc.

We present a new aspect language, *AspectU*, for modularizing crosscutting concerns within the use-case model. AspectU provides a join-point model based on elements in this model: use cases, steps, and extensions. Many crosscutting concerns that can be naturally expressed in terms of the use cases can be formally expressed using AspectU. AspectU aspects capture changes to a system's dynamic behaviour as expressed in its use cases. These behavioural changes are expressed as steps and extensions added to or replacing elements of the existing behaviour.

In addition, we present an exploration of aspect modularity between models. To this end, we have implemented a tool for partially translating between AspectU and AspectJ. In particular, the translation focuses on translating AspectU pointcuts. Our translation makes use of a third language, *AspectSD*, that targets the sequence diagram model. AspectSD bridges AspectU and AspectJ in the same way that sequence diagrams bridge the use-case model and the implementation. Our translation relies on a mapping between the use case model and the sequence-diagram model, as well as correspondence between the sequence-diagram model and the implementation. Given such a mapping, our tool translates an AspectU pointcut into an AspectSD representation, and then into AspectJ advice that identifies the join points in the implementation corresponding to the specified join points in the use case. In addition to identifying join points, the generated AspectJ advice can trigger user supplied Java code, bind objects for that code to operate on, and control the flow of the application in a number of ways.

Using AspectU and our translation tool together allow a developer to write aspects using use case level pointcuts—affecting both the use-case model and the behaviour of the running system. We claim that for many concerns such a pointcut will be more natural and intensional than a corresponding pointcut in AspectJ or another (source level) aspect-oriented programming language.

We present the AspectU language in detail in Section 2. Two complete AspectU examples are shown in Section 3. The AspectSD language is presented in Section 4. The details of our translation between aspect languages is in Section 5. We discuss the benefits of our approach in Section 6, and conclude with a discussion of related work in Section 7.

2 The AspectU Language

AspectU extends the use-case model with support for modularizing crosscutting behaviour. This support makes it possible to define additional behaviour at certain points in the model. Crosscutting in AspectU is based on a small set of constructs based on AspectJ's constructs: *join points*, which are points in the model, *pointcuts*, which are a means to identify join points, and *advice*, which is a means of affecting the behaviour at the join points. Taken together, these constructs define a *join-point model* that specifies how the crosscutting behaviour relates to the underlying use-case model.

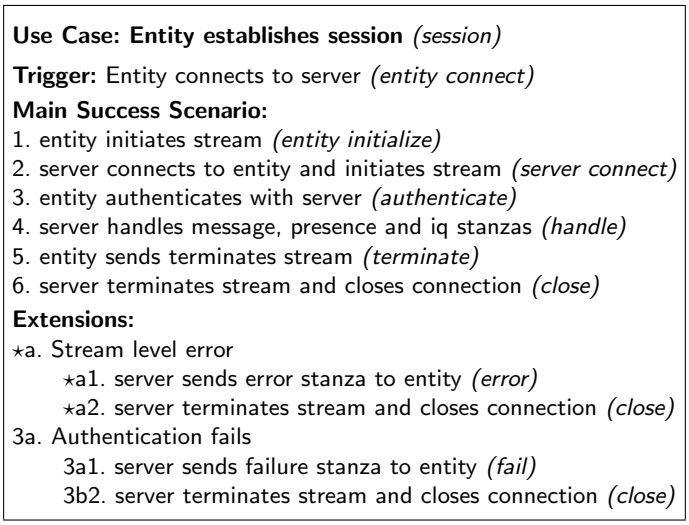

Fig. 1. Use case describing the behaviour of the system when an entity (a client or other server) establishes a session with a XMPP server.

This section contains several small examples targeting use cases (see Figures 1, 3, and 4) based on the XMPP [15] specification. This specification describes a set of client server-messaging protocols. The use cases we present in this paper describe parts of those protocols in terms of steps and extensions. More details on the XMPP specification, along with more complete examples are presented in Section 3. These use cases are simplified slightly for presentation convenience; primarily that not all use-case extensions are shown. Also, we provide an identifying name next to each step, each use case, and some extensions.

One use case, shown in Figure 1, indicates to the steps involved when a client or other server connects to a server. It consists of six steps: the first three involve the entity connecting and authenticating with the server. The fourth step handles each stanza, or communication unit, between the entity and the server. There are several different kinds of stanzas; each is handled in a different way as documented in separate use cases. Finally, when the entity wishes to disconnect, the last two steps are performed. This use case also contains two extensions. The first extension is triggered whenever there is a stream-level error anywhere in the use case, and ensures that the server attempts to cleanly terminate the connection. The second extension specifies that the session should be terminated when authentication fails.

2.1 Join Points

Use cases are a behavioural model of a system, and can be thought of as something that can be *executed*. Elements in the model are nested: a use case is made up of extensions and steps; an extension is made up of steps. This can be taken

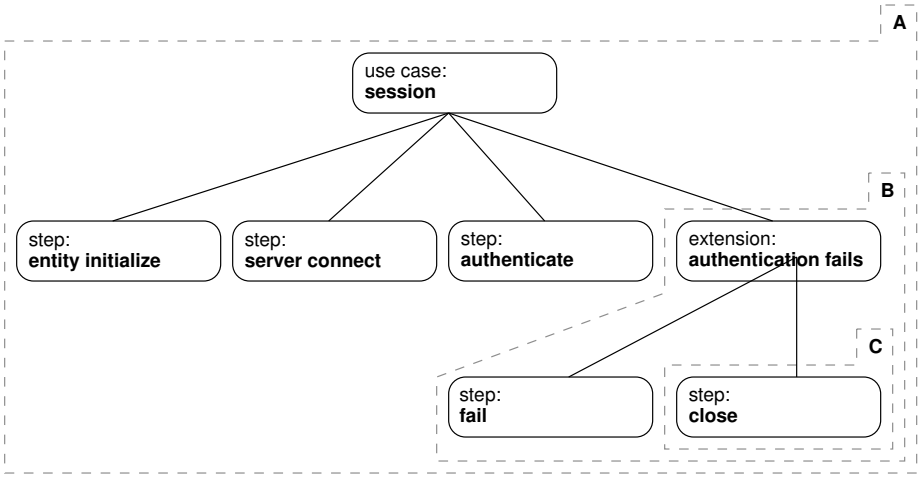

Fig. 2. A tree describing one possible execution of the *session* use case shown in Figure 1. The tree captures the execution corresponding to an authentication failure. The enclosed areas labelled **A**, **B** and **C** refer to three subtrees rooted at the use-case *session*, the extension *authentication fails* and the step *close* respectively.

further in that steps may be further elaborated as other use cases. Because of this nesting property of the model, each execution of a use case can be thought of as forming a tree. Execution order is represented in left-to-right, depth-first traversal of this tree.

As an example, one possible execution of the *session* use case (see Figure 1) is shown in Figure 2. In this particular execution, authentication fails and so steps **4**, **5**, and **6**, and extension ∗**a** are never executed. The system behaviour captured by this execution of the use case is broken down as three steps followed by one extension, which is is further broken down as two additional steps.

Join points in AspectU are elements in the execution of the use-case model. These points can be considered as subtrees in the execution of that model. In particular there are three types of join points: use-case, extension, and step join points. The entire subtree labelled **A** in Figure 2 corresponds to a join point of type use case; the subtree labelled **B** corresponds to a join point of type extension, and the subtree labelled **C** corresponds to a join point of type step.

2.2 Pointcuts

An AspectU pointcut identifies a set of join points. Pointcuts can be thought of in terms of matching certain join points (subtrees) in the execution. Three primitive pointcuts, which can be composed to form more sophisticated pointcuts, are defined:

1. `usecase(id)`—matches the join point for any *use-case* subtree with name matching `id`,

2. `extension(id)`—matches the join point for any *extension* subtree with name matching `id`, and
3. `step(id)`—matches the join point for any *step* subtree with name matching `id`.

The identifiers used as arguments in the above primitive pointcuts can contain the wild-card character `*`, which matches any substring. For example, `step(send*)` would match the join point for any step with a name beginning with *send*.

Pointcuts can be combined using *and* (`&&`) and *or* (||) operators. The *and* operator combines two pointcuts to build a new pointcut that matches any subtree matched by one of the pointcuts and lies *within* a subtree matched by the other pointcut. For example,

```
usecase(session) && extension(authentication fails)
```

identifies subtrees that are either (a) matched by the pointcut `usecase(session)` *and* lie within a subtree matched by the pointcut `extension(authentication fails)`, or (b) matched by the pointcut `extension(authentication fails)` *and* lie within a subtree matched by the pointcut `usecase(session)`. For the execution tree shown in Figure 2, the example pointcut identifies the subtree labelled **B**, because it is matched by `extension(authentication fails)` and is within a subtree matched by `usecase(session)` (i.e. it is withing subtree **A**).

The *or* operator combines two pointcuts to build a new one that matches execution trees where either subordinate pointcut matches. For example,

```
usecase(session) || extension(authentication fails)
```

identifies execution trees that are in use-case *session* or in extension *authentication fails*. For the execution tree shown in Figure 2, the previous pointcut identifies the subtree labelled **A** as well as the subtree labelled **B**.

The *and* and *or* pointcut combinators allow powerful effects when used in concert with wild cards. The pointcut

```
step(deliver*)
```

would apply to the execution subtrees corresponding to the *deliver presence* step in the *handle presence* use case (see Figure 4), as well as the *deliver message* step in the *handle message* use case (see Figure 3). However when this pointcut is combined as follows

```
usecase(handle message) && step(deliver*)
```

only execution trees corresponding to the *deliver message* step in the *handle message* use case are matched by the pointcut.

In addition to identifying join points, a pointcut can also provide access to values in the execution context of those join points. This is done using the `bind(bind-id, entity-id)` pointcut, which binds the use-case entity named `entity-id` to the identifier `bind-id`. This pointcut allows values associated with the matching join point to be available within the advice body. For example,

```
usecase(session) && bind(m,message)
```

provides a name, `m`, for the message entity within the *session* use case. As in this example, `bind` is most often used with *and* and *or* operators. For a complete example of this, see the privacy aspect in Figure 6, discussed in Section 3.

2.3 Advice

Advice is a mechanism used to declare that certain *additional* behaviour should execute at each of the join points in a pointcut. A piece of advice has three parts:

1. a pointcut indicating *where* the additional behaviour should be performed;
2. an advice body describing *what* the additional behaviour is, expressed in terms of steps and extensions; and,
3. a qualifier indicating *how* the additional behaviour combines with the behaviour at the join point.

AspectU supports three types of qualifiers, with meanings analogous to the corresponding AspectJ qualifiers. Each qualifier has a `binding-list` associated with it, which specifies the bound entities available in the advice body:

1. `before(binding-list)`—denotes advice to apply immediately before the execution of the matched subtrees,
2. `around(binding-list)`—denotes advice to apply around, and possibly instead of, the matched subtrees (more on this below), and
3. `after(binding-list)`—denotes advice to apply immediately after the execution of the matched subtrees.

A simple example of a complete piece of advice (with comments on the right identifying the advice parts) is

```
after(m) :                                          // qualifier
   step(handle*) && bind(m, message)                // pointcut
{                                                   // body
   steps:
      - server logs delivery of message m
}
```

The additional behaviour given in advice can combine with the pre-existing behaviour in three ways. First, advice can be strictly additive with respect to the normal execution. That is it can simply add behaviour at the join points identified by the pointcut. This can be done with `before` or `after` advice to specify additional steps to be applied at entry to or exit from a join point (subtree).

Second, advice can influence the behaviour in a way that is not strictly additive. To that end `around` advice has the special capability of selectively preempting the normal execution at the join point. An `around` advice can, alternatively, allow the execution to continue normally by including a step named `proceed`. This special step, analogous to AspectJ's `proceed()`, is used only in `around`

Use Case: Server handles message stanza *(handle message)*

Trigger: Entity sends message *(send)*

Main Success Scenario:

1. server processes and verifies message *(verify)*
2. server determines recipient of message *(determine recipient)*
3. server delivers message to recipient *(deliver message)*

Extensions:

2a. Non-local recipient
    2a1. route message *(route)*

2b. No such client
    2b1. reply with 'recipient unavailable' *(error)*

3a. Delivery failed *(delivery failed)*
    3a1. reply with 'recipient unavailable' *(error)*

Fig. 3. Use case describing how an XMPP server handles a message stanza: a simple push scheme is followed.

advice. A `proceed` step instructs the execution to proceed into the subtrees that the advice surrounds. An `around` advice lacking a `proceed` step replaces the original behaviour of the use case. One example of this is shown in the storage aspect (see Figure 5). That example's `around` advice introduces a step that executes in place of the execution of the *delivery failed* extension.

Third, advice can introduce use-case extensions. Like normal use-case extensions, this specifies behaviour that is executed when a certain condition is met. In AspectU, an added extension can end with a step named `rejoin`, which instructs the execution to return to (i.e. rejoin with) the use case at the location the advice was started from. If no `rejoin` is specified, the extension is understood to *terminate* the use case. The result is that the subtree corresponding to the extension is the last in that particular execution of the use case. An example of this is shown in the privacy aspect (see Figure 6). In the event that the privacy check fails, the control will pass to the added extension. As this extension does not rejoin, the remaining steps in the use case will be skipped.

Taken together, join points, pointcuts and advice support the expression of crosscutting behaviour. Aspects package these constructs in a modular way. Two complete example aspects are presented in the next section.

3 AspectU Examples

Throughout this paper we use examples based on a set of XML protocols and technologies that enable entities (clients and servers) to exchange communication units called stanzas. Stanzas can be messages, presence, and other structured information. The Internet Engineering Task Force has formalized the core protocols under the name *Extensible Messaging and Presence Protocol* (XMPP). The official documentation is in several parts or layers. The base specification is called *XMPP Core* [15].

Use Case: Server handles presence stanza. *(handle presence)*

Trigger: Entity sends presence *(send)*

Main Success Scenario:

1. server processes and verifies presence *(verify)*
2. server obtains entity's subscription information *(subscription)*
3. server determines recipients based on subscription information *(determine recipients)*
4. server delivers stanza to each recipient *(deliver presence)*

Extensions:

1a. Presence probe
    1a1. server determines target of probe *(determine target)*
    1a2. server obtains target's last reported presence *(obtain presence)*
    1a3. server replies to probe with this presence *(reply)*

4a. Non-local recipient
    4a1. route presence *(route)*

Fig. 4. Use case describing how an XMPP server handles presence stanza. The basic approach is based on a subscribe and broadcast scheme.

Based on XMPP Core specifications, we have developed several use cases. The first of these was presented in Section 2. Two more are given in Figures 3 and 4. The use case shown in Figure 3 details the steps taken to handle a message stanza sent by a connected entity. The *handle message* use case comprises three steps: verification, addressing, and delivery. If addressing indicates that a message cannot be locally delivered, extension **2a** is triggered to route the message to a remote location. Addressing could fail because the recipient does not exist, which is handled by extension **2b**. Delivery can fail because the recipient is not currently connected, which is handled by extension **3a**.

The use case shown in Figure 4 details the steps taken to handle presence stanzas sent by a connected entity. Presence stanzas are used for communicating status (e.g. online, offline, or away) between entities. While handling presence stanzas, the server follows a publish-subscribe scheme in which the information is sent to all subscribed entities. An exception to this is a presence probe which is a query for a particular entity's presence. Presence probes continue to be handled in extension **1a**.

XMPP Core is intended to provide a general framework for building messaging applications. One such application is instant messaging (IM) similar to AIM or ICQ. A separate specification document, XMPP Instant Messaging (XMPPIM) [16], extends XMPP Core with features needed to support such instant-messaging applications. In addition to adding more use cases the XMPPIM specification adds some concerns that crosscut the XMPP Core use cases. Two examples of such crosscutting concerns are from the *storage* and *privacy* features. We present each these as examples of AspectU aspects in the following two subsections. Expressing these concerns in AspectU, rather than directly modifying the affected use cases, is natural—it supports a modularization consistent with the specification.

```
aspect storage {
    around(stanza) : usecase(handle message) &&
        extension(delivery failed) && bind(stanza,message)
    {
      steps:
         - server stores stanza for later delivery (store)
    }

    before(client) : usecase(session) && step(handle) &&
        bind(client,entity)
    {
      steps:
          - server delivers any stored messages to client (deliver)
    }
}
```

Fig. 5. An AspectU aspect expressing the storage concern.

3.1 Storage Aspect

Message storage is a store-and-forward feature similar to that found in email servers. Describing the effect of this feature in terms of the *session* and *handle message* use cases described above is straightforward: when handling a message for a local recipient, if they are offline (i.e. in extension **3a**) then store the message; and, when a client connects to and successfully authenticates with the server (i.e. before the *handle* step) then deliver any stored messages.

This concern, written in AspectU, (see Figure 5) is similarly straightforward. It contains two pieces of advice, one for each new behaviour: the first stores undeliverable messages, and the other delivers deferred messages once a new connection is established. Each of the pieces of advice contains a `bind()` in the pointcut. In both cases this identifies objects that are referred to in the advice body. The `bind()` in the `around` advice, for example, states that the stanza referred to in the added step is the message object referred to in the use case. It is noteworthy that this advice does not contain a `proceed` step.

3.2 Privacy Aspect

The aspect in Figure 6 packages the behaviour needed to implement crosscutting associated with the privacy feature. Privacy, like the message storage concern, is introduced in the XMPPIM specification as a concern layered on top of the core protocol. The privacy concern deals with a client's ability to limit communication to or from other users. Supporting privacy requires the addition of several use cases for managing privacy lists, but these are already well modularized. But, it also introduces some additional required behaviour crosscutting the *handle message* and *handle presence* use cases shown in Figures 3 and 4 respectively. Rather than modify the affected use cases directly, AspectU supports modularization of these concerns.

```
aspect privacy {
   before (user, stanza) :
   (step(deliver message) && bind(user, recipient) &&
       bind(stanza, message)) ||
   (step(deliver presence) && bind(user, entity) &&
       bind(stanza, presence))
   {
     steps:
         - server verifies stanza against user's privacy settings
           (check privacy)

     extensions:
         - name: privacy check failed
           source: check privacy
           steps:
               - silently drop stanza (drop)
    }
}
```

Fig. 6. An AspectU aspect expressing the privacy concern.

The privacy aspect in Figure 6 implements this additional behaviour. The pointcut matches executions of the *deliver message* step from the *handle message* use case and executions of the *deliver presence* step from the *handle presence* use case. The body adds a step before each of the identified steps and also adds an associated extension. The inserted step involves verifying that the stanza can be sent without violating the appropriate privacy settings. The extension ensures that if the privacy check fails then the stanza is silently dropped rather than delivered.

The body of the privacy aspect is written in terms of a user and a stanza: the arguments to the qualifier. There are some differences between how privacy should be enforced in the *handle message* use case and in the *handle presence* use case. In the *handle message* use case, the check verifies that the privacy rules of the recipient allow the given message. In the *handle presence* use case, the check verifies that the privacy rules of the connecting entity allow the given presence notification to be sent. AspectU's name binding mechanism allows the body of the privacy aspect to be general enough to apply privacy in both of these cases. For handling messages the `bind` statements in the pointcut support this by mapping *recipient* to *user* and *message* to *stanza*. Similarly, for handling presence, *entity* is mapped to *user* and *presence* is mapped to *stanza*.

The AspectU language is supported by a tool that takes as input use cases (stored in a machine-readable format, based on the *yaml* [4] mark-up language) and AspectU aspects and produces transformed use cases. This tranformation process is sometimes called called *weaving*. Figure 7 shows the results of weaving both the privacy and storage concerns into the message handling use case. The steps and extensions shown in bold represent the additions made by the aspects.

Use Case: Server handles message stanza *(handle message)*

Trigger: Entity sends message *(send)*

Main Success Scenario:

1. server processes and verifies message *(verify)*
2. server determines recipient of message *(determine recipient)*
3. server verifies message against recipient's privacy settings (check privacy)
4. server delivers message to recipient *(deliver message)*

Extensions:

2a. Non-local recipient
    2a1. route message *(route)*
2b. No such client
    2b1. reply with 'recipient unavailable' *(error)*
3a. Privacy check failed
    3a1. silently drop message (drop)
4a. Delivery failed *(delivery failed)*
    4a1. server stores message for later delivery (store)

Fig. 7. Illustrates the effect of weaving the privacy and storage aspects into the *handle message* use case. Added behaviour added by the aspects is shown in bold.

Notice that in step number 3, *stanza* and *user* have been replaced by *message* and *recipient*.

While this can be a useful tool, ultimately the goal is not to modify the use case but rather to keep these concerns separate. Given this separation it is possible to pursue modularization between models, as discussed in the next two sections.

4 The AspectSD Language

For our purposes a *sequence diagram* is a sequence of messages along with the objects that send and receive them. Figure 8 contains a sample sequence diagram. During the design of a system, sequence diagrams may be used to refine the behaviour specified by the use cases in terms of object interactions.

Each sequence diagram corresponds to a single scenario of a use case and, therefore, to a single execution tree. For example, the sequence diagram shown in Figure 8 corresponds to the main success scenario of the *handle message* use case. Sequence diagrams can be viewed as a further elaboration of the behaviour captured by a use-case execution tree in terms of messages between objects.

Like use cases, we store sequence diagrams in a machine readable format based on the yaml mark-up language. Our format for storing sequence diagrams supports various annotations that document what sequences of messages correspond to which use-case steps. Given this mapping it is possible to think about a sequence diagram as the fringe of one execution tree. This is illustrated in Figure 9 where, for example, the execution of the *verify* step is decomposed as the execution of messages $m_1 \ldots m_4$.

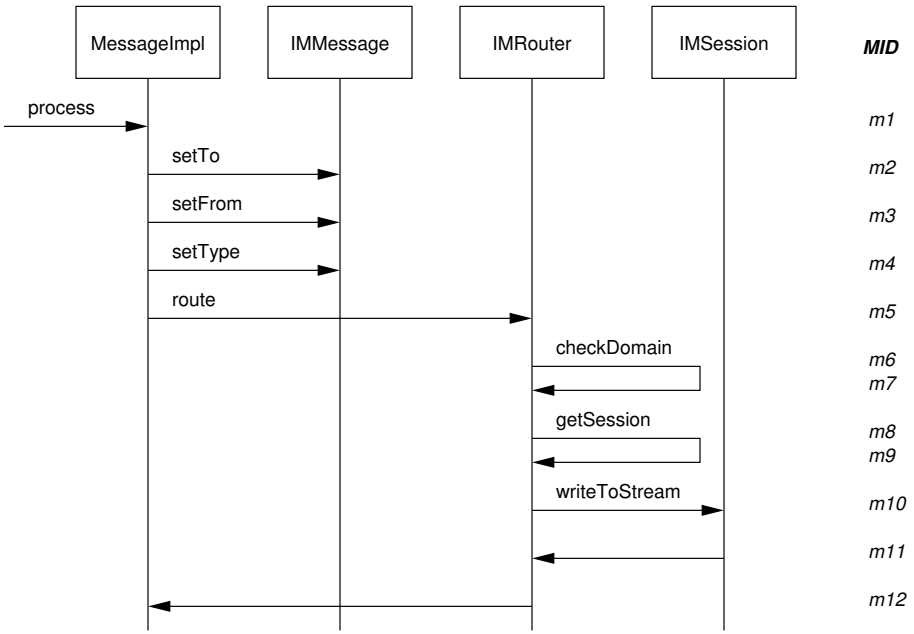

Fig. 8. Sequence diagram corresponding to the main success scenario of the *handle message* use case. A message identifier (MID) for each message is shown on the right of the diagram. We use simple MID's for presentation convenience.

This mapping provides a correspondence between use case subtrees and sequences of messages. Based on this correspondence, we present a join point model for AspectSD, a language that targets the sequence-diagram model. We use AspectSD as a bridge between AspectU and AspectJ in the same way that, during system design and development, sequence diagrams bridge the gap between use cases and source code. In this section we describe the AspectSD language, the mapping and translation between models is discussed more fully in Section 5.

The join points exposed by AspectSD are simply sequences of consecutive messages which correspond to the subtrees that form AspectU's join points. As an example consider the tree in Figure 9. In that particular execution tree, the AspectU join point for the *verify* use-case step, corresponds to the AspectSD join point (i.e. the message sequence) $m_1 \ldots m_4$.

AspectSD pointcuts identify join points using two primitive pointcuts. These pointcuts can be composed using the *and* (`&&`) and *or* (||) operators, which operate analogously to those in AspectU. The two primitive pointcuts in AspectSD are:

1. `messages(message-list)`—matches consecutive message sequences (join points) identical to the sequence of messages in `message-list`, and

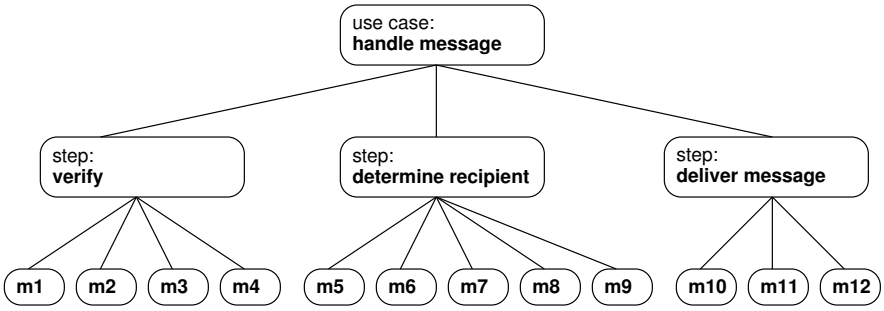

Fig. 9. An execution tree corresponding to the main success scenario of the *handle message* use case shown in Figure 3. The leaves of this tree correspond to the messages from the sequence diagram in Figure 8.

2. `in-flow(message-list)`—restricts the pointcut to match only immediately *after* the sequence of messages in `message-list` (in order) has occurred. This provides the context where the `messages` pointcut will match.

With these join points and pointcuts it is straightforward to map an AspectU pointcut to a corresponding AspectSD points. For example, with respect to the particular execution illustrated in Figure 9, the AspectU pointcut `step(determine recipient)` would correspond directly with the AspectSD pointcut

```
in-flow(m1,m2,m3,m4) && messages(m5,m6,m7,m8,m9)
```

This pointcut matches the sequence of messages $m_5 \ldots m_9$ when it occurs immediately after the sequence $m_1 \ldots m_4$. In general, AspectU pointcuts will correspond to multiple subtrees in multiple possible execution trees and so, multiple sequences of messages will correspond to a given AspectU pointcut.

In addition to identifying join points, an AspectSD pointcut can also provide access to values in the execution context of those join points. In the case of sequence diagram messages, values in the execution context include the sender of the message, the receiver of the message, the return value of the message and values passed as arguments to the message. Binding these values is done using the `bind(id, msg-id, entity-id)` pointcut, which provides a binding for the identifier `id` (part of the advice qualifier's `binding-list`) to the entity `entity-id` in message `msg-id`; `entity-id` can be any of `sender`, `receiver`, `argument_`n (where n is the n^{th} argument), and `return-value`. For example,

```
bind(stanza, m2, receiver)
```

binds the receiver of message m_2 to the name *stanza*.

AspectSD advice, like AspectU advice, can have one of three qualifiers: `before`, `around`, and `after`. For each of these, the `message` sequence identifies the advised join point. When applying `after` advice this will be the sequence just

seen. When applying `before` or `around` advice this will be the sequence about to begin. This implies that the `in-flow` sequence in the pointcut uniquely identifies the advised join point.

In our work we have only used AspectSD in the context of our translation process, rather than as a language a developer may use directly. As a result the body of a piece of AspectSD advice is of limited use. However, a developer can include sequences of messages to AspectSD advice. The primary use of this mechanism is to provide information for the translator described in the next section.

5 Translation Between Models

We want to leverage AspectU in a way that allows us to affect a running system. To this end, we have implemented an AspectU to AspectJ translator. The inputs to the translator, all described in more detail below, are the system's use cases, several AspectU aspects based on those use cases, the system's sequence diagrams (including mapping information), and a small amount of Java code. Given this input, the tool generates an AspectJ aspect that is able to affect the system in a way consistent with the behavioural changes specified by the AspectU advice.

This translation is challenging for a number of reasons. The decomposition of the system generally will not match the decomposition of the use cases. The messages in the sequence diagram may be implemented as method calls, returns from methods or exceptions being thrown. Also, much of the behaviour (i.e. many of the method calls, etc.) of the running system will not be specified in the sequence diagrams.

The details of the translation are presented along with an example AspectU aspect (the privacy aspect introduced in Section 3.2) and an example system. The example system is OpenIM [1], which is a partial implementation of the XMPP specification. The use cases were derived from the specification and the sequence diagrams (including the annotations) were developed by investigating the source code of the system.

The translation is a two stage process: AspectU to AspectSD and then AspectSD to AspectJ. The implementation of the translation tool requires explicit traceability links between the use cases, sequence diagrams, and source code. Our format for storing sequence diagrams allows basic message information to be specified along with mapping information to support our translation. Basic information includes: sender, receiver and message name. The primary pieces of mapping information specify which use-case *step* or *condition* (i.e. a condition triggering an extension) the message relates to. Other information can map high-level objects mentioned in the use case to parts of messages: sender, receiver, arguments, and return values.

Figure 8 shows the sequence diagram that corresponds to the main success scenario of the *handle message* use case. The following is an example of the information that can be captured for a message in our sequence diagram format, it corresponds to message m_5 in Figure 8:

```
sender: MessageImpl
receiver: IMRouter
name: route
step: determine recipient
arguments: entity, message
```

The first three lines provide basic information about the message. The last two lines provide mapping information; this message contributes to the *determine recipient* use-case step and the arguments to the message correspond to the *entity* and *message* objects referred to in the use case. Put another way, the object passed as the first argument to the *route* message, in this scenario, corresponds to the *entity* discussed in the use case. While discovering and documenting this information was a manual process for us, techniques described in [5, 10, 14] could be used to assist in the creation and maintenance of these explicit links either automatically or semi-automatically. Another approach to facilitate this mapping could involve work in the area of model-driven architectures [6].

The two stages of our translation process are described in the following subsections. The description of the process is based on the privacy-concern example introduced as an AspectU aspect in Section 3. The concern is converted from AspectU aspect to AspectSD aspect and then to an AspectJ aspect that was then applied to the OpenIM system. This yielded an extended and runnable application that contains the new features.

5.1 AspectU to AspectSD

The annotations attached to messages in our sequence-diagram format provide a mapping between use-case steps and the messages that implement those steps. For each distinct message (usually a triple: sender, receiver, and method) in a system's sequence diagrams, our translation tool assigns a *message identifier* (MID). For example, the messages in the sequence diagram shown in Figure 8 have been assigned the MID's $m_1 \ldots m_{12}$. As mentioned above, in order to map from use cases to sequence diagrams, sequences of messages can be annotated as participating in various use-case steps. Figure 9 shows how the messages correspond to one particular use case execution tree. The following is an alternative illustration of this scenario, which shows what use-case steps are implemented by what sequences of messages in this scenario:

$$\underbrace{m_1\ m_2\ m_3\ m_4}_{\textit{verify}}\ \underbrace{m_5\ m_6\ m_7\ m_8\ m_9}_{\textit{determine recipient}}\ \underbrace{m_{10}\ m_{11}\ m_{12}}_{\textit{deliver message}}$$

Section 4 described the correspondence between AspectU pointcuts and AspectSD pointcuts. Given all relevant sequence diagrams (appropriately annotated) and a piece of AspectU advice, the AspectU translator computes the effect of the added steps, added extensions, and replaced subtrees on the sequence diagrams. In the advice for the privacy aspect described in Section 3.2, we added one step (we will refer to this with message id $\mathbf{s_1}$) and one extension (we will refer to this by message id $\mathbf{e_1}$).

The translation process from AspectU to AspectSD comprises three steps.

1. Insert identifiers for steps—For each block of steps inserted by the AspectU advice body, the associated MID is inserted into each message sequence that matches the appropriate sequence of use-case steps.

For example, the step added by the privacy advice is intended to go *before* the *deliver message* step in the *handle message* use case. Therefore, the above sequence would be modified as follows:

$$\underbrace{m_1\ m_2\ m_3\ m_4}_{\textit{verify}}\ \underbrace{m_5\ m_6\ m_7\ m_8\ m_9}_{\textit{determine recipient}}\ \mathbf{s_1}\ \underbrace{m_{10}\ m_{11}\ m_{12}}_{\textit{deliver message}}$$

2. Insert identifiers for extensions—Next, the effect of adding extensions is computed. While adding steps expands a sequence of messages, adding an extension splits one sequence into two sequences: one where the extension does occur and one where it does not. Again from the privacy aspect, when the privacy check fails the extension is triggered, this corresponds to the following message sequence being added:

$$\underbrace{m_1\ m_2\ m_3\ m_4}_{\textit{verify}}\ \underbrace{m_5\ m_6\ m_7\ m_8\ m_9}_{\textit{determine recipient}}\ \mathbf{s_1}\ \mathbf{e_1}$$

In this example, the use case is terminated by the extension $\mathbf{e_1}$; thus, ending the sequence. In situations where the added extension rejoins the scenario the modified sequence would be followed by the additional messages from the original sequence.

3. Translate `bind` designators—Next, the tool determines how `bind` designators are to be translated. For each matching bind in an AspectU advice's pointcut, our translation tool computes a corresponding AspectSD bind. This is done by considering all messages that preceded the match point, starting from the match point and working toward the start of the message sequence. In the case of the privacy example, the messages $m_9 \ldots m_1$ would be considered based on the sequence above.

For each of these messages, the tool searches for a message with a sender, receiver, argument or return value that matches the operand of the AspectU

advice. In the case of the first AspectU `bind` in the privacy aspect the *recipient* matches with return value of message m_9 so the corresponding AspectSD pointcut is `bind(user, m9, return_value)`. For the second AspectU `bind` in the privacy aspect the *stanza* matches with the second argument of message m_5 so the AspectSD pointcut is `bind(user, m5, argument_2)`.

Once the modified sequences have been computed along with the associated bindings the AspectSD pointcut can be generated. The location identified by the pointcut for the privacy aspect (and the *handle message* use case) is the *deliver message* step, which corresponds to the message sequence m_{10} m_{11} m_{12}, corresponding to the entire AspectSD pointcut:

```
before(user,stanza) :
    in-flow(m1,m2,m3,m4,m5,m6,m7,m8,m9) &&
    messages(m10,m11,m12) &&
    bind(user, m9, return_value) &&
    bind(stanza, m5, argument_2)
```

We have shown one sequence diagram being modified along with one resulting AspectSD pointcut. In general translating a piece of AspectU advice will result in multiple sequence diagrams being modified. One AspectSD pointcut is generated for each of these modified points.

5.2 AspectSD to AspectJ

The translation from AspectU to AspectSD produces a description of how the the AspectU advice impacts a system in terms of its sequence diagrams. The goal of the second stage of the AspectU translator tool is to generate AspectJ advice that can appropriately impact the execution of the system.

To support the privacy feature, we wrote Java code to implement various parts of concern including code to check that a given message or presence stanza should be accepted. The translation process automatically determines when in the execution of the system the check should be performed along with binding the objects needed for the check (the *user* and *stanza* objects). So that the correct operation will be carried out at that point, we provide the translator with some minimal information about what to call. This is done by supplying to the translator one block of Java source for the steps in an advice body and one for each extension added by the advice. These blocks connect the generated AspectJ code and the Java code it is expected to trigger at the appropriate point in the execution. In the case of the *check privacy* step introduced by the privacy aspect the block simply contains a call to a static method:

```
{ UserPrivacy.accept(user, stanza); }
```

For each AspectU aspect, the output of our tool is one AspectJ aspect. In addition to advice (explained below), the aspect contains an `Event` class, an `eventList` field, a `post` method, a `match` method, and a `get` method. An event corresponds to the sending and receiving of one sequence-diagram message. These components of the generated aspect help manage these events.

- `Event`—instances of this class capture information about a particular message event such as its name (which is the MID of the corresponding message as a `String`) and name-value pairs binding objects corresponding to the sender, receiver, arguments and return type of the message.
- `eventList`—a list that stores events in the order they occur[2].
- `void post(String messageName)`—pushes a new event onto the event list.
- `boolean match(String messageList)`—checks the event list to see if the messages in `messageList` exist in the same order at the top of the list.
- `Object get(String mid, String objectName)`—searches the list of events starting from the most recent event for the first event whose name matches `mid`; returns the object bound to the `objectName` in the matched event.

Also part of this generated aspect are three kinds of AspectJ advice: *tracing advice*, *triggering advice*, and *blocking advice*, which interact with the application at the source-code level. Each of these categories of advice are described below along with examples from the AspectJ aspect generated when applying the privacy AspectU aspect to the OpenIM system.

Tracing advice keeps track of messages as they occur. For each distinct message that appears in a system's sequence diagrams, a piece of AspectJ advice is generated that calls the `post` method when this message occurs. Name-value pairs are attached to the event to support the variable binding mechanism. The generated tracing advice for the m_1 message, part of the sequence diagram in Figure 8, looks like this:

```
before(MessageImpl receiver) :
    execution(* MessageImpl.process(..)) &&
    target(receiver)
{
    Event event = new Event("m1");
    event.set("receiver", receiver);
    post(event);
}
```

The above example shows the posting of an event that corresponds to a method being called. Tracing advice is also generated to capture events associated with returns from methods. It is possible that a return message specified in a sequence diagram could be associated with a specific value being returned or with a specific exception being thrown. For example, a scenario described in one sequence diagram may contain a return message that corresponds to an exception being thrown from a particular method, while a different scenario may contain a return message that corresponds to a normal return from the same method. We generate pieces of AspectJ advice to distinguish between various types of returns so that the appropriate event is posted.

[2] The event list can be viewed as unbounded, but for practical purposes, a maximum size can be enforced.

Triggering advice triggers the actual work of the AspectU advice body. It determines when the blocks of Java code supplied by the user get executed. The execution of this code is conditional on whether the appropriate sequence of messages has occurred.

In the special case where the sequence of messages correspond to nested method calls, AspectJ's `cflow` construct could be used to check the context. However, in the more general case, AspectJ provides no direct support for this kind of checking. As a result, our generated aspects rely on the event history supplied by the tracing aspects and stored in the `eventList`. This history is checked at runtime by calling the `match` method with a message pattern that captures the expected context. Only if that method call to `match` returns true is the block executed.

One piece of triggering advice will be generated for each situation in which a user-supplied block of Java code may need to be executed. The following is the triggering advice associated with the steps added by the AspectU privacy aspect (i.e. the actual check that the stanza should be accepted). As shown in Section 5.1, the check is intended to be done after the message sequence $m_1 \ldots m_9$. Before the advice is executed the `match` method is called to verify that the sequence has indeed occurred. Message m_9 corresponds to the return of the `getRegisteredSession` method call, as shown in the sequence diagram in Figure 8. As a result this generated piece of AspectJ advice is implemented as `after` advice on that call.

In addition to the call to the `match` method and the user supplied block of code, the generated advice contains a call to `post` to add the event associated with this portion of the AspectU advice (identified by the message id $\mathtt{s_1}$), and calls to the `get` method to bind the variables expected by the block.

```
after(IMRouterImpl receiver, IMRouterImpl sender) :
    call(* IMRouterImpl.getRegisteredSession(..)) &&
    this(sender) && target(receiver)
{
    // check context (pattern is in reverse order)
    if (match("m9,m8,m7,m6,m5,m4,m3,m2,m1")) {
        Event event = new Event("s1");
        post(event);

        // generated bindings
        Object user = get("m9","return_value");
        Object stanza = get("m5","argument_2");

        // user supplied source block
        UserPrivacy.accept(user, stanza);
    }
}
```

Blocking advice is the final type of generated advice. In situations where AspectU advice adds an extension that terminates the use case (i.e. an extension with no `rejoin`) or replaces part of the behaviour (due to `around` advice), there may be some remaining steps that should be skipped. One such situation is illustrated by the AspectU privacy aspect shown in Figure 6, which adds an extension to handle a privacy-check failure. In this case, the *deliver message* step should be skipped. As a result, the method calls implementing these steps should be blocked. Preventing these calls is handled by `around` advice on each method call that may need to be blocked. The `around` advice first checks whether or not the extension has been executed. If it has not, then the `proceed` is called, permitting the method to execute. Otherwise, the `proceed` is not called and the method call is skipped.

```
void around() :
    call(* net.java.dev.openim.IMSession.writeToStream(..))
{
    if (!match("e1,s1,m9,m8,m7,m6,m5,m4,m3,m2,m1")) {
        proceed();
    }
}
```

In the OpenIM application's privacy concern, this approach works well. However, in general there could be problems with this approach. If there is some code unrelated to the given use case that is tangled with the code that is blocked from executing, incorrect behaviour may result as the unrelated code will be blocked as well. The limitations of our approach are discussed further in the next Section.

6 Discussion

We have analyzed the storage and privacy instant messaging concerns, two concerns that crosscut a system's use cases and source code. The specification for XMPPIM describes instant messaging as a layer added on top of a general messaging protocol (XMPP Core). Following this separation of concerns, we presented storage and privacy as aspects that add IM features to a system implementing the core protocol. Describing the various parts of these concerns in terms of where they affect existing use cases is relatively straightforward:

- when delivery of a message fails because the client is not connected, store the message,
- when an entity connects to the server, before handling any message stanzas deliver any deferred messages,
- before delivering a message stanza check the recipient's privacy rules, and
- before forwarding a presence stanza check the sender's privacy rules.

The AspectU pointcut and advice body for expressing these concerns is similarly straightforward, in fact the advice reads almost like their English descriptions. For these concerns (and many others, we believe), expressing them at a use-case

level is natural and makes the developer's intention clear. Further, our approach results in an aspect that captures the concern in a way that can be understood in terms of the use-case model; and, when combined with our translator, affects the runtime behaviour.

We believe that use cases and the types of concerns we have discussed, map naturally to sequences of method calls in the source code. Capturing such concerns in an aspect-oriented programming language like AspectJ can be difficult. This is because AspectJ does not provide direct support for pointcuts based on such sequences. An AspectJ implementation may require manually developing support for bindings and tracing along the lines of what is automatically generated by our translation tool. Furthermore, explicitly describing the concern in terms of these sequences is less intensional than expressing them in terms of use-case steps and extensions.

AspectU advice along with translation based on a mapping between models provides a level of indirection that provides a degree of independence from implementation details. However, this independence comes at the cost of creating and maintaining this mapping between models. If this cost is too high, our approach becomes impractical; hence, the full benefits depend on the extent to which this maintenance can be automated.

In addition to the benefits we experience from using use case level pointcuts, we have a natural mechanism for allowing high-level manipulation of the control flow of the system in some situations. In particular, our tool can automatically translate an AspectU extension that terminates a use case into AspectJ code that suppresses the remainder of the use case after it is terminated by the extension. However there are limitations with our approach in its ability to control the flow of an application. First, as discussed above, in some cases suppressing method calls could produce unintended side effects. Second, ideally an added extension would be able to rejoin at an arbitrary step in the use case (possibly repeating a previous step or skipping ahead to another). Supporting these arbitrary changes to the flow of the application is not feasible with our approach.

A different approach, that may not suffer this limitations could involve applying aspect languages similar to AspectU in the context of a model-driven approach [6]. Using such an approach, higher-level models are used to drive the generation of the source model. In this context, aspect languages based on the appropriate models could be used to influence the source code generation. This approach might give the AspectU aspects more control of the flow of the system, avoid the cost associated with maintaining a mapping between models, and overcome key limitations with our current approach.

The work we have presented has been inspired by the success of aspect-oriented languages in modularizing crosscutting concerns in source code. Our goal has been to extend these ideas beyond the source code to include other behavioural models; the use-case model, in particular. In addition to this, our approach allows a developer to express advice with use case level pointcuts while still affecting the runtime behaviour of the system modelled by the use cases.

7 Related Work

Much of the work in developing other aspect languages has been the foundation of our work with AspectU. *Hyper/J* [13] supports Multi-Dimensional Separation of Concerns [18] by allowing a software engineer to define separate, but overlapping hyperslices in a software system simultaneously. The *Caesar* aspect programming language [12] offers an alternative to AspectJ as an implementation language. It offers a higher-level module concept on top of join-point interception, helping with the encapsulation of aspects. However, like AspectJ, neither of these aspect languages help encapsulate higher-level concerns; they interact with the source code only.

Work by Jacobson [8] suggests that aspect-oriented programming can provide a link between use cases and implementation. In particular aspects could support a decomposition of a system based on use cases, including extension use cases. In contrast to this, our work focuses on modularizing concerns that crosscut the structure of the use-cases model and translating those to a form that can be applied to a system with a conventional decomposition.

Gray, et al. [7] introduce *Embedded Constraint Language*, a domain-specific aspect modelling language and tool that implement constraints as aspects. Like AspectU, their tool recognizes and weaves aspects that apply to non-code artifacts. While Gray's approach targets domain specific models, ours targets use cases.

Batory, et al. [2, 3] add a notion of aspects to product-line architectures and domain specific languages so that features can be added or removed from a product by a simple reconfiguring of its architecture. Their technique targets cross-cutting concerns that can span multiple document types. Our approach also supports aspects that cross-cut behavioural models.

Our work is also related to work in Early Aspects, which refers to the identification and encapsulation of crosscutting concerns that occur in requirements and architecture. One example of work in this area is *Cosmos* as described in [17]. Cosmos offers an alternative means to encode the mapping between requirements level artifacts. It is a concern-modelling schema for creating a rich set of relationships between all types of software artifacts. However, Cosmos offers no means of applying advice on any of the concerns that it models.

References

1. A. Agahi. OpenIM Java jabber server, September 2003. http://openim.jabberstudio.org/.
2. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 187–197. IEEE Computer Society Press, 2003.
3. D. S. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering Methodolology (TOSEM)*, 11(2):191–214, 2002. http://doi.acm.org/10.1145/505145.505147.

4. O. Ben-Kiki, C. Evans, and B. Ingerson. Yaml ain't markup language (yaml^{TM}1.0). http://yaml.org/spec/.
5. A. Egyed and P. Grünbacher. Automating requirements traceability — beyond the record and replay paradigm. In *Proceedings 17th International Conference Automated Software Engineering (ASE)*, pages 163–171, September 2002. http://citeseer.nj.nec.com/egyed02automating.html.
6. D. Frankel. *Model Driven Architecture*. Wiley Publishing, Inc, 1 edition, 2003.
7. J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM (CACM)*, 44(10):87–93, 2001. http://doi.acm.org/10.1145/383845.383864.
8. I. Jacobson. Use cases and aspects – working seamlessly together. *Journal of Object Technology*, 2(4):7–28, July 2003.
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072, pages 327–355, 2001. http://citeseer.nj.nec.com/kiczales01overview.html.
10. K. Koskimies, T. Systä, J. Tuomi, and T. Männistö. Automated support for modelling OO software. *IEEE Software*, 15(1):87–94, January 1998.
11. H. Masuhara and G. Kiczales. Modelling crosscutting in aspect-Oriented mechanisms. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2003.
12. M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings International Conference on Aspect-Oriented Software Development (AOSD '03)*, 2003. http://citeseer.nj.nec.com/mezini03conquering.html.
13. H. Ossher and P. Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 734–737. ACM Press, 2000. http://doi.acm.org/10.1145/337180.337618.
14. Use case management with Rational Rose and Rational RequisitePro, 2000. http://www.therationaledge.com/content/feb_03/rdn.jsp.
15. P. Saint-Andre. XMPP core, September 2003. http://www.jabber.org/ietf/draft-ietf-xmpp-core-18.html.
16. P. Saint-Andre. XMPP instant messaging, September 2003. http://www.potaroo.net/ietf/ids/draft-ietf-xmpp-im-18.txt.
17. S. M. Sutton, Jr. and I. Rouvellou. Modeling of software concerns in Cosmos. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD '02)*, pages 127–133. ACM Press, 2002. http://doi.acm.org/10.1145/508386.508402.
18. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society Press, 1999.