# Term Rewriting Meets Aspect-Oriented Programming

Paul Klint, Tijs van der Storm, and Jurgen Vinju

Centrum voor Wiskunde en Informatica (CWI),
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands
{Paul.Klint, T.van.der.Storm, Jurgen.Vinju}@cwi.nl

*Dedicated to Jan Willem Klop on the occasion of his 60th anniversary.*

*Term rewriting is in the intersection of our interests and physical distance has never been large. Nonetheless we seem to be living at opposite ends of the term rewriting galaxy. Here is a story from the other side of that galaxy.*

**Abstract.** We explore the connection between term rewriting systems (TRS) and aspect-oriented programming (AOP). Term rewriting is a paradigm that is used in fields such as program transformation and theorem proving. AOP is a method for decomposing software, complementary to the usual separation into programs, classes, functions, etc. An aspect represents code that is scattered across the components of an otherwise orderly decomposed system. Using AOP, such code can be modularized into aspects and then automatically weaved into a system.

Aspect weavers are available for only a handful of languages. Term rewriting can offer a method for the rapid prototyping of weavers for more languages. We explore this claim by presenting a simple weaver implemented as a TRS.

We also observe that TRS can benefit from AOP. For example, their flexibility can be enhanced by factoring out hardwired code for tracing and logging rewrite rules. We explore methods for enhancing TRS with aspects and present one application: automatically connecting an interactive debugger to a language specification.

## 1 Introduction

Software engineering is about conquering the complexity of real life software systems. Can large systems be organized such that they remain manageable? Many solutions have been tried from structured programming to abstract data types, modules, objects, components and agents. In specific areas some of these approaches have been successful but the problem of structuring and organizing software remains mostly open for research. In more recent years, *aspects*, *concerns* or *dimensions* of software systems have been investigated [19]. These approaches aim at encapsulating functionality that cuts across boundaries of conventional modularization. In this way, software would become composable along different

A. Middeldorp et al. (Eds.): Processes... (Klop Festschrift), LNCS 3838, pp. 88–105, 2005.

axes and the desired flexibility and composability could be achieved. While providing potential solutions to the software composition problem, they pose new problems as well: how can such new methods of modularization be combined with existing languages and how can they be supported by tools?

*Term rewriting* [29] is a well-known paradigm used in program transformation, and thus a natural candidate for developing language-oriented tool support. We first give quick introductions to aspect-oriented programming (Sect. 1.1) and applications of term rewriting (Sect. 1.2) and then we explain why it is interesting to explore connections between these two fields and how they can benefit from each other (Sect. 1.3).

The contributions of the paper can be summarized as follows:

- It raises the awareness that term rewriting techniques can be relevant for the implementation of aspect-oriented programming (Sect. 2).
- It explores the application of aspect-oriented techniques to term rewriting systems themselves (Sect. 3 & 4).
- It formulates research questions in the field of term rewriting that are brought forward by the previous two points (Sect. 5).

### 1.1 Aspect-Oriented Programming

One of the most important principles in software engineering is the principle of *separation of concerns*. Separating concerns in modules (e.g., functions, classes etc.) promotes maintainability and reuse, because the dependencies between modules are loose and explicit.

There are, however, concerns that cannot be adequately modularized using conventional mechanisms. Typical examples of these so-called crosscutting concerns are profiling, tracing, debugging, error handling, origin tracking, caching, and transaction management. In all these cases, the code to implement these concerns occurs in many modules since all these modules are affected by the concern in question. This situation is referred to as *code scattering*.

Aspect-Oriented Programming (AOP) [19] is an approach to ameliorate this situation by introducing a new modularization concept: *aspects*. An important characteristic of AOP is *quantification* [11]. For example, "whenever condition $C$ arises in program $P$, do $X$" is a quantified statement over program $P$. The scattering of code for crosscutting concerns is avoided by automatically *weaving* the aspect code $X$ in places where condition $C$ holds.

In many AOP implementations, quantification over a program is achieved by specifying *pointcuts*. A pointcut is an addressing mechanism for the static or dynamic identification of execution points in the base program. These execution points are called *joinpoints* since at these points in the source code, the aspect code is joined with the base code.

To illustrate the notion of a pointcut, consider the example in the upper left part of Fig. 1 expressed in AspectJ [18], the aspect language for Java. The pointcut `creatingFoo` captures all calls to constructors of class `Foo`, disregarding the argument signature.