Using Aspect Oriented Modeling to localize implementation of executable models

Zaid Altahat, Tzilla Elrad, Didier Vojtisek

Using Aspect Oriented Modeling to localize implementation of executable models

Zaid Altahat
GE Healthcare
Illinois Institute of Technology
Zaid.Altahat@ge.com

Tzilla Elrad
Illinois Institute of Technology
Elrad@iit.edu

Didier Vojtisek
INRIA
Didier.Vojtisek@inria.fr

Abstract

Executable models are essential to define the behavior of models, such as constraints put on model elements. However their implementation crosscut multiple model elements. Model semantics will facilitate Model Driven Development, without it, Design and Implementation won't necessarily represent different abstractions of the same system. This paper introduces a mechanism to query executable models and weave constraints in order to localize their implementation, which improves code redundancy and modularity.

Keywords

Executable Models, Aspect-Oriented Modeling (AOM), Model Driven Development (MDD), Model Driven Architecture (MDA), Aspect-Oriented Software Development (AOSD).

1. Introduction

Models have been limited in use to design and documentation. Designs are lost by interpretation when moved from system architects (Design) to software engineers (Implementation). The design doesn't dictate the models semantics. Semantics are "the underlying meaning of exchanged models, that is, the *constraints* that models place on the runtime behavior of the specified system." [7]. Design By Contract (DBC) is an example of *constraints* on the model behavior; however their implementation is not localized and crosscut [3] multiple model elements.

A programming language consists of syntax and semantics. Syntax is the language constructs, such as UML class diagrams; while, semantics give the syntactic constructs their meaning. Leaving out the semantics of models created a gap that lead to a wide range of interpretations of the same model. The gap also created a chain of tools that can only exchange the syntax of models. Executable models came to fill in this gap.

Executable models *constrain* how models *behave* at run time. Code generated from models should have a *unique* execution behavior. Unique in the sense that if different codes, programming languages such as Java or C++, to be generated, all should have the same execution behavior. UML is in the process of fully defining Executable UML [7]. KerMeta [5] on the other hand has already defined full behavioral language to specify semantics of models. Section 3 briefly presents KerMeta.

One way to constrain the behavior of a model element, a Class for example is to define *Invariant* condition on the class, and *pre* and *post* conditions on its operations. These three are what is referred to as DBC, which KerMeta already provides capabilities of; however it achieves that *individually* for each class and operation. To manually define constraints for each class and operation may lead to code redundancy and reduction in modularity. The added constraints

crosscut [3] multiple classes and operations. Aspect-Oriented Modeling (AOM) can help in obviating this problem.

AOM provides separation of crosscutting concerns at the models level. Most popular among these models are behavioral models, which are used in software development, not just for design and documentation but for code generation as well. To set foundations for the code generation and model transformation, new standards are being defined as part of Model Driven Architecture (MDA) group. MDA standards are being set in parallel to AOM. MDA *transforms* a model from high abstract level to platform specific level then to code. AOM also help in keeping crosscutting models separate, as well as transforming Platform Independent Models (PIM) to Platform Specific Models (PSM) by weaving in platform dependent model implementation.

Using AOM approach we will demonstrate how to *localize* the implementation of a crosscutting behavior that intersect multiple classes and/or operations. In AOM a pointcut model, and an *advice* model are defined. Both models and the original models are fed into a weaver. The weaver adds the advice, added behavior, to the join points matched by the pointcut in the model. This paper introduces a novel approach for the modularization and weaving of executable models.

The main contribution of this paper is to provide a model driven approach to query and weave executable model elements into models. The problem this approach tackles is to localize the DBC constraints for executable models; moreover, localize the implementation of operations. Which reduces code redundancy and increases modularity. The project was done in KerMeta for both querying and weaving executable model elements, it is a pure model driven approach that operates on executable models.

Paper is organized as follows: Section 2 presents related work and section 3 briefly describes KerMeta. Section 4 is the core of this paper; it presents the details of the metamodels used, as well as the weaving process. Besides, Section 5 demonstrates the querying and weaving process on an example model. Original model and modified model are presented in Appendices A and B, respectively.

2. Related Work

There are other attempts to localize DBC constraints. However they were designed with a specific programming language in mind. A C++ approach [10] presented a mechanism to localize DBC implementation using Constraint-Specification Aspect Weaver (C-SAW) [9]. ECL [4] was used to locate operations, in addition to weave *assertions* at the beginning and end of an operation to represent *pre* and *post* conditions, respectively. No support for *Invariant* condition. Another approach, Contract4J [2], uses AspectJ to support DBC in Java. It uses Java 5 annotations to mark elements to be amended and define the *pre*, *post*, and *Invariant* conditions. It uses AspectJ behind the scenes to weave in the added code. In contrast with my approach, both of these approaches are geared more towards a specific programming language and are not based on executable models.

3. KerMeta

Meta-languages such as MOF1.4 [5] ,MOF2.0 [6] ,and Ecore[1] are used to specify the structural and syntax parts of a model but not its behavior. For example EMOF specifies an operations signature and stops there, without defining its behavior. A mix of pseudo code and natural language is used to define its behavior. KetMeta on the other hand uses an operational semantic

to specify the precise behavior of models. The example [11] presented in Listings 1 and 2 show how the definitions of the same method in both MOF and KerMeta.

Operation ***isInstance(element : Element) : Boolean***
"*Returns true if the element is an instance of this type or a subclass of this type. Returns false if the element is null*".

Listing 1. MOF definition of method isInstance()

```
operation isInstance(element : Element) : Boolean is do
        // false if the element is null
        if element == void then result := false
        else
                // true if the element is an instance of this type
                // or a subclass of this type
                result := element.getMetaClass == self or
                element.getMetaClass.allSuperClasses.contains(self)
        end
end
```

Listing 2. KerMeta definition of method isInstance()

KerMeta proposes a rich model oriented environment for metamodeling. It provides support for many use cases, including:

- Implementation of operations directly in metamodels,
- Execution of simulation of metamodel behavior,
- Transformation and weaving of models,
- Verification and validation of models against metamodels (as given by a set of static and dynamic constraints),
- Building new Domain Specific Languages under the shape of metamodels,
- Building any model-driven tools, including tools that generate tools (generative programming).

This work is a demonstration of several of them. The most important is that it reflectively applies MDA to itself[12]. In this paper, KerMeta is applied at two levels. First, the language is used to define the transformation that constitutes the metamodel weaver. Second, the weaving is applied to KerMeta itself by introducing the advices in models written in KerMeta.

4. Metamodel Weaver

The weaver consists of several metamodels, a pointcut metamodel, an advice (added behavior) metamodel, a link metamodel, and the weaver itself, presented in Figure 1. The following sections present each of these metamodels in details.

Weaver::weave, shown in Figure 1, is the starting operation, it's passed a collection of *Link* and a collection of *Model*. *Link* defines a relation between a pointcut *MatchPattern* and an *Advice*. For each join point matched by a *pointCut behavior* is added.

Advice has multiple operations *getInvariant*, *getPreCond*, and *getPostCond* to retrieve *Invariant*, *pre* and *post* condition, respectively. An instance advice inherits from *Advice* and overwrites *operationAbs* in order to define *pre* and *post* conditions. Instance advice can also hold other operations definitions that needs be added to the model. They are retrieved using the operation *getOps*. This is to provide an operation implementation into a class. Section 5 presents an example with added behavior.

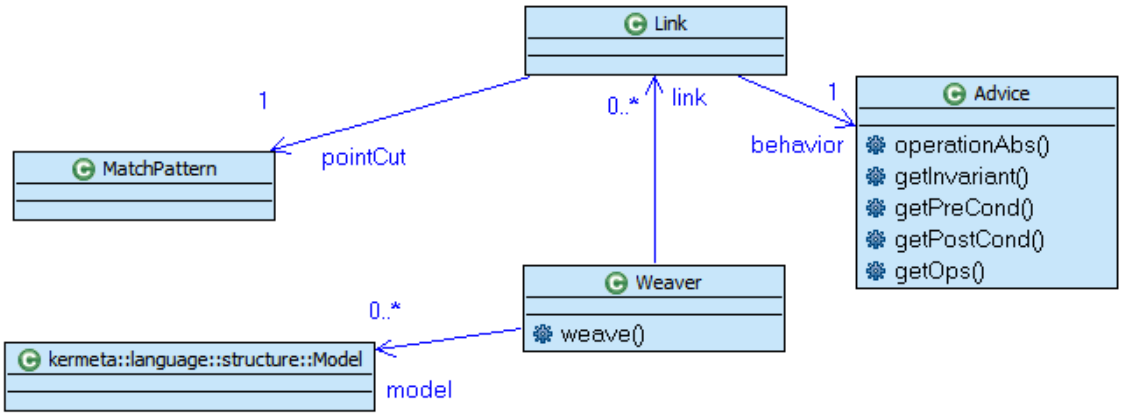

Figure 1 Weaver Metamodel

Figure 2 shows the pointcut metamodel, *MatchPattern*. All elements, except *MatchPattern*, inherit from *MatchPattern* and with it they inherit the string *namePattern* to define its name signature. The matching signature consists of a *ClassPattern* class that has a collection of *AttributePattern* and a collection of *OperationPattern*, which in turns has a collection of *ParamPattern*. All elements inside *ClassPattern* are optional, that's a pointcut in its simplest format is a class name pattern, for example **Account* that will match all classes that end with *Account*. Match patterns used here are similar to AspectJ name matching. Section 5 presents an example with pointcut.

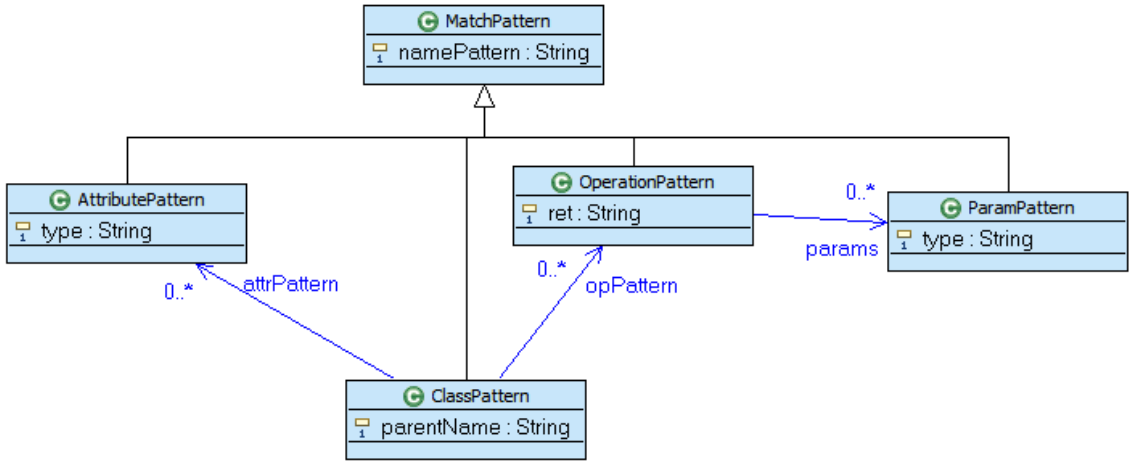

Figure 2 Pointcut metamodel

5. Example

Next we'll introduce an example where blocks of executable models were weaved into model elements, classes and operations. Figure 3 introduces a basic Bank system with different type of accounts. One thing to note about the class *Account* is that the class itself doesn't have *Invariant* condition and none of its operation has a *pre* or a *post* condition, which will be added using the

weaver. Also the operation *applyInterest* is abstract where its implementation will be weaved in for two of *Account* subclasses only. KerMeta representation of the model is presented in Appendix A.

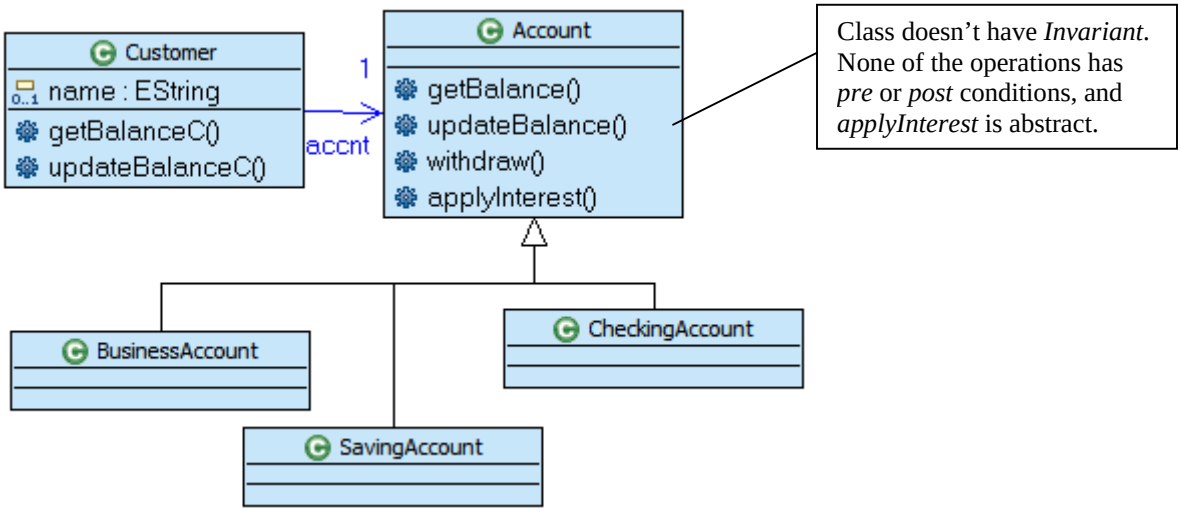

Figure 3 Bank metamodel

The model in Figure 3 represents the element *model* in Figure 1. The *Weaver* needs *link* element(s) to define what behavior to add for a matched pointcut. Figure 4 introduces two of these *Link* elements. In Figure 4-1 a *Link* is created with a *pointCut* that matches the operations *updateBalance* and *withdraw*. *Advice1* defines the behavior to be added, it introduces the *Invariant* condition to the matched class, and the *post* condition to the matched operations.

Figure 4-2 defines another Link with a *pointCut* that matches classes *CheckingAccount* and *BusinesAccount* that inherit from the class *Account*. The behavior to be added is an implementation for the operation *applyInterest*. More elements could be used to define more model queries, like number and type of parameters to an operation and its return type. More involved queries were run on larger models, but for sake of simplicity I introduced these queries on the Bank system.

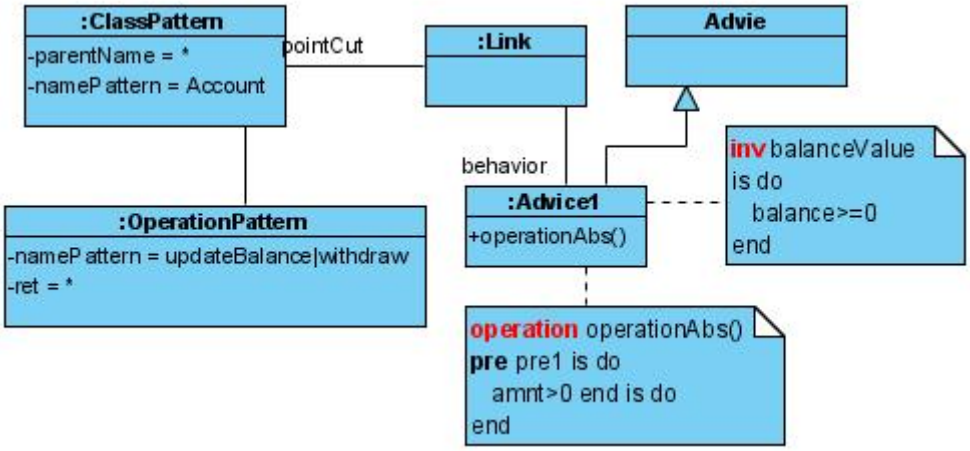

Figure 4-1 an instance of Link with a pointcut that matches operations *updateBalance* and *withdraw* in class *Account*, and *pre* and *post* conditions.

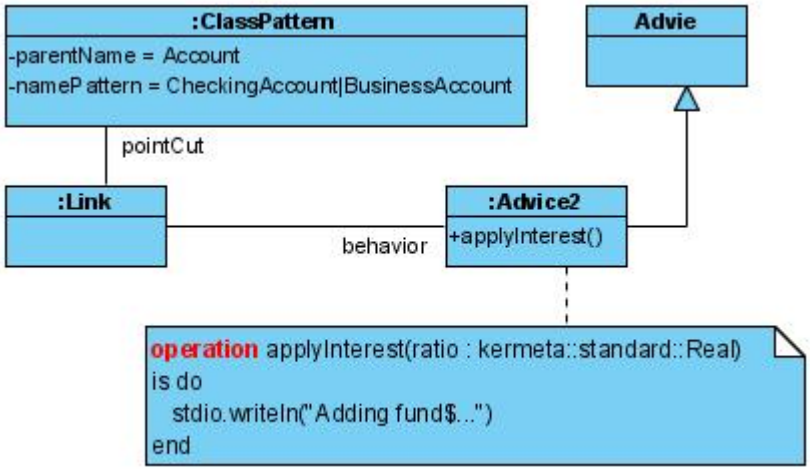

Figure 4-2 an instance of Link with a pointcut that matches classes *CheckingAccount* and *BusinessAccount* whose parent are *Account,* and implementation for operation *applyInterest.*

The weaver iterates on each element in the Bank model and applies each link on it. It iterates on the elements twice, once for each link. In the first pass it adds the *Invariant* condition to the class *Account* and the *post* condition to the operations *updateBalance* and *withdraw*. In the second pass it adds the operation *applyInterest* to the classes *CheckingAccount* and *BusinessAccount*. Appendix B presents the generated modified model, and Appendix A present the original model before any modifications.

6. Conclusion and future work

Executable models are getting high attention in order to add semantics to PIM models. In this paper we presented a novel way to query and weave executable models. We chose to localize the implementation of DBC constructs and shared operations implementations in order to improve code redundancy and modularity.

In pointcut metamodel we used strings to define many of the match pattern elements, as shown in Figure 2. In the future we'd like to change the parameter type and operation return type to *kermeta::language::structure::Type*. This will enable us to check for types and super-types, such as *Integer* and *Collection*, without having to use strings. It will also enable us to check for validity of operation arguments. However, using the element *Type* will complicate writing queries and the actual querying process.

7. References

1. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R. and Grose, T. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.
2. Dean Wampler, "AOP@Work: Component design with Contract4J". http://www-128.ibm.com/developerworks/java/library/j-aopwork17.html
3. Diotalevi, F., "Contract Enforcement with AOP," *IBM DeveloperWorks*, July 2004, http://www-106.ibm.com/developerworks/library/j-ceaop/
4. Gray, J., Sztipanovits, J., Schmidt, D., Bapty, T., Neema, S., and Gokhale, A., "Two-level Aspect Weaving to Support Evolution of Model-Driven Synthesis," in *Aspect-*

Oriented Software Development, (Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhán Clarke, eds.), Addison-Wesley, 2004.
5. OMG. Meta Object Facility (MOF) Specification 1.4, Object Management Group,http://www.omg.org/cgi-bin/doc?formal/2002-04-03, 2002.
6. OMG. MOF 2.0 Core Final Adopted Specification, Object Management Group,http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2004.
7. OMG, “Semantics of a Foundational Subset for Executable UML Models” 2006.
8. http://www.kermeta.org/
9. http://www.gray-area.org/Research/C-SAW/
10. Jing Zhang, Jeff Gray and Yuehua Lin. "A Model-Driven Approach to Enforce Crosscutting Assertion Checking".
11. Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer and Jean-Marc Jézéquel. “On Executable Meta-Languages applied to Model Transformations”
12. Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. -- Reflective model driven engineering. -- In G. Booch P. Stevens, J. Whittle, editor, Proceedings of UML 2003, volume 2863 of LNCS, pages 175--189, San Francisco, October 2003. Springer.

Appendix A Bank.kmt

```
/* Class Customer was not modified, it was left out from this
appendix.*/

class Account
{
	attribute balance : kermeta::standard::Integer

	operation updateBalance(amnt : kermeta::standard::Integer)
	is do
		balance := amnt
	end

	operation getBalance() : kermeta::standard::Integer
	is do
		result := balance
	end

	operation withdraw(amnt : kermeta::standard::Integer)
	is do
		balance := balance - amnt
	end

	operation applyInterest(ratio : kermeta::standard::Real) is
	abstract
}

class CheckingAccount inherits Account
{ }

class SavingAccount inherits Account
{ }

class BusinessAccount inherits Account
{ }
```

Appendix B Generated Bank.kmt

This Appendix shows only modified classes and operations. Unmodified classes were left out and are identical to the originals presented in Appendix A. Weaved code is in the red box.

Weaved models.

```
class Account
{
	inv balanceValue is
		do
			balance.isGreaterOrEqual(0)
		end
	attribute balance : kermeta::standard::Integer

	operation applyInterest(ratio : kermeta::standard::Real) :
		kermeta::standard::~Void is	abstract

	operation updateBalance(amnt : kermeta::standard::Integer) :
		kermeta::standard::~Void
		post post1 is do
		 result.isNotSameAs(void).~and(balance.isGreaterOrEqual(0))
		end
	is do
			balance := amnt
	end

	operation getBalance() : kermeta::standard::Integer is do
			result := balance
	end

	operation withdraw(amnt : kermeta::standard::Integer) :
		kermeta::standard::~Void
		post post1 is do
		 result.isNotSameAs(void).~and(balance.isGreaterOrEqual(0))
		end
	is do
		balance := balance.minus(amnt)
	end

}
class CheckingAccount inherits Account
{
	operation applyInterest(ratio : kermeta::standard::Real) :
		kermeta::standard::~Void is do
			stdio.writeln("Adding fund$...")
		end
}

class BusinessAccount inherits Account
{
	operation applyInterest(ratio : kermeta::standard::Real) :
		kermeta::standard::~Void is do
			stdio.writeln("Adding fund$...")
		end

}
```