# Aspects in MDSD as an Extension of a Meta-Protocol of a Transformation System

Thomas Cleenewerck
Vrije Universiteit Brussel
tcleenew@vub.ac.be

## 1 Introduction

Model-driven developed software is not a monolithic piece of code. The software is modeled with different domain models, each capturing a concern of the software (e.g. gui, business logic, etc.). Although the concerns are separately modeled, at the implementation-level they often are tangled. Code artifacts fulfill different roles and need to be capable to interact with one another. Therefore, the artifacts contain tangled code which conceptually belongs to different concerns. The generators for each model thus are required to produce scattered code.

Likewise, the models themselves are also not monolithic, but are comprised of different concepts and relationships. A modular generator for a single model consists of a set of transformations responsible for the transformation of each concept and relationship into code or another model. Some of these transformations are fairly simple: one-to-one rules that only take a single entity as input and produce a single code fragment as output. But frequently more complex transformations are encountered. These can be categorized into two groups [vWV03]: global-to-local (G2LTs) and local-to-global transformations (L2GTs). In the former, transformations operate on additional context information to execute the transformation. In the latter, the transformations additionally affect the results produced by other transformations. So even at the level of a single model, scattered code is produced by G2LTs.

Although the basic one-to-one rules are sufficient to formalize an entire transformation process, interesting properties from a software engineering point of view like evolvability, maintainability and reusability get lost when dealing with those complicated transformations. Contemporary MDSD systems do not offer a mechanism to compose such transformation results. Hence, the crosscutting nature of the local-to-global transformations and of the different models render implementations of L2GTs, using solely the basic one-to-one rules, scattered and tangled with the other transformations [Cle05].

A prevalent technique to tackle scattering and tangling is aspect-oriented software development (AOSD) [FECA05]. AOSD provides a mechanism to separate the crosscutting behavior from the other concerns by a point-cut description and advice code.

Obviously MDSD could benefit from AOSD. Current approaches [KR03, GBNG, GBNT01, SSR+05] let the transformations and the generators produce aspects. After the models have been transformed, the produced code could be fed into an aspect-aware compiler. Despite the benefits like traceability [AGM05], these approaches raise more problems then they solve as the tangling and scattering problem is not tackled, but merely shifted towards an aspect compiler and restrictions for MDSE. First, every programming language or model (in case of model refinement) should be aspect-aware e.g. aspect-oriented programming languages and aspect-oriented modules. This is difficult, as it requires changing all existing programming languages and models. Especially in case of domain models where the notion of an aspect does not always make sense. Secondly, the generators and the transformations have to be designed such that only one generator or transformation produces the basic concern and the other ones produce the aspects. For specific cases this distinction is feasible. Simonds et.al. [SSR+05] for example, distinguishes aspect-oriented models dedicated to model quality of service requirements like persistence, etc. However in general, this not only breaks the modularity of the generators [Bri05], it is also not always possible to achieve, as we cannot predetermine

at design time of the transformations which of those transformations will be triggered [DGL+03]. Thirdly, aspect languages only support limited composition facilities (before, after, around advice) whereas generators and transformations may have arbitrary composition semantics.

The driving idea behind this position paper is to introduce the AOSD techniques into realm of transformation systems for the implementation of L2GTs and generators. Note that the position taken in this paper is formulated in terms of the background technologies of MDSD: transformation techniques and domain-specific languages.

## 2 Position

We'll exploit two remarkable parallels between AOSD and MDSD. First, the transformations of MDSD can be easily categorized, according to their crosscutting nature, into either a base-level concern or an aspect. The one-to-one transformations are the base-level concerns and the local-to-global transformations are the aspects. Second, aspects are a special case of local-to-global transformations. This is true for both their point-cuts and advices. Point-cut languages reify only a particular set of execution events whereas L2GTs reason about a complete meta-representation of the program, but at a lower abstraction-level. Furthermore, a point-cut description always denotes a set of events during the execution of the program whereas a L2GT may have effects that can only be applied once. Finally, as previously stated, the composition semantics of aspects (before, after, around advices) is to restricted. In our approach we introduce crosscutting capabilities into the transformation system to implement the L2GTs and we generalize the composition semantics as required.

*Our position statement is that a transformation system equipped with a meta-protocol allows to implement crosscutting transformations and generators as separate concerns.*

In contrast to aspects, where the base code is oblivious to the aspect code, the transformations are not necessarily oblivious to the L2GT i.e. the other transformations may depend on their effects. To enable these interactions, we integrated the implementation of L2GTs within the transformation system itself. In other words, L2GTs are implemented by changing the behavior of the transformation system, so that the system handles the crosscutting code fragment differently from other code fragments. Instead of merely producing these crosscutting code fragments, the fragments must invasively be integrated in the correct places of the program/model that was produced by other transformations.

We used the open implementation design technique to construct a new transformation system were an individual transformation can change the implementation strategy of the transformation system. The transformation system exposes its internal datastructures and processes by offering a meta-object protocol. This meta-protocol is called the primary meta-object protocol.

Instead of letting each L2GT specify how the transformation should handle the crosscutting code fragments, the primary meta-object protocol of the transformation system is extended with additional methods that will contain the common parts. Such extensions are called secondary meta-object protocols. Each crosscutting code fragment is furthermore accompanied by a specification to complete the missing parts or to further specialize the protocol, usually stating the specific conditions of where and how that code fragment should be added to the rest of the produced program.

The complete extension of the meta-protocol to handle crosscutting code fragmenets is shown below. It consists of four methods `relocate:`, `integrate:`, `corresponds:` and `combine:`.

```
RELOCATE:
   INTEGRATE:
      CORRESPOND:
         COMBINE:
      MEMBER:ADD:
```

The `relocate` method moves the code fragment to correct place of the produced program. It invokes the `integrate` method to try and integrate the code fragment in a particular place of the produced program. In the integration of the code fragment we first check whether the code fragment already corresponds to an

existing code part by calling the `correspond` method. If a correspondence is found, the two fragments are combined, otherwise the fragment is added to the program using the `member:add:` function.

Although aspects can be considered as a specific case of L2GTs, the latter are (as previously stated) formulated in a lower abstraction-level. L2GTs operate on a static meta-representation of the source code instead of the domain of run-time execution events of a program. This abstraction level can be obtained by further extending the protocol. Due to the page limit, this protocol extension could not be included here.

# 3 Conclusion

During the construction of compilers or transformers for MDSD with transformations systems we frequently encounter crosscutting generators and complex transformations that go beyond the primitive one-to-one transformation. Local-to-global transformations are amongst others such complex transformations. As these transformations additionally change the results produced by other transformations, they can not be implemented as a separate concern. AOSD is remarkably closely related to this challenge in MDSD. We managed to transfer the AOSD ideas to the transformation community and at the same time generalized the notion of an aspect. The position defended in this paper is that a transformation system equipped with a meta-protocol allows to implement local-to-global transformations as separate concerns. In this setting aspects are a special case which can be implemented by creating a specialized and extended meta-object protocol of a transformation system.

# References


[AGM05] Pablo Amaya, Carlos Gonzalez, and Juan M. Murillo. Towards a Subject-Oriented Model-Driven Framework. *The 1st International Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems*, November 2005.

[Bri05] J. Brichau. *Integrative Composition of Program Generators.* PhD thesis, Vrije Universiteit Brussel, 2005.

[Cle05] Thomas Cleenewerck. Disentangling the Implementation of Local-to-Global Transformations in a Rewrite Rule Transformation System. In *Proceedings of the Symposium on Applied Computing Conference*, 2005.

[DGL+03] Keith Duddy, Anna Gerber, Michael Lawley, Kerry Raymond, and Jim Steel. Model Transformation: A declarative, reusable patterns approach. In *EDOC*, pages 174–185. IEEE Computer Society, 2003.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[GBNG] J. Gray, T. Bapty, S. Neema, and A. Gokhale. Aspect-Oriented Domain-Specific Modeling.

[GBNT01] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Commun. ACM*, 44(10):87–93, 2001.

[KR03] Vinay Kulkarni and Sreedhar Reddy. Separation of Concerns in Model-Driven Development. *IEEE Software*, 20(5):65–69, Sept/Oct 2003.

[SSR+05] D. Simmonds, A. Solberg, R. Reddy, R. France, and S. Ghosh. An Aspect-Oriented Model Driven Framework. *Ninth IEEE The Enterprise Computing Conference (EDOC)*, September 2005.

[vWV03] Jonne van Wijngaarden and Eelco Visser. Program Transformation Mechanics. Technical Report UU-CS-2003-048, Universiteit Utrecht, 2003.