A Process for Aspect-Oriented Platform-Specific Profile Checking

Thiago Gottardi
Computing Department – Federal University of São Carlos – UFSCar
Rodovia Washington Luís, Km 245, CEP 13.565-905
São Carlos, São Paulo, Brazil
thiago_gottardi@dc.ufscar.br

Rosângela Aparecida Delosso Penteado
Computing Department – Federal University of São Carlos – UFSCar
Rodovia Washington Luís, Km 245, CEP 13.565-905
São Carlos, São Paulo, Brazil
rosangela@dc.ufscar.br

Valter Vieira de Camargo
Computing Department – Federal University of São Carlos – UFSCar
Rodovia Washington Luís, Km 245, CEP 13.565-905
São Carlos, São Paulo, Brazil
valter@dc.ufscar.br

ABSTRACT

Several modeling profiles for aspect-oriented software have been proposed in the literature; however, many of them lack important concepts or have deficiencies when used for code generation. These problems indicate a disparity between the required basic concepts of the paradigm and the concepts provided by the notation. In this paper we propose a process to evaluate UML profiles in order to allow the detection of inconsistencies between what is provided by the profile and what is required by the paradigm. As a result, we found several inconsistencies after applying our process to evaluate a real aspect-oriented profile; these detections are beneficial to attain correct profiles allowing complete use of concepts and correct code generation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*; D.3.2 [**Programming Languages**]: Language Classifications—*Design languages, UML, AspectJ*; D.2.10 [**Design**]: Representation

General Terms

Algorithm,Design,Verification

Keywords

UML Model, UML Profile, Profile Checking, Aspect-Oriented

1. INTRODUCTION

Aspect-Oriented Programming (AOP) was created to allow better software modularization, however, aspect-oriented software development needs further efforts to allow concern separation since earlier phases. Among efforts to solve this problem, researchers proposed UML (Unified Modeling Language [10]) extensions, providing new modeling notations to represent aspect-oriented software [1] [4] [2] [5] [7] [3] [8].

A software model is an alternative representation to software code, it may be equivalent to code or with higher levels of abstraction. Also, a model can be linked to a graphical view, referred as diagram. Models can be formally defined with models, which are also known as metamodels. UML formal definition uses metamodels and metametamodels [10].

UML extensions can be made by either modifying or appending the metamodel, respectively Heavy Weight or Light Weight [8]. Heavy Weight extensions are UML metamodel modifications while Light Weight are UML profiles, a set of stereotypes, tags and constraints. Stereotypes are definitions allowing to extend elements in model level, tags are stereotype properties (attributes) and constraints are logical expressions evaluated to assure better model semantics. The standard language to define constraints for UML is the OCL (Object Constraint Language) [10].

It is important that the profile is correctly defined to avoid incorrect definitions, impracticable software projects or restricting project from using valid concepts. When dealing with MDD (Model Driven Development)[11][12], where software code can be generated directly from models, incorrect notations may lead to erroneously generated code that cannot be compiled.

In this paper, a process to generate evaluation guidelines for UML profiles is proposed, allowing to detect profile inconsistencies. Inside a case study, we detected several inconsistencies after applying our process to evaluate a real aspect-oriented profile. The detections are also helpful for corrections, which should be done in order to attain a better semantic level, improving concept use and code generation.

This paper is structured as follows: In Section 2 is shown an earlier proposed profile; in Section 3, our validation process is presented; in Section 4, there are found inconsistencies and in Section 5, there are the conclusions.

2. ASPECTJ PROFILE

AspectJ Profile is a UML Profile which allows detailed design models for aspect-oriented systems. It was proposed by Evermann specifically for AspectJ [3].

According to Evermann, his approach is light weight and addresses all aspect-oriented programming concepts and does not include textually declarations in diagrams as is done by other approaches[8]. Heavy weight approaches are defined by modifying UML metamodel [10] and its use demands specific tools [8]. As a light weight extension, the profile can be used in any available UML tool which supports profiling [3].

In Figure 1 there is a diagram representing the most important elements of AspectJ Profile. The boxes marked with ≪stereotype≫ are stereotype specifications and those marked with ≪enumeration≫ are enumerations. By using generalizations (transparent tipped arrow), a stereotype may inherit all properties of the generalized stereotypes and their ancestors. Inside each box, above its name, (between brackets) there is the name of the specialized metaclass. Enumerations are type definitions, when applied to stereotypes, they list possible advice types ("before","around" and "after"), possible pointcut types ("not", "or" and "and") and aspect instanciation types ("per this", "per target", etc.).

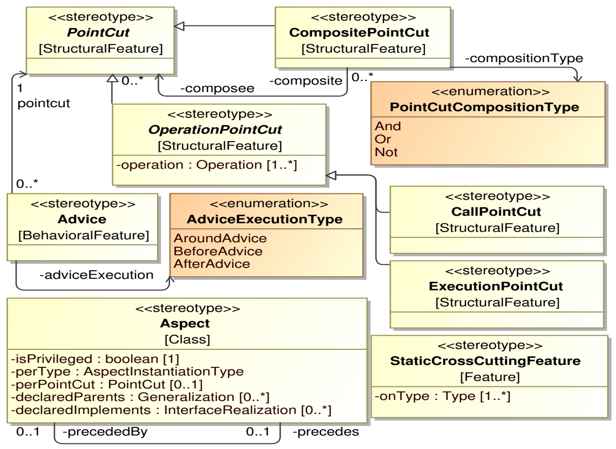


Figure 1: AspectJ Profile Subset [3].

The ≪Aspect≫ stereotype extends the *Class* metaclass, because of their similarities. In model level, an aspect has dynamic and static properties and may extend classes and implement interfaces. Since classes contain behavioral and structural features, ≪Aspect≫ does not need another tag for pointcuts and advices. The ≪Advice≫ stereotype extends *BehavioralFeature*, for being similar to operations. It is associated with its type and a pointcut. *Feature* metaclass is extended by ≪StaticCrosscuttingFeature≫ to allow intertype declarations. This is done because *Feature* is superclass of *BehavioralFeature* and *StructuralFeature*, allowing properties and operations being inserted by an aspect into another class. ≪PointCut≫ was created to represent pointcuts in model level. It is abstract, then only substereotypes may be applied in model level, for example, ≪CallPointCut≫ and ≪ExecutionPointCut≫.

A UML profile can be used in every UML tool which supports profiling mechanism, making it more accessible to use than heavy weight approaches. After loading the profile in a specific tool, stereotypes may only be applied to instances of the extended metaclass (or submetaclass), for instance, ≪Aspect≫ extends *Class*, then every UML class may receive an ≪Aspect≫ stereotype to represent an aspect. In order to represent an advice, an operation inside an aspect must be stereotyped ≪Advice≫, also, set *execution type* and a *pointcut* to define its join points. To represent a ≪PointCut≫, a property of an aspect must be stereotyped with a specific pointcut, for instance, stereotypes ≪CallPointCut≫ or ≪SetPointCut≫. PointCuts may be combined using ≪CompositePointCut≫ and setting a boolean composition operator.

However, it must be verified if the profile can correctly represent aspect-oriented software and can be used to generate AspectJ code.

3. PROPOSED PROCESS

In this section, it is proposed a process to check if a UML profile is representing required concepts correctly and if it is not allowing invalid models. The process is an algorithm whose inputs are required concepts, profile elements, extended metamodel and mapping between concepts and profile elements. The first and the last informations must be entered by the user, since they cannot be completely automatized. The process is explained along with a practical example in order to comply with size limitations.

The possible inconsistencies of a UML profile are divided into two categories, as shown on Table 1; these categories are referred as "Validation Categories" when detected by our process.

Table 1: Validation Categories

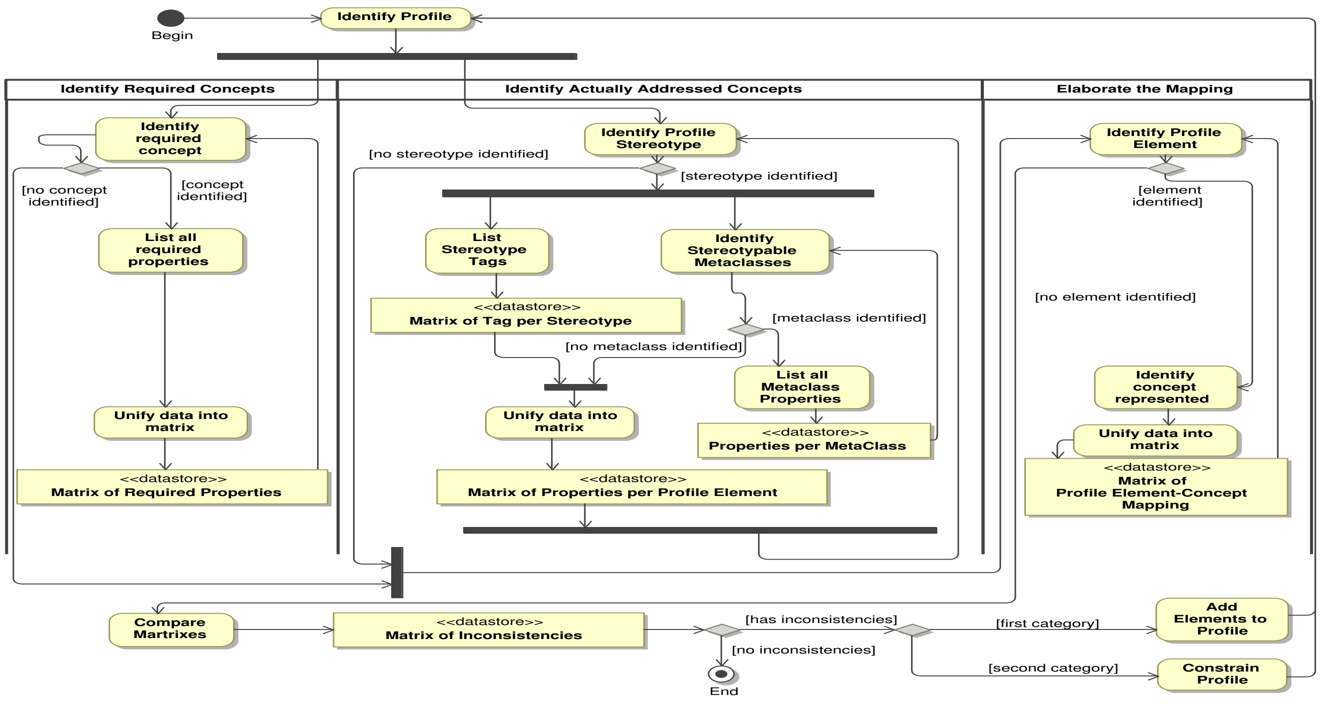
Category	Name	Description
1st	Mandatory	Usage must be allowed
2nd	Prohibited	Usage should not be allowed

In first category, which is named "Mandatory", are the problems caused when the profile does not allow to model valid concepts, consequently restricting models from representing valid usages, problems of this category inhibit software developers from creating software taking advantage from specific concepts.

The second category of inconsistencies is named "Prohibited". These inconsistencies are caused by incorrect defined stereotypes being excessively permissive, allowing models representing software which cannot be compiled.

In Figure 2 there is an activity diagram illustrating the proposed process. After identifying which profile to analyze, in this case AspectJ Profile, the process is divided into three activities: Identify Required Concepts, Identify Actually Addressed Concepts and Elaborate the Mapping.

Identify Required Concepts: The first activity is intended to identify which concepts are mandatory for being represented by models applying the profile. AspectJ profile requirements are found on AspectJ Language Specifications,



Begin
Identify Profile
Identify Required Concepts
Identify Actually Addressed Concepts
Elaborate the Mapping
Identify required concept
[no concept identified]
[concept identified]
List all required properties
Unify data into matrix
<<datastore>> Matrix of Required Properties
Identify Profile Stereotype
[no stereotype identified]
[stereotype identified]
List Stereotype Tags
Identify Stereotypable Metaclasses
<<datastore>> Matrix of Tag per Stereotype
[metaclass identified]
[no metaclass identified]
List all Metaclass Properties
Unify data into matrix
<<datastore>> Properties per MetaClass
<<datastore>> Matrix of Properties per Profile Element
Identify Profile Element
[element identified]
[no element identified]
Identify concept represented
Unify data into matrix
<<datastore>> Matrix of Profile Element-Concept Mapping
Compare Martrixes
<<datastore>> Matrix of Inconsistencies
[has inconsistencies]
[no inconsistencies]
End
[first category]
[second category]
Add Elements to Profile
Constrain Profile

Figure 2: Validation Process.

in this example, the AspectJ grammar defined for ABC (AspectBench Compiler) [9] is considered.

For example, the Identified Concept for this iteration is "Advice" and in Figure 3 it is defined the grammar for an *Advice* declaration. Identifiers between less ("⟨") and greater ("⟩") symbols are non terminals, between single quotes (" ' ") are terminals. Non terminals ending in "opt" are optional.

```
⟨advice_declaration⟩ ::=

        ⟨modifiers_opt⟩ ⟨advice_spec⟩ ⟨throws_opt⟩ ':'
        ⟨pointcut_expr⟩ ⟨method_body⟩

⟨advice_spec⟩ ::=

          'before' '(' ⟨formal_parameter_list_opt⟩ ')'
        | 'after' '(' ⟨formal_parameter_list_opt⟩ ')'
        | 'after' '(' ⟨formal_parameter_list_opt⟩ ')' 'returning'
        | 'after' '(' ⟨formal_parameter_list_opt⟩ ')' 'returning'
            '(' ')'
        | 'after' '(' ⟨formal_parameter_list_opt⟩ ')' 'returning'
            '(' ⟨formal_parameter⟩ ')'
        | 'after' '(' ⟨formal_parameter_list_opt⟩ ')' 'throwing'
        | 'after' '(' ⟨formal_parameter_list_opt⟩ ')' 'throwing'
            '(' ')'
        | 'after' '(' ⟨formal_parameter_list_opt⟩ ')' 'throwing'
            '(' ⟨formal_parameter⟩ ')'
        | ⟨type⟩ 'around' '(' ⟨formal_parameter_list_opt⟩ ')'
        | 'void' 'around' '(' ⟨formal_parameter_list_opt⟩ ')'
```

Note: The only valid modifier for an advice_declaration is strictfp.

Figure 3: Grammar for Advice Declaration [9].

By analyzing the grammar, it is possible to identify several concepts, for instance, advices may be defined with: optional *strictfp* modifier; exactly one *pointcut*; *execution type*; *formal parameters*. Every non fundamental typed property should be taken as another concept; this recursive process should then be executed until all concepts are analyzed. The *strictfp* modifier is used to ensure that float point calculations, within the method body scope, are made in conformance to IEEE 754 [6]. *Execution type* and *formal parameters* are concepts discovered at this iteration and should be analyzed in next iterations. *Execution type* is a redundant definition found in grammar, each type can be listed as an enumeration, which are later identified as *before*, *after*, *after throwing*, *after returning* and *around*; these must be evaluated in next iterations as well. After identifying properties of *Around* concept, it is identified that around advices must have one type.

The information identified is then unified into Matrix of Required Properties, containing owner concept with its properties and multiplicities. For instance, Table 2 is the Matrix of Required Properties for profile validation regarding *Advice* concept. The multiplicities contain lower bounds (minimum) and higher bounds (maximum). When the lower bound is 0, the property is optional, otherwise, obligatory if it is 1. Other possible multiplicity is "enumeration". Enumerations are listed values. A concept defined as an enumeration must be used with exactly one of listed values.

Identify Actually Addressed Concepts: The concepts actually addressed by profile are identified during the execution of this activity. Each stereotype of the profile should be identified and all tag definitions listed. Every non fundamental typed property should be analyzed recursively, including enumeration definitions. Every stereotype extends at least one metaclass. These metaclasses must be identified as well, since they contain properties which are also important for stereotyped element usage. For example, when analyzing ≪Advice≫, there are two tags: *pointcut* and *adviceExecution*. ≪Advice≫ extends *BehavioralFeature*, in model level, the stereotype can be applied to *Operation*. Every property from *Operation* must be also listed and appended to

Table 2: Matrix of Required Properties

Concept	Property	Multiplicity
Advice	strictfp	0..1
Advice	pointcut	1..1
Advice	execution type	1..1
Advice	throws option	0..1
Advice	method body	1..1
Advice	parameter (list)	0..*
execution type	before	enumeration
execution type	after	enumeration
execution type	after throwing	enumeration
execution type	after returning	enumeration
execution type	around	enumeration
around	type	1..1

≪Advice≫ addressed properties. The origin of each property is visible on table (between parenthesis). Both lists are then unified into Matrix of Profile Element-Concept Mapping. For instance, Table 3 shows some interesting properties addressed by the profile.

Table 3: Matrix of Properties per Profile Element

Profile Element	Property	Mult.
≪Advice≫ (stereotype)	pointcut	1..1
≪Advice≫ (stereotype)	adviceExecution	1..1
≪Advice≫ (metaclass)	Abstract	0..1
≪Advice≫ (metaclass)	Static	0..1
≪Advice≫ (metaclass)	Final	0..1
≪Advice≫ (metaclass)	Return type	0..1
≪Advice≫ (metaclass)	Parameters	0..*
AdviceExecutionType	Before	enumeration
AdviceExecutionType	After	enumeration
AdviceExecutionType	Around	enumeration

Elaborate the Mapping: The Matrix of Profile Element-Concept Mapping for AspectJ Profile validation links profile elements to concepts found as required. This phase is important for comparing both matrices. For instance, Table 4 shows some interesting relations.

Table 4: Matrix of Profile Element-Concept Mapping

Concept	Profile Element
Advice	≪Advice≫
execution type	AdviceExecutionType

Notice that the "Matrix of Required Properties" is a data artifact containing requirements; it is related to the first validation category, while the "Matrix of Properties per Profile Element" is the reverse way which is related to the second validation category.

After completing the earlier matrices, by using "Matrix of Profile Element-Concept Mapping", it is possible to compare the requirements to what was addressed. Therefore, there are three possible results from each execution:

1. If every required concept property matches one profile property, then profile is considered correct.
2. If there is a required concept unmatched by the profile properties, then there is a first category problem: profile is not allowing a valid usage.
3. If there is a profile property unmatched by required properties, then there is a second category problem: profile is being excessively permissive.

However, it is frequently needed to review the requirements matrix, as identifiers regarding the same property may be different. If an inconsistency is already constrained by correctly defined OCL code, then the inconsistency should be ignored. It is important to identify these issues before assuming a profile as incorrect.

After comparing both matrices, the unmatched properties indicate inconsistencies, as seen on Table 5. Different multiplicities also indicate inconsistencies, although not shown in the example. It is possible to identify that profile does not allow to use *strictfp* modifier, *after throwing* and *after returning*, requirements specified on Matrix of Required Properties. These are first category inconsistencies.

The profile allows advices with *abstract*, *static* and *final* modifiers. *Return type* (marked with an asterisk) is specific for around advices, therefore, it should not be available for every advice. These are second category inconsistencies.

Table 5: Matrix of Inconsistencies

Cat.	Concept	Property	Req.	Add.
1st	Advice	"strictfp"	Yes	No
1st	execution type	"after throwing"	Yes	No
1st	execution type	"after returning"	Yes	No
2nd	Advice	"Abstract Modifier"	No	Yes
2nd	Advice*	"return type"	No	Yes
2nd	Advice	"Static Modifier"	No	Yes
2nd	Advice	"Final Modifier"	No	Yes

The outcome of the execution was detecting inconsistencies of first and second categories, other important inconsistencies can be found in Section 4.

4. PROFILE INCONSISTENCIES

In this Section there are more detected inconsistencies of AspectJ Profile, identified by our process. Only a few of them are shown for each concept, however, they are enough to notice the complexity of evaluating the profile and that the inconsistencies are not restricted to advices. It is important to apply the process exhaustively in order to attain optimal detection and, consequently, improved problem correction.

Validating *PointCuts*, on Table 6, there are differences between the requirement and what was addressed by the profile, respectively, *PointCut* from AspectJ and ≪PointCut≫ stereotype, which are applied on *Operations*. As identified earlier as required, *PointCuts* can be abstract, Also, among addressed properties, it was identified that profile allows *PointCuts* with *type*, *type modifier* and *default value*, properties unmatched by the requirements matrix.

Table 6: ≪PointCut≫ applied on *Property*

Cat.	Property	Required	Addressed
2nd	"Type"	No	Yes
2nd	"Type Modifier"	No	Yes
2nd	"Default Value"	No	Yes
1st	"Abstract"	Yes	No

On Table 7 there are differences between *Aspect* from AspectJ and ≪Aspect≫ stereotype, which is applied to to classes, allows in model level. It was identified that the profile allows class extending aspects, concrete aspects being inherited, and intertype declarations edit static definitions of aspects; these uses are not allowed for AspectJ codes.

Table 7: ≪Aspect≫ applied on *Class*

Cat.	Property	Req.	Add.
2nd	"Classes may Inherit Aspects"	No	Yes
2nd	"Concrete Aspect Inheritance"	No	Yes
2nd	"InterType Targeting Aspects"	No	Yes

On Table 8 there are differences between *InterType declaration* from AspectJ and ≪StaticCrosscuttingFeature≫ stereotype, which is applied on *Features*.

Table 8: ≪StaticCrosscuttingFeature≫ applied on *Operation* or *Property*

Cat.	Property	Req.	Add.
2nd	"InterType Declarations Inside Classes"	No	Yes
2nd	"External PointCut declaration"	No	Yes
2nd	"External Advice declaration"	No	Yes

5. CONCLUSIONS

A process to check UML profiles was proposed in this paper. It is capable of detecting inconsistencies in AspectJ Profile, which can be fixed in order to reach a better semantic level. The main objective is to make model validations easier, improving the quality of profiles and models.

Aspect-oriented software developers are in need of a detailed design notation and UML profiles may solve this problem, which encouraged us to create an improved version of the profile. Most of the inconsistencies found in second category and every of the first category were corrected. A prototype of a tool helping to apply the process was also created. These are not shown due to size limitations.

This paper, however, shows only one profile being evaluated. It is preferable to apply the proposed process in more UML profiles and, optionally, comparing different profiles to choose the best option for each use.

Another limitation is that we did not verify if the exhaustive application of the process is enough to identify all possible inconsistencies. Also, we could not demonstrate if a given profile is completely correct. It is also unclear how to detect inconsistencies caused by inheritance among model elements.

As a future work, we intend to analyze the possibility of using an fixed version ofthe profile with Model Driven Development for AspectJ code generation. It is also intended to conduct an experiment similar to the one performed by Uetanabara [13].

6. ACKNOWLEDGMENTS

Thiago Gottardi would like to thank CNPq for funding (Processes 132996/2010-3 and 560241/2010-0). Also, FAPESP for grant number 2009/16173-6.

7. REFERENCES

[1] S. Clarke and E. Baniassad. *Aspect Oriented Analysis and Design: The Theme Approach*. Addison-Wesley Professional, 1st edition, 2005.

[2] T. Cottenier, A. van den Berg, and T. Elrad. Motorola WEAVR: Aspect orientation and model-driven engineering. *Journal of Object Technology, Special Issue: Aspect-Oriented Modeling*, 6(7):51–88, August 2007.

[3] J. Evermann. A meta-level specification and profile for aspectj in UML. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 21–27, New York, NY, USA, 2007. ACM.

[4] L. Fuentes and P. Sánchez. Designing and weaving aspect-oriented executable uml models. *Journal of Object Technology (JOT), Special Issue on Aspect-Oriented Modelling*, 6(7):109–136, August 2007.

[5] G. Georg, R. France, and I. Ray. Composing aspect models. In *The 4th AOSD Modeling With* UML *Workshop*, San Francisco, CA, 2003.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java (TM) Language Specification*. Addison Wesley, third edition, 2005.

[7] I. Groher and T. Baumgarth. Aspect-orientation from design to code. In *Workshop on Early Aspects, AOSD*, Lancaster, UK, 2004.

[8] Y. Han, G. Kniesel, and A. Cremers. Towards visual aspectj by a meta model and modeling notation. In *AOSD - AOM*, 2005.

[9] L. Hendren, O. D. Moor, A. S. Christensen, and the abc team. The ABC scanner and parser. http://abc.comlab.ox.ac.uk/documents/scanparse.pdf, November 2010.

[10] Object Management Group. Unified Modeling Language Infrastructure Specification. http://www.omg.org/spec/UML/2.3/Infrastructure/, May 2010.

[11] P. Sánchez, A. Moreira, L. Fuentes, J. Araújo, and J. Magno. Model-driven development for early aspects. *Inf. Softw. Technol.*, 52(3):249–273, 2010.

[12] A. Solberg, D. Simmonds, R. Reddy, S. Ghosh, and R. France. Using aspect oriented techniques to support separation of concerns in model driven development. In *Proc. 29th Computer Software and App. Conf. COMPSAC 2005*, volume 1, pages 121–126, 2005.

[13] J. Uetanabara Jr., R. A. D. Penteado, and V. V. Camargo. An overview and an empirical evaluation of UML-AOF: A UML profile for aspect-oriented frameworks. In *ACM Annual Symposium On Applied Computing (ACM-SAC)*, pages 1–6, Cross-Montana, 2010.