From Aspect-Oriented Design To Aspect-Oriented Programs: Tool-Supported Translation of JPDDs Into Code

Stefan Hanenberg, Dominik Stein, and Rainer Unland
Institute for Computer Science and Business Information Systems (ICB)
University of Duisburg-Essen, Germany
{ stefan.hanenberg, dominik.stein, rainer.unland }@icb.uni-due.de

Abstract

Join Point Designation Diagrams (JPDDs) permit developers to design aspect-oriented software on an abstract level. Consequently, JPDDs permit developers to communicate their software design independent of the programming language in use. However, developer face two problems. First, they need to understand the semantics of JPDDs in addition to their programming language. Second, after designing aspects using JPDDs, they need to decide how to map them into their programming language. A tool-supported translation of JPDDs into a known aspect-oriented language obviously would ease both problems. However, in order to achieve this goal, it is necessary to determine what a "good" JPDD translation looks like, i.e. it is necessary to have a number of principles that determine the characteristics of a "good" translation. This paper describes a tool-supported translation of JPDDs to aspect-oriented languages. Principles for translating JPDDs are described and a concrete mapping to the aspect-oriented language AspectJ is explained.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques. K.6.3 [**Management of Computing and Information Systems**]: Software Management – *software development*. D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation*.

Keywords

Aspect-Oriented Software Development; Aspect-Oriented Design; Query Models

1. Introduction

«Join Point Designation Diagrams» (JPDDs [22, 23, 24]) are a graphical means to represent join point selections in a programming language-independent manner. They are intended to help developers communicate the underlying conceptual ideas of their join point selections to others (cf. [24]). However, in order to use JPDDs in daily software development, developers have to solve two problems: First of all, they need to learn and understand the notation and the semantics of JPDDs. Otherwise, they wouldn't be able to create "correct" JPDDs, or read the ones of others. Furthermore, they need to have a well-defined mapping between the symbols used in JPDDs and their programming language constructs. Otherwise, it would be hard to map a given join point selection, outlined by a JPDD, to program code and, moreover, to exploit an existing pointcut design, expressed by a JPDD, in a different software development project.

One way to tackle these problems would be to use a tool that is capable to translate JPDDs into code. Such a tool would help developers to solve both of the mentioned problems: It would help them in mapping JPDDs to their "native" programming language, i.e. it would help them in detecting and exploiting a given JPDD in their own program code. Thereby, it would also help them to learn and understand the notation and semantics of JPDDs since the translation would codify/reproduce the semantics of a given JPDD in terms of the semantics of an aspect-oriented programming language that is known to the developer.

In particular to enable the latter, it is essential that the tool generates pointcut code that is easy to understand and that is in-line with the JPDD it was generated from. Therefore, the quality of the mapping implemented in the tool is significant. In this paper, we present a (non-exclusive) set of principles operationalizing a set of requirements that "good" mappings should obey in our opinion, i.e. mappings

- which are easy to comprehend,
- which meet the "spirit" of the target programming language, and
- which are in-line with the respective JPDD.

Based on these principles, we furthermore present a concrete mapping for the aspect-oriented programming language AspectJ [11], and discuss how the mapping complies to the principles.

The remainder of this paper is structured as follows: In section 2, we motivate the contributions of our work with help of an example. In section 3, we introduce JPDDs as the graphical design notation that should be transformed into program code. In section 4, we discuss alternative mappings of JPDDs to program code and conclude principles that "good" mappings should obey. In section 5, we outline concrete mappings to the aspect-oriented programming language AspectJ. In section 6, we relate our work to existing work. Section 7 concludes the paper.

2. Problem Statement

The contributions of this work yielded to the development of aspect-oriented software is best motivated with help of an illustrative example: Imagine a JAsCo [26] programmer and that is interested in implementing a data mining application on top of an existing online shop (as described in [28], for example). One of the data to be mined is the response of customers to promotional offers. In particular, the developer is interested in selecting *low promotion-prone customers*, i.e. customers who seem to discover promotional offers merely "by accident" (example adopted from [28], with slight modifications).

The JAsCo programmer probably wants to exploit the *stateful aspect language constructs* of JAsCo to realize the data mining aspects. The following code snippet is adopted from [28] (with slight modifications), and shows only those lines of code that are responsible for the join point selection (we abstract from the rest since all other lines of code are irrelevant for the subsequent considerations).

```
//defining join point selection
hook LowPromotionProneCustomerHook {
  LowPromotionProneCustomerHook(
    browseProducts(Category category),
    accessPromotions(CustomerID customer)) {

    start > helperTrans;
    helperTrans : execute(browseProducts) >
              helperTrans || browsePromotionTrans;
    browsePromotionTrans : execute(accessPromotions);
  }
  //classify customer as low promotion-prone
  after browsePromotionTrans () {...}
} [...]

//binding join point selection to concrete method calls
LowPromotionProneCustomerHook hook1 =
  new LowPromotionProneCustomerHook(
    * OnlineShop.browse*(*),
    * OnlineShop.getPromotions(CustomerID)
  ) [...]
```

After having implemented these lines of code, let's assume that the occasion arises that the developer needs to communicate the aspect (i.e. your join point selection, in this case) to an audience that is not familiar with JAsCo – for example, to other developers who are interested in the underlying pointcut design and who have not studied JAsCo, yet. However, if the audience has not studied JAsCo yet, it is impossible to communicate the pointcut design using the source code – because this requires in-depth knowledge about JAsCo.

With help of a JPDD, it is possible to explain the join point selection to a non-JAsCo-aware audience. The JPDD in Figure 1, for example, can be used to illustrate the join point selection realized by the code snippet shown above. The JPDD outlines that the join point selection LowPromotionProneCustomerHook distinguishes between different states. The transitions between these states outline a protocol that customers must perform in order to be classified as a low promotion-prone customer. First, customers have to browse the online shop (to any location/product) at least once. Then, they may continue browsing as long as they feel like. Ultimately, though, they must find (and follow) the promotions link. This is the situation when the online shop receives the message `getPromotions`. At this point, the aspect intercepts, and performs the desired action. This correspond to `after browsePromotionTrans` in the JAsCo code. In the JPDD this point is indicated by identifier `<?jp>`.

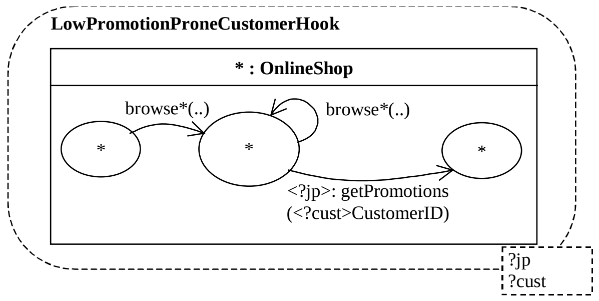

Figure 1: Documenting Join Point Selections With JPDDs.

Having such a JPDD, a means to represent your join point selection is provided which abstracts from the programming language being used actually to implement the system. This should permit others to comprehend a join point selection without the need to understand the underlying aspect-oriented language. In order to actually do so, of course, those others must be capable to read and understand JPDDs. Imagine, for example, an AspectJ developer who is not familiar with JAsCo, and with state-based join point selections in general, and who is interested in the selection semantics underlying the JPDD in Figure 1. Having never seen, and heard of, stateful aspects, he/she might wonder what the JPDD in Figure 1 is referring to and what join points it is actually selecting. In this case, it helps to provide the AspectJ developer with a plain AspectJ implementation of the join point selection. Such a mapping of the JPDD to his/her "native" programming language should help him/her to understand the JPDD, as well as similar JPDDs he/she will look at in the future. Furthermore, such a translation would help the AspectJ developer to use the join point selection (designed by a JAsCo developer), while at the same time the JAsCo developer (designing the join point selection) does not have to be familiar with AspectJ.

Translating JPDDs into programming languages is not unproblematic, though. Usually, there exists a large variety of possible ways to translate a given JPDD into an aspect-oriented programming language. Hence, it is more than likely that, all too often, developers will be unsure about the best way to map a given JPDD into program code. However, having each developer realize his/her own mapping would make the detection of an implementation of a JPDD in the program code difficult. Moreover, having such developer-dependent, and thus non-uniform, translations of JPDDs would obstruct a "learning-by-example" approach as it has been proposed above to acquire the join point selection semantics of JPDDs. Indeed, there are mappings that can be considered more suitable to this way of learning JPDDs than others. For these reasons, it can be considered essential to have carefully thought-out mappings that (a) describe a uniform way of translating JPDDs into program code and (b) that generate program code which is easy to trace back to the respective JPDDs (thus, facilitating the learning of JPDDs).

The contribution of this paper is to give principles for the translation of JPDDs into aspect-oriented program code. They can be seen as an evaluation framework for JPDD translations that are considered to be "good" and "sustainable" with respect to the goals (a+b) mentioned above. Furthermore, a concrete mapping is given for the aspect-oriented programming language AspectJ which aims to support the understanding of JPDDs when pursuing a "learning-by-example" approach. It is discussed how far the proposed mapping matches the postulated goals.

3. Join Point Designation Diagrams

«Join Point Designation Diagrams» (JPDDs [22, 23, 24]) are a means to specify queries on software artifacts. As such, they are particularly suited to express aspect-oriented join point selections, such as pointcuts [11], traversal strategies [12], match or type patterns [27, 11], or alike.

JPDDs outline a search pattern, which means that they specify a set of selection constraints which is mapped against a given software artifact (i.e. a program code or a running program). Within that pattern, (a tuple of) identifiers (see Figure 2a) may be assigned to particular elements in order to designate those parts of the queried software artifact that should be actually selected by the JPDD. JPDDs possess a export parameter box at their lower right corner (see Figure 2b) which lists the identifiers of all elements to select.

JPDDs come with a set of means to specify deviations in search patterns. Examples are the asterisk wildcard (`*`) which abstracts from an arbitrary number of characters in element names, or the dot-dot-wildcard (`..`) which abstracts form an arbitrary number of parameters in a parameter list (see Figure 2c). A vertical bar (`|`) may separate alternative search patterns for element names (see Figure 2c). Furthermore, JPDDs provide means to postulate the existence of paths (of arbitrary length) along class/object associations (—//—), the inheritance hierarchy (◁//—), state transitions (—//▷), or the call graph (—//►) (see Figure 2c). Further explanations on the effects of these deviation specification means and examples of their usage will be given in the subsequent subsections.

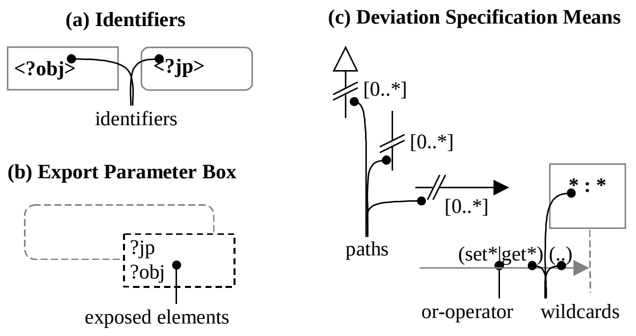

Figure 2: JPDD-Specific Symbols (cf. [23]).

JPDDs are defined with help of a graphical notation. That notation makes use of symbols from existing and well-established graphical notations, such as message sequence charts, state charts, flow charts[1], and class/object diagrams. The precise notation used in JPDDs is the one which can be found in the UML (the UML terms the aforementioned notations differently, and talks about interaction sequence diagrams, state diagrams, activity diagrams, and class/object diagrams, respectively). JPDDs chose to use symbols of various notations so that developers can choose those symbols that suit their conceptual view on a join point selection best (see [24] for further explications). The UML has been chosen as the base notation for JPDDs since its symbols are considered to be well-known to a broad range of developers. Nonetheless, it is important to note that JPDDs alter the semantic of the original UML symbols, and extend them with few new elements (see Figure 2), in order to suit and attend the specific needs of a query notation (see [23] for a more detailed discussion of the selection semantics of JPDDs).

3.1 Structure JPDDs

JPDDs make use of symbols of different graphical notations in order to provide the developer with appropriate means to express the intention of their join point selection. Class and object diagram symbols are used, for example, to express selections (i.e. selection constraints) on the structure of programs and program instances.

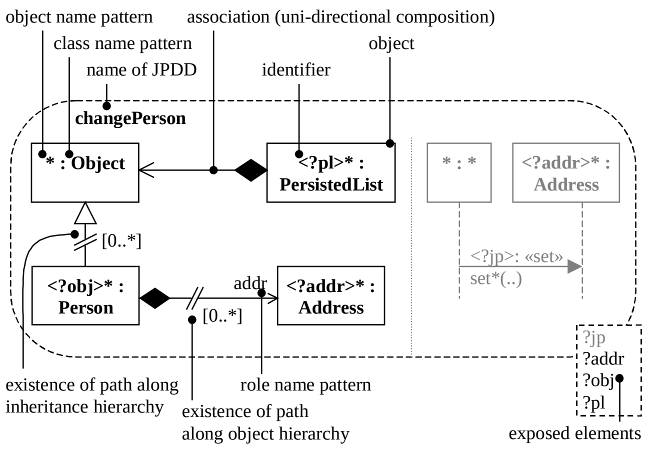

Figure 3: A Structure JPDD (cf. [23]).

Figure 3 shows a JPDD demonstrating how this is accomplished in detail: The example visualizes a join point selection that is taken from [1] and which deals with the selection of state changes of persisted objects. Its goal is to select also state changes that occur to objects being affiliated with (owned by) the persisted objects. In the example JPDD, all modifications[2] of `Address` objects affiliated with (owned by) a (persisted) `Person` object are selected. `Person` objects are persisted by (contained in) a `PersistedList` (which considers its contents to be of the general type `Object`). The JPDD makes use of an indirect association symbol (—//—) in order to indicate that the `Address` object does not need to be an immediate neighbor of the `Person` object. Its association ends assert, however, that the (indirectly) affiliated `Address` object must be reached via a composite relationship, and that it must play the role `addr`. Furthermore, the JPDD makes use of an indirect generalization symbol (◁//—) in order to indicate that `Person` objects must be specializations (subkinds) of `Object` objects – notwithstanding that there may exist further specialization steps along the inheritance hierarchy. Both (all) kinds of indirect relationships are adorned with a multiplicity restricting the number of relationships (i.e. (associations or generalizations, in this case) that need to be traversed on the path from one object to the other. In its export

[1] Program/data flow-model JPDDs will not be introduced in this paper; the interested reader is pointed to [24] for further details.

[2] I.e. any field assignment to fields of `Address` objects, indicated in the JPDD (see Figure 3) as a stereotyped method invocation of (pseudo) setter methods on `Address` objects (cf. [21]); for further details on behavioral JPDDs, see subsequent sections.

parameter box, the JPDD exposes the affiliated `Person` and `Address` objects (`<?obj>` and `<?addr>`), as well as the `PersistedList` (`<?pl>`) which is used to persist the `Person` objects[3].

Join point selection criteria on the program structure – which are specified with the symbols described above – may be combined with join point selection criteria on the program behavior – which will be considered next. Figure 3 (right part) outlines how such combinations could look like. Note how the identifier `<?addr>` is used to concatenate the left part to the right part. Combining JPDDs this way means that the selection constraints of both JPDDs must be satisfied for the join points to be selected.

3.2 Message Invocation Model-JPDDs

One way of expressing join point selections on the behavior of programs in JPDDs is by using symbols which are adopted from UML interaction sequence diagrams (message sequence charts). For example, the JPDD shown in Figure 4 uses lifelines (¦), message symbols (⟶), and activation bars (▯) to render a join point selection that selects all invocations of method `search` on objects of type `DiseaseRepositoryDBMS`, having one argument of type `int` and returning a value of type `DiseaseType`, which come to pass within the control flow (⫽▶) of any method (`*(..):*`) invoked on an object (`<?s>`) of (sub)type `ListServlet`. The purpose of this join point selection, which is adopted from [20], is to control access coming from the Internet (i.e. via a `ListServlet`) to the `Disease-RepositoryDBMS`. The JPDD returns the method invocation (`<?jp>`) as a join point, together with the argument (`<?arg>`) being passed as a parameter.

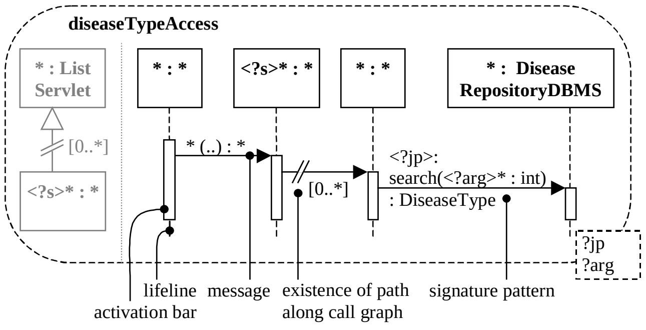

Figure 4: A Message Invocation Model-JPDD (cf. also [23])

Message invocation model-JPDDs may be combined with other JPDDs, e.g. structure JPDDs – as shown in Figure 4 (left part), – meaning that the selection constraints of both JPDDs must be satisfied for the join points to be selected. Note how the identifier `<?s>` is used here to connect both kinds of JPDDs.

3.3 State Model-JPDDs

Another way to express join point selections on the behavior of programs in JPDDs is by using symbols from UML state diagrams (state charts). Figure 5 outlines an example how the symbols of state model-JPDDs are used to represent a state-based join point selection. The example is adopted from [19] and is used to prevent any access to deleted objects. Thereto, it selects any invocation on messages beginning with `set` or `get` taking any number of arguments, as well as any invocation on message `toString` taking no argument, sent on any object (`<?obj>`) of (sub)type `PersistentRoot`. Note how the join point selection makes use of state symbols (⬭) and state transition symbols (⤻) in order to emphasize that it is only interested in message invocation events issued on objects that have received a `delete` message before (taking no arguments) and are thus in state "deleted". The join point selection returns the intercepted method call events (`<?jp>`) together with the object (`<?obj>`) receiving such events.

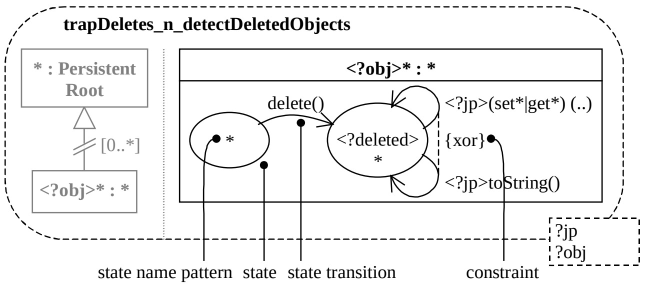

Figure 5: A State Model-JPDD (cf. [24]).

State model-JPDDs may comprise a special symbol (⫽⤻) in order to denote a path along state transitions (i.e. a sequence of arbitrary state transitions) – similar to the way in which the special symbol ⫽▶ denotes a path along the control flow (i.e. a sequence of arbitrary messages) in message invocation-model JPDDs (cf. previous section 3.2). Furthermore, state model-JPDDs may be combined with other JPDDs, e.g. structure JPDDs – as shown in Figure 5 (left part), – meaning that the selection constraints specified in both JPDDs must be satisfied so that the join points will be selected. Note how the object identifier `<?obj>` is used here to connect the JPDDs.

4. Principles Of JPDD Translation

There typically exists an infinite number of possible solutions for translating a JPDD to an aspect-oriented language. Hence, in order to come to a reasonable conclusion on what a translation should look like and what language features of the target language should be used, guidelines must be found that help to determine under which circumstances one mapping is considered better than another. Therefore, in the following, we discuss a couple of sample JPDDs with corresponding – alternative – language mappings, and we elucidate why we consider the one mapping better than the other. Then, we conclude from these considerations and formulate a number of principles of "good" mapping design.

In order to ease the understanding of our argumentation, we neglect to use other notational means than message sequence model-JPDDs and other programming languages than AspectJ. This is because we estimate that these means are most-familiar to most of the readers of this paper. This does not mean, though, that the following principles only pertain to the translation of message sequence model-JPDDs into AspectJ. Furthermore, it should be noted that this work only deals with code generation such as it is known from forward engineering approaches (from JPDDs to pointcut code). No assertions are being made about (principles of) reverse engineering approaches (from pointcut code to JPDDs).

[3] Apart from the modification join point `<?jp>`, itself; see subsequent sections for further explanations on the specification of join point selections on program behavior.

4.1 Use of Pointcut Language Constructs

Figure 6 illustrates a sample JPDD based on interaction diagrams. It states that an object of type `A` sends a message `m1`. Later on, an object of type `B` sends a message `m2` to an object of type `C`. The last message `m2` represents the join point to be selected.

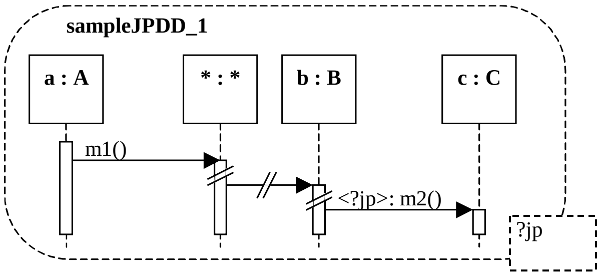

Figure 6: Sample JPDD (1).

In case we want to translate this JPDD into the target language AspectJ, there are different possibilities how a corresponding pointcut could look like.

One option is to specify a pointcut that selects the first message `m1` sent from `a`, and to specify an advice that stores this information (together with the current thread that is about to be executed) into a suitable data structure. Then, it is possible to specify another pointcut selecting the message `m2` sent from `b` to `c`, and let it check whether there has been previously a message `m1` (by searching for it in the data structure).

```
static Hashtable d = new Hashtable ();
pointcut pc1(A a):
  call(*.m1()) && this(a);
pointcut sampleJPDD_1():
  call(*.m2()) && this(B) && target(C) &&
  if(contained());
static boolean contained() {
  return d.containsKey(Thread.currentThread());}

before(A a): pc1(a) {
  ..storing thread with a to dictionary d…
}
```

Figure 7: Translation of JPDD (1) into AspectJ.

Figure 7 illustrates what a corresponding implementation in AspectJ looks like. An advice that refers to the pointcut `pc1` stores the current thread into a dictionary along with the join point specific data `a` and retains that way that message `m1` has been sent. The pointcut `sampleJPDD_1` refers to that dictionary and determines whether the current thread is contained as a key (using AspectJ's `if` pointcut). Only if this is the case, `sampleJPDD_1` selects message `m2` (for reasons of simplicity we did not specify the complete code for storing the data to and reading from the dictionary). Note, that it is also necessary to remove the current thread from the dictionary at the right point in time (when there is no longer any method `m1` on the call stack) which is not discussed here.

Although this translation is technically correct, it suffers from the problem that the translation does not consider the pointcut construct `cflow` in AspectJ, which permits to specify the desired join point selection directly and without the need of maintaining a special data structure. Figure 8 illustrates a translation of JPDD (1) using a `cflow` pointcut. A comparison of Figure 7 and Figure 8 reveals that the first translation is relatively hard to understand (even for experienced AspectJ programmers), while the second one is not. The reason is that in the second case a dedicated language construct of AspectJ is used that AspectJ developers are well-familiar with and that they would usually exploit when they specify a join point selection like the one considered here: AspectJ programmers are familiar with AspectJ's control flow abstraction and use it when it is necessary.

```
pointcut sampleJPDD_1():
  cflow(call(*.m1()) && this(A)) &&
  call(*.m2()) && this(B) && target(C);
```

Figure 8: Translation of JPDD (1) into AspectJ using cflow.

From these considerations we conclude the first principle for translating JPDDs to program code, which is that whenever there is a construct in the underlying aspect-oriented programming language that is dedicated to a special join point selection, this feature should be used.

> **Principle 1 (Use of Pointcut Language Constructs):** *A JPDD translation should express a JPDD's join point selection semantics in terms of those language constructs of a target language that are particularly designed for the given selection task.*

Although this principle seems to be somewhat intuitive, its intention is to make developers of JPDD mappings aware of the individual capabilities of the target language, and to choose that particular mapping that outlines best the semantics of a join point selection in terms of the available target language constructs.

4.2 Reduction of Aspect State

Although join point selection constructs of the target programming language should be used whenever appropriate (i.e. in those situations which the constructs were designed for), sometimes a programming language may fail to provide suitable join point selection means. In that case, defining additional state within aspects (or using other data structures) is a frequent technique for overcoming the restrictions of the respective pointcut language.

A common case in which this is necessary, for example, is the realization of a state-based join point selection [5] with an aspect-oriented programming language that does not provide suitable selection means to reference states and/or state transitions.

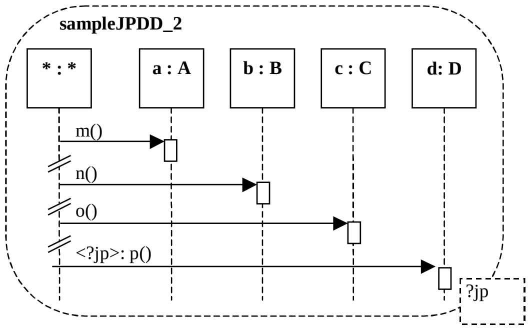

Figure 9: Sample JPDD (2).

For example, let's consider the join point selection that is visualized in Figure 9, and which selects a message `p` to an object of type `D` as a join point – provided that previously an object of type `A` has received a message `m`, an object of type `B` has received a message `n`, and an object of type `C` has received a message `o` (in that order). A join point selection like this is commonly known as an application of a state-based join point selection (cf. [28, 24])[4]. Now, if the target aspect-oriented programming language happens not to be able to refer to states and/or state transitions within its pointcut language, it is usually necessary to add further state information to the aspect as a workaround.

Figure 10 outlines a possible mapping of the JPDD shown in Figure 9 to AspectJ. For each message in the message interaction model-JPDD (except the last one), a Boolean field is specified within the aspect. Furthermore, for each message (except the last one), a pointcut is added to the aspect selecting the message and verifying whether the predecessor of the message has already been reached. If this is the case, a `before` advice sets the respective Boolean field to `true`. While all of these pointcuts and advice merely exist to maintain the aspect state, only the last pointcut selects the actual join point of interest: The last pointcut refers to the message `p` sent to an object of type `D` – and verifies that all the other messages have been executed before by evaluating the state variable `reachedThird`. Similar to the previous section, it is also necessary to specify the right point in time when the states are to be set to `false` which is not discussed here.

```
static boolean reachedFirst = false;
static boolean reachedSecond = false;
static boolean reachedThird = false;

pointcut pc1():  call(*.m()) && target(A);
pointcut pc2():  call(*.n()) && target(B) && if (reachedFirst);
pointcut pc3():  call(*.o()) && target(C) && if (reachedSecond);

before(): pc1() { reachedFirst = true;}
before(): pc2() { reachedSecond = true;}
before(): pc3() { reachedThird = true;}

pointcut sampleJPDD_2():
  call(*.p()) && target(D) && if (reachedThird);
```

Figure 10: Translation of JPDD (2) into AspectJ.

Although the semantics of the mapping presented in Figure 10 is correct, we consider this mapping to be not satisfying. With a mapping like this, the number of state variables in the aspect tends to grow quite large very fast, and so is the number of pointcuts and advice in the aspect. We consider this to be confusing for the developer because, when looking at the generated program code, the developer is faced with an immense number of states, pointcuts, and advice – although only one pointcut of them is actually concerned with the selection of the join point of interest. Therefore, we consider this mapping to be problematic (with respect to using the generated code in order to understand the JPDD semantics) because it does not emphasize the actual join point selection (`sampleJPDD_2`).

Consequently, when working with (i.e. modifying or using) the generated aspect code, the developer may easily happen to refer to one of the generated "helper" pointcuts rather than the actual join point selection – even though he/she did not intend to do so (and should not do so).

```
static int currentState=0;
pointcut innerSelectionState():
        (call(*.m()) && target(A) && if (currentState == 0)) |
        (call(*.n()) && target(B) && if (currentState == 1)) |
        (call(*.o()) && target(C) && if (currentState == 2)));
before(): innerSelectionState() { currentState++;}
pointcut sampleJPDD_2():
  call(*.p()) && target(D) && if (currentState == 3);
```

Figure 11: Translation of JPDD (2) into AspectJ.

Figure 11 illustrates another mapping of the JPDD above. In contrast to the previous one, the code in Figure 11 provides only one additional state member (`currentState`). That field is intended to remember the current state of the aspect. Furthermore, there is only one pointcut (`innerSelectionState`) that monitors whether a new state has been reached, and only one advice that increments the `currentState` member. In contrast to the previous mapping, the interface of this aspect is less overloaded with state-representing and state-maintaining elements. Note, that Figure 11 does not denote when the current state is set back to `0` (in correspondence to the previous examples).

Following the discussion above, we consider this mapping superior to the previous one, and conclude the following principle:

> **Principle 2 (Reduction of Aspect State):** *A mapping should keep the number of elements to represent and maintain join point selection-dependent aspect state as less as possible.*

The term «elements» refers to any kind of program code that is used to represent and maintain aspect state, here, such as field definitions, methods, class definitions, pointcuts as well as advice.

4.3 Side Effect Free Pointcuts

Carrying on the considerations from the previous example, and taking it to ultimate perfection, would lead to a mapping where the maintenance of aspect state is included in the "real" pointcut itself (i.e. the one concerned with the selection of the actual join point of interest) – rather than having them separated into extra pointcuts and advice.

```
static int currentState=0;
pointcut myPointcut():
  (call(*.m()) && target(A) && if(setState(0,1))) |
  (call(*.n()) && target(B) && if (setState(1,2))) |
  (call(*.o()) && target(C) && if (setState(2,3))) |
  (call(*.p()) && target(D) && if (setState(3,4));
public static boolean setState(int from, int to) {
  if (currentState == from) currentState = to;
  return currentState > 3;
}
```

Figure 12: Translation of JPDD (2) into AspectJ.

Such a mapping is illustrated in Figure 12. Similar to the mapping outlined in Figure 11, there is only one field (`currentState`) that represents the current state of the aspect. In contrast to Figure 11, though, there is only one pointcut that subsumes the (entire) join point selection outlined by the JPDD (see Figure 9). Apart from that, there is a method `setState` which is invoked from within the pointcut (see `if` pointcut designators), and which is responsible for maintaining the aspect state. The method is invoked with two integer parameters where the first one

[4] Nevertheless, we have decided to visualize it by means of a message interaction model-JPDD since we are going to investigate its implementation using a "un-stateful" aspect-oriented programming language. Furthermore, it could be argued that this visualization represents the underlying conceptual model of the developer (cf. [24] for a more detailed discussion).

represents the number of the current state and the second one represents the number of the next state. If the delivered current state corresponds to the value of the variable `currentState`, then the next state is assigned to this variable. The method returns `true` if the current state is larger than 3. Hence, the pointcut triggers the execution of a related advice only if the last state (number 4) is reached.

A straightforward interpretation of principle **2** (see previous section) would suggest that the mapping in Figure 12 is superior to the mapping in Figure 11 because there is only one pointcut, one field, and one method – as opposed to two pointcuts, one field, and one advice as in Figure 11.

Nonetheless, we consider this mapping to be problematic because the pointcut in Figure 12 does not only check whether certain conditions hold (e.g. whether a particular state has been reached). Rather, the pointcuts also takes care about updating the aspect state: The evaluation of the `setState` method in the pointcut's `if` clauses is not side-effect free. We feel that this breaks with the expectations developers have when they study an AspectJ pointcut. Moreover, a mapping to non side-effect free pointcuts might confuse developers trying to understand the semantics of a JPDD by studying the translated program code – because JPDDs, being mere selection patterns, should not have any side effects by default. Hence, we think that side-effects in the pointcut evaluation reduce the understandability of the generated code. Consequently, as a restriction to principle **2**, we postulate:

> **Principle 3 (Side-Effect Free Join Point Selection):** *Join point selections in a mapping should be side-effect free, i.e. the evaluation of a pointcut must not change the state of the system/the aspect.*

Due to some characteristics of the underlying aspect-oriented programming language, the situation may arise in which this principle cannot be fully satisfied. For example, the use of the `if` pointcut designator in AspectJ may change the state of the system by referring to a method that triggers some state-changing aspect. The main intention of the principle still holds despite this possibility, though; i.e. developers should try to reduce any (potential) side-effect within their join point selections.

4.4 Pointcut Headers

As already mentioned before, the occasion may arise where a given JPDD cannot be directly transformed into a target language. Such situations come to pass, for example, when a JPDD developer makes use of a JPDD construct that does not have a semantic counterpart in the target language. While in the previous sections we dealt with workarounds for missing join point selection means (which cause the creation of state within aspects), in this section we are going to deal with workarounds for missing means for join point context exposure.

Figure 13 illustrates a JPDD whose join point is a message invocation of method `m1` on an object of type `Person`. Furthermore, in its structural selection constraints, the JPDD requires that the `Person` object must be related to an `Address` object via an association (in which the `Address` object is playing the role `address`). As indicated in the export parameter box, the JPDD exposes both the `Person` object (bound to the variable `?p`) as well as the `Address` object (bound to the variable `?a`).

If this JPDD has to be translated into AspectJ, the main problem is that AspectJ only permits to expose particular objects to an advice. This means that – with respect to the example given above – only the `Person` object `?p` can be exposed (using AspectJ's `target` pointcut designator). The `Address` object `?a`, on the contrary, cannot be exposed with help of AspectJ's proper pointcut language constructs. In consequence, a workaround implementation must be found.

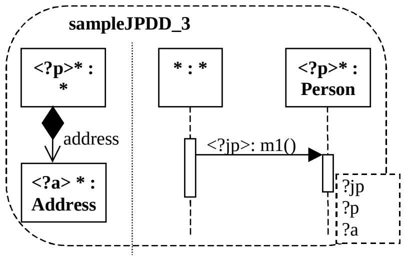

Figure 13: Sample JPDD (3).

The first option would be to store the `Address` object `?a` into a variable of the aspect. That variable would be initialized properly during the pointcut evaluation, and could be accessed by the advice afterwards. Figure 14 illustrates a corresponding AspectJ pointcut implementation. It makes use of an `if` pointcut designator which refers to a method that (a) assigns the `address` attribute[5] of the `Person` object `?p` (exposed by the `target` pointcut designator) to the variable of the aspect, and (b) verifies if the assigned object is an instance of type `Address`. Afterwards, each advice affiliated with `pc1` can access the respective `Address` object `?a` via the variable of the aspect. As such, the same join point context is exposed as defined by the corresponding JPDD.

```
static Address a;
pointcut sampleJPDD_3(Person p):
  call(*.m1()) && target(p) && if(hasAddress(p));
private static boolean hasAddress(Person p) {
  return ((a=p.address) instanceOf Address)
}
```

Figure 14: Translation of JPDD (3) into AspectJ.

Although this approach is feasible, we consider this solution to be not desirable because the exposure of the `Address` object is difficult to detect. As a consequence, the resulting join point selection is hard to understand. Most aspect-oriented programming languages (such as AspectJ, abc, JAsCo, etc.) decided to make the exposure of join point context explicit in the pointcut header – rather than “hiding” them in the aspect state. Consequently, most aspect-oriented developers expect to have a complete list of all exposed join point parameters in the pointcut header – rather than having to search for them among the aspect’s variables. "Hiding" exposed join point context in the aspect state also imposes problems when it comes to pointcut composition. If some join point parameters can be accessed only via an aspect’s variable, developers wishing to compose a pointcut need to be aware of these "implicit" join point parameters in order to implement a proper composition – which is an additional source of errors. Therefore, we consider it desirable to construct pointcuts whose signatures (headers, including the exposed

[5] Accessing the `address` attribute of `Person` object `?p` is not unproblematic; see section 4.6 for further explications.

variables) are closely related to the signatures of the corresponding JPDDs (i.e. its export parameter boxes).

Figure 15 illustrates an AspectJ implementation that we consider more appropriate. The code contains a new aspect method `constructedJoinPoint`, which is going to be invoked whenever a `Person` object receives a message `m1`. The `around` advice that is responsible for evaluating if the `address` attribute of the `Person` object `?p` (exposed by the `target` pointcut designator) is an instance of type `Address`. If this is the case, it calls the newly generated aspect method and passes the `Person` object `?p`, the `Address` object `?a`, as well as an anonymous `ProceedObject` as parameters (the sole purpose of the last parameter is to be able to invoke the originally intercepted message call, selected by pointcut `pc1`, from within the aspect method `constructedJoinPoint`; cf. [8]). The invocation of this newly generated aspect method can now be intercepted by another pointcut `sampleJPDD_3`, which exposes the passed arguments – in particular, the `Person` object `?p` and the `Address` object `?a` – to a corresponding advice. As a result, the same join point context is exposed as defined by the corresponding JPDD.

```
Object constructedJoinPoint(Person p, Address a, ProceedObject o)
  { return o.doProceed(p); }

abstract class ProceedObject {
  abstract Object doProceed(Person p); }

pointcut pc1(Person p):
  call(*.m1()) && target(p);

Object around(Person): pc1(p) {
  Address a;
  if ((a=p.address) instanceOf Address)
   return constructedJoinPoint(p, a,
     new ProceedObject {
       Object doProceed(Person p) { return proceed(p); }
     });
  else
     return proceed (p);
}

pointcut sampleJPDD_3(Person p, Address a, ProceedObject po):
   call(* *.constructedJoinPoint(Person, Address, ProceedObject))
   && args(p, a, po);
```

Figure 15: Translation of JPDD (3) into AspectJ using generated aspect methods.

Reflecting on these considerations, the principle underlying the mapping presented in Figure 15 can be summarized as follows:

Principle 4 (Pointcut Headers): *A JPDD translation should generate pointcuts whose headers correspond to the exported parameters as defined in the JPDD.*

Having postulated that principle, a short note needs to be given to aspect-oriented languages that do not provide dedicated mechanisms for context exposure. One example for such a language is AspectS [9], which basically exposes all parameters of a method call to an advice. We think that even in those cases the aforementioned principle should be obeyed. In case of AspectS, this could mean to generate a new method (in analogy to the previous AspectJ example) such that each of the export parameters of the JPDD are parameters of the new method.

4.5 Restricting JPDD Translations

JPDDs are permitted to specify constraints on the join point which either cannot be implemented in certain circumstances, or even in general (i.e. never). One example for such a JPDD would be one that refers to *future data* (cf. [7, 16, 25]).

For example, Figure 16 illustrates a JPDD which selects a message `m` being sent to an object `a` of type `A` as the join point (`<?jp>`). However, this join point is only selected if the target object `a` receives another message `n` at a later point in time. Furthermore, at an even later point in time, the object `a` is required to sent a message `o` to an arbitrary object (which is bound and exposed by the variable `?o`). This last message also represents a join point.

The problem with this JPDD is that it refers (for join point `<?jp>`) to some future events in the system (i.e. messages `n` and `o`). This leads to a twofold problem. First, it cannot be guaranteed in general whether such a join point selection (`<?jp>`) is computable at all. Second, most aspect-oriented languages (with minor exceptions; cf. [16]) do not support the reference to future events within their pointcut language. Nevertheless, there are possible ways to still realize a translation of a JPDD like this. First, in case the aspect-oriented target language does provide suitable join point selection means to refer to future events, those language features should be used. Second, if the aspect-oriented target language does not provide those join point selection means, a JPDD translation could at least “try” to determine whether such events *could* happen (by means of static code analysis).

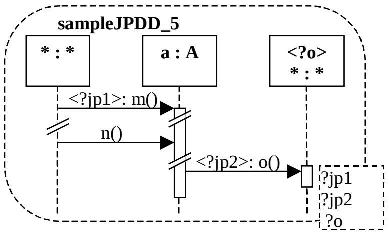

Figure 16: Sample JPDD (4).

From our point of view, only the former solution seems justifiable (because it would comply to principle 1; see section 4.1). The latter solution is considered undesirable, though, because it would go beyond the implementation of (reasonable) workarounds and would mean to superimpose the "spirit" of the target programming language with the "spirit" of JPDDs. With other words, we do not consider it to be the goal of translations to extend a given target programming language such that every selection pattern specified by a JPDD can be performed in/executed with that target language. Instead, we consider it sufficient that the mapping translates “as much as possible” of the join point selection and informs the developer which parts of a JPDD cannot be translated correctly. We consider such approximation of the join point selection to be still helpful because it permits developers to understand at least some parts of the join point selection in terms of the known target language and makes explicit what parts are not translated in the correct way.

Concluding from these contemplations, we propose:

Principle 5 (Restrictions and Approximations of JPDD Mappings): *A description of a JPDD mapping needs to define reasonable circumstances under which the mapping refuses to translate a JPDD into a target language completely.*

Furthermore, reasonable approximations for a JPDD translation should be specified (if possible).

In case a developer tries to translate a JPDD for which no reasonable mapping exists, suitable error messages should be thrown that point the developer to those elements in the JPDD that need to be changed in order to gain a translatable JPDD.

4.6 Static Analysis of Program Code

Another – related – problem implementers of JPDD mappings have to deal with is related to the different flavors in which aspect-oriented languages perform static code analysis on the generated pointcut code in order to verify if it is valid. An example for such static analysis is type checking: While languages like AspectJ are based on a typed programming language (Java), languages like AspectS are not (Smalltalk). Consequently, a translation of JPDDs into a typed programming language would be expected to be conform to the underlying type system.

However, the question is whether the JPDD translation itself is responsible for guaranteeing such conformance, or whether the resulting code should be simply handed over to the aspect-oriented system so that the system can test it for conformity.

A suitable example to elucidate the problem has been given in section 4.4 (see JPDD in Figure 13): The matter of interest is the relationship between the `Person` object and the `Address` object. According to the JPDD, it can be easily concluded that the actual type of the variable `?o` must be `Person` because it is a `Person` object that receives message `m1` (which is the actual join point in the join point selection). However, it is not clear whether the type `Person` has a field named `address` (as specified by the role name at the association end pointing to the `Address` object). Hence, from the (typed) perspective of AspectJ, it is not clear whether this field can be simply addressed by using an expression like `p.address` (where p is a variable of type `Person`)– or if this would mean a type error.

One solution in AspectJ (respectively in Java) to overcome this problem is to use the Reflection API. Instead of accessing the field directly, we would first access the class of object `?o` (i.e. `Person`) in order to compute whether this class (or any superclass) provides the appropriate field `address`, and would try to access it only if the computation has yielded a positive result. Figure 17 illustrates how such a check for the existence of a field `address` would look like.

```
static boolean checkAddress(Person p) {
  try {
    Field f = p.getClass.getField("address");
    return (f.get(p)!=null);
  } catch (Exception ex) {}
  return false;
}
```

Figure 17: Translation of JPDD (3) into AspectJ.

However, although this translation certainly meets the semantics of the JPDD, we would not consider it to be useful with respect to an easy comprehension of the generated pointcut code. Instead, we think that a mapping may make positive assumptions about the validity of a JPDD and therefore should translate JPDDs into straightforward implementations. With other words, we propose to leave some necessary static checks of the resulting code to the target language. Consequently, we accept that the resulting code possibly contains some errors that need to be fixed by the developer after the pointcut code generation by hand – for the sake of an increased understandability the resulting code.

Hence, we conclude:

Principle 6 (Static Constraints Checks Left to Aspect-Oriented Language): *A mapping of JPDDs should make positive assumptions about the validity of a JPDD with respect to static analysis checks that need to be performed by the underlying aspect-oriented language. A mapping does not need to perform such checks itself, but needs to document situations where the resulting code might fail and provide strategies how a defective translated code can be fixed.*

Principle 6 is closely related to principle 5. Both principles refer to situations where the generated join point selections in a given target language either do not match the semantics of the underlying JPDD (principle 5) or are semantically incomplete, i.e. the generated code may contain errors that are left to be checked by the target language (principle 6). Nevertheless, it should be noted that the focus of each principle is different. Principle 5 refers to situations where it is (maybe inherently) not possible to map a JPDD to code that matches the semantics of a JPDD completely. In contrast to that, principle 6 refers to situations in which a semantically correct mapping of a JPDD would be possible, but where straightforward (though erroneous) solutions exist that still permit developers to understand the underlying join point selection.

5. AspectJ Mapping

This section describes mappings from JPDDs to the aspect-oriented language AspectJ[6]. Thereto, we describe for AspectJ how interaction-based join point selections, state-based join point selections and structural JPDDs are translated into the language and explain furthermore the mapping restrictions. The algorithms underlying the mapping are explained informally and illustrated using a non-trivial example.

Note, that we intentionally focus on message invocation models and structure JPDDs here since they have been mainly used for motivating the principles (cf. section 4).

5.1 Interaction JPDDs

As already explained in some examples in section 4, it is not possible to translate interaction diagrams into pointcuts straight forward, since it is possible that some state information need to be generated.

In order to consider principle 1 (Use of Pointcut Language Constructs) we need to determine what AspectJ language features need to be considered for translating interaction diagrams.

First, the pointcuts `call` and `execution` can be directly used for the translation, since they directly refer to single elements in the JPDD (e.g. [23]). Second, AspectJ provides with the `cflow` language construct means to specify indirect messages directly on

[6] We have also implemented already some preliminary tool-supported translation to AspectS (http://dawis.icb.uni-due.de/research/ aosd/join_point_designation_diagrams_jpdds/), but since we assume that readers are rather familiar with AspectJ, we do not discuss the AspectS mapping here in detail and just give some insights about this mapping in the conclusion.

the programming language level. Third, actual types that can be specified within the object description of an interaction diagram can be described using `target`, `this` and `args` under some circumstances. The mapping has to consider that `if` pointcuts cannot be used freely within `cflow` pointcuts [11]. Hence, whenever there is a situation where it is desirable to use a `cflow` pointcut but where additional constraints on objects, types or messages are required, we need to construct a work-around.

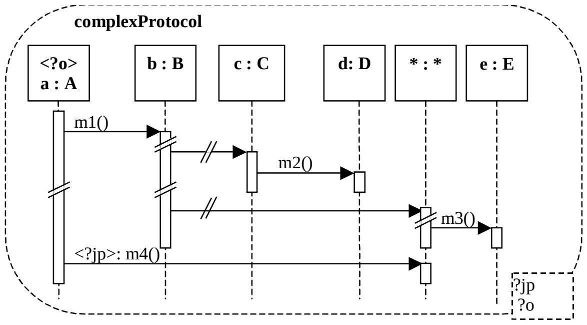

Figure 18. Example for Interaction-Based Selection.

Once we have a JPDD such as defined in Figure 18 we start from the first message (which is message `m1`) and analyse whether this message (and all messages caused by `m1`) can be described by a `cflow`. In such a case, we would simply generate the `cflow` pointcut. In the other case, we need to construct pointcuts and advice which construct and maintain the state information required for the join point selection.

We use the following rules to determine whether a message sequence can be specified using the `cflow` construct:

1. each object participating in a message sequence (as a sender or receiver) does not send or receive more than one message (since this requires the construction of additional state information)
2. all messages in the sequence are separated by the indirection symbol (otherwise this means that one message has to follow the previous message directly which does not correspond to the semantics of `cflow`)
3. structural constraints on objects participating in the message (sender, target, parameters) can be expressed in terms of the pointcuts `this`, `target` and `args` (i.e. no `if` pointcuts are necessary, which cannot be used within `cflow`, cf. [11])
4. only the elements of the last message in the sequence are exposed (or described identifiers that are used by further elements in the join point selection).

Applying these rules to the interaction model in Figure 18, we see that the complete interaction cannot be translated at once using the `cflow` pointcut. Although rules number 3. and 4. are valid for all elements in the interaction diagram, the straight forward translation is not possible for the following reasons:

1. object `a` sends more than one message (`m1` and `m4`, which contradicts 1.)
2. object `b` sends more than one message (indirect messages to c and to an unnamed object, which contradicts 1.)
3. message `m2` is the first message of a method in c (which contradicts 2.)

However, we see that parts of the interaction can be represented using `cflow`: Message `m3` should occur within the control flow of an object `b` (while there are further constraints on this object `b`)

In order to reduce the number of state variables (principle 2), we use the proceed object idiom (as explained in section 4.4) whenever there is a message at a later point in time that explicitly needs to refer to an object of a previous message. Since the message `m4` needs to be sent from the object `a` (i.e. the same instance that sends message `m1`), such a situation exists. We construct an abstract class `Proceed01` that is instantiated when an object of `A` is executed (`_state01_`). At the same time, the aspect's state variable is set to state `1`. When message `m1` is sent we refer to the previous object `a` via a `cflow` construct that refers to the method `_state01_`. The `cflow` exposes the parameter of that method. In that way it is not necessary to store object `a` itself in an aspect's variable. A message `m1` is part of the interaction if `states` has the value `1` and if the sender of the message equates to the parameter exposed by the `cflow`.

Following the same approach as for `_state01_`, we construct a new method `_state02_` that sets the variable `states` to `2`.

Figure 19 illustrates the resulting code, whereby the state with the value 1 represents the state "object `a` is currently executing" and the one with value 2 states "object `a` sends object `b` message `m1`".

```
static int states = 0;

abstract class _Proceed01 {  abstract Object doProceed(A a);}
pointcut _state01_(A a):
   execution(* *.*(..)) && this(a) && if(states == 0);

Object around(A a):_state01_(a) {
   return _state01_(a, new _Proceed01() {
      Object doProceed(A a) {return proceed(a); }});}

// method that permits to expose object a
Object _state01_(A a, _Proceed01 p) {
   states = 1; return p.doProceed(a);}

// Step 1: message m1 from a to b
abstract class
  _Proceed02 {  abstract Object doProceed(A a, A a2, B b);}
pointcut _state02_(A a, A a2, B b):
   cflow(execution(* *._state01_(A, ..)) && args(a, ..)) &&
   call(* *.m1()) && this(a2) && target(b) && if(a == a2)
   && if(states==1);

Object around(A a, A2 a2, B b):_state02_(a, a2, b) {
   return _state02_(a, b, new _Proceed02() {
      Object doProceed(A a, B b) {return proceed(a, a2, b); }});}

Object _state02_(A a, B b, _Proceed02 p) {
   states = 2; return p.doProceed(a, b);
}
```

Figure 19. AspectJ Translation of Figure 18 (step 1): Translating message m1 from a to b.

Please note, that according the examples throughout this paper Figure 19 does illustrate how the state variable is set back to some previous values. However, this turns out to be not that complicated. For example, the state 1 is abandoned, if the there is no executing object a on the call stack, i.e. right after the execution of an object a (that does not appear below the cflow of an object of type A) the variable states is set to 0.

Applying each of the previous steps to all messages leads to the additional code as illustrated in Figure 20. Although we did not illustrate for the previous examples how to abandon a certain state, we illustrate it here for the message m2: The special thing about m2 is that it needs to be the first message in a method in c. Hence, abandoning the state is an essential characteristics of the selection here.

```
pointcut _state03_():
   cflow(execution(* *._state02_(A, B, ..))) &&
   execution(* *.*(..)) && this(C) && if(states==2);
before(): _state03_() { states = 3; }

// Checking that m2 is the first message in c
pointcut _state04_():
   cflow(execution(* *._state02_(A, B, ..))) &&
   call(* *.m2()) && this(C) &&  target(D) && if(states==3);

Object around():_state04_() { states = 4; return proceed();}

pointcut _state04Abandoned_():
   cflow(execution(* *._state02_(A, B, ..))) && call(* *.*(..)) &&
   !(call(* *.m2()) && this(C) &&  target(D) && if(states==3));
before():_state04Abandoned_ () { states = 2; }

pointcut _state05_():
   cflow(execution(* *._state02_(A, B, ..))) &&
   call(* *.m3()) && target(E)
   && if(states==4);

Object around():_state05_() { states = 5; return proceed();}

// Mapping  the parameters to match JPDD param box
abstract class
  _Proceed03 {  abstract Object doProceed(A a, A a2);}

pointcut _complexProtocol_(A a, A a2):
   cflow(execution(* *._state00_(A, ..)) && args(a2, ..))
   && call(* *.m4()) && this(a) && if(states == 5) && if(a==a2);

Object around(A a, A a2): _complexProtocol_ (a, a2) {
   return complexProtocol (a, a2, new _Proceed03() {
     Object doProceed(A a, A a2) {return proceed(a, a2); }});}

Object complexProtocol (A a, A a2, _Proceed03 p) {
   return p.doProceed(a, a2);}

// Checking that m2 is the first message in c
pointcut complexProtocol(A a):
  execution(* *.complexProtocol(A)) && args(a,..);
```

Figure 20. AspectJ Translation of Figure 18 (continued).

The problem with stating "it is the first message in a method in c" is that it also requires an additional state that describes that object c has been entered. Since we also have the constraint that c needs to be executed in the control flow of method _state02_ we construct a corresponding pointcut that makes use of cflow (see pointcut_state03_). A corresponding advice that refers to _state03_ sets the variable states to value 3. Furthermore, we generate a pointcut _state04_ that determines whether states has the value 3 and message m2 is currently sent. In order to guarantee that the message m2 is the first message, we generate an advice that sets states back to value 2. The corresponding pointcut _state04Abandoned_ checks, whether states has already he value 3 and a messages is currently sent that is not the desired message m2.

The implementation of the pointcut _state05_ is rather straightforward: we already determined in the beginning of this section that message m3 can be selected using the cflow construct. Hence, we only need to determine the corresponding start of the control flow (which is the method _state02_).

Finally, we need to create a pointcut that matches the parameter box of the JPDD (principle 4, Pointcut Headers). The pointcut that determines whether the last message is reached (_complexProcotol_) requires 2 parameters in order to check whether the sender of the message corresponds to the object a. The original JPDD just requires one parameter (in addition to the join point). Hence, we create a method complexProcotol that also contains the objects a and a2 (in addition to the proceed object) and create finally the pointcut complexProtocol which refers to the method execution and exposes its first argument.

5.2 Structural JPDDs

Structural JPDDs differ from interaction JPDDs in the way that they do not define join points but structural constraints on those join points. Consequently, a translation of a structural JPDD does not necessarily lead to the definition of a pointcut, but rather to the definition of a new method that needs to be invoked from within a pointcut or an application of a primitive pointcut.

Since AspectJ already provides with the operator + as well as with the pointcuts this, target and args the ability to specify some structural constraints (on the actual type) it is desirable to use them when they are needed (according to principle 1). Whenever from within a join point a structural constraint is required we check whether this structural constraint can be directly mapped to these constructs.

In all other cases (which means that object relationships need to be evaluated and potentially collected) we check,

1. whether all object information necessary to evaluate the relationships are accessible from the join point itself (which we call local accessibility).
2. whether related objects are being addressed from within or exposed from the JPDD.

We consider a (directed) relationship to be locally accessible if the source object of the relationship is bound to a variable that is used in a join point selection within an interaction- or state-JPDD. If all necessary information are locally accessible we construct a method that has the necessary local information as input parameters (the structural check-method).

If this is not the case, we generate pointcuts and corresponding advice that collect all necessary information. From within the structural check method such data structures need to be accessed.

In case object relationships are named, we construct directly field accesses according to the principle 6 (Static Constraints Checks Left to Aspect-Oriented Language). In case there are object relationships that are not named, we generate reflective accesses to an object's fields. In case indirect object relationships are used, we generate traversal methods that run over the object graph.

If objects are not being addressed from within the JPDD and if they are not exposed, the structural check method is simply a boolean method that checks whether the constraints of the structural JPDD hold.

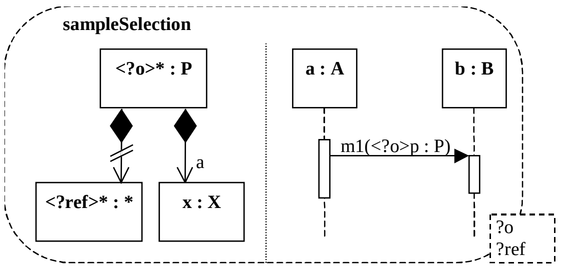

Figure 21. Example for Structural JPDD.

If objects are addressed or exposed, it is necessary to bind them to a variable (or a collection). Furthermore, because of the principle 4 (Pointcut Headers), we generate a method that (also) has this variable (or collection) as parameter. This method is used by other pointcuts in order have an explicit representation of the bounded objects.

Figure 21 illustrates a structural JPDD that should be translated. There are two relationships between the object bound to the variable o and related objects. All relationships are local, because the object o is bounded to the parameter of type P that is passed at the join point. Furthermore, related objects are exposed by the JPDD (the objects ref) while the related object a is not directly exposed.

```
abstract class _Proceed01 {  abstract Object doProceed(P p);}
pointcut _state01_ (P p):
  call(*.m(P)) && this(A) && target(B) && args(p) &&
  if(checkA(p));

static boolean checkA (P p) {return (p.a instanceOf X);}

Object around(P p): _state01_ (p) {
  Collection ref = bindRef(p);
  return _state01_(p, ref,
    new _Proceed01 {
      Object doProceed(P p) { return proceed(p); }
    });
}

Collection bindRef(P p) {
  Collection c = …a depth first search on p…;  return c ;}

Object _state01_(P p, Collection c, _Proceed01 p1) {
  return p.doProceed(p);}

pointcut sampleSelection(P p, Collection c):
  call(* *._state01_(P, Collection, ..)) && args(p, c);
```

Figure 22. AspectJ Translation of Figure 21.

First, we construct for the non-exposed constraint on field a a boolean method that accesses a from p and checks, whether the actual type matches (type X). This boolean method is invoked from a first pointcut (_state01_, see Figure 22).

Second, for the variable ref we construct a method bindRef that traverses the object hierarchy using the reflective API, stores all reachable objects in a collection and returns that collection. For exposing this method, we generate a method that has all exposed objects (p as well as ref) as a parameter and which proceeds with the original join point. This method is being invoked when the original join point is reached via an advice that refers to the pointcut _state01_. This advice creates the proceed object passed as an additional parameter to the method.

The pointcut usable by the developer which reflects on the complete JPDD is the pointcut sampleSelection which has the desired interface.

5.3 Stateful JPDDs

Our translation of state JPDDs is very closely related to the translation of interaction JPDDs. To be more precisely, the algorithm for translating state JPDDs is a subset of the one used for translating interaction JPDDs: while the algorithm for interaction JPDDs needs to determine how many states need to be constructed, this information is already explicitly contained in a state JPDD. Consequently, we just give a very short overview of the mapping of stateful JPDDs.

The translation enumerates all states (to be stored in a corresponding state field) and generates for all transitions between two states a pointcut that checks the current state (which needs to match the number of the source state of the transition), the constraints in the join point (which corresponds to the translation of constraints as discussed in section 5.1 and 5.2).

For each constructed pointcut a corresponding advice is generated that updates the state (according to the number of the following state). In case further elements need to be exposed, the translations as explained in 5.2 have to be performed.

Although the principle 2 says that the number of states should be minimized, this is not valid for the explicitly declared states in the JPDD: Since the intention of such state is to have an explicit representation, a reduction of the states would in such a situation decrease the comprehensibility of the translated JPDD.

5.4 Restrictions of Translations

The here proposed translation does not translate any occurrence of future data of join points. I.e. it is not possible translate a JPDD which requires to evaluate information from the future of the system. Instead, the translation approximates the join point selection by leaving out all future data (which corresponds to the explanation given in section 4.4).

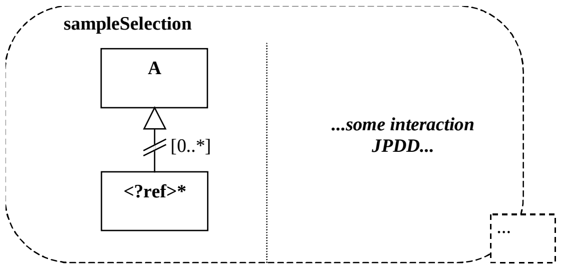

Figure 23. Restriction of Structural JPDD translation.

However, there is another restriction of the translation that is closely related to the background of the problem of future events. The translation does not permit to translate structural JPDDs that relate to subtype relationships where only the upper type is known.

Figure 23 illustrates a structural JPDD were the variable ref should be bound to all subtypes of A. The problem with translating such a JPDD to AspectJ is that Java does not directly permit to access any information about subtypes using the reflection API. The reasons lies mainly in the architecture of Java that permits to load classes later on via the class loader, i.e. not all classes need to be known at a certain point in time. Consequently, at a certain point in time (i.e. when the corresponding join point is reached) the number of all subtypes cannot be computed. In order to achieve this, it would be necessary to access the class loader in order to perform some modifications to it which is from our point of view not the task of a JPDD translation.

6. Related Work

The generation of program code from graphical notation such as JPDDs has been supported by a vast variety of CASE tools since a long time, and much research has been focused on how to generate that program code best (cf. [3]). However, no tool to our knowledge has been tackling the generation of pointcut code from visual representations of join point selections (such as JPDDs).

The principles introduced in section 3 can be likely compared with the effort do reduce so-called *code-smells* which have been articulated for object-oriented software construction in the context of refactorings (cf. [6]) as well as recently in the context of aspect-oriented software development (cf. e.g. [13]): the intention is to give some guidelines which prevent the resulting code to be not readable or understandable. Furthermore, the guidelines can be compared with coding guidelines as they exist for many programming languages(cf. e.g. [18] for programming guidelines on C++).

Other tool support exists in the domain of aspect-oriented software development, such as ActiveAspects [4], or Asbro [17], etc. The focus of these tools is to visualize the effects an aspect has on a given target system. They are usually not concerned with the generation of aspect code.

For the Theme/UML approach [2], a tool has been implemented based on KerMeta [14] which enables the composition and execution of themes (cf. [10]). Theme/UML does not provide a means for specifying join point selections in isolation, though. Consequently, no code generation has been necessary to do so.

7. Conclusions

In this paper, we have discussed the need for suitable mappings for JPDDs to aspect-oriented programming languages. We have come up with a (non-exclusive) list of principles according to which these mappings have to be specified. We furthermore have outlined a concrete mapping for the aspect-oriented programming language AspectJ which comply to these principles.

Apart from the translations, which can be used in a forward engineering approach to generate pointcut code from JPDDs, one of the major contributions of this work has been the detailed deduction of principles for that code generation. Some of these principles are focused on an appropriate alignment of the generated pointcut code with the structure of the JPDD (e.g. principle 4); other specify what to do if the target programming language does not provide sufficient join point selection means to realize the JPDD appropriately (principles 5 and 6). While these kinds of principles are very closely related to the translation of JPDDs to pointcut code, other principles (such as the principles 1-3) turn out to be even applicable as general design principles for "good" pointcut design in the general case.

We also think that further principles are necessary to have a good translation of JPDDs into code. For examples, we consider the use of an appropriate naming convention to be necessary for states that need to be constructed. However, the intention of this paper was not to discuss such principles only, but the application of these principles in a concrete mapping, too. Therefore, we did not describe further principles here.

In section 5 we only described the mapping from JPDDs to AspectJ. Nevertheless, we already have some preliminary tool-supported translation of JPDDs into AspectS. An interesting observation is that the resulting code largely differs from the corresponding translation into AspectJ due to the different philosophy of AspectS to consider aspects as ordinary classes which need to be instantiated and explicitly woven. Furthermore, since the pointcut language of AspectS widely differs from AspectJ, it is the usual case that even simple join point selections end up with the generation of pure Smalltalk code, where the connection to the underlying aspect-oriented system is hard to follow. For other target language (e.g. JAsCo) we expect that the existence of stateful pointcuts widely eases the construction of states (in comparison to the state constructions that were necessary in section 5.1). However, we are currently not able to provide a complete mapping from JPDDs to JAsCo and consider this to be future work.

One goal of providing translations for JPDDs into program code was to help developers learn and understand the semantics of JPDDs by enabling them to study equivalent pointcuts in a programming language they are familiar with. The idea was that this would help them also to read and understand similar JPDDs in other situations – in which they cannot resort to a corresponding pointcut implementation. However, as the examples given in section 5 have shown, this is not always the case. As it turned out, a complex join point selection specified in a JPDD is very likely to result into a likewise complex pointcut specification in a given programming language. Hence, we can conclude that – even if all mapping principles outlined in section 4 haven been obeyed – translations of JPDDs into program code facilitate the comprehension of the selection semantics of JPDDs only up to a certain complexity. Nevertheless, a translated JPDD still gives the developer the chance to understand a JPDD semantics by “playing around” with the join point selection in the sense that the developer can try out how the join point selection behaves in a given application.

It remains to mention that the principles specified in this paper are not meant to be complete. Indeed, the authors already apply further principles in their JPDD mappings (specifying how to deal with combinations of JPDDs, for example, or how to use (i.e. use to which extent) the reflection capabilities of the target language) which haven't been presented here due to space limitations. It is considered essential to maintain and complement this list of principles as part of future work/research. Likewise the concrete mappings should be extended with further mappings to other programming languages.

References

[1] Al-Mansari, M., Hanenberg, S., *Path Expression Pointcuts: Abstracting over Non-Local Object Relationships in Aspect-Oriented Languages*, in Proc. of NODe'06, Erfurt, Germany, LNI, 2006, pp. 81-96

[2] Baniassad, E., Clarke, S., *Aspect-Oriented Analysis and Design - The Theme Approach*, Addison-Wesley, 2005

[3] Bunse, C., Atkinson, C., *The Normal Object Form: Bridging the Gap from Models to Code*, in Proc. of UML'99, Fort Collins, CO, October 28-30, 1999, LNCS 1723, pp. 675-690

[4] Coelho, W., Murphy, G., *Presenting crosscutting Structure with Active Models*, in Proc. of AOSD'06, Bonn, Germany, March 20-24, ACM, pp. 158-168

[5] Douence, R., Fradet, P., Südholt, M., *Composition, Reuse and Interaction Analysis of Stateful Aspects*, in: Proc. of AOSD 2004, Lancaster, UK, March 2004, ACM, pp. 141-150.

[6] Fowler, M., Beck, K., Brant, J., Opdyke, W.F., Roberts, D., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999

[7] Hanenberg, S., *Design Dimensions of Aspect-Oriented Systems*, Dissertation, Institute for Computer Science and Business Information Systems, University of Duisburg-Essen, 2006.

[8] Hanenberg, S., Schmidmeier, A., *AspectJ Idioms for Aspect-Oriented Software Construction*, in: Proc. of EuroPLoP'03, June, 25-29, 2003, Irsee, Germany, pp. 617-644

[9] Hirschfeld, R., *AspectS - Aspect-Oriented Programming with Squeak*, in Proc. of NODe'02, Erfurt, Germany, October 2002, LNCS 2591, pp. 216-232

[10] Jackson, A., Klein, J., Baudry, B., Clarke, S., *Testing Executable Themes*, Workshop on Models and Aspects, at ECOOP'06, Nantes, France, July 3, 2006

[11] Laddad, R., *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, Greenwich, 2003

[12] Lieberherr, K., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996

[13] Monteiro, M., Fernandes, J.M., *Object-to-Aspect Refactorings for Feature Extraction*, in Proc. of AOSD'04 (industry track), Lancaster, UK, March 22-24, 2004, ACM

[14] Muller, PA., Fleurey, F., Jézéquel, JM., *Weaving executability into object-oriented meta-languages*, in Proc. of MoDELS'05, Montego Bay, Jamaica, October 2005, LNCS 3713, pp. 264-278

[15] OMG, *Unified Modeling Language Specification*, Version 1.5, 2003 (OMG Document formal/03-03-01)

[16] Ostermann, K., Mezini, M., Bockisch, Chr., *Expressive Pointcuts for Increased Modularity*, in: Proc. of ECOOP'05, Glasgow, UK, July 2005, ACM

[17] Pfeiffer, JH., Gurd, J., *Visualization-Based Tool Support for the Development of Aspect-Oriented Programs*, in Proc. of AOSD'06, Bonn, Germany, March 20-24, 2006, ACM, pp. 146-157

[18] Plum, T., Saks, T., *C++ Programming Guidelines*, Plum Hall, 1991

[19] Rashid, A., Chitchyan, R., *Persistence as an Aspect*, in: Proc. of AOSD 2003, Boston, MA, March 2003, ACM, pp. 120-129

[20] Soares, S., Laureano, E., Borba, P., *Implementing Distribution and Persistence Aspects with AspectJ*, in: Proc. of OOPSLA '02 (Seattle, WA, Nov. 2002), ACM, pp. 174-190

[21] Stein, D., Hanenberg, St., Unland, R., *A UML-based Aspect-Oriented Design Notation For AspectJ*, Proc. of AOSD '02; Enschede, Netherlands, April 2002, ACM, pp. 106-112

[22] Stein, D., Hanenberg, S., Unland, R., *A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models*, in: Proc. of MDA-FA '04, Linköping, Sweden, June 2004, Springer, LNCS 3599, pp. 77-92

[23] Stein, D., Hanenberg, S., Unland, R., *Query Models*, in Proc. of UML '04, Lisbon, Portugal, October 2004, Springer, LNCS 3273, pp. 98-112

[24] Stein, D., Hanenberg, S., Unland, R., *Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design*, in Proc. of AOSD'06, Bonn, Germany, March 2006, ACM, pp. 15-26

[25] Störzer, M., Hanenberg, S.: *Classification of Pointcut Language Constructs*. Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT) held in conjunction with AOSD 2005, Chicago, Illinois, USA, March 15, 2005.

[26] Suvee, D., Vanderperren, W., Jonckers, V., *JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development*, in Proc. of AOSD'03, Boston, USA, March 2003, ACM, pp. 21-29

[27] Tarr, P., Ossher, H., *Hyper/J User and Installation Manual*, IBM Corp., 2000

[28] Vanderperren, W., Suvee, D., Cibran, M., De Fraine, B., *Stateful Aspects in JasCo*, in: Proc. of SC 2005, LNCS, April 2005