Towards a Model-Driven Join Point Model (Extended Version)

Walter Cazzola
Dipartimento di Informatica e Comunicazione,
Università degli Studi di Milano
cazzola@dico.unimi.it

Antonio Cicchetti, Alfonso Pierantonio
Dipartimento di Informatica,
Università degli Studi di L'Aquila
{cicchetti,alfonso}@di.univaq.it

ABSTRACT

Aspect–Oriented Programming (AOP) is increasingly being adopted by developers to better modularize object–oriented design by introducing crosscutting concerns. However, due to tight coupling of existing approaches with the implementing code and to the poor expressiveness of the pointcut languages a number of problems became evident. Model–Driven Architecture (MDA) is an emerging technology that aims at shifting the focus of software development from a programming language specific implementation to application design, using appropriate representations by means of models which could be transformed toward several development platforms. Therefore, this work presents a possible solution based on modeling aspects at a higher level of abstraction which are, in turn, transformed to specific targets.

1. INTRODUCTION

Aspect–Oriented Programming (AOP) [9], is a relatively new paradigm introduced to better deal with the problem of crosscutting concerns, i.e. concepts which are orthogonal to the object definition dimension. In the Object–Oriented (OO) paradigm, the implementation of these concerns affects several classes jeopardizing the functional modularization. Nowadays, the number of tools supporting AOP is rapidly increasing and consequently the number of developers that use it is also increasing; by the way, using it massively, some problems come to light. The aspect–oriented code is not well re–usable while the resulting code tends to be difficult to scale and maintain [10, 13]. This problem is mainly due to the tight coupling of current aspect–oriented approaches and the implementation language of the application. Therefore, we believe that a solution could be found by adopting techniques based on MDA world, in order to leverage the abstraction for a language independent solution.

This paper proposes an approach to coordinate pointcuts, advices and join points by means of appropriate models. In particular, models are used to specify the weaving among pointcuts and advices from both structural and behavioral perspectives. Hence, the specification of the modalities pointcuts and advices are weaved together (in the sense of AOP) is given by automated transformations over the weaving models. In other words, the work pursues the following AOP enhancements:

– a better pointcut definition by using a "semantical" description;

– an advice weaving improvement enabling more complex merge operations;

– a higher possibility of concerns specification re–use.

The paper is organized as follows: the Section 2 is an overview of current AOP approaches in which we highlight common solutions and related issues; the Section 3 introduces some MDE concept, and explain how to successfully use it in the AOP field; the Section 4 speaks about related works on model weaving and AOP; finally the Section 5 draws some conclusions and ideas for future works.

2. AOP: POTENTIALITIES AND LIMITS

AOP is both a designing and programming technique that takes another step towards increasing the kinds of design concerns that can be cleanly captured within source code. Its main goal consists of providing systematic methods for the identification, modularization, representation and composition of crosscutting concerns such as security, mobility and real-time constraints. The captured aspects (both functional and nonfunctional) are separated into well-defined modules that can be successively composed in the original or in a new application.

The basic mechanisms for separating the crosscutting concerns in aspects and for weaving them together again are: *join points*, a means of identifying join points (*pointcut*), and a means of semantic affect at join points (*advice*). *Join points* represent well-defined points in the execution of a program, such as method calls, object field accesses and so on. *Pointcut* is a construct whose evaluation picks out a set of join points based on given criteria. Pointcuts also serve to define which advice has to be applied at a specific join point. An advice defines additional code to be executed at the join points and some rules about its applicability (e.g., before, after, and so on). Finally, an aspect represents a crosscutting concern and is composed of pointcuts definition and advices to be weaved at the corresponding join points. The framework that renders possible the proper execution of the assembled program is called *join point model* (JPM).

AspectJ [8] has been the pioneer of the aspect-oriented languages and it is still one of the most relevant frameworks supporting the AOP methodology.

The join point model and in particular its mechanism to identify the join points (the pointcut definition language) has a critical role in the applicability of the aspect-oriented methodology. As

SAC'06 April 23-27, 2006, Dijon, France

stated by Kiczales in his keynote at AOSD 2003 [7], the pointcuts definition language has the most relevant role in the success of the aspect-oriented technology but most of them (either static or dynamic) rely on a mechanism too tailored on the syntax of the program to manipulate. Similar approaches suffer of several flaws but mainly they hinder the join points matching in a more semantic way. Let us see the problem in details[1].

Dependency from the Syntax. The AspectJ pointcut language offers a set of *primitive pointcut designators*, such as **`call`**, **`get`** and **`set`** specifying a method call and the access to an attribute. These primitive pointcut designators can be combined using logical operations (`||`, `&&`, `!`) forming more complex pointcuts. All the pointcut designators expect, as an argument, a string reassembling (a part of) the prototype of a method or of a field declaration.

Therefore, we have just two ways for describing the join points: i) by listing or ii) by using a combination of wildcards and naming conventions. For example, to identify the methods in a specific package that change in some extent the status of the system we have to use the following pointcut (or a very similar one):

```
call(public * com.xerox.printers..*.set*(..))
```

Thus, we have to update also the pointcut when we add a new methods to the package that do not respect the naming conventions used by the pointcut or that is not explicitly listed in the pointcut. A problem that affects also the classic object-oriented development but what happens when we carry out a deeper code refactoring? Something like changing the naming convention, or changing the package name? Simply the aspects that use such a pointcut will not be applied anymore. Since the pointcut is tightly coupled to the syntax of the base code, a refactoring of the base code implies a refactoring also of the aspects applied on it. This problem is known as the *problem of the fragile pointcuts* [10].

Unrecognized Join Points. Neglecting the pointcuts' fragility aspect, they are still not usable in all possible contests. There are some computational patterns (also very simple) that cannot be captured by a pointcut. The sequence of two or more calls is the most simple example of this problem. In some extent, we can bypass the problem of matching the join point that precedes a sequence by extruding a new dummy method from the sequence and expressing the pointcut in term of this dummy method. Notwithstanding this work around, the problem still remains when the sequence is embedded in the unrolling of a loop. Moreover, not always this kind of refactoring can be applied or gives the expected results.

In general, it is difficult to use complex patterns to recognize the join points but the difficulties increase when the pattern we are looking for is based on the properties of the computational flow and does not rely on the syntax of the base code itself. Something like capturing all the join points from where starts a specific computational trace expressed abstracting from the specific code up to use a more abstract description language (such as the natural language) to express them. Decoupling the pointcuts from the base code representation is essential to modularize a general and then reusable concern and to free the aspect modularization from the syntactic limits of the programming languages.

Dynamic Join Points Selection or Restriction. Dynamic join points identification is neglected by most of the available join point models. Sometimes we have the necessity of describing a set of join points in terms of other join points, e.g., all the join points that can be reached from a given join points in two hops. To determine this kind of join points the framework needs a deepen knowledge of the program's dynamic execution. The **`cflow`**[2] pointcut declarator, that allows to select all the reachable join points from the currently matched join point, is one of the few exceptions. Notwithstanding that **`cflow`** cannot be used as a brick to build more complex sets of join points, e.g., by intersection, union or complement with another set of join points. A similar selection mechanism would be the basic tool to select the join points on a more semantic way looking after the properties more than after the program syntax.

In spite of the potentialities of the aspect-oriented methodology, its applicability is limited from the poorness of the proposed mechanisms to the join points identification. As stated by Gregor Kiczales [7] a lot of work has still to be done to render the mechanism more expressive and therefore usable in all contests. In his talk, Kiczales suggested to define a more precise pointcut declarator, named **`pcflow`**, based on the prediction of the computational flow but we think that to widen the mechanism applicability we need an approach more trustworthy and not based on heuristics.

3. RAISING ABSTRACTION

Shifting the focus of software development from a programming language specific implementation to application design, using appropriate representations by means of models is likely the major characteristic of MDA [12]. The main artifacts of this emerging technology are the models, which can be manipulated by means of automated transformations in order to define, for instance, mappings towards different target platforms | just to mention one of the most common application of model–to–model transformations [14].

Different concerns of a software system are often modeled by means of distinct models which are kept consistent and connected by linking the corresponding concepts among the models itself. Weaving models [6] can be given in order to specify such correspondences and to define weaving operations which merge semantically similar concepts from different ones[3].

Accordingly to such vision, modeling techniques can be used to select in a more semantics–based fashion join points and weave code into those points (i.e., a JPM). Pointcuts can be seen as queries to the code, used to weave advices into the original code at the selected join points. This work aims at lifting the reasoning from the code to the model level. A pointcut provides with patterns that could be found in the computational flow of the program, these patterns identify a set of join points. Every time a certain pattern is recognized, the related join point is picked up in order to apply the correspondent advice. In the proposed approach, models are defined for pointcuts, join points, and advices, respectively. Thus, the overall development process consists in starting from an application model, which describes the business logic, and enriching it by weaving it with the models describing several crosscutting concerns, such as data persistence, security, user auditing and so on. At this stage, how those different features interact with the initial application, both from the structural and the behavioral point of view, must be provided. Concerning the structural view, we have the application and the advice models, that we call *introduction model*. Moreover, we need to specify how the previous models will be merged in a single model; in fact, it is often necessary to refac-

[1]Note that, in the rest of the paper we will present examples written in AspectJ but similar consideration and examples can be done also adopting a different AOP language.

[2]Of course, the same consideration also applies to the `cflowbelow` pointcut declarator.

[3]Because of the ambiguity between MDA weaving concept and AOP one, we will use model weaving for the former and simply weaving for the latter when not clear from the context.

tor the application model (base) with the new introduced elements (extension), and not just compose them straight away. Hence, we describe this operation in a detailed way by using the *structural weaving model*; this model relates base and extension by introducing more abstract mappings among the elements. In particular, we need to express three distinct cases:

- an extension element is inserted as stand alone in the base model;
- an extension element is linked to some base elements;
- an extension element is used to modify some base elements.

In the first case, we use as base element a class with a specific stereotype, namely «`application`», which is used to identify the whole base model, and as extension what we would like to insert from the *introduction model*; these elements will be linked with a bidirectional association. In the second case, base and extension will be linked by the desired kind of association (generalization, composition and so on). Finally, the elements will be linked by a bidirectional association, decorated with two tagged values, namely `type` and `scope` which are used as a refinement. The former can be `insert` or `merge`, meaning that the extension will overwrite the base or it will update that respectively. The latter tag is used to narrow the manipulation to the set of base: `attributes`, `methods`, or both. If the just mentioned tagged values are not specified, we assume by default `type=insert` and as `scope` the class as a whole.

Once described the structural cooperation, we need to specify the behavioral one. As said above, we consider pointcuts as patterns describing computational flows; thus, we use activity diagrams [11] for modeling them. Every action name describes the semantic of a certain operation; therefore, a computational flow is represented by a sequence of actions. Between those actions we have to identify the join point we want to capture by using the stereotype «`joinpoint`». In this way, every time the flow is recognized the advice can be weaved to the desired point; we can either add new operations or substitute a set of them. In the case of substitutions we also need to mark the starting and the ending points of such modification; to this purpose, it is possible to use the stereotypes «`startjoinpoint`» and «`endjoinpoint`» respectively, and the advice will replace the surrounded block of actions. It is also possible to reuse the block inside the advice by using the stereotype «`proceed`». Finally, when describing computational flows, it could be useful (or necessary) to specify a sort of wildcard, «`anyflow`», meaning that we are not interested in matching a particular computational flow, rather any flow of the base can match that part of the pointcut. In this case, the advice model is very similar to the pointcut one; it is described by computational flows and it can contain context information passed during the weaving phase by means of an object. As in the structural case, we need to describe the weaving in a third model, namely *behavioral weaving model*; in this model we specify the links between the join points selected by the pointcuts and the advices by means of directed associations from a pointcut to an advice. This link can be refined by using a tagged value, `parameter`, which contains a list of arguments to be passed to the advice.

To better explain the approach, we show the classical *logging* example realized by using our model–oriented approach. Hence, we consider an ATM application to extend with the logging non–functional concern. For sake of clarity and space, we show a simplified ATM application model (Fig. 1); in that figure you can see the representation of three kind of `User`, namely `Administrator` (of the ATM system), `Developer` (of the ATM application) and `Customer`, which can perform the operations proper of her/his role. Then, we

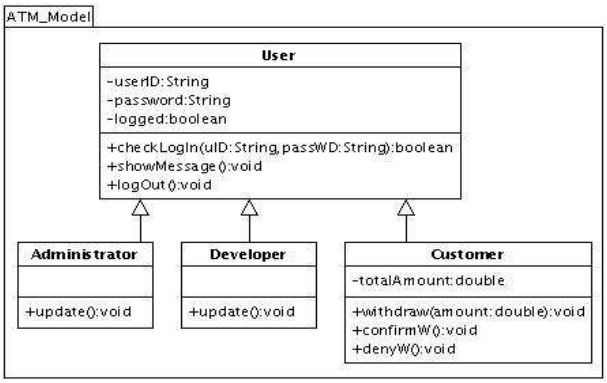
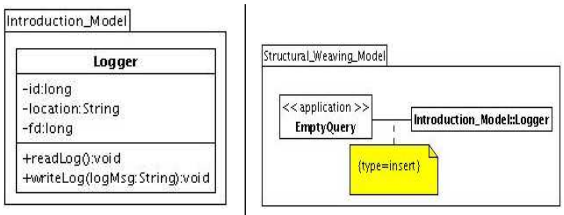


Figure 1: The ATM model

would like to log to a file some operations: for example, when a `User` tries to login, when a `Developer` commits some updates to the system, and when a `Customer` makes a withdraw attempt. This logfile will be used by an administrator for management and security reasons. As in the classical AOP realizations, we have to define pointcuts, advices and join points. In Fig. 2.a there is the *introduction model* that contains the logger class, i.e. the class that will be added to the application model while the *structural weaving model* (Fig. 2.b) contains the links application elements to *introduction model* ones. In the figure we can see that the `Logger` class has

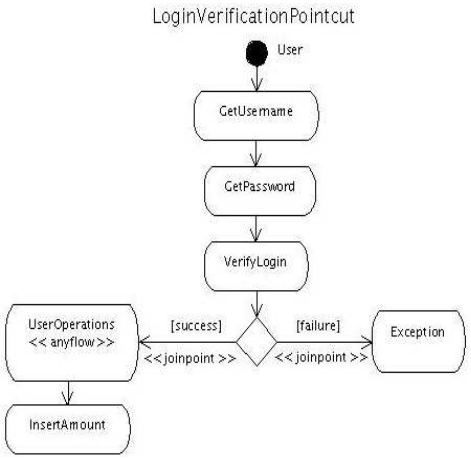


a) Introduction Model b) Structural Weaving Model

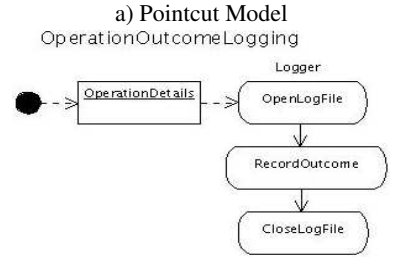
Figure 2: The Logger structural models

to be inserted as a stand–alone class (we have used the stereotype «`application`»). Once we have defined the refactoring, we need to weave new behaviors to the application; we will have to specify the operation flows to be modified (pointcuts) and the new flows to be added (advices). In Fig. 3.a, we can see the flow of a general login operation; so we read a userid, a password and then we check the correctness of the introduced data. The log behavior is added to the two join points identified by the «`joinpoint`» stereotype on the links exiting from the operation outcome diamond. The use of the «`anyflow`» stereotype permits to exclusively select, in case of success, and then log those computations that include withdrawal operations (`InsertAmount`). As depicted in Fig. 3.b the advice receives context information (represented in this case by the `OperationDetails` object), and describes the new operations. Finally, in Fig. 3.c there is the description of the necessary links; thus, we relate the `LoginVerificationOutcome` pointcut to the `OperationOutcomeLogging` advice. Any join point can be captured by several different pointcuts but there is no direct association with the advices and we need just a weaving rule to connect them.

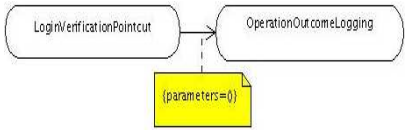
In the second example, we want to change users' authorization



a) Pointcut Model



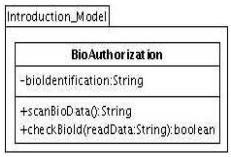
b) Advice Model



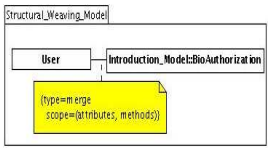
c) Behavioral Weaving Model

Figure 3: The Logger behavioral models.

procedure enabling the biometric recognition[4] non–functional concern. In the structural definition we add a new attribute and two new methods to the `User` class by a `merge` operation (see Fig. 4.b). In fact, we want to update the source and not to overwrite it. Moreover,



a) Introduction Model



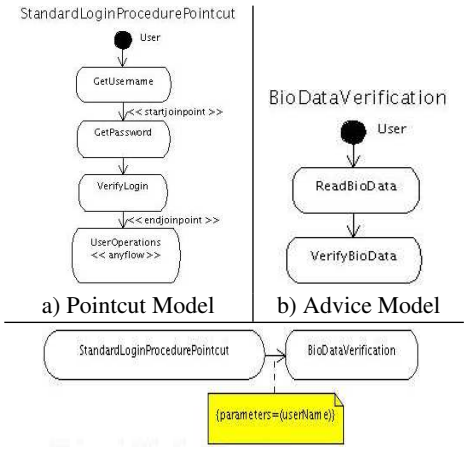
b) Structural Weaving Model

Figure 4: The new login procedure structural models

we specify that we want to merge attributes and methods. In the behavioral models, we substitute the old login procedure with the new one. So, we model the login operation flow and we mark the limits of the substitution by means of the «`startjoinpoint`» and «`endjoinpoint`» stereotypes (see Fig. 5.a); in the advice model we describe the new flow of operations, hence the biological scanning and the verification with user data (see Fig. 5.b), that will substitute the flow captured by «`startjoinpoint`» and «`endjoinpoint`».

[4]This is the authorization procedure based on the recognition of the fingerprint or iris scanning.

In this case, the advice completely replaces the captured flow so the «`proceed`» stereotype is not used in the advice. In this exam-



a) Pointcut Model b) Advice Model

c) Behavioral Weaving Model

Figure 5: The new login procedure behavioral models.

ple, when we weave the advice (`BioDataVerification`) into the join point coming from the (`StandardLoginProcedurePointcut`) pointcut, we have also to pass the actual instance of `userName` to the advice; in fact, this is necessary to compare read biological data with the one corresponding to the user in the database (see Fig. 5.c).

Starting from the previous models, we realize both the structural and the behavioral weavings. In the former case, the transformation of the application model is performed accordingly to what specified in the *structural weaving model*, while in the latter the bindings with the "real" names of methods involved are realized, obtaining a more detailed model in which is represented the sequence of operations (by means of a sequence diagram). Then, by using a transformation supported with all the models we will be able to generate the target code.

Compared with current approaches, we can observe some advantages; first of all, we have a more semantics–related pointcuts definition. In fact, when we describe operational flows, we are identifying the precise meaning of what we would like to weave with advices, and not the name of the method that performs it. This makes the definition more understandable, less dependent on the implementation and easier to maintain. Besides, we are able to define complex weaving; as mentioned in Section 2, there are particular cases where it is difficult to define pointcuts. By using this approach, thanks to the higher level of abstraction we are able to model all those cases; we can describe cycles in pointcuts, we can make a join point dependent on the operations that follows it, included other join points. Moreover, we are free from name conventions. Another improvement is the precise specification of the refactoring due to an aspect, which makes it possible to easily modify and maintain also this portion of the model (and hence of the whole application). Finally, we have a complete independence of the implementing programming language. This makes pointcuts and advices applicableÂ both to code written by using different programming languages, and to specifications different from the one used to define them.

There are also some drawbacks. Mainly, transforming the model toward a specific target could be not so simple and obvious; in fact, if on one hand the level of abstraction enables complex definitions, on the other hand it makes the transformations difficult to specify.

Besides, we could choose to transform the whole application in a "flat" code, where aspects are no more observable. In such a way, from one hand we make easier transformations definition, but from the other hand we have to perform a complete code generation every time we make changes to the model. Another choice could be to generate AspectJ like code to apply to the real code, with the risk to have the same problems mentioned in Section 2.

4. RELATED WORK

Several papers focused on problems related with current AOP approaches. Most of the approaches address only specific aspects which derive from the problems described in the first part of this paper. In particular, in [2] the authors propose a solution to improve pointcuts and advices definitions for supporting software evolution by means of statecharts tackling mainly the problem of software refactoring. In [10] a solution for the maintenance of the aspect code by mining differences between several versions. A better pointcut definition and maintenance, which has been also the main objective of this paper, is proposed in [13] where they propose inductively generated pointcuts; the main difference lies in the nature of their proposal which remains intrinsecally based on syntax–related mechanisms. Extensions of the crosscut languages have been investigated in [5], but still the approach is not model–based and remains dependent on the code. Regarding model weaving, in [4] there is the attempt to define a meta–model for that operation; in that paper, they propose to relate semantically similar concepts from different domains by means of links, supported by the human knowledge of that domains (and eventually heuristics). By the way, this operation deal with the static modeling, and so there is no information about dynamic behavior.

5. CONCLUSIONS AND FUTURE WORK

Current AOP approaches suffer from well known problems which this paper tried to address by leveraging the abstraction of the crosscutting concern specification. The approach proposes to relate information encoded in different role–specific models by using model weaving, the technique has been illustrated throughout the paper by means of examples which showed how typical problem patterns can be better faced with more abstraction in the join point model.

Currently, we are investigating on suitable transformations to automate the generation toward a target programming language from the source models by using techniques already proposed by the authors and validated in other domains [1, 3].

6. REFERENCES

[1] Mauro Caporuscio, Davide Di Ruscio, Paola Inverardi, Patrizio Pelliccione, and Alfonso Pierantonio. Engineering MDA into Compositional Reasoning for Analyzing Middleware-Based Applications. In *Proceedings of the 2nd European Workshop on Software Architecture (EWSA'05)*, LNCS 3527, pages 130–145, Pisa, Italy, June 2005. Springer.

[2] Walter Cazzola, Sonia Pini, and Massimo Ancona. AOP for Software Evolution: A Design Oriented Approach. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, pages 1356–1360, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.

[3] Davide Di Ruscio and Alfonso Pierantonio. Model Transformations in the Development of Data-Intensive Web Applications. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)*, LNCS 3520, pages 475–490, Porto, Portugal, June 2005. Springer.

[4] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: A Generic Model Weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, Paris, France, June 2005.

[5] Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 60–69, Boston, Massachusetts, April 2003.

[6] Jan Hendrik Hausmann and Stuart Kent. Visualizing Model Mappings in UML. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 169–178, San Diego, CA, USA, June 2003. ACM.

[7] Gregor Kiczales. The Fun Has Just Begun. Keynote AOSD 2003, Boston, March 2003.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.

[10] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, September 2004.

[11] OMG. Unified Modeling Language (UML) Specification version 1.4 (Draft). OMG Document ad/01-02-13, February 2001.

[12] Bram Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September 2003.

[13] Tom Tourwé, Andy Kellens, Wim Vanderperren, and Frederik Vannieuwenhuyse. Inductively Generated Pointcuts to Support Refactoring to Aspects. In *Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'04)*, Lancaster, UK, March 2004.

[14] Laurence Tratt. Model Transformations and Tool Integration. *Software and Systems Modeling*, 4(2):112–122, May 2004.