

Logic Meta Programming, a Framework for Domain-Specific Aspect Languages

Kris De Volder†, Johan Brichau*, Kim Mens, Theo D'Hondt

The *logic meta programming* (LMP) approach to AOP allows non-expert programmers to define, implement and use their own high-level, domain-specific aspect languages. Indeed, LMP provides a powerful and general framework for expressing aspects and implementing extensible weavers. It has an innate capability for expressing crosscutting using logic expressions as a natural mechanism for defining pointcuts. Furthermore, logic rules can be used to build domain-specific aspect languages easily. Last but not least, logic rules enable an expressive aspect combination technique.

How it works

In our *LMP* approach, an aspect language is embedded in a logic programming language. This provides a general framework for declaring and implementing aspects. Aspects are declared as logic facts, which annotate the component program with additional declarations or code fragments. The facts trigger logic rules, which reason about the aspects, draw conclusions and finally generate woven output code. For example, consider the following aspect-declaration facts.

```
advicBefore(methodJoinPoint(Stack, pop), { Logger.Log("Entry Stack pop") ; }, { }).  
advicAfter(methodJoinPoint(Stack, pop), { Logger.Log("Exit Stack pop") ; }, { }).
```

These facts inform the weaver that some code fragments must be executed before and after the method `pop` in the class `Stack`. In our LMP framework program text is delimited by `@` and `@` and is treated as a special kind of logic term.

The aspect-declaration facts are loaded into the logic system, together with a set of logic rules constituting a weaver implementation. The weaver understands these facts and knows how to produce correctly woven code.

LMP Supports Crosscutting

A logic meta program consists out of a number of facts and rules which together describe the desired output program. This code generation metaphor allows dealing with crosscutting modularity structure in a natural way. Consider the following analogy that illustrates the intuition behind this idea: in one sentence I could state a fact about the prime minister of Belgium and in the next, unconstrained by geography, talk about the president of the United States. In a similar way we can group facts and rules into logic modules without being constrained by the modularity structure of the program they describe. A visualization of this idea is shown in figure 1.

* Department of Computer Science, The University of British Columbia, 309-2366 Main Mall Vancouver, B.C. V6T 1Z4, Canada.
† Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

Domain-Specific Aspect Languages

The earliest aspect languages were domain specific. AML [2] is an AOP extension to Matlab for sparse matrix computation, and RG [3], a specific language for creating image-processing systems. Unfortunately, while very powerful within their specific scope, such domain-specific aspect languages cannot be applied outside of the specific domains they were designed for. More recently, AspectJ [4] has taken a definite turn towards being a more general aspect language. But this generality comes at a price: a general-purpose aspect language is typically more low-level than domain-specific aspect languages [5]. This leads to more complicated programs which are more involved with implementation-specific issues.

The LMP approach to AOP is a framework, that allows users to define, implement and use their own domain-specific aspect languages [1]. The implementation of domain-specific aspect languages (DSLs) is eased because weavers for DSLs do not have to be implemented from scratch. Domain specific notations can be implemented more easily in terms of more general notations using logic rules that deduce general aspect declarations from more domain-specific ones.

We have experimented with this approach by implementing a family of simple aspect languages for implementing synchronization code. Figure 2 illustrates this, showing three equivalent aspect programs in declaring a synchronization implementation for a stack. It is interesting to note that the aspect program in each subsequent notation becomes more concise. Also, each subsequent program is less involved with *how* synchronization is implemented. The first notation, though already synchronization specific, is still low-level and requires the user to implement synchronization by means of flag variables and Boolean method guards. The next program directly expresses which patterns of mutual exclusion are intended by the complicated mesh of flags and guards. The next program is even more high level and expresses the reason why mutual exclusion between methods is necessary, allowing the mutex patterns to be deduced automatically.

Figure 1: Logic meta modules naturally crosscut base modularity.



A particular kind of domain specificity occurs when dealing with interactions among multiple aspects. For example, combining a logging aspect with a synchronization aspect poses some complications not observed when both are considered in isolation. For one, we need to consider how to log methods that may block. Do we log entry to a method before or after checking the guards? Or perhaps we may even want to log when a method actually blocks.

Combination of Aspects

These two simple rules in fact constitute the entire implementation of notation 3 in terms of notation 2!

```
mutex(?c, ?m1, ?m2) if modifies(?c, ?m1, ?state) and inspects(?c, ?m2, ?state) .
mutex(?c, ?m1, ?m2) if modifies(?c, ?m1, ?state) and modifies(?c, ?m2, ?state) .
```

explicit in two simple rules: modifies the state of the stack but peek and isEmpty only read the state. We can make this reasoning peek as mutually exclusive, but not peek and isEmpty? The reason is that we know that push directly the reasons why methods are declared mutex. For example, why is it that we declare push and expressing the underlying logic of the mutex declarations. Instead of expressing mutex pairs, we express As a second example, consider the rules that implement the modifies/inspects notation by directly implementation in the more low-level guard notation.

To fully implement the mutex notation, we have a few more rules that are equally simple. Note that the mutex notation is much more clear about its intention: to engender a specific pattern of mutual exclusions between methods. The rules declare in an intuitive way how the intention is realized in terms of an

```
onEntry(?class, ?method, { disabled<?other>+; })
if mutex(?class, ?method, ?other)
or mutex(?class, ?other, ?method) .
```

method in order to disable it while any one of its mutex buddies is active. From the mutex declarations this rule deduces onEntry declarations that increment the disabled flag of a As a first example, consider the following rule that is part of the implementation of the mutex notation.

We show a few example rules to illustrate the idea and style. The implementations of these languages are expressed in terms of one another. Each notation is implemented by a set of rules expressing the logic deductions that translate the higher-level notation into implemented by rules deducing equivalent before/after advice in the most general aspect language. The implementation rules are intuitive, which facilitates the process of building new domain-specific notations.

Figure 2: Three equivalent synchronization aspects for a stack

<p>Notation 1: using boolean 'required' guards.</p> <pre> ... onExit(Stack, peek, {disabledpop--;}) . onEntry(Stack, peek, {disabledpop--;}) . required(Stack, pop, {disabledpop==0}) . introduceVar(Stack, int, disabledpop, {0}) . introduces(Stack, push, {disabledpush==0}) . onEntry(Stack, peek, {disabledpush++;}) </pre>	<p>Notation 2: 'mutex' pairs.</p> <pre> ... onExit(Stack, pop, {disabledpeek--;}) . onEntry(Stack, pop, {disabledpeek++;}) . required(Stack, peek, {disabledpeek==0}) . introduceVar(Stack, int, disabledpeek, {0}) . introduces(Stack, peek, push) . mutex(Stack, pop, pop) . mutex(Stack, pop, push) . mutex(Stack, push, push) . </pre>	<p>Notation 3: modify/inspect.</p> <pre> ... inspects(Stack, peek, this) . modifies(Stack, pop, this) . modifies(Stack, push, this) . </pre>
---	---	---

The order of execution 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. For example, the following rule prioritizes the before advice:

```
combined.adviseBefore(methodJoinpoint(?class,?method),{?syncode ?logcode })
if logaspect.adviseBefore(methodJoinpoint(?class,?method),?logcode)
and syncaspect.adviseBefore(methodJoinpoint(?class,?method),?syncode) .
```

Logging when a method blocks, however, is conceptually more difficult and 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 domain-specific terms. Here, our approach really pays off [6]. By making use of the high-level declarations, logic rules can be defined that provide a specialized declaration for the combination. For example:

```
syncaspect.onBlock(?class,?method,{Logger.Log(?class ?method blocks@); })
if loggingaspect.loggedMethod(?class,?method)
and syncaspect.synchronizeMethod(?class,?method) .
```

Conclusion

Aspect-oriented programming and domain-specific languages are concepts that go very well together. Domain specificity in an aspect-language is highly desirable because it results in more concise and more readable aspect declarations. While this kind of statement can be made in general, about any kind of programming language, we believe domain-specificity to be of particular interest for aspect languages. A testament to this is the fact that the first aspect languages were domain specific and *not* general purpose aspect languages. We believe that domain specificity is both more natural and more important to AOP than it is simply to programming languages in general.

The true power of an LMP framework like ours then, is that the logic paradigm has an innate capability for expressing crosscutting and is at the same time an intuitive and expressive meta-programming paradigm for defining DSAs. Thereby defining and implementing DSAs can be brought within reach of the non-expert, end-user programmer.

[1] Kris De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 250-272. Springer Verlag, 1999.

[2] J. Irwin, J.-M. Longinier, J. R. Gilbert, and G. Kiczales. Aspect-oriented programming of sparse matrix code. *Lecture Notes in Computer Science*, 1343:249-256, 1997.

[3] A. Mendhekar, G. Kiczales, and J. Lampping. RG: A case-study for aspect-oriented programming. *Technical Report SPL97-009P9710044*, Xerox PARC, Feb. 1997.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of ECOOP 2001* (to appear), 2001.

[5] Cristina Vidreira Lopes and Gregor Kiczales. Recent developments in aspectj. In Serge Demeyer and Jan Bosch, editors, *ECOOP 98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 398-401. Springer Verlag, 1998.

[6] Johan Brichau, Declarative Composable Aspects, Position paper at *workshop on Advanced Separation of Concerns*, OOPSLA 2000.