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@cwi.nl T.van.der.Storm@cwi.nl 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 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 [30] 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.

Advice specifications describe how the aspect code should be weaved at the joinpoints captured by a certain pointcut. There are three kinds of advice: before, after or around. Figure 1 contains an example of around advice. The directive **proceed**`()` is used to continue the delayed execution of the joinpoint, in this case constructing a new `Foo`-object. Figure 1 also shows on the left an example program and on the right the result of applying the given pointcut and advice to it.

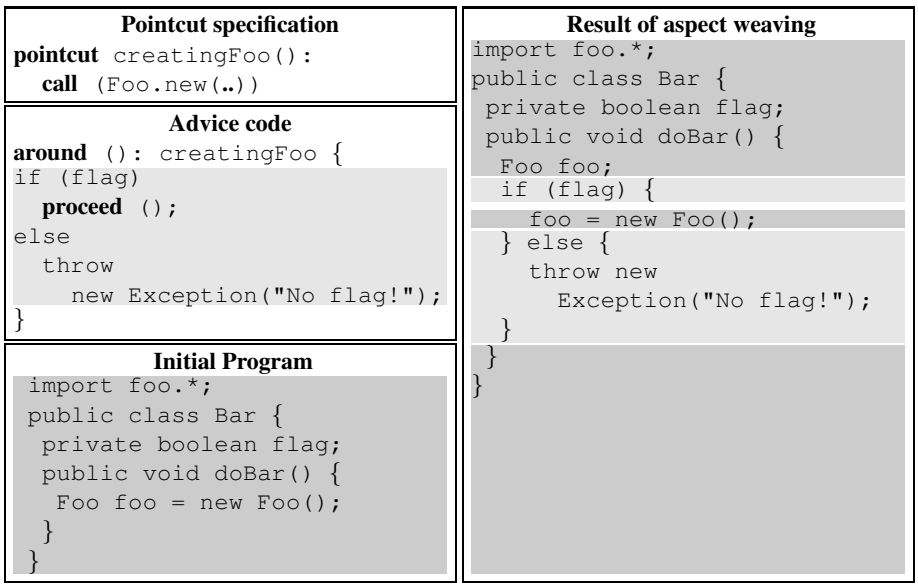

Fig. 1. Pointcut and advice are used to weave code into a small program.

1.2 Applications of Term Rewriting

In this paper we consider term rewriting to be a programming paradigm. The concept of term rewriting systems is used in many application areas, such as functional programming, program transformation, theorem proving, and language semantics [14, 30]. From the viewpoint of these application areas term rewriting systems are *programs* [21].

To cater for different kinds of applications, most rewriting implementations extend basic rewrite rules with additional features. Such features include, for instance, concrete (mixfix) syntax [22], conditional rewrite rules [4], ordered rules [2], list matching and AC matching [10], traversal functions [31], strategies [6] and more. For this paper, the conditional rewrite rules are essential, while concrete syntax and traversal functions are practical utilities for the application we present.

A conditional rewrite rule is a normal rewrite rule, extended with a list of predicates. Now a redex must also satisfy all such predicates, before it can be contracted. Different kinds of predicates are allowed. For example, in our system we only allow (in)equality between two terms that are first normalized, or an (un)successful match between a normalized term and an open term.

A full account of the theoretical foundations of term rewriting and of systems implementing it is given in [30]. We will give examples using the notation found in the ASF+SDF META-ENVIRONMENT [7, 22].

1.3 Connections Between the Two Fields

Why is it interesting to explore the connections between AOP and term rewriting?

Aspect-Oriented Programming Can Profit from Term Rewriting. The ideas for AOP contribute to software composition and maintenance and have been applied to mainstream programming languages (Java [18], C/C++ [28], C# [20], SmallTalk [15], Caml [29], Cobol [24]). In all these cases language-specific tool support has been developed. It is worthwhile to wonder whether AOP is applicable in the context of other languages, such as Perl, PHP, legacy languages, or even domain specific languages. The question, then, is how to develop tool support to evaluate such hypotheses. Term rewriting is used in other kinds of program transformation, so it would be natural to apply it to aspect weaving as well. In Sect. 2 we will investigate whether term rewriting is a good choice for rapidly implementing aspect weavers for new languages.

Term Rewriting Can Profit from Aspect-Oriented Programming. In many applications the *side-effects* of term rewriting systems are important. However, such side-effects are usually hardwired into a particular term rewriting engine. For example, each engine typically implements one kind of reduction tracing or debugging. We propose to separate these hard-wired aspects from the engine, and promote them to programmable aspects on the term rewrite system level. The result is that the application of existing engines can be made much more flexible and reusable. In Sect. 3 we explore a way to add reusable side-effects to an ASF+SDF term rewriting system, by employing aspect-oriented programming.

2 Aspect-Oriented Programming Can Profit from Term Rewriting

Since term rewriting is well equipped to deal with program transformation, aspect weaving is also a natural application area. This section provides an example of how to use term rewriting to implement aspect weavers.

The view that aspect weaving is a kind of program transformation is not new. For example, in [12] a case is made for a term rewriting approach to weaving aspects. Using special pattern-matching operators the authors are able to succinctly specify how aspects should be weaved. Graph writing for aspect weaving is discussed in [1]. It is argued that graph rewriting is more suitable than term rewriting because the base language consists of class graphs. Semantic information can be stored naturally in graphs, so more complex pointcuts may be expressible.

In most approaches that use rewriting for aspect weaving, rewrite rules directly function as the aspect weaving language. In [13] the authors present an aspect-oriented programming language for ObjectPascal which is implemented on top of DMS [3]. DMS contains a term rewriting component that forms the basis of their weaving algorithm, but this fact is hidden from the user. The strong motivation for using term rewriting is rapid development of aspect weavers for legacy languages.

In this section we adopt the latter approach, and demonstrate the implementation of a simple aspect-oriented programming language called μAspectJ. It consists of a very simple weaver written in ASF+SDF. It is able to weave advice similar to the example of Fig. 1. After this example, we will evaluate the fitness of term rewriting in this application area.

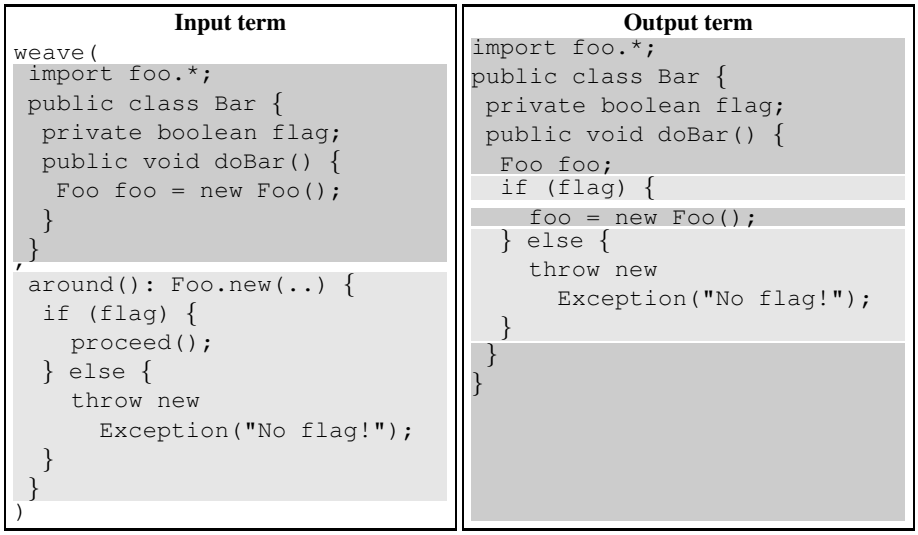

Fig. 2. Weaving is rewriting: a μAspectJ program is weaved by reducing the `weave` symbol.

2.1 Implementing a Weaver for μAspectJ

The expected behavior of μAspectJ on the example from the introduction is displayed in Fig. 2. The function `weave` is applied to two arguments: a Java compilation unit and an advice specification (shown on the left). The result is a new Java compilation unit that is the result from weaving the advice in the original Java code (shown on the right).

This weaver can be defined in one module of ASF+SDF code, which we present here. ASF+SDF modules consist of two parts. The syntactic part defines the types of input terms and functions. These types are defined using context-free syntax productions. The semantic part contains equations between terms expressed in concrete syntax. These equations are rewrite rules when read from left to right.

The definition of μAspectJ consists of three parts: the syntax of pointcuts and advice (Fig. 3), the signature of the weave traversal function (Fig. 4), and the equations defining this function (Fig. 5). The module `MuAspectJ` imports the syntax of Java and a generic parameterized module for substitution. In the syntax section, the syntax of very simple advices is defined. For the sake of brevity, we only allow pointcuts that capture arbitrary constructor invocations. The pointcuts follow the syntax of AspectJ. Advice consists of a kind (before, after or around), a pointcut and a Java statement block.

The `weave` function maps a compilation unit together with an advice specification to a compilation unit with the advice weaved in. To avoid writing a lot of boiler-plate code for traversing a compilation unit, the weave function is defined as traversal function [31]. Weave matches lists of statements containing a constructor invocation for classes that match the pointcut contained in the advice. For each kind of advice the associated advice code is weaved in accordingly. In these equations, pattern variables such as `S*`, `Class`, and `Type` are used to capture lists of statements, class names and type identifiers, respectively.

```
module MuAspectJ
imports Java Substitute[Statement Statements]
exports context-free syntax
  AdviceKind "(" ")" PointCut ":" Block → Advice
  "before" | "after" | "around"         → AdviceKind
  "call" "(" Signature ")"              → PointCut
  Class "." "new" "(" ".." ")"          → Signature
```

Fig. 3. μAspectJ: syntax of pointcuts and advices.

```
  weave(CompilationUnit, Advice)  → CompilationUnit {traversal}
```

Fig. 4. μAspectJ: signature of the weave traversal function.

Consider equation `[1]` in Fig. 5 which matches Java statements of the form "`Type Id = new Class(Param*)`" in the context of a list of statements and applies a **before** advice to it. Note that the name of the class (variable `Class`) in the statement and the advice should be the same (non-left-linearity).

In the resulting code on the right-hand side of the rewrite rule, the body of the advice, which is matched by the variable `S*`, is placed just before the original statement. The other equations work in a similar fashion.

2.2 The Fitness of Term Rewriting for Aspect Weaving

The example shows some characteristics of aspect weaving. It needs at least complex pattern matching, pattern construction, tree traversal, and non-local information (the advice code) at redex positions. The first two features are provided by basic term rewrite rules, the last two features emerge from using traversal functions [31]. As an alternative, traversal strategies and dynamic rewrite rules could be used to implement the same behavior [33]. Matching modulo associativity (list matching) in ASF+SDF, makes the weaver deal easily with lists of statements.

The example does not show how to scale up to more advanced pointcut specifications, such as exist in AspectJ. Such specifications need name space resolution, or even control flow information. Any aspect weaver implemented using term rewriting will therefore have to be preceded by a static semantic analysis phase to collect additional context information for matching pointcuts, and take this as an extra argument to the weave function. Alternatively, code could be generated to dynamically resolve semantic issues, but this may have significant repercussions on run-time efficiency.

3 Term Rewriting Can Profit from Aspect-Oriented Programming

We now shift perspective and explore whether aspect-orientation can contribute to term rewriting. There are at least two directions in which aspect-oriented term rewriting can be considered:

```
equations
[1] weave(S*1 Type Id = new Class(Param*); S*2,
          before() : call(Class.new(..)) { S* }) =
          S*1 S* Type Id = new Class(Param*); S*2

[2] weave(S*1 Type Id = new Class(Param*); S*2,
          after() : call(Class.new(..)) { S* }) =
          S*1 Type Id = new Class(Param*); S* S*2

[3] S*' := substitute(proceed();, Id = new Class(Param*);, S*)
    ================================================
    weave(S*1 Type Id = new Class(Param*); S*2,
          around() : call(Class.new(..)) { S* }) =
          S*1 Type Id; S*' S*2
```

Fig. 5. Equations defining μAspectJ weaving.

- *Starting from pure term rewriting systems* we can identify the crosscutting concerns in large sets of rewrite rules and factor them out as programmable rewrite rule aspects.
- *Starting from the applications of term rewriting systems*, we notice that crosscutting concerns already occur naturally in the form of side-effects. Usually the implementation of these side-effects is hardwired in the rewriting engine that is used.

The first direction is particularly interesting in the field of language definitions. In this field we try to define rewriting based semantics for programming languages. The combination of aspect orientation and programming language definitions has received some attention. This has mainly focused on making language definitions more modular and extensible.

For example, in [32] the idea of implementing language extensions as aspects is explored in the context of attribute grammars. A similar strategy is explored in [23]. In this paper, declarative language definitions (e.g., in SOS) are evolved by transforming the semantic rules. This allows for the incremental addition of language facets (e.g., state, input/output, exception handling, etc.) to some base language. Primitives to achieve this include adding parameters to semantic functions, adding conditions to conditional rewrite rules, and the like.

The second direction is interesting in many applications of term rewriting. Figure 6 (Step 1) displays the process of rewriting a term. An engine takes a TRS and a term, and produces a normal form and some side-effects. Typical side-effects of the rewriting process include:

- Tracing: an exported trace of a reduction sequence represents a proof. Each reduction step contained in this trace is an equational deduction step.
- Profiling: measure the execution behavior (frequency, call graph, timings) of the term rewriting system.
- Debugging: instrument the term rewriting system with debugging information and interaction.

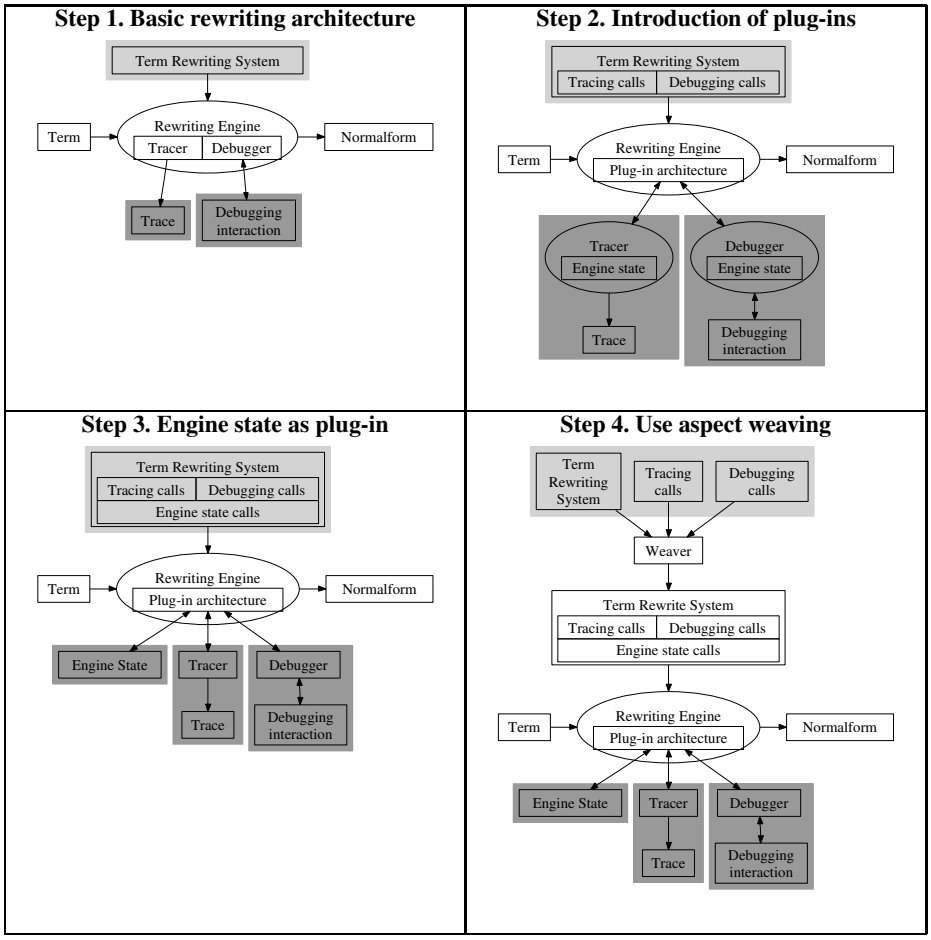

Fig. 6. From a rewriting engine with hardwired side-effects, to a completely decoupled engine with pluggable side-effects.

For some applications of term rewriting, the side-effects are even more important than the normal form. Depending on the domain, or on a specific application, in which a term rewriting engine is applied, these side-effects are specialized in different ways.

For example, a term rewriting engine that is used in concert with a proof assistant (e.g., Elan with Coq [25, 16]) generates a trace that communicates with a specific deduction process. Specializing the rewriting engine to emit such a trace makes an otherwise generic term rewriting system less applicable in a different context, let alone in another domain. For example, if we want to connect the same engine to a different brand of proof assistant, a large part of its implementation must be adapted. In [25], this problem is also recognized, and attacked by providing a separate translation scheme from a canonical representation to the specific proof term syntax of the proof assistant. In this

paper, we try to generalize this separation of concerns, and make it available for other aspects of term rewriting besides tracing.

Another problem with hardwired side-effects is the lack of control available to the user. For example, execution traces can be huge, but the user might be only interested in a localized section of a term rewriting system. In terms of space efficiency, such lack of control can lead to a lack of scalability. The possibilities that open up when the user could specialize side-effects for specific term rewriting applications are significant. In Sect. 4 we will present how a language specific debugger is obtained by adding debugging side-effects to a term rewriting system that implements the operational semantics of a language (i.e., a language interpreter).

We know of one instance of weaving side-effects into a language definition. In [34] a debugging aspect is weaved into an ANTLR [27] language definition of a domain specific language. Our example also shows how to weave in debugging support, but we use a dedicated aspect language instead of a general purpose transformation language (the authors of [34] use DMS). Note that a debugging feature does not change the semantics of a programming language, so there is no need to evolve the signatures of semantic functions.

3.1 From Hardwired Side-effects to Programmable Aspects

Figure 6 depicts how a term rewriting engine can be refactored from an architecture with hardwired side-effects to a flexible plug-in architecture with programmable side-effects. We will discuss each step in turn.

Step 1 represents a basic implementation of term rewriting. Side-effects such as debugging, profiling and tracing are hardwired into the implementation. All side-effects are initiated by the term rewriting engine itself.

Step 2 represents the case that a TRS can be extended with explicit calls to library functions that might have side-effects. In line with the convention in other frameworks, we call these library functions *plug-ins* and add a plug-in architecture to the rewrite engine. This is achieved by adding a plug-in API (Application Programming Interface) to the engine to communicate a large part of the engine's state information to a plug-in. The engine will now be able to communicate with arbitrary plug-ins and achieve arbitrary side-effects. Unlike the situation in Step 1, side-effects are no longer initiated by the engine, but explicit calls to plug-ins must be added to the original term rewriting system.

Step 3 deals with the fact that interaction between different plug-ins is largely determined by the engine state. Therefore, we separate the engine state as a new plug-in. The engine state can now be queried from a term rewriting system dynamically. This information can then be used to trigger tracing, debugging or any other side-effect in a user-defined manner.

Step 4 introduces an aspect language and separates all calls to plug-ins into separate aspects. Note that the original term rewriting system of Step 1 is back as a separate input. By weaving in several aspects we automatically obtain the complete rewriting system of Step 3. From now on, libraries of reusable side-effects can be provided for the general use-cases of a term rewriting engine, while at the same time user-defined specializations can be created without much effort.

We have now separated hardwired functionality, and replaced it by plug-ins and aspect weaving. By doing this we have gained flexibility and reusability. For example, the syntax of a reduction trace can be adapted to the input syntax of a particular proof assistant, and a language semantics can be used with and without debugging support. In both examples, the original term rewriting system does not have to be changed, but only aspects have to be weaved in.

3.2 Introducing AspectASF

The aspect weaver in Fig. 6 has been introduced for a reason. Side-effects often represent crosscutting concerns: many, if not all, rules of a TRS should be modified to include calls to the library of plug-ins. Take for instance the tracing of a reduction sequence: all firing rewriting rules have to communicate context information to the tracing plug-in. This would mean adding plug-in calls to all rules.

It would be advantageous if one could quantify over the set of rules and thus declaratively specify which rules should be adapted to incorporate the side-effect in question. For the tracing functionality, one would then say something like "add a call to trace to every rule". The actual adaptation of rewrite rules is subsequently enacted by automatically transforming the TRS.

This section introduces a simple aspect language for aspect-oriented programming in ASF+SDF. Using this language, the calls to the plug-ins can be specified separately. Invasive modifications of the TRS are not needed. Moreover, aspects have the additional advantage that we can now use a TRS with and without side-effects, or even add side-effects to parts of the TRS.

The next paragraphs will focus on how the declaration of these aspects would look like in a language called AspectASF. Firstly we will define pointcut patterns that are used for identifying sets of equations. Secondly, we show how these patterns are used in specifying pointcuts and advice.

AspectStratego [17] is another experiment in adding aspects to rewriting. That language allows much more complex pointcut specifications for inserting code along a reduction sequence. AspectASF, however, aims to primarily illustrate the viability of combining TRS with aspect-oriented techniques.

Pointcut Patterns The pointcut pattern language is a pattern matching language on the structure of equations[1]. The examples in Fig. 7 illustrate the approach. The _ pattern functions as a placeholder for concrete terms that are not of interest. The * pattern is a wild-card for quantifying over parts of literals.

For the sake of exposition we only allow pattern matching on labels of equations and left-hand sides. Patterns are expressed in concrete syntax. They do not contain any meta-variables. So, in the third example, `Env` matches with a regular ASF variable named `Env`; no binding is taking place at weaving time.

Note that this pointcut pattern language can be made more expressive by adding, for example, higher order matching, meta-variables, associative matching on conditions,

[1] Pointcut patterns are referred to as *signatures* in the AspectJ community. We avoid this term for clarity.

```
[_] _                    captures all equations
[_] eval(_,_)            ... with outermost symbol eval
[_] eval(_, Env)         ... with 2nd arg an Env variable
[int*] _                 ... with label like int..
[int*] _ or [real*] _    ... with label like int.. or real..
```

Fig. 7. A number of example pointcut patterns in AspectASF.

sort assertions etc. For this presentation, we restrict ourselves to first-order matching and simple boolean connectives (**and** and **or**) for constructing composite patterns.

Pointcuts and Advice Pointcut patterns are used in the definition of pointcuts. We identify two kinds of pointcuts in AspectASF: entering an equation (after a successful match of the left-hand side), and exiting an equation (just before returning the right-hand side). In an equation without conditions, **entering** and **exiting** are equivalent. The pointcut

entering `[_] eval(_)`

captures the points in the reduction sequence where the left-hand side of an equation with outermost function symbol `eval` has been matched successfully against a redex. The **exiting** pointcut is interpreted similarly.

Pointcuts are used in advice specifications. The kinds of advice that are allowed, exactly correspond to the kinds of pointcuts: **after entering** an equation, and **before exiting** an equation. Note that **before entering** and **after exiting** are meaningless, since such expressions do not correspond to identifiable points in the code.

As is common in aspect languages, the host language is reused for specifying advice code. In our case this language is the language of ASF conditions. To weave conditions **after entering** means prepending the advice conditions to the list of conditions of the equation that is matched by the pointcut. Similarly, advice **before exiting** corresponds to appending the advice conditions to the list of conditions of the subject equation.

Advice code can benefit from access to the context of the equation that it is weaved in. This information is provided in part by the weaver (e.g., the equation name), in part by calls to the library of plug-ins in the advice conditions itself (e.g., to obtain the depth of the evaluation stack). In the next section we will see an example of a call to such a library plug-in.

4 Applying Aspect-Oriented Term Rewriting

In this section we apply aspect-oriented programming to an ASF+SDF-specification of a small programming language. As a small case-study a plug-in for debugging side-effects was constructed. This plug-in sends and receives messages from a generic visual debugging tool called Tide [26]. With this plug-in, we can instrument term rewriting systems such that they stop at certain points in the execution and allow inspection of

```
[1] Env' := evst(Stat, Env),
    Env'' := evs(Stat*, Env')
    ==============================
    evs(Stat ; Stat*, Env) =  Env''

[2] eve(Exp, Env) != 0
    ============================================
    evst(if Exp then Series1 else Series2 fi, Env) =
                                    evs(Series1, Env)

[3] eve(Exp, Env) == 0
    ============================================
    evst(if Exp then Series1 else Series2 fi, Env) =
                                    evs(Series2, Env)
```

Fig. 8. Fragment of the original Pico interpreter

the current state. To avoid the pollution of the term rewriting system with calls to the debugging plug-in, our goal is to automatically add such calls to the specification.

Starting from a term rewriting system that implements the semantics of the toy programming language Pico, we can now easily obtain an interactive debugger. We are interested in two functions of this semantics definition: `evs(Series, Env) -> Env` (used for evaluating series of statements) and `evst(Stat, Env) -> Env` (used for evaluating a single statement). For the sake of brevity we omit some of the equations that define these functions. The equations are displayed in Fig. 8.

Recall that ASF+SDF allows matching on terms in concrete syntax. So in the first equation, the string `evs(Stat ; Stat*, Env)` is the redex pattern. Variables in these patterns start with an uppercase character (`Stat`, `Stat*`, `Env`, etc.).

Communication with the Tide debugger occurs via one library function, called `tide-step`. It receives one argument, which represents the source location of the active element of the Pico program. The source code location of the active element is obtained by calling the function `get-location`. This function is implemented as a separate plug-in. In a similar fashion one could query for other aspects of the term rewriting engine state.

Setting a breakpoint on a statement or conditional expression (in while- and if-statements) has the effect of pausing the execution. To obtain this functionality for the Pico language, the equation for statement sequencing, as well as the equations dealing with if and while statements, should include a call to `tide-step`.

The aspect declaration in Fig. 9 is used to weave in calls to the debugging side-effect at the points of interest. The pointcut `statementStep` captures the points of entry of the equation for statement sequencing. Pointcut `conditionStep` matches the `evst` equations defining the if- and while-statements. Both pointcuts are used in advice specifications, which ensure that a `tide-step` is executed at the appropriate places. Note how the advice uses some of the variables from the pointcut definitions.

The result of weaving this aspect into the Pico interpreter TRS is shown in Fig. 10. Each equation is appended with a special (tautological) condition that communi-

```
pointcut statementStep: entering [_] evs(Stat ; Stat*, Env)
pointcut conditionStep:
  entering [_] evst(if Exp then Series1 else Series2 fi, Env)
         or [_] evst(while Exp do Series od, Env)
after: statementStep tide-step(get-location(Stat))
after: conditionStep tide-step(get-location(Exp))
```

Fig. 9. A debugging aspect for the Pico semantics.

```
[1] tide-step(get-location(Stat)),
    Env' := evst(Stat, Env),
    Env'' := evs(Stat*, Env')
    ==============================
    evs(Stat ; Stat*, Env) =  Env''

[2] tide-step(get-location(Exp)),
    eve(Exp, Env) != 0
    ================================================
    evst(if Exp then Series1 else Series2 fi, Env) =
                                   evs(Series1, Env)

[3] tide-step(get-location(Exp)),
    eve(Exp, Env) == 0
    ================================================
    evst(if Exp then Series1 else Series2 fi, Env) =
                                   evs(Series2, Env)
```

Fig. 10. Pico interpreter fragment with debugging support automatically weaved in.

cates with the Tide debugger. Any previously existing condition is evaluated after the debugger is informed. A screen-shot of the resulting interactive debugger is shown in Fig. 11.

We conclude that the original Pico interpreter (Fig. 8) remains completely separated from the debugging aspect (Fig. 9) and that the Pico interpreter with debugging support (Fig. 10) can be generated fully automatically. The example clearly shows the benefits of aspect-oriented programming. Firstly, the concerns for evaluation (base TRS) and debugging (aspect) are completely separated. Secondly, the scattering of calls is avoided: two aspects affect five equations. In larger specifications this ratio (2/5) is expected to be even better.

5 Discussion and Further Research

What can we conclude from these explorations of the connection between term rewriting and aspect-oriented programming? We have shown that term rewriting is a natural choice for implementing aspect weavers as has been illustrated in the μAspectJ case (Sect. 2). This regards primarily the transformations carried out by a weaver. Another source of complexity in weavers is the amount of type information that is needed for

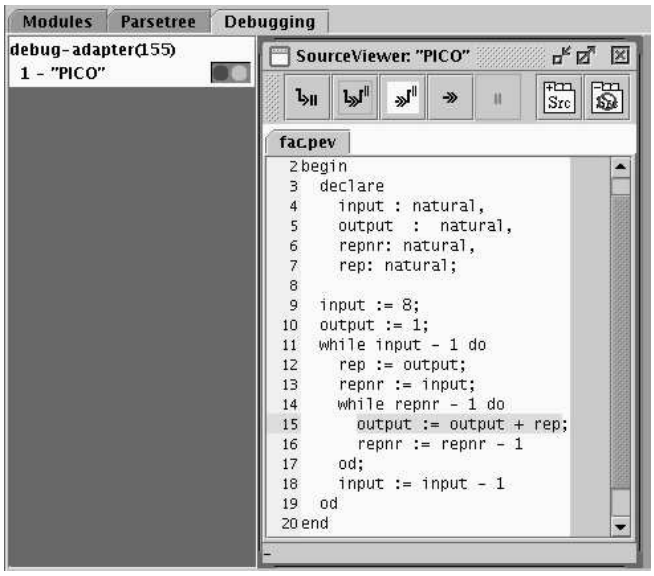

Fig. 11. A screen-shot of the generated Pico debugger in action.

the weaving process, like, for instance name resolution. Here, term rewriting has no particular advantage over other techniques.

Another conclusion is that the application of aspect-orientation to term rewriting itself opens up several new possibilities and research questions as has been illustrated by the AspectASF case (Sect. 3).

On the theoretical side, the composition of term rewriting systems has been studied for the relatively simple case of taking the union of rule sets. Aspect weaving, however, introduces a composition operation with more computational power thus decreasing the chances of predicting properties of the weaving result. Questions are:

- Is it possible to impose restrictions on the weaving TRS such that properties of the weaving result can be guaranteed? For instance, if the original TRS has a certain property (e.g. confluence, termination) how should the weaving TRS be restricted in order to guarantee certain properties of the weaving result?
- Can AOP be helpful to restructure an existing TRS in such a way that it becomes easier to proof properties of the AOP version?

On the practical side, other questions abound:

- Is it possible to design a sufficiently flexible aspect language for term rewriting systems or should one resort to full meta-programming as provided in, for instance, Maude [8]? Instead of the pointcuts and advices used in this paper, one can then use the full expressive power of term rewriting by transforming complete (collections of) rewrite rules.

– What are the implications of static versus dynamic weaving? In the former case, the initial TRS is changed by the weaver before rewriting. In the latter case, the weaving is done during rewriting. The meaning and implications of weaving during rewriting are unexplored.
– In the examples given in this paper, side-effects play a crucial role in the definition of the various aspects. Is it the case that AOP is *the* manner of introducing side-effects in a TRS without completely disturbing the underlying rewriting semantics, or are there alternatives?
– In most AOP implementations, origin information is lost. That is, if a weaved program is compiled or executed and there is an error, finding the source of this error is hard: is it in the base code, the advice code, or the interaction of the two? Origin tracking for term rewriting is a well-researched subject [9, 5] and provides a solution for this problem. Aspect weaving may, however, require specialization or extension of that technique.
– Another question is related to origin tracking as well. As the authors of [9] state, different applications of origin tracking, such as program animation, error handling and debugging, require different notions of origin tracking. It would be interesting to investigate whether the code to propagate origins could be weaved in the TRS using aspect-oriented techniques.

All these questions show that cross-fertilization between the areas of term rewriting and aspect-oriented programming is possible and desirable.

Acknowledgments

We would like to thank Pieter Olivier and Bas Cornelissen for their work on debuggers for term rewriting systems and Mark van den Brand, Jan Heering and Ralf Laemmel for comments on drafts of this paper.

References

[1] U. Aßmann and A. Ludwig. Aspect weaving with graph rewriting. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 24–36. Springer-Verlag, 2000.
[2] J.C.M. Baeten, J.A. Bergstra, J.W. Klop, and W.P. Weijland. Term rewriting systems with rule priorities. *Theoretical Computer Science*, 67:283–301, 1989.
[3] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: program transformation and practical scalable software evolution. In *International Conference on Software Engineering*, May 2004.
[4] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: confluence and termination. *Journal of Computer and Systems Sciences*, 32:323–362, 1986.
[5] I. Bethke, J.W. Klop, and R. de Vrijer. Descendants and origins in term rewriting. *Information and Computation*, 159:59–124, 2000.
[6] P. Borovansky, C. Kirchner, H. Kirchner, P. Moreau, and C. Ringeissen. An overview of elan. In C. Kirchner and H. Kirchner, editors, *Second Intl. Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, 1998.

[7] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[9] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.

[10] S.M. Eker. Associative-commutative matching with bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[11] R. Filman and D. Friedman. Aspect-Oriented Programming is quantification and obliviousness. In *Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.

[12] P. Fradet and M. Südholt. AOP: towards a generic framework using program transformation and analysis. In *Proceedings of ECOOP '98 AOP Workshop*, 1998.

[13] J. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, pages 36–45. ACM Press, 2004.

[14] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, March 2000.

[15] R. Hirschfeld. AspectS – aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer, 2003.

[16] G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 8.0*, April 2004. `http://coq.inria.fr`.

[17] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. In Horatiu Cirstea and Narciso Marti-Oliet, editors, *Workshop on Rule-Based Programming (RULE'05)*, April 2005. (To appear).

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[19] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[20] H. Kim. AspectC#: An AOSD implementation for C#. Master's thesis, Trinity College, November 2002.

[21] H. Kirchner and P.-E. Moreau. Promoting Rewriting to a Programming Language: A Compiler for Non-Deterministic Rewrite Programs in Associative-Commutative Theories. *Journal of Functional Programming (JFP)*, 11(2):207–251, March 2001.

[22] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[23] R. Lämmel. Declarative aspect-oriented programming. In O. Danvy, editor, *Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, San Antonio (Texas), BRICS Notes Series NS-99-1*, pages 131–146, January 1999.

[24] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In *Proceedings of Aspect-Oriented Software Development (AOSD 2005)*. ACM Press, March 2005. 12 pages; To appear.

[25] Q-H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. volume 29(3–4), pages 309–336. Kluwer Academic Publishers, 2002.

[26] P. A. Olivier. *A Framework for Debugging Heterogeneous Applications*. PhD thesis, Universiteit van Amsterdam, 2000.

[27] T. J. Parr and R. W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Software – Practice & Experience*, 7(25):789–810, 1995.

[28] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, February 2002.

[29] H. Tatsuzawa, H. Masuhara, and A. Yonezawa. Aspectual caml: an aspect-oriented functional language. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *Proceedings Foundations of Aspect-Oriented Languages Workshop held at AOSD 2005 (FOAL 2005)*, pages 39–49, March 2005.

[30] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[31] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[32] E. Van Wyk. Aspects as modular language extensions. In *Proc. of Language Descriptions, Tools and Applications (LDTA)*, volume 82.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003.

[33] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.

[34] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *ACM Symposium for Applied Computing (SAC) – Programming for Separation of Concerns Track*, Santa Fe NM, March 2005.