Building Composable Aspect-Specific Languages with Logic Metaprogramming

Johan Brichau[1]*, Kim Mens[2], and Kris De Volder[3]

[1] Vrije Universiteit Brussel,
Vakgroep Informatica,
Pleinlaan 2,
B-1050 Brussel, Belgium
johan.brichau@vub.ac.be

[2] Université catholique de Louvain,
Place Sainte-Barbe 2,
B-1348 Louvain-la-Neuve, Belgique
kim.mens@info.ucl.ac.be

[3] University of British Columbia,
309-2366 Main Mall,
V6T 1Z4 Vancouver, BC, CANADA
kdvolder@cs.ubc.ca

Abstract. The goal of aspect-oriented programming is to modularize crosscutting concerns (or aspects) at the code level. These aspects can be defined in either a general-purpose language or in a language that is fine-tuned to a specific aspect in consideration. Aspect-specific languages provide more concise and more readable aspect declarations but are limited to a specific domain. Moreover, multiple aspects may be needed in a single application and composing aspects written in different aspect languages is not an easy task.
To solve this composition problem, we represent both aspects and aspect languages as modularized logic metaprograms. These logic modules can be composed in flexible ways to achieve combinations of aspects written in different aspect-specific languages. As such, the advantages of both general-purpose and aspect-specific languages are combined.

1 Introduction

The idea of *separation of concerns* [16] is that the implementation of all concerns in a software application should be cleanly modularized. Today's existing programming techniques have succeeded to support the separation of concerns principle at the code level to a reasonable degree. However, some concerns cannot be modularized using the existing modularizations and tend to crosscut with other concerns. Common examples of such concerns are synchronisation, persistence and error-handling. Aspect-oriented programming (AOP) [8] modularizes such crosscutting concerns as aspects. These aspects are expressed in one or

* Research assistant of the Fund for Scientific Research – Flanders (Belgium) (F.W.O.)

more aspect languages and they are composed with the rest of the program by an aspect weaver.

An aspect language designed to express a specific kind of aspect is highly desirable because it results in more concise and more intentional aspect declarations, making it easier to read and write aspect code. A testament to this is the fact that many of the first aspect languages were aspect-specific [6,10,11] and *not* general purpose. On the other hand, aspect-specific languages are very powerful within their specific scope but they can only be applied to the specific aspect they were designed for. Because of this, many AOP-related techniques [1,7,15] offer a general-purpose aspect language which allows to express many kinds of aspects as well as combinations and interactions between them. The latter becomes more complex when the aspects are implemented in different aspect-specific languages. Our approach to AOP is based on logic metaprogramming (LMP) [12,13,14,20,21]. In a previous paper [4], we explored the use of LMP as an open and extensible aspect-weaver mechanism that facilitates the specialization of a general-purpose aspect language and, as such, facilitates the building of aspect-specific languages (ASLs). In this paper we focus on the combinations and interactions between aspects written in different ASLs. We extend the work in [4] and introduce logic modules to encapsulate aspects and implementations of ASLs. These logic modules provide a composition mechanism that allows us to combine aspects or implement interactions between aspects written in different ASLs. This is made possible because all our ASLs share the same Prolog-like base language. In other words, we obtain a modular aspect-weaver mechanism that offers the generality of a general-purpose aspect language without loosing the ability and advantages of defining aspects in aspect-specific languages. In addition, we offer a means of composing and regulating the interactions among different aspect-specific languages. In section 2, we introduce a software application that is used as a running example throughout the remainder of the paper. In section 3, we describe what an aspect language embedded in a logic language looks like and how it supports modularization of crosscutting concerns. Section 4 explains how ASLs are implemented and section 5 shows how aspect composition and interaction issues are handled. We briefly introduce a prototype research tool for our approach in section 6. Sections 7 and 8 discuss future and related work.

2 Case: The Conduit Simulator

2.1 Basic Functionality

Our running example is a conduit simulator, implemented in Smalltalk and which allows to simulate the flow of fluid in a network of conduits. A conduit system can be built using 4 types of conduits: pipes, sources, sinks and joins. An example of a conduit-system is shown in figure 1 and a simplified class diagram of the implementation is shown in figure 2.

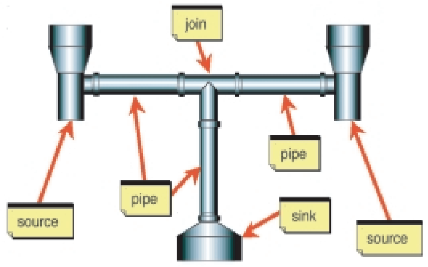


Fig. 1. A Conduit-system.

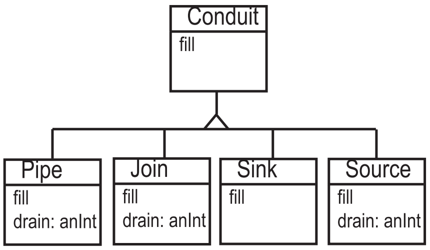


Fig. 2. Class diagram of the Conduit Simulator.

Each type of conduit is implemented by a single class. A conduit-system is built by linking each conduit to an incoming conduit from which it should receive fluid[1]. The basic behaviour is implemented in two methods:

#drain: Each drainable conduit (source, pipe and join) understands the message `#drain:` which can be used to drain an amount of fluid from it.

#fill The `#fill` method of each conduit tries to fill the conduit by draining the incoming conduit(s). A source conduit fills itself based on a profile.

All conduits simultaneously run a looping process that executes the `#fill` method, i.e. conduits are active objects that continuously drain their incoming conduit(s). As a result, fluid will flow from sources to sinks.

2.2 Crosscutting Functionality

Making the conduit simulator work correctly requires us to deal with some crosscutting concerns.

Synchronizing and Order of Execution. A conduit can only be drained after each time it has been able to fill itself. Therefore, the `#fill` and `#drain:` methods can only be executed in alternating order.

This can be done by inserting synchronisation code at the beginning and at the end of both these methods. Obviously, this leads to tangled functional and non-functional code. In the rest of the paper, we will refer to this aspect as the 'order of execution' aspect.

User Interface (UI). We also need to visualize the real-time simulation. Therefore, we create views for each type of conduit and use an Observer design pattern to couple them. The code for this pattern also crosscuts the implementation of all conduit types.

[1] Join conduits are linked to two incoming conduits.

Logging. For debugging purposes, we want to log the execution of the `#drain:` and `#fill` methods, which also amounts to the introduction of code in the beginning and at the end of those methods. Once again, this would tangle logging code with functional code. Also, this addition requires the insertion of very similar code in many places. It is also important to note that writing to the log should also be synchronized.

In the following section, we explain how to write the logging aspect in a general-purpose aspect language (implemented using our LMP approach) and we elaborate on building aspect-specific languages for all aspects of the conduit simulator in section 4.

3 Aspects in a Logic Language

In LMP, we use a logic programming language to reason about object-oriented base programs. The metalevel description of the base-language program consists of logic facts and rules [20]. In the context of AOP, the logic language also serves as the surrounding medium in which we embed our aspect languages. This provides a general framework for declaring and implementing aspects.

3.1 Aspects as Logic Modules

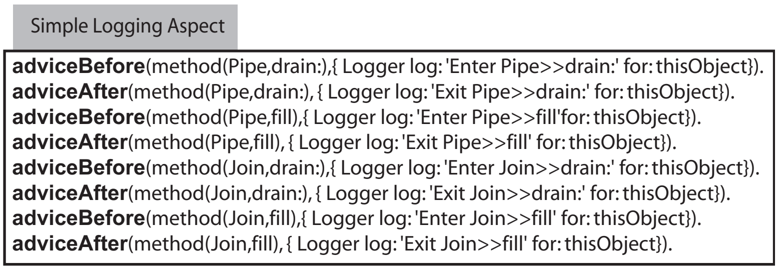
An aspect language is specified as a set of logic predicates, as is shown in table 1 for a general-purpose 'advice' aspect language (similar to advices in AspectJ [7]). An aspect in this aspect language is implemented as a set of logic declarations of these predicates, contained in a logic module. An example aspect is shown in figure 3 (this is only a first and simple version of the aspect that will be improved in later sections). The logic inference engine becomes the weaver, which gathers all the logic declarations that are present in such a module. The weaver (for a particular aspect language) understands the declarations and knows how to weave the aspect into the base program. In the code fragments, all predicates that are part of an aspect language are shown in bold.

We first provide some details about the syntax of our logic language:

– We have a special logic term (delimited by curly braces: '{' and '}') to embed base program text in the logic declarations. This term can even contain logic variables (their use is explained later, also see [4]).
– Logic variables start with a question mark (e.g. `?variable`).
– The modules basically encapsulate logic declarations. Each logic declaration belongs to a module and is only visible in the module where it is defined.
– Modules can be composed to use declarations of another module or to make declarations in one module visible in another module. How this is done is explained later.

Table 1. A simple advice aspect-language (similar to advices in AspectJ).

Predicate	Description
adviceBefore(?m,?c)	Execute code fragment ?c before executing method ?m
adviceAfter(?m,?c)	Execute code fragment ?c after executing method ?m



Simple Logging Aspect

```
adviceBefore(method(Pipe,drain:),{ Logger log: 'Enter Pipe>>drain:' for: thisObject}).
adviceAfter(method(Pipe,drain:), { Logger log: 'Exit Pipe>>drain:' for: thisObject}).
adviceBefore(method(Pipe,fill),{ Logger log: 'Enter Pipe>>fill'for: thisObject}).
adviceAfter(method(Pipe,fill), { Logger log: 'Exit Pipe>>fill' for: thisObject}).
adviceBefore(method(Join,drain:),{ Logger log: 'Enter Join>>drain:' for: thisObject}).
adviceAfter(method(Join,drain:), { Logger log: 'Exit Join>>drain:' for: thisObject}).
adviceBefore(method(Join,fill),{ Logger log: 'Enter Join>>fill' for: thisObject}).
adviceAfter(method(Join,fill), { Logger log: 'Exit Join>>fill' for: thisObject}).
```

Fig. 3. Logging aspect in the advice aspect language

The aspect shown in figure 3 implements the logging for the conduit-system in the aspect language defined in table 1. The `Logger` class keeps a log for each conduit. The logic declarations inform the weaver that some code fragments must be executed before and after the methods `#drain:` and `#fill` in the classes `Pipe` and `Join`. Technically, the weaver gathers all `adviceBefore` and `adviceAfter` declarations and weaves the code fragments in the indicated methods. The `thisObject` keyword is understood by the weaver and is a reference to the object in which the code fragment is executed.

3.2 Logic Pointcuts

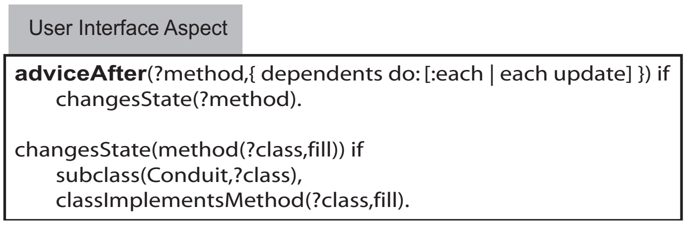
An aspect describes where or when its functionality should be invoked in terms of joinpoints. Pointcuts are sets of joinpoints. In our approach, joinpoints are specific points in the base program's code[2].

The primitve example above expressed an aspect by directly applying advices to individual joinpoints. Adequately expressing aspects also requires a mechanism to abstract over sets of joinpoints (pointcuts) and factor out commonalities between aspect code applied over all of them [3]. This involves (1) a pointcut mechanism to characterize sets of joinpoints, (2) a mechanism of parameterization that allows the aspect's code to have joinpoint-specific behavior. In the LMP approach, both these mechanisms are supported through the use of logic rules. We now discuss each mechanism in more detail.

Defining Pointcuts. Part of the implementation of the observer pattern (for the UI-aspect) is the insertion of code at well defined joinpoints, that triggers the observable's update mechanism (hence, updating the UI). Defining a separate

[2] Some experiments with dynamic joinpoints in LMP have been conducted in [5].

advice fact for each of those joinpoints would result in a lot of code duplication, because the advice is identical for each joinpoint. A better way to define the UI-aspect is through the use of logic rules, which are a way to define a set of similar of facts.



User Interface Aspect

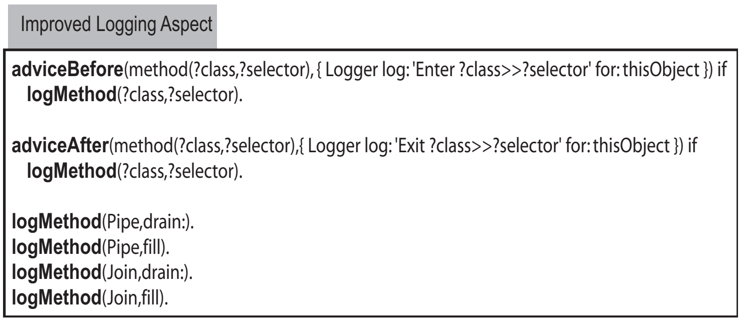
```
adviceAfter(?method,{ dependents do: [:each | each update] }) if
     changesState(?method).

changesState(method(?class,fill)) if
     subclass(Conduit,?class),
     classImplementsMethod(?class,fill).
```

Fig. 4. Logic module implementing the UI-aspect.

The implementation of the ‘update’ part of the UI aspect is shown in figure 4. The first logic rule declares `adviceAfter` facts for each joinpoint that is matched by the pointcut defined by `changesState` declarations. The second logic rule defines this pointcut as the `#fill` method of each subclass of `Conduit`. The `subclass` and `classImplementsMethod` predicates are part of a predefined logic library of predicates to reason about Smalltalk code (see [20]).

This example was particularly easy, because the code is identical for each joinpoint of the pointcut. However, an aspect becomes significantly more complex if the code requires variations dependent on the specific joinpoint.

Joinpoint-Dependent Variations. The logging aspect in figure 3 is an example of an aspect that inserts a pattern of code containing joinpoint-dependent variations, i.e. the name of class and selector. We can capture these variation points using logic variables, embedded in the code pattern. Using this technique, we can implement a more generic logging aspect, as is shown in figure 5.



Improved Logging Aspect

```
adviceBefore(method(?class,?selector), { Logger log: 'Enter ?class>>?selector' for: thisObject }) if
   logMethod(?class,?selector).

adviceAfter(method(?class,?selector),{ Logger log: 'Exit ?class>>?selector' for: thisObject }) if
   logMethod(?class,?selector).

logMethod(Pipe,drain:).
logMethod(Pipe,fill).
logMethod(Join,drain:).
logMethod(Join,fill).
```

Fig. 5. Logic module implementing the logging aspect.

The code pattern in the `adviceBefore` and `adviceAfter` declarations is now parameterized by logic variables `?class` and `?selector`. The weaver uses the inference engine to substitute them with their specific bindings, dependent on the joinpoint that the advice is woven into.

We have now explained what an aspect language embedded in a logic language looks like and how it is suited to describe crosscutting concerns that should be woven in the base program code. We now elaborate on the use and composition of logic modules to implement and use aspect-specific languages.

4 Aspect-Specific Languages

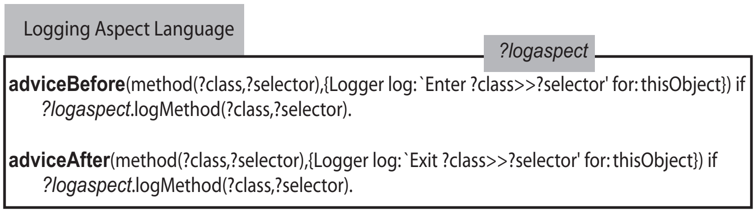
Aspect languages are implemented through logic rules in logic modules. These rules define the meaning of an aspect language in terms of a more primitive aspect language. This eases the implementation of ASLs because weavers for them do not have to be implemented from scratch. Typically, we have a primitive aspect weaver that implements a low-level, general-purpose aspect language on which other aspect-specific languages can be built. In the following subsections, we illustrate this with the construction of two aspect languages for logging and 'order of execution' in our conduit simulator. Both of these aspect languages are defined in terms of the more general-purpose advice aspect language.

Table 2. A simple logging aspect language.

Predicate	Description
logMethod(?c,?m)	Log the execution of the method ?m in class ?c.

Logging Aspect Language. The logging aspect of figure 5 already defined a simple aspect language for logging. The aspect language consists of a single predicate and is shown in table 2. The first two rules in figure 5 define the meaning of the logging aspect language in terms of the advice aspect language. The remaining facts constitute the implementation of the aspect. But the logic module in figure 5 contains both the logging aspect itself and the implementation of the logging aspect language. To facilitate reuse of the implementation of the aspect language, we prefer to separate the aspect implementation from the aspect language's implementation. To achieve this, we split this module in an *aspect language module* and an *aspect definition module* and provide a composition mechanism to compose both modules.

The logic rules that implement the logging aspect language are placed in a separate aspect-language module. But now, these logic rules should gather the required `logMethod` declarations in a separate aspect-definition module, which depends on the particular application in which the aspect language is used. Therefore, we parameterize the aspect-language module using a module-variable, which can be bound to a specific aspect-definition module (containing logMethod facts) in the context of a particular application. In the code fragments, all module-variables are shown in italic.



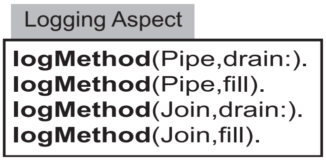
Logging Aspect Language

?logaspect

```
adviceBefore(method(?class,?selector),{Logger log:`Enter ?class>>?selector' for: thisObject}) if
    ?logaspect.logMethod(?class,?selector).

adviceAfter(method(?class,?selector),{Logger log:`Exit ?class>>?selector' for: thisObject}) if
    ?logaspect.logMethod(?class,?selector).
```

Fig. 6. Aspect-language module implementing the logging aspect language.

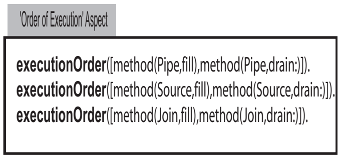
Figure 6 shows the aspect-definition module for the logging aspect language implemented in terms of the advice aspect language. The logMethod declarations will be gathered in the logic module that is bound to the ?logaspect module-variable. Hence, the ?logaspect module-variable parameterizes this logic module with another logic module. In the implementation of a particular application, we can bind the module-variable with an aspect definition module implementing a logging aspect, such as the one shown in figure 7.



Logging Aspect

```
logMethod(Pipe,drain:).
logMethod(Pipe,fill).
logMethod(Join,drain:).
logMethod(Join,fill).
```

Fig. 7. Logging aspect that is understood by the logging aspect language implemented in figure 6.



'Order of Execution' Aspect

```
executionOrder([method(Pipe,fill),method(Pipe,drain:)]).
executionOrder([method(Source,fill),method(Source,drain:)]).
executionOrder([method(Join,fill),method(Join,drain:)]).
```

Fig. 8. 'Order of execution' aspect that is understood by the 'order of execution' aspect language of table 3.

While this example is rather simple, the use of ASLs is particularly interesting for aspects that can be reused in many different contexts (such as the more technical aspects that implement non-functional requirements like synchronization, distribution, ...) and where the code of the aspect is more complicated. The ASL shields the developer from the burden of the implementation while still enabling him to tailor the functionality of the aspect (to the extent that the ASL allows it).

'Order of Execution' Aspect Language. The above logging aspect language only allows to specify joinpoints for the logging aspect. The 'order of execution' aspect is much more interesting because the aspect language we constructed for it provides ‘hooks’ that allow us to add behaviour to the aspect. The logic module shown in figure 8 is an aspect-definition module, implementing an aspect for our conduit simulator in the ‘order of execution’ aspect language. This language consists of three logic predicates, described in table 3. The last two predicates define hooks that allow the user of the aspect language to specify additional code that will be inserted in the implementation of the aspect.

Table 3. ‘Order of Execution’ aspect language.

Predicate	Description
executionOrder(?list)	The `?list` argument of this predicate is a list of methods that should be executed in mutual exclusion and in order of occurence in the list. After the last method in the list is executed, the first method can again be executed.
onBlock(?m,?c)	Execute code `?c` when the method ?m is blocked by the synchronisation guards.
onStart(?m,?c)	Execute code `?c` when the method ?m is given permission to execute by the synchronisation guards.

The ‘order of execution’ aspect for our conduit simulator is shown in figure 8. In figure 9, we show part of the aspect-language module implementing the ‘order of execution’ ASL in terms of the advice language. Basically, the first two rules respectively describe before advices and after advices that insert a simple synchronization algorithm (using semaphores) on each of the methods given in the `executionOrder` declarations. The auxiliary `orderDependence` rule is part of the internal implementation of the ASL to compute which semaphores should be used by that particular advice and to gather the additional code fragments in the optional `onBlock` and `onStart` declarations.

'Order of Execution' ASL

```
adviceBefore(?jp,{ globalSema wait. (semaphores at: ?position)
                                    waitAndExecute:[?onBlock. globalSema signal].
                    ?startCode }) if
  orderDependence(?jp,?position,?nextPosition,?blockCode,?startCode).

adviceAfter(?jp,{ (semaphores at: ?nextPosition) signal }) if
  orderDependence(?jp,?position,?nextPosition,?blockCode,?startCode).

orderDependence(?jp,?currentPos,?nextPos,?blockCode,?startCode) if
  ?orderAspect.executionOrder(?list),
  computePositions(?list,<?jp,?currentPos,?nextPos,?blockCode,?startCode>).
  ...
```

Fig. 9. Part of the aspect-language module implementing the ‘Order of Execution’ ASL.

We have shown how to build aspect-specific languages on top of a more general-purpose aspect language. These ASLs can be reused as black-box entities in many different development contexts through the use of logic rules in logic modules. We do not claim that the actual implementation of an ASL is a simple process. One still has to design an appropriate language and implement its semantics in terms of another (more low-level) aspect language. Also, the fact that all ASLs in LMP remain embedded in the same logic language, obviously bears some advantages as well as disadvantages. On the one hand, it ensures a common medium to express the composition of all aspects in these aspect-languages. On the other hand, no aspect-specific syntax is provided. However, we feel that the advantage is far more greater than the disadvantage because the combination of multiple aspects can raise many subtle and difficult issues that should be tackled by the programmer [9,19]. Also, nothing prohibits us to extend the approach to allow for aspect-specific syntax on top of the underlying logic description. We are currently looking into that issue.

5 Composition and Interaction

Combining multiple aspects in a single application can raise problems that do not exist when the aspects are considered in isolation. For example, in our conduit simulator, combining the logging aspect with the 'order of execution' aspect poses some complications:

A: Logging of methods that are 'ordered'. How do we log methods that may block because of the 'order of execution' aspect?

- A1 Do we log entry to a method before or after checking the guards?
- A2 How do we log the fact that a method blocks?

B: Reducing synchronisation overhead for logging. To synchronize the `Logger` class, we use a synchronisation aspect. But the 'order of execution' aspect also synchronizes methods of each conduit and the log itself is different for each conduit. This means that if logging is only executed in the critical sections that are created by the 'order of execution' aspect, that it is safe to omit a supplementary synchronisation aspect. On the other hand, in some cases, we also might want to log other methods of a conduit than `#drain:` and `#fill`. In those cases, we do need proper synchronisation for the log aspect.

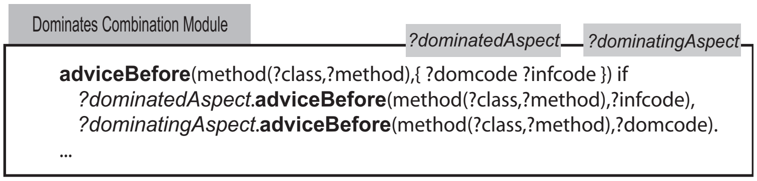
- B1 How do we automatically apply a synchronisation aspect to the logging aspect?
- B2 How do we reduce the amount of synchronisation code to be executed, based on interaction with the 'order of execution' aspect?

In the following subsections, we show how logic modules can be used to implement the aspect-combination complications mentioned above.

5.1 Combining Aspects

Aspects are combined using *aspect-combination modules*. A combination module is a logic module that is parameterized with several other modules and contains rules that describe how the functionality of these other modules is to be combined. This composition mechanism can be used to compose *aspect-definition modules*, as well as *aspect-language modules*. The aspect combination module then acts as their composition and therefore it is the only module to be handed to the weaver.

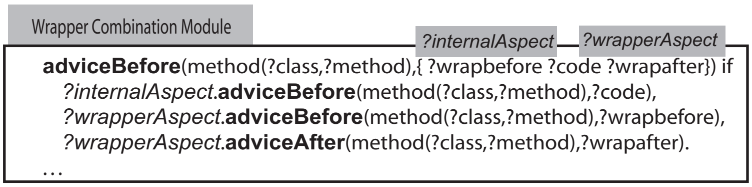
Dominates Combination Module. The aspect-composition problem A1 (order of the aspects) could be solved by prioritizing the aspects. This is a general solution which requires no domain-specific knowledge of the aspects themselves. Therefore it can be handled by a general type of prioritization rule. Figure 10 shows part of the 'dominates' combination module that prioritizes the advices generated by the aspect-language modules for logging and 'order of execution'. The rule that handles `adviceAfter` is identical. In figure 12, we show how the 'dominates' combination module is used to ensure that the 'order of execution' aspect (dominating aspect) is executed before the logging aspect (dominated aspect). The actual composition of the modules, as depicted in figure 12, can either be defined by a logic program or in a visual composition tool (see [2] for a prototype of such a tool).



```
adviceBefore(method(?class,?method),{ ?domcode ?infcode }) if
  ?dominatedAspect.adviceBefore(method(?class,?method),?infcode),
  ?dominatingAspect.adviceBefore(method(?class,?method),?domcode).
...
```

Fig. 10. The Dominates combination module to prioritize an aspect.

Wrapper Combination Module. Another kind of aspect-combination module is required for problem B1, where we merely want to wrap synchronisation code around the logging code. Using the previous 'dominates' combination module, this would result in wrapping synchronisation code around the entire method, instead of only around the logging code. Figure 11 shows part of a 'wrapper' combination module that produces the desired result. The rule fetches the before and after advice of the 'wrapper' aspect for every before advice of the 'internal' aspect and concludes a combined before advice. This combined before advice contains the 'internal' aspect's before advice, surrounded with the 'wrapper' aspect's advices. In our particular case, the 'wrapper' aspect is the synchronisation aspect and the 'internal' aspect is the logging aspect. Once again, rules that handle `adviceAfter` predicates are similar and are omitted. Also, for simplicity, we do not include the implementation of this synchronisation aspect here.



```
adviceBefore(method(?class,?method),{ ?wrapbefore ?code ?wrapafter}) if
  ?internalAspect.adviceBefore(method(?class,?method),?code),
  ?wrapperAspect.adviceBefore(method(?class,?method),?wrapbefore),
  ?wrapperAspect.adviceAfter(method(?class,?method),?wrapafter).
...
```

Fig. 11. The Wrapper combination module to wrap an aspect's advices around another aspect's advices.

Completely solving problem B and A2 requires some more interacting aspects. These are explained in the following section.

5.2 Interacting Aspects

Logging when a method blocks (problem A2), is conceptually more difficult, it cannot be solved simply using the 'dominates' or 'wrapper' combination module. It requires an explicit specialization of one of the aspects to adapt to the other one. This requires knowledge about both aspects and is most easily expressed in aspect-specific terms. In our approach, we make use of such high-level declarations and define intuitive logic rules that implement an interaction.

An *aspect interaction module* is implemented as a logic module, parameterized with module-variables. The difference with combination modules is that they do not combine several aspects in one aspect but implement a dependency or interaction between aspects. In other words, they modularize a crosscutting aspect. Interaction modules contain logic rules that are triggered by one aspect and add logic declarations to the other aspect. Furthermore, interaction modules do not compose aspects. To succesfully compose aspects that require an interaction, the interaction module should be used together with a combination module, as shown in figure 13.

Log methods that block. The logic module in figure 14 is an aspect interaction module that implements the desired interaction between the logging aspect and 'order of execution' aspect to solve problem A2. The logic rule adapts the 'order of execution' aspect by adding an additional `onBlock` declaration to it for each method that needs to be logged and 'ordered'. This is specified by starting the conclusion of the logic rule with a module variable, which will be bound to the 'order of execution' aspect module. As such, the rule makes it's conclusion visible in this module. The rule in figure 14 adds an `onBlock` declaration to the 'order of execution' aspect if the method needs to be logged and 'ordered'.

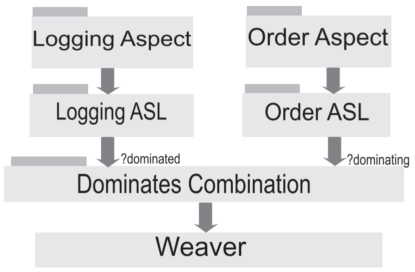


Fig. 12. Combining Logging and Order of Execution.

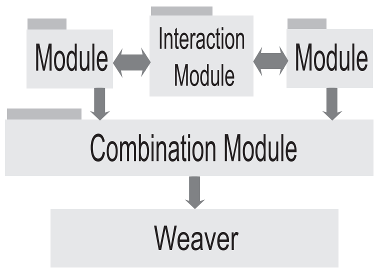
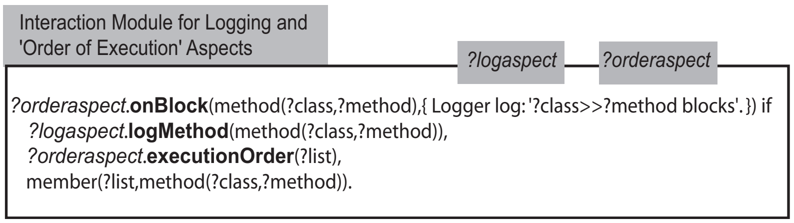


Fig. 13. Composing logic modules to implement interactions between aspects.


Interaction Module for Logging and 'Order of Execution' Aspects
?logaspect
?orderaspect

```
?orderaspect.onBlock(method(?class,?method),{ Logger log: '?class>>?method blocks'.}) if
   ?logaspect.logMethod(method(?class,?method)),
   ?orderaspect.executionOrder(?list),
   member(?list,method(?class,?method)).
```

Fig. 14. Interaction to log methods when they block.

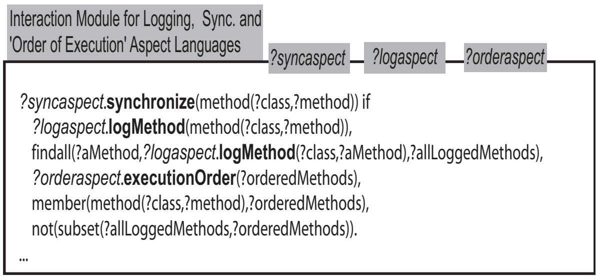
Synchronising the log. Another interaction module is required for problem B. In this example, the synchronisation aspect does not include a pointcut definition. Instead, it should fetch its pointcut definition from the logging aspect. The following logic rule could be used to implement such an interaction:

```
?syncaspect.synchronize(?method) if
    ?logaspect.logMethod(?method).
```

However, the logic rule above is too simple to tackle problem B2, which requires a more complex interaction module that also needs to interface with the 'order of execution' aspect. Since the log is different for each conduit and the 'order of execution' aspect synchronizes the `#drain:` and `#fill` of each conduit, it is safe to omit the synchronisation code of the log if logging only occurs in the critical sections of these methods. As we have seen in section 5.1, in the combination aspect for the 'order of execution' and logging aspects, logging code is 'dominated' by the 'order of execution' aspect's code.

The interaction module implementing this functionality uses the rule shown in figure 15. It specifies the pointcut of the synchronisation for the logging code. This pointcut contains all methods that need to be logged under the condition that all these methods are not a subset of the methods that are wrapped with the 'order of execution' aspect. Indeed, if it would be a subset, the pointcut is empty because in that case synchronisation of the log is already done by the 'order of execution' aspect.

To completely solve problem B, the interaction module should also be used with the dominates and the wrapper combination modules (from section 5.1), using the composition structure as shown in figure 16.


Interaction Module for Logging, Sync. and 'Order of Execution' Aspect Languages
?syncaspect
?logaspect
?orderaspect

```
?syncaspect.synchronize(method(?class,?method)) if
  ?logaspect.logMethod(method(?class,?method)),
  findall(?aMethod,?logaspect.logMethod(?class,?aMethod),?allLoggedMethods),
  ?orderaspect.executionOrder(?orderedMethods),
  member(method(?class,?method),?orderedMethods),
  not(subset(?allLoggedMethods,?orderedMethods)).
...
```

Fig. 15. Interaction to reduce synchronisation overhead.

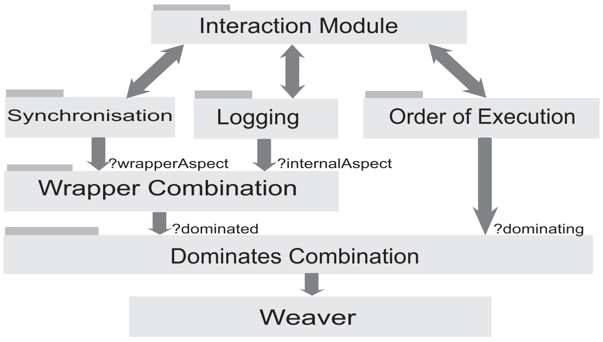


Fig. 16. Composition of logic modules to solve problem B.

6 Tool Support

The Soul/Aop system [2] is a prototype aspect-weaver that implements our logic metaprogramming approach to AOP in Smalltalk. It provides a hard-coded basic aspect language on which we can build our own ASLs using the techniques explained in this paper. The experiment in this paper was conducted using Soul/Aop.

The basic aspect language (table 4) supports wrapping of methods with aspect code as well as the definition of aspect-instance variables. Furthermore, the aspect code (defined in `wrap` declarations) can contain two special keywords (`original`, `thisObject`) that respectively allow access to the wrapped method and the executing object with which the aspect is woven.

The weaver itself consists of two parts: the logic inference engine and the low-level (Smalltalk) weaver. This low-level weaver launches queries to gather the logic declarations written in the basic aspect language and generates the appropriate Smalltalk constructs to produce the required behaviour described by these declarations. We can also say that the low-level weaver actually is the kernel of the Soul/Aop aspect-weaver and that the logic inference engine weaves the extensions described by the logic declarations in the various logic modules together and transforms them into the basic aspect language.

Table 4. Basic SOUL/Aop aspect language.

Predicate	Description
wrap(?m,?code)	Wrap/shadow the method `?m` with `?code`
instvars(?list,?scope)	Declares a list `?list` of aspect-instance variables of which the scope is defined as `?scope`. How this scope is specified is out of the scope of this paper (see [2]).

7 Future Work

Although the experiment with the conduit simulator is rather small, it was chosen specifically to illustrate how the LMP approach can be used to implement composable ASLs. This approach will now be used to investigate the many complex and interesting problems that can arise when combining aspects as well as aspect languages.

For the reason above, the logic modules have a flexible composition mechanism, which could even be more flexible when we extend it with the ability to override predicates in a logic module. For now, the ability to express interaction issues between ASLs is limited in terms of the expressiveness of the ASLs themselves. For example, the interaction to solve problem A2 relies on the *onBlock* predicate of the 'order of execution' ASL. Overriding of predicates would allow an interaction aspect to change the implementation of the ASL itself. As such, an interaction module itself could also have added the `onBlock` predicate to the 'order of execution' ASL.

Furthermore, the LMP approach presented in this paper uses static joinpoints, which are locations in the source code. LMP has also been used to express crosscutting on a dynamic joinpoint model [5]. The issue remains open wether this joinpoint model can be easily merged with the LMP approach we discussed. We also envision more fine-grained weaving than method-level wrapping as one of the future improvements of the Soul/Aop aspect-weaver.

8 Related Work

In [4], we explained how to use logic metaprogramming as a technology to implement extensible aspect weavers. However, no means for modularization of aspects and aspect languages was discussed, nor did we address the combination and interaction of aspects and aspect languages.

AspectJ [7] is an aspect-oriented extension to Java. Aspects are written like normal java classes, extended with pointcuts and advices. The *dominates* keyword accomplishes the same as our dominates combination. However, a combination such as the wrapper combination is harder to achieve because pointcuts cannot refer to advices. The modularization of interactions between aspects (or crosscutting aspects) is not yet supported. AspectJ also features the definition of abstract aspects through the use of abstract methods and abstract pointcuts. This allows to write aspects that can be reused and adapted and hide much of the implementation from the reuser. This is somewhat similar to what ASLs accomplish. But all combinations and interactions in AspectJ need to be expressed in general-purpose terms and not in more intuitive, aspect-specific terms.

An approach to validate combinations of aspects is presented in [9]. Aspects are augmented with specifications that describe the mutual exclusiveness or dependencies with other aspects. This allows to detect or prevent some faulty combinations of aspects. The approach provides a conflict-detection mechanism, but does not discuss how conflicting aspects could be combined.

In JAC (Java Aspect Components) [17], aspects can be wrapped around objects at run time. The precedence of wrapping is addressed by an explicit composition aspect written in a general-purpose language. Other adaptations to aspects, such as the interactions we discussed, are not addressed in this technique. An advantage is that the composition aspect can use dynamic information to decide on the composition.

In [18], a number of approaches to modularize crosscutting concerns are combined in a hybrid system. This system allows a developer to use the most applicable approach for the implementation of a given concern. Interactions between the different concerns are possible because the different approaches have been integrated in a (general-purpose) object-oriented approach.

9 Conclusion

In this paper we explained how aspect-specific languages can be implemented and combined using a logic metaprogramming approach. Logic metaprogramming provides a uniform and intuitive mechanism that reconciles the ability to build aspect-specific languages with the ability to compose aspects. The common logic medium facilitates the combination and interaction of aspects written in different aspect-specific languages. Furthermore, the logic modules that govern the interactions and combinations can use aspect-specific terms, which allows an intuitive description of the desired combination and interaction of the aspects involved.

References

1. L. Bergmans, M. Aksit, and B. Tekinerdogan. Aspect composition using composition filters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*, pages 357–382. Kluwer Academic Publishers, 2001.
2. J. Brichau. Soul/Aop weaver. http://prog.vub.ac.be/research/aop/soulaop.html.
3. K. De Volder. Code reuse, an essential concern in the design of aspect languages? Position Paper on Workshop on Advanced Separation of Concerns at ECOOP 2001, 2001.
4. K. De Volder and T. D'Hondt. Aspect-oriented logic meta-programming. In *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *LNCS*, pages 250–272. Springer-Verlag, 1999.
5. K. Gybels. Expressing crosscutting on a dynamic joinpoint structure using logic meta programming. Graduation thesis, Vrije Universiteit Brussel, 2001.
6. J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE*, volume 1343 of *LNCS*. Springer-Verlag, 1997.
7. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, LNCS. Springer-Verlag, 2001.
8. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.
9. H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect composition applying the design by contract principle. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering*, volume 2177 of *LNCS*, pages 57–69. Springer-Verlag, 2000.
10. C. V. Lopes and G. Kiczales. *D: A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, 1997.
11. A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, February 1997.
12. K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, 2000.
13. K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, pages 236–243. Knowledge Systems Institute, 2001.
14. T. Mens and T. Tourwe. A declarative evolution framework for object-oriented design patterns. In *Proceedings of Int. Conf. on Software Maintenance.* IEEE Computer Society Press, 2001.
15. H. Ossher and P. L. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of ICSE 2000*, pages 734–737, 2000.
16. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
17. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible and efficient solution for aspect-oriented programming in Java. Submitted to the Reflection Conference, 2001.

18. A. Rashid. A hybrid approach to separation of concerns: The story of SADES. In *Proceedings of the 3rd International Conference on Meta-Level Architectures and Separation of Concerns Reflection 2001*, volume 2192 of *LNCS*, pages 231–249. Springer, 2001.
19. P. L. Tarr, M. D'Hondt, L. Bergmans, and C. V. Lopes. Workshop on aspects and dimensions of concerns: Requirements on, and challenge problems for, advanced separation of concerns. In *ECOOP Workshop reader*, volume 1964 of *LNCS*, pages 203–240. Springer-Verlag, 2000.
20. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA '98*, 1998.
21. R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.