

# Enhancing model-based AOP with behavior representation<sup>1</sup>

Alan Cyment, Nicolas Kicillof and Fernando Asteasuain  
University of Buenos Aires

[acyment, nicok, fasteasuain]@dc.uba.ar

## 1. INTRODUCTION

Recent research has shown that the implementation of the obliviousness principle introduced by Filman in [1] brings about serious coupling between aspects and base code [2][3]. Several authors have proposed to define an intermediate and more conceptual layer that should stand between aspects and base code. This layer, to which base code does not need to be completely oblivious [4], should allow aspects to remain unaware of implementation-level changes in base code [5]. We call this approach **model-based aspects**.

Despite the fact that arguments for the existence of the intermediate layer are quite solid, many questions remain open: What is the most useful way to represent a given domain? What kind of expressiveness is required from the chosen knowledge-representation language? Are domain-specific languages needed or is a general-purpose one expressive enough? What mechanism should be used for relating base code to the conceptual model? Will the use of model-based aspects add a significant performance overhead? How will the conceptual model cope with structural changes in base code?

Our contribution towards answering these and other open questions is to present our practical experience in the subject. Almost two years ago, we developed the first version of SetPoint [6], an AOP tool for .NET that mainly focused on making aspects rely on domain models. We used OWL [7] as the knowledge-representation mechanism.

Analyzing this first experience, we summarized pros and cons of the OWL approach and concluded that we were missing the ability to do **complex mapping** between base code and aspects, and the possibility of **maintaining state** in the conceptual model that could **evolve in response to system events**. As a result, we decided to use object-orientation as an alternative knowledge-representation mechanism, in order to tackle these issues.

## 2. REPRESENTING DOMAINS WITH OWL

We decided on the use of OWL without a deep analysis on its expressiveness, but rather based on practical facts. We found a nice analogy between our problem and the one the Semantic Web aims at tackling: *lack of explicit semantics in the WWW*. Moreover, we wanted to use a widely adopted and supported (both by a community and by the availability of tools) mechanism for knowledge representation, at least in our first attempt at building a conceptual model for a model-based AOP tool.

Conceptually, OWL is a faithful instantiation of Description Logics [8]. Our proposal was to represent domain knowledge using *ontologies* (i.e. abstract metamodels of certain domains) and *instances* of those ontologies (i.e. individuals that correspond

to viewing the application from one of the declared ontologies' perspective). Following the work in [11], we decided that *source code* was going to be one of the ontologies to be used. This would allow us to base pointcuts both on the conceptual model (i.e. "the new way") and the low level structure of source code (i.e. "the AspectJ way"). Moreover, this approach would render a more uniform knowledge representation metamodel: we chose to map base code to domain knowledge using program annotations; so having source code elements as first class citizens in the knowledge base would let us refer to annotations as mere relationships between concepts.

We used Sesame [10] as the engine for storing all domain related data using RDF triples. RDF is a basic knowledge representation language that defines resources (i.e. *things* that have an identity) and triples (i.e. relationships between two resources). OWL is nothing more than a vocabulary for RDF: it gives precise semantics to a given set of resources. The triples stored in Sesame would then represent the *join point universe* (i.e. every possible join point for the running application). Sesame offers a very powerful query language called SPARQL. Queries defined in SPARQL would become our *pointcuts*.

This experience proved quite fruitful [13]. But we mainly found two problems in the approach. Sometimes the *mapping between two concepts* (e.g. between a source code class and an architectural component) was not trivial. There was no way we could represent this behavior using OWL. A closely related problem was that in some cases we needed the conceptual model to store some kind of *state* to be updated in response to system events (e.g. when representing architectures using Petri nets, we wanted to query the net about its current state). In contrast to the *static* nature of OWL, we needed to represent the system *dynamics from a conceptual viewpoint*.

## 3. MODELING BEHAVIOR

As described in works such as [14], object-orientation is less expressive than Description Logics when considering static structure. But, as mentioned in the previous section, Description Logics provides no way of defining the dynamics of classes and individuals, whereas objects have the associated notion of **behavior**. We consequently designed a domain representation metamodel that mimicked our previous *ontologies, instances and annotations* approach, making use of object-orientation's behavioral capabilities and trying to make up for the lack of expressiveness of this new mechanism.

The model consists now of what we call *concepts*. To represent them, we use C# *interfaces*. A given set of concepts can be seen as an *ontology*. All transformations from base code to the model are made by *mappers*, realized by *objects* that implement the

<sup>1</sup> This work was partially funded by Microsoft Research's Phoenix—Excellence in Programming RFP Awards and ANCYT PICT 11738

above-mentioned interfaces. They make use of behavior to produce complex mappings. Lastly, *annotations* relate base code elements (such as methods or classes) to concepts and are implemented using .NET *attributes*. There are basically two kinds of concepts: *actions* and *entities*. Entities have *properties*; actions, in turn, have *roles*, played by entities.

Actions represent system-wide events, as seen from a conceptual level. They are basically *abstract join points* to which aspect developers can refer in a more conceptual way than syntactic join points, decoupled from low-level implementation details. They can also be used to update the state of conceptual objects, but this capability has not been developed yet.

Interfaces representing actions and entities merely define, using .NET properties, the names of the roles and properties they will have, respectively. Mapper classes are then declared so that they implement these interfaces. *Action mappers* specify which entity performs each role in terms of the *low level join points* being mapped. Low-level join points basically describe runtime events, such as method or constructor calls in a pure object-oriented way: they contain the message sender, receiver, arguments and selector. *Entity mappers*, in turn, wrap objects or groups of objects. They therefore have access to all public members, in order to define an entity's roles.

As mentioned before, annotations are implemented using .NET attributes. Attributes<sup>2</sup> are actually special classes, which means developers can write their own attribute hierarchies, with a different behavior for each of their members. In this case, we have chosen to differentiate entity and action annotations, as we did for mappers. They can only be attached to classes and methods, respectively.

This approach has tackled the complex mapping question through the use of mappers. We are still lacking a way to maintain updatable state in the model, but actions can be considered as a first step in that direction.

## 4. CONCLUSIONS AND OPEN QUESTIONS

We have shown two domain-modeling formalisms for decoupling aspects from base code. Practical experience, rather than formal expressiveness analysis, was our basis when choosing the knowledge representation formalism to use. Both approaches employ general purpose knowledge representation languages. But it is worth mentioning that several domain-specific issues were raised during the experiments. We tackled them using ontologies, which in turn were built using a general purpose knowledge-representation language. Program annotations were used as a means to relate concepts to source code. Other implementations, such as [5], use a more intensional approach based on predicates. Formal benchmarking has been postponed, mainly because we follow the "model first – optimize later" approach (and we are still looking for the best model). In contrast to [5], no special mechanism has been put forward in our implementation to cope

with source code evolution. This issue is critical and we will work on it in the near future.

The OO model we presented allows us to define interesting *mappings* between model concepts and base code elements, in a way similar to CAESAR [16]. We are working on improving it so that it can accommodate for concepts that have an extended lifetime (i.e. beyond the instant of the mapping). This would let us tackle the *dynamics from a conceptual viewpoint* problem.

A potential benefit of using OO mechanisms as a modeling formalism is that their syntax and concepts are more familiar to the common developer than those of Description Logics.

Regardless of the knowledge representation formalism used for domain modeling, we would be invariably facing the intractability problem of second order logics presented in [15]. A workaround to this issue needs to be devised by the community, perhaps using the *spanning object* concept also presented in that paper.

## 5. REFERENCES

- [1] R. Filman and D. Friedman. *Aspect-oriented programming is quantification and obliviousness*. Proc. Advanced Separation of Concerns, OOPSLA 2000.
- [2] W.G. Griswold et al. *Modular Software Design with Crosscutting Interfaces*. IEEE Software, vol. 23, no. 1, pp. 51-60, Jan/Feb, 2006.
- [3] C.Clifton and G. T. Leavens. *Obliviousness, modular reasoning, and the behavioral sub typing analogy*. Technical Report TR03-01a, Iowa State University, 2003.
- [4] J. Aldrich. *Open Modules: Modular Reasoning about Advice*. Proceedings of the European Conference on Object-Oriented Programming, July 2005.
- [5] A. Kellens et al. *Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts*. To be published in ECOOP 2006.
- [6] R. Altman and A. Cyment. *SetPoint: a semantic approach for the pointcut resolution in AOP*. Msc. Thesis, Universidad de Buenos Aires, 2004.
- [7] <http://www.w3.org/2004/OWL>
- [8] F. Baader et al: *The description logic handbook: theory, implementation, and applications*. Cambridge University Press (2003)
- [9] <http://www.w3.org/2001/sw>
- [10] <http://www.openrdf.org>
- [11] C. Welty. *An Integrated Representation for Software Development and Discovery*. Ph.D. Thesis, Rensselaer Polytechnic Institute 1996.
- [12] <http://www.w3.org/RDF>
- [13] R. Altman, A. Cyment and N. Kicillof. *On the need for SetPoints*. EIWAS 2005.
- [14] D. Calvanese et al. *Unifying class-based representation formalisms*. Journal of Artificial Intelligence Research, 11:199--240, 1999.
- [15] Welty, Chris and Ferrucci, Dave. 1994. *What's in an Instance?* RPI Computer Science Technical Report.
- [16] M. Mezini and K. Ostermann. *Conquering Aspects with Caesar*. AOSD 2003.

<sup>2</sup> Properties are a native construct of .NET similar to fields, but getter and setter methods have to be defined in order to access the value. They behave analogously to their EJB counterpart.