On the Role of Method Families in Aspect Oriented Programming

Katharina Mehner and Annika Wagner
Department of Mathematics and Computer Science, University of Paderborn
33095 Paderborn, Germany
Email: {mehner, awa}@uni-paderborn.de

Introduction

Aspect oriented programming aims at avoiding cross-cutting and code tangling phenomena by separating the basic functionality from aspects like synchronisation, distribution or memory optimisation. Different techniques for this purpose have been proposed [1, 2, 4, 5, 6, 8, 9]. All of them have in common that methods form the interface between the basic functionality and the aspects.

In this paper we propose method families, i.e. equivalence classes of methods, together with operations to modify them as an important concept for the reusable definition of aspects. Usually different methods show the same behaviour seen under the perspective of an aspect. Hence it seems to be very natural to put additional structure on this interface between the basic functionality and the aspect. Organising methods in families allows to factor out common behaviours increasing the reuse of aspect code within an aspect. Moreover the possibility of dynamically deriving method families from existing method families using operations also improves the reusability of the whole aspect.

Method families go back to the concept of method sets in concurrent object oriented programming languages. Hence the proposed operations are set theoretic ones like union, extension, intersection and complement. Nevertheless we are not speaking of method sets but of method families to point out that these sets are defined under the perspective of common properties.

In the following we demonstrate two different uses of method families within concurrent object oriented programming. The first following the classical idea of accept sets [3, 7], the second driven by the aspect oriented paradigm. At the end we discuss the consequences of the two domain specific proposals for general aspect oriented programming approaches.

The origin of method families in object oriented concurrent programming

In this section we look at the origin of method families and analyse the way in which they are used. Method families were first used in non-standard object oriented concurrent programming in order to describe synchronisation schemes [3, 7]. As an example we present the approach of Yonezawa et al. [7].
Synchronisation schemes restrict the executability of the methods in the interface of an object. Apart from exclusion constraints it is most often the case, that method invocation depends on the abstract state of the object. Moreover different methods may have the same dependencies. Under this perspective it is natural to classify methods according to their execution conditions, i.e. all methods which require the same abstract state belong to the same method family.

This idea is followed with the concept of accept sets. Accept sets are declared in the scope of a class. In figure 1 the synchronisation scheme of the well-known bounded buffer example is shown on the left hand side. In this example we have three accept sets EMPTY, FULL and PARTIAL containing the methods put, get and put as well as get respectively. The accept set PARTIAL is derived from the other two sets by using the union operation.

$$\text{Class b-buffer}$$

method sets:

mset EMPTY #{put};
mset FULL #{get};
mset PARTIAL EMPTY | FULL;

synchronizers:

(0 < size && size < MAX_SIZE) enables PARTIAL;
(size == 0) enables EMPTY;
(size == MAX_SIZE) enables FULL;

$$\text{Class b-buffer2}$$

method sets:

mset AFTER-PUT #{gget};
mset FULL super FULL | AFTER-PUT;

transitions:

transition put() {
  disable_once AFTER-PUT;
}

$$\text{Figure 1: Synchronisation of bounded buffer using accept sets}$$

In order to apply these sets for achieving synchronisation we need the notion of active or enabled accept sets. If a set is enabled, its methods can be executed. An accept set is either in the state „enabled“ or „disabled“. There are two different mechanisms to determine the enabled accept set, i.e. to set the accept set's state to enabled: synchronizer statements and set transitions. Using synchronizers the condition for execution of methods is defined in terms of the abstract state of an object, which can be determined via accessing the fields of the object. However, a guard cannot access information about execution history, which of course should not be coded into the fields of an object.

An example of synchronizers is again the synchronisation scheme of the bounded buffer depicted on the left hand side of figure 1. Here conditions dealing with the amount of objects already contained in the buffer are bound to the accept sets EMPTY, FULL and PARTIAL. Note that a method is accepted if it is contained in at least one enabled accept set.

In order to have more powerful execution conditions, set transitions are introduced. With this technique the accept sets are not only seen as the methods executable in a certain state. Additionally accept sets are used as place holders for the states themselves, which makes it possible to define transitions between accept sets.

If the bounded buffer is extended with a method gget that is only executable directly after a call to method get the synchronisation scheme is defined using transitions, as shown on the right hand side of figure 1. Here the accept set AFTER-PUT is disabled by a put operation for the next execution step.
Also in this example we have a derived method set. \texttt{FULL} is defined as the union of its extension in the superclass and \texttt{AFTER-PUT}. Moreover the derived method set \texttt{PARTIAL} defined in the superclass is automatically updated in the subclass, since method \texttt{gget} is added to the method set \texttt{FULL}, which is used in the definition of \texttt{PARTIAL}. With this mechanism one avoids the redefinition of already existing parts of the synchronisation scheme for new methods.

To summarise method families serve to group methods according to common restrictions on their execution. Moreover we have seen that this allows a compact definition of synchronisation schemes. Operations as exemplified by the extension and union of method sets allow the definition of derived method sets, which are dynamically updated. This mechanism eases reuse of synchronisation schemes.

\textbf{An alternative use of method families}

In this section we are going to present a language which follows another idea for defining equivalence classes on methods. Our approach is heavily influenced by the ideas of aspect oriented programming. In detail, we exploit the use of method families based on a prototype language for concurrent AOP, namely COOL \cite{5,8}.

Our approach is similar to Yonezawa et al. \cite{7} in that we also use method sets in combination with execution requirements. The requirements are specified similar to the synchronizer statements. However, we have nothing like transitions. To encode information of the abstract object state and especially history information the synchronisation code is extended with boolean variables called \texttt{conditions}. These conditions can be used in specifying requirements for method execution. After method execution they can be assigned new values, which can be determined in terms of the actual values of conditions and of the state of the object. The \texttt{requirement} and \texttt{changes} sections are associated with method families. The idea of requires-and changes- sections is taken from COOL \cite{5,8}.

We demonstrate this using again the example of a bounded buffer (cf. figure 2). It starts with a declaration of two condition variables which are assigned default values. The declaration of a method family is followed by a block structure in which the requirements for method execution are defined as well as the changes to be carried out after method execution.

```
class condition FULL=false;
class condition EMPTY=true;

class mset #GET = (get) {
    requires (!EMPTY);
}

class mset #PUT = (put) {
    requires (!FULL);
}

class mset #PUTGET = #GET|#PUT {
    changes {
        FULL to (count == MAX);
        EMPTY to (count == 0);
    }
}
```

Figure 2: Bounded Buffer with method families with requires and changes
The changes- and requires- sections are optional. Our languages is also able to specify exclusion constraints, to handle requirements for execution which involves the actual parameters, and apart from condition variables it also allows variables of other simple types, e.g. to realise counters.

Compared to Yonezawa et al. [7] our approach uses only one technique to define requirements, while in his approach one has to choose between synchronizers and set transitions. With our approach, method families are freed from the task to represent abstract state or history information, as synchronisation state can be represented with condition variables. We think that we simplify things for programmers, as it might be easier to think in terms of side effect on condition variables than to manage a state transition system.

At this point we can already compare this approach to [7] concerning the equivalence classes behind method families. Our approach is similar to [7] approach in that it also allows to group methods according to the execution requirements. It is different in that it allows to define a second kind of equivalence class, as it is possible to group methods also according to their changes on the abstract state and history state. Concerning the design of programmes, this gives more flexibility for factoring out common properties and thus different encapsulation strategies can be pursued.

Next we investigate how we can add suitable method operations for refinement to our language. First of all we have to note, that COOL does not provide techniques for refining or inheriting aspects. So these operations where taken over from Yonezawa et al. [7]. We illustrate the refinement techniques of our language with the same extension for the bounded buffer as before (cf. figure 3). Adding the method \texttt{gget()} requires to encode a history information which is added with the condition \texttt{GETTED}. The keyword \texttt{super} is used in the same way as in [7] approach. Another important method operation applied here is the \texttt{all-except} constructor for method sets, which was also taken over from [7]. It is used to define a complement and is a powerful construct as its result will always be recomputed in derived classes. This means, that newly added methods are automatically added to \#\texttt{NOTGET}, if they are not added to \#\texttt{GETONLY}. Thus the properties specified for \#\texttt{NOTGET} serve as the default value in derived classes for new methods if not specified otherwise.

\begin{verbatim}
condition GETTED=false;
   mset \#GET = super \#GET|\{gget\};
   mset \#GETONLY = \{get\} {
    changes \{GETTED to true;\}
   }
   mset \#NOTGET   = all-except \#GETONLY {
    changes \{GETTED to false;\}
   }
   mset \#GGETONLY = \{gget\} {
    requires \{GETTED && !EMPTY\};
   }
\end{verbatim}

Figure 3: Refinement of synchronisation aspects

It is important to note that our approach also avoids the classical inheritance anomalies, because the refinement techniques allow to incrementally add additional requirements and changes to existing method families.
To summarise we have demonstrated that method families can not only serve as *equivalence class for common execution conditions* but also as *equivalence class for common changes on the abstract state an history state*. In our approach these two equivalence classes can also be combined, if a method family is associated with both requirements and changes. We have also introduced a more imperative style of specifying the requirements and changes. This style was influenced by AOP, because in general aspects are concerned with additional actions to be carried out before and after method execution.

Furthermore we have illustrated how method set operations can foster reuse inside an aspect and also across derived aspects. Method families can be extended and new method families can be derived of existing ones. We paid special attention to dynamic derivation of method families, i.e. constructs which affect also future subclasses. They are a powerful means for reuse.

**Method families in general aspect oriented programming**

So far we have considered method families only for domain specific languages. In this section we sketch our ideas how method families can be applied to general purpose aspect oriented programming languages and show that they can be a useful extension, however this is not intended to be a complete treatment of this subject. As our example serves AspectJ from Xerox Parc [6].

AspectJ allows to define different kind of aspects, e.g. aspects for all instances of a class and aspect instances to be used with one or more instances of a class or several classes. Within an aspect one can define so called *weaves*. These contain pieces of normal code which are woven into the class code according to the weave mode. The weave mode determines e.g. whether a weave is executed *before* or *after* method execution or on *initialisation* of objects. Weaves can be defined also for a set of methods, although this is done in writing down a list of methods. These sets are not assigned to identifiers and hence not reusable.

```java
aspect LogAccess {
    static DataOutputStream log = null;
    static {//initialise
        try {
            log = new DataOutputStream (new FileOutputStream("log"));
        } catch ( IOException e) { }
    }
    static after Point.set, Point.setX, Point.setY,
    Line.set, Line.setX1, Line.setX2, ...
    {
        try {
            log.writeBytes (thisObject.toString() + "write access" + "\n")
        }catch ( IOException e) { }
    }
    static after Point.get, Point.getX, Point.getY,
    ...
    {
        try {
            log.writeBytes (thisObject.toString() + "read access" + "\n")
        }catch ( IOException e) { }
    }
}
```

Figure 4: A general purpose aspect for writing a log
AspectJ provides inheritance between aspects. Here we want to investigate how weaves can be reused in inherited aspects. We look at two cases: (1) the use of existing weaves for new methods and (2) the extension of weaves with new code. We illustrate these cases with two examples. Consider several classes `Point`, `Line`, etc. with read and write methods. The calls to them have to be logged in a file. Figure 4 shows how this is done with an aspect.

Now consider the requirement, that a new write operation is added e.g. to class `Point`, which also should be logged to the same file. As there is no possibility to add it to the method list of an existing weave we need to define a new weave for this method. Then we would have to copy the whole piece of code of the weave, if it is not accessible via one method. But what if the code was not encapsulated in a method because it was not taken into account that someone would reuse it? Having an identifiable method family which could be extended with new methods however would allow an easy extension of this aspect in any case.

```java
// aspect
mset WRITERS = {Point.set, Point.setX, Point.setY,...}
static after WRITERS {...}
// derived aspect
mset WRITERS = super WRITERS | {newMethod}
```

The constructor `all-except` could provide for an automatic categorisation of all new methods as writers as a default if not specified otherwise.

```java
mset READERS = {Point.get, Point.getX, Point.getY,...}
mset WRITERS = all-except READERS
```

In the second case we consider the requirement, that several weaves should be extended with the same code. E.g. in the example in figure 4 the two existing weaves for readers and writers should both be extended with code to write also the time into the log file for each method call. Even if the weave code would have been encapsulated in two methods e.g. `writerToLog(obj: Object)` and `readerToLog(obj: Object)` it would be necessary to override both methods using calls to super and the method encapsulating the writing of the new log information:

```java
// derived aspect
writerToLog(obj: Object) {
  super;
  writeTimeToLog(),
}
readerToLog ...
```

However if the two weaves would be based on two method families, then we would proceed as follows:

```java
R&W := READERS|WRITERS
static after R&W { writeTime() }
```

Our examples show that method families are a helpful extension also for general AOP. They have the same advantages as for the domain specific languages presented before. They provide reuse inside an aspect and between aspects and derived aspects. Also the concepts of dynamically derived method families which are automatically updated are applicable.
Conclusion and future work

In this paper we have shown that method families are an important concept for achieving reuse of aspect code. Together with set theoretic operations on method families we can achieve reuse of aspect code within an aspect and especially during inheritance of aspects where we can have derived method families which are dynamically updated. This was discussed in detail for two approaches from the domain of concurrent object oriented programming. When we examined a general purpose AOP language it turned out that it does not yet apply method families although we could demonstrate already exemplary that it would improve reuse within and across aspects during inheritance.

Within domain specific (aspect) languages for synchronisation we showed that method families are used as equivalence classes defined according to execution conditions or changes in execution conditions. These kinds equivalence classes are bound to the language support. With general AOP this is not the case, but it could be a conceptual means for a good design.

As a next step we would like to examine what we can improve, if we apply method families to Composition Filters. Here the situation appears to be the same as with AspectJ in that filters are also specified method oriented, but without declaring method families.

Currently we are implementing our proposed language using the weaving technology. We are considering the implementation at the same time to be a kind of feasibility study which will show how the the reuse mechanisms including method families as proposed for aspects can be weaved respectively how much additional framework is needed for the implementation of the dynamic operations on method families.

Literature