RULE 2005 Preliminary Version

Combining Aspect-Oriented and Strategic Programming

Karl Trygve Kalleberg [1]

Department of Informatics, University of Bergen, P.O. Box 7800, N-5020 BERGEN, Norway

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands

Eelco Visser [2]

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands

Abstract

Properties such as logging, persistence, debugging, tracing, distribution, performance monitoring and exception handling occur in most programming paradigms and are normally very difficult or even impossible to modularize with traditional modularization mechanisms because they are cross-cutting. Recently, aspect-oriented programming has enjoyed recognition as a practical solution for separating these concerns. In this paper we describe an extension to the Stratego term rewriting language for capturing such properties. We show our aspect language offers a concise, practical and adaptable solution for dealing with unanticipated algorithm extension for forward data-flow propagation and dynamic type checking of terms. We briefly discuss some of the challenges faced when designing and implementing an aspect extension for and in a rule-based term rewriting system.

Key words: aspect-oriented programming; language extension; rule-based programming; unanticipated extension; strategic programming

1 Introduction

Good modularization is a key issue in design, development and maintenance of software. We want to structure our software close to how we want to think

[1] Email: `karltk@ii.uib.no`
[2] Email: `visser@cs.uu.nl`

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

about it [21], by cleanly decomposing the properties of the problem domain into basic function units, or *components*. These can be mapped directly to language constructs such as data types and functions. Not all properties of a problem decompose easily into components, but rather turn out to be non-functional and frequently cross-cut our module structure. Such properties are called *aspects*. The goal of aspect-oriented software development [13] is the modularization of such cross-cutting concerns. By making aspects part of the programming language, we are left with greater flexibility in modularizing our software, as the cross-cutting properties need no longer be scattered across the components. Using aspects, these properties may now be declared entirely in separate units, one for each property. Examples of general aspects include security, logging, persistence, debugging, tracing, distribution, performance monitoring, exception handling, origin tracking and traceability. All these occur in the context of rule-based programming, in addition to some which are domain-specific, such as rewriting with layout. Existing literature predominantly discusses aspect-based solutions to these problems for object-oriented languages, and the documentation of paradigm-specific issues and deployed solutions for the rule-based languages is scarce.

In this paper we describe the design and use of aspects in the context of rule-based programming. We introduce the AspectStratego language for declaration of separate, cross-cutting concerns and discuss the joinpoint model on which it is built. We demonstrate its practical use and highlight some of its implementation details through three small case studies motivated by the problem of constant propagation. The contributions of this paper include:

(i) The description of an aspect language extension implemented for and in a rule-based programming language.

(ii) An example of its suitability for adding flexible dynamic type checking of terms in a precise and concise way.

(iii) A demonstration of its application to unanticipated algorithm extension by showing how it can help in generalizing a constant propagation strategy to a generic, adaptable forward propagation scheme, using invasive software composition [2].

We proceed as follows. In the next section, we describe the Stratego language for term rewriting, with examples. In Section 3, we introduce an extension to Stratego which allows separate declaration of cross-cutting concerns and show how this extension facilitates declarative code composition. In Section 4, we discuss three cases where the aspect extension is found to be highly useful: logging, type checking of terms and algorithm adaption. In Section 6, we discuss previous, related and future work.

2 The Fundamentals of Stratego

In order to discuss aspects for rule-based programming, we shall briefly introduce the Stratego language for term rewriting. Readers familiar with the language may want to skip ahead to Section 2.2.

2.1 The Stratego Language

Stratego is based on the concept of rewriting strategies; algorithms for transforming a term with respect to a set of rewrite rules. In most rewrite engines these strategies are fixed, and often require the set of rules to be confluent and terminating. In Stratego, the strategies are user-defined, thus providing the user with fine-grained control over the selection of rules and the order of their application. While a term may represent anything, the Stratego library and tool chain, together named Stratego/XT, is geared towards computer languages and program transformation.

2.1.1 Signatures and Terms

A *signature* describes the structure of a first-order term. It consists of a set of *constructors*, each taking zero or more arguments. A *term* over a signature S is a syntactic expression generated from its signature by application of the constructors. For example, `Assign(Var("x"),Plus(Int("1"), Var("x")))` is the term representation of `x := 1 + x`.

2.1.2 Match and Build

Stratego is built around the two primitive operators *match* and *build*. Build, written `!`, is used to construct a term from constructors and primitive sorts. Match, written `?`, is used to match a pattern against a term. A *pattern* is a term that might contain variables. If a variable in a term pattern is already bound to a term, then the variable in the pattern will be replaced with this value. If variables in the pattern are not yet bound, then these variables will be bound to the actual values in the current term to which the pattern is applied. Building and matching is always done against an implicit *current term*. Assuming the signature in Fig. 1 and that x is bound to `"0"`, y unbound, the following code fragment will first replace the current term with the `Plus` term, then match against the new current term and bind y to `"2"`:

```
   !Plus(Int(x), Int("2")) ; ?Plus(_, Int(y)))
```

The underline (_) here is a wildcard. It matches any term and ignores it.

2.1.3 Rewrite Rules

A *rewrite rule* describes the transformation from one term to another, and might be guarded by a condition. It has the form `R : l -> r`, where `R` is the rule name, `l` the left-hand side pattern and `r` the right-hand side pattern. The condition may be added in a `where`-clause, which contains a strategy expression, described next. The left-hand side pattern is matched against a term and if the match succeeds, the right-hand side pattern is instantiated to construct the new term. Multiple rules may have the same name, and rules are always invoked by name. When multiple rules with the same name exist, all are tried until one matches, and its result is returned. If multiple rules could match, only one will succeed. The language semantics does not specify which. In Fig. 1, `EvalBinOp` is an example of a rewrite rule with a condition. A rewrite rule with a dynamic (as opposed to lexical) scope is called a *dynamic rewrite rule*. The scoping is controlled entirely by the programmer by inserting special scope constructs into strategy expressions[20]. In Fig. 1, `rules( PropConst : ... )` is an example of the declaration of the dynamic rule `PropConst`, and `rules( PropConst :- ... )` shows its deletion.

2.1.4 Rewriting Strategies

A *rewriting strategy* is an algorithm for transforming a term. If it succeeds, the result is the transformed term. If it fails, there is no result. Strategies control the order of application of rules or other strategies. They can be combined into *strategy expressions* using a set of built-in *strategy combinators*, such as *sequential composition* (`;`) and *deterministic choice* (`<+`). The language has two primitive strategies: `id` will always succeed and return the identity term and `fail` will always fail. `prop-const-assign` in Fig. 1 is an example of a strategy definition.

2.1.5 Rules and Strategy Parameters

Strategy and rule definitions may contain two kinds of parameters, *higher-order strategy parameters* and *term parameters*. Higher-order strategy arguments work mostly like higher-order functions in functional languages. Term parameters are used to pass context information into rules or strategies without involving the current term. The call to `log` in Fig. 3 is an example of strategy invocation with only term arguments. The strategy arguments are separated from the term arguments by a `|`. Arguments before the `|` are taken to be strategy arguments, arguments after are taken to be term arguments. In the case of no term arguments, `|` is omitted.

2.1.6 Term Traversal

Term traversal strategies apply rewriting strategies throughout the structure of a term. The library provides a host of primitive strategies for this. To apply a strategy `s` to all immediate subterms of a term, one may write `all(s)`. Other examples of provided traversals are `bottomup` and `topdown`. An alternative mechanism for term traversal is by using *congruence operators*. For each n-ary constructor `C` there is a corresponding congruence operator defining a strategy with the same name and arity, $\mathtt{C}(s_1,\ldots,s_n)$. When applied to the term $\mathtt{C}(t_1,\ldots,t_n)$ strategy applies s_1 to t_1, s_2 to t_2, and so on, yielding the term $\mathtt{C}(t'_1,\ldots,t'_n)$, provided all s_i strategies succeed. If any s_i fails, so does

```
module prop-const
signature
 constructors
  Var    : Id -> Var
         : Var -> Exp
  Int    : String -> Exp
  Plus   : Exp * Exp -> Exp
  If     : Exp * Exp * Exp -> Exp
  While  : Exp * Exp -> Exp
  Assign : Var * Exp -> Exp
rules
  EvalBinOp : Plus(Int(i), Int(j)) -> Int(k)
              where <addS>(i,j) => k
  EvalIf    : If(Int("0"), e1, e2) -> e2
strategies
  prop-const =
    PropConst <+ prop-const-assign <+ prop-const-if
    <+ prop-const-while <+ (all(prop-const) ; try(EvalBinOp))
  prop-const-assign =
    Assign(?Var(x), prop-const => e)
    ; if <is-value> e then rules( PropConst : Var(x) -> e )
                      else rules( PropConst :- Var(x) ) end
  prop-const-if =
    If(prop-const, id, id)
    ; (EvalIf ; prop-const <+
       (If(id,prop-const,id) /PropConst\ If(id,id,prop-const)))
  prop-const-while =
    ?While(e1, e2)
    ; (While(prop-const,id)
       ; EvalWhile
         <+ (/PropConst\* While(prop-const, prop-const)))
```

Fig. 1. An excerpt of a Stratego program defining an intraprocedural conditional constant propagation transformation strategy for a small, imperative language.

the strategy `C`. `If(prop-const, id, id)` in the strategy `prop-const-if` in Fig. 1 is an example of a congruence for `If`.

2.1.7 Modularization

Modules are the coarsest elements of the Stratego program structure. A module is a file starting with the declaration `module` *`name`*. Following this declaration are *sections*, each with a *section header*, of which `signature`, `rules` and `strategies` are relevant to us. Constructor declarations must occur inside a `signature` section. Rules and strategies may be freely mixed within

```
x := 1 + 1 ;
if x then y := 1
     else y := 0 fi
```

⇒

```
x := 2 ; y := 1
```

Fig. 2. Example showing the constant propagation strategy.

the `rules` and `strategies` sections, but good form is to place the definition according to its type.

2.2 Constant propagation

The code in Fig. 1 shows a constant propagating strategy for an imperative language with assignment, `While` and `If` constructs. The principle of the constant propagation algorithm is straightforward: Whenever there is an assignment of a constant to a variable, we record this using a dynamic rewrite rule. If the variable is subsequently assigned a non-constant value, the rule is deleted. This is done in `prop-const-assign`. We use the dynamic rewrite rule to replace every constant variable with its value, if known. This opens up for elementary evaluation rules `EvalBinOp` and `EvalIf` to simplify expressions over `Plus` and `If`. An example of its application is given in Fig. 2.

The `prop-const` strategy is the top level driving strategy which takes care of recursively applying the constant propagation to a term. It works by calling the rule `PropConst`, which will replace any variable term for which we know the value with its constant. In fact, `PropConst` is a set of dynamic rules with the same name. Since `PropConst` is dynamic, rules may be added to and removed from the set as the traversal progresses. This is done in the strategy `prop-const-assign`. As we can see in `prop-const`, `prop-const-assign` is invoked on any term where `PropConst` fails, i.e. whenever we cannot replace a variable with its constant. If, when the `prop-const-assign` is invoked, we are on an `Assign` term and it is assigned a constant value, we generate a new rule with the name `PropConst`, thus adding it to the `PropConst` rule set. The freshly generated rule is a lookup from a variable (a `Var`) to a constant value. When applied, the new `PropConst` rule will rewrite an occurrence of that variable to its associated value.

Refer back to the `prop-const` strategy. If the current term is not a `Var` we know the constant value for, nor an `Assign` we can pick out a constant value form, `prop-const` will try the other strategies, for `If` (`prop-const-if`) and `While` (`prop-const-while`). If all fail, `prop-const` will fall back to applying itself recursively to all subterms of the current term, then try the `EvalBinOp` rule on the result.

The `prop-const-if` strategy will match an `If` construct using a congruence operator, while at the same time applying `prop-const` to the condition expression. If the congruence succeeds, the `prop-const-if` strategy proceeds by either (1) simplifying the `If` using the `EvalIf` rule and then recursively continuing the `prop-const` algorithm on the result, or (2) applying `prop-const` recursively to the then-branch and else-branch in turn, and keeping only `PropConst` rules which are valid after both branches, i.e. those that are defined and equal in both branches.

The dynamic rule intersection operator `s1 /PropConst\ s2` applies both strategies `s1` then `s2` to the current term in sequence, while distributing (clones of) the same rule set for the dynamic rule `PropConst` to both strategies. Afterwards, only those rules which are consistent in both branches are kept. A similar explanation holds for `prop-const-while`, where the fixpoint operator `/PropConst\* s` is used instead. This operator will apply `s` repeatedly until a stable rule set is obtained. Each iteration will apply `s` to the *original* term, and the result of the final iteration is kept as the new term.

2.2.1 Generalization and Adaptation

As written, the algorithm already has some variation points where the user can extend it using the language features we have presented, without modifying the algorithm itself. For example adding another evaluation rule for `EvalIf` that deals with non-zero constants. But there are also other extensions and adaptations we may want to apply to this algorithm. Section 4.1 shows how we can extend it with logging capabilities to record all rule invocations, and in Section 4.2, we show to extend it with dynamic type checking of terms to ensure the result is a correct term. Finally, in Section 4.3, we show how the algorithm can be refactored into a more generalized schema for forward propagating data-flow transformations. All extensions and adaptations are performed with the help of our aspect extension to the Stratego language, described next.

3 AspectStratego

AspectStratego is an extension to the Stratego language which addresses the problem of declaring cross-cutting concerns in a modular way. The language extension bears some resemblance to the AspectJ language [12]. We have tried to keep much of the terminology, as well as some properties of its joinpoint model, though the latter has been adapted to fit better within the paradigm of rule-based rewriting systems. Similar to AspectJ, we provide the programmer with expressions called pointcuts (ours are boolean involving predicates instead of set theoretic) on the program structure used to pick out well-defined points in the program execution, called joinpoints. Pointcuts are used in advice to pinpoint places to insert code before, after or around. The inserted code is declared as part of the advice. Advice are in turn gathered in named entities called aspects. The act of composing a program with its aspects is called weaving.

Fig. 3 shows how we may use an aspect to extend the constant propagator with trivial logging. In the following section we will give a more advanced example of logging. Here, we will cover the new language features introduced

```
module prop-const-logger
imports logging prop-const
aspects
 pointcut call = strategies(prop-const)
 aspect prop-const-logger =
  before : call = log(|Debug, "Invoking constant propagator")
```

Fig. 3. Aspect extending the constant propagation module with logging. The `log` is from `logging` module, part of the Stratego library.

in this example.

3.1 Joinpoints

A *joinpoint* is a well-defined point in the program execution through which the control flow passes twice: once on the way into the sub-computation identified by the joinpoint, and once on the way out. The purpose of the aspect language is allowing the programmer to precisely and succinctly identify and manipulate joinpoints.

3.2 Pointcuts

A *pointcut* is a boolean expression over a fixed set of predicates and the operators `;` (*and*), `+` (*or*) and `not`, and is used to specify a set of joinpoints. There are two kinds of predicates in a pointcut, joinpoint predicates and joinpoint context predicates [3]: A *joinpoint predicate* is a pattern on the Stratego program structure used to pick out a set of joinpoints. A *joinpoint context predicate* is a predicate on the runtime environment which can be used in a pointcut to restrict the set of joinpoints matched by a joinpoint predicate. Fig. 4 lists the supported joinpoint and joinpoint context predicates. A *pointcut declaration* is a named and optionally parameterized pointcut, intended to allow easy sharing of identical pointcuts between advice. The parameters are used to expose details about the pointcut to the advice. The declaration `pointcut call = strategies(prop-const)` from Fig. 3 shows a parameterless pointcut named `call` with the joinpoint predicate `strategies` and no joinpoint context predicates. It picks out all definitions of strategies named `prop-const`.

3.3 Advice

An *advice* is a body of code associated with a pointcut. There are three main kinds of advice, *before*, *after* and *around*, which specify where the body of code should be placed relative to the joinpoint matched by its pointcut. Fig. 4 lists

[3] This terminology and implementation differs from the AspectJ language which provides *primitive pointcut designators* instead, see [12].

Joinpoint	*Matches*
`calls(`*name-expr* `=>` *n*`)`	strategy or rule invocations
`strategies(`*name-expr* `=>` *n*`)`	strategy executions
`rules(`*name-expr* `=>` *n*`)`	rule executions
`matches(`*pattern* `=>` *t*`)`	pattern matches
`builds(`*pattern* `=>` *t*`)`	term constructions
`fails`	explicit invocations of fail
Joinpoint context	*Matches*
`withincode(`*name-expr* `=>` *n*`)`	joinpoints within a strategy or rule
`matchflow(`*flow-regex* `=>` *t*`)`	joinpoints with given execution history
`args(`n_0`,`n_1`,...,`n_n`)`	joinpoints with given arity
`lhs(`*pattern* `=>` *t*`)`	rule left-hand sides
`rhs(`*pattern* `=>` *t*`)`	rule right-hand sides
Advice	*Action on pointcut*
`before`	run before
`after`	run after
`after fail`	run after, iff code in pointcut failed
`after succeed`	run after, iff code in pointcut succeeded
`around`	wrapped around

Fig. 4. Synopsis of the AspectStratego joinpoint, joinpoint context predicates and advice variants. The *name-expr* can either be a complete identifier name, such as `EvalBinOp` or a prefix, such as `prop-*`. The result of a *name-expr* is a string, and may optionally be assigned to a variable using the `=>` x syntax. The *pattern* is an ordinary Stratego pattern, which may contain both variables and wildcards. The *flow-regex* is a regular expression on the execution stack and may be considered as an extended, dynamic version of `withincode`. It will not be discussed further in this article.

the available advice types for AspectStratego. The code `before : call = log(|Debug, "Invoking const....")` in Fig. 3 is an example of a before advice. The strategy `log` is provided by the library, and will be discussed later. This code will be weaved into the `prop-const` strategy in Fig. 1 as follows:

```
  prop-const = log(|Debug, "Invoking const....")
  ; (PropConst <+ prop-const-assign <+ prop-const-if
  <+ prop-const-while <+ (all(prop-const) ; try(EvalBinOp)))
```

Composing code by inserting advice like this opens up the possibility for manipulating the current term. Exactly how the strategy or rule invocations inside the advice body changes the current term can be controlled in two ways: the advice body is a strategy expression and may be wrapped (entirely or partially) in a `where` to control how and if the current term is modified. In the above case, `log` only takes term arguments and is designed to leave the current term untouched, making `where` superfluous.

This manipulation of the current term turns out to be extremely useful in `around` advice, where the implementer of the advice has full control over how the pointcut should be executed. The placeholder strategy `proceed` is available for this purpose. By placing the `proceed` within a `try` or as part of a choice (+), it is trivially possible to add failure handling policies. The flexibility of `around` allows the aspect programmer to completely override and replace the implementation of existing strategies and rules, by not invoking `proceed` at all. This can even be applied to strategies found in the Stratego standard library.

The usefulness of current term manipulation stems from the fact that terms are normally passed as the current term from one strategy to another, not as term arguments. E.g. in the following example, `strat2` will be applied to the current term left behind by `strat1`:

```
strat1 ; strat2
```

An alternative, more imperative formulation of the same would be:

```
strat1 => r ; strat2(|r)
```

but this is not within the style of Stratego, as it becomes cumbersome to use when we replace `;` with the other strategy combinators, such as `<+`. Current term manipulation is thus mostly analogous to manipulating input parameters and return values in AspectJ.

Wrapping a rule (or strategy) with an advice subsumes overloading a rule (or strategy) in the following ways:

- *controlling order*; when using overloading to extend a set of rules with the same name with new cases, we cannot control the order of application of our extension. If this is required, the rules must be renamed and hidden behind a strategy with the old name, which expresses their priority.
- *controlling context*; by extending a rule using overloading, the extension will always be available, program-wide. Using aspects, we may control the context where its extension should be available, say by use of `calls` and `withincode`.

3.4 Weaving

The pointcuts are designed to be evaluated entirely at compile-time, `matchflow` notwithstanding. Given an advice declaration, the compiler will interpret its pointcut declaration on the Stratego abstract syntax tree (AST) to find the location where to weave the advice body. The code in the advice body is then inserted into the AST before, after or around the joinpoint.

The body of the advice has a rudimentary reflective capability, which is also resolved at compile-time. From Fig. 4, we see that the advice body has access to rule and strategy names. The Stratego runtime has no reflective nor code-generating capabilities, so these names are mostly useful for logging purposes. Advice body code also has access to patterns from match expressions, and may evaluate these patterns at runtime. We will show how this is useful in Section 4.2.

3.5 Modularization

As with all Stratego code, aspects must reside in modules. We think this sensible, as aspects are about modularizing cross-cutting concerns. An `aspect` or `pointcut` can only be declared within an `aspects` section. While `aspects` sections may be interleaved with the other Stratego sections (`strategies`, `rules`, `signature`), we encourage each aspect to be declared in its separate module. Firstly, this helps keep aspects — separate, cross-cutting concerns – truly separate, both in design and implementation. Secondly, this also allows them to be selectively enabled or disabled using compiler flags without any code modification at all.

AspectStratego keeps the pointcuts declarations outside the aspects declarations. This allows pointcuts to be shared between aspects. In object-oriented renditions of aspects, such as AspectJ, sharing of pointcuts between aspects are captured using inheritance: a subaspect inherits all pointcuts from its superaspect.

We show how shared pointcuts are useful in Section 4.3. The mechanisms and language features required for controlling the application of aspects on the module level are still subject to research.

4 Case Studies

We describe the use of AspectStratego for three case studies relevant to rule-based programming. The first is a simple logging aspect, which is included to show similarities and differences with the AspectJ language. The second is a dynamic type checker of terms realized entirely as an aspect, and shows how aspects may sometimes be used as an alternative to compiler extensions. The final case is a discussion of how aspects can be useful in expressing variation points when implementing generalized, adaptable algorithms.

```
module simple-logger
strategies
 invoked(|s) = ![ "Rule '", s, "' invoked" ]
aspects
 pointcut log-rules(n) = rules(* => n)
 aspect simple-logger =
  before       : log-rules(r) = log(|Debug, <invoked(|r)>)
  after fail   : log-rules(r) = log(|Debug, <failed(|r)>)
  after succeed: log-rules(r) = log(|Debug, <succeeded(|r)>)
  after        : log-rules(r) = log(|Debug, <finished(|r)>)
```

Fig. 5. A complete logging aspect in AspectStratego. The definitions of `failed`, `succeeded` and `finished` are similar to `invoked`. The direction of information flow through the pointcut declaration arguments is somewhat uncommon: they specify information going *out* of the declaration.

4.1 Logging

Logging of program actions is often useful when developing software, and is therefore a problem we want to encode in a concise fashion. The program points we want to trace frequently follow the program structure, for instance the entry and exit of functions. In these cases the established solution is to wrap the function definitions in syntactical or lexical macros which do simple code composition. The numerous shortcomings of this technique, such as decreased code readability, lack of flexibility, interference with meta-tools (especially for documentation and refactoring) and typographic tedium are all addressed by aspects and their weaving. The aspect language also allows pervasive insertion of logging code in locations unanticipated by the original implementor, such as inside rule conditions and failures deep inside the Stratego library.

The code in Fig. 5 shows an aspect called `simple-logger` that may be used to insert logging code around all rules in a program by adding it to the `imports` list. If we were to import `simple-logger` into the module in Fig. 2, all executions of `EvalBinOp` and `EvalIf` would be logged. The weaver will *shadow* both declarations by prefixing them to obtain a new, unique identifier. It will then generate a wrapper strategy from the template in Fig. 7[4] The final result of this weaving for `EvalBinOp` is shown in Fig. 6. The wrapper will first perform the code from the before advice followed by the shadowed code. If the shadowed code fails, the `after fail` advice is run, followed by the `after` advice. The enclosing `try` and `if-then-else` are there to allow `after fail` and `after succeed` advice to change a failure into success or success into

[4] The actual implementation uses the Stratego guarded choice operator, which is a lot less readable, not `if-then-else`. `if-then-else` insulates its condition in a `where`, leaving the current term unchanged. In our case, the condition is the pointcut code and must be allowed to modify the current term.

```
EvalBinOp =
  log(|Debug, invoked("EvalBinOp")) ;
  if shadowed-EvalBinOp then
    if log(|Debug, succeeded("EvalBinOp")) then
      try(log(|Debug, finished("EvalBinOp")))
      else
      log(|Debug, finished("EvalBinOp")) ; fail
    end
  else
     // identical to the then-clause,
     // with succeeded replaced by failed
  end
```

Fig. 6. The declaration of `EvalBinOp` from Fig. 1 after weaving in the `simple-logger` aspect.

```
before ;
if pointcut-code then
  if after-succeed then try(after) else after ; fail end
  else
  if after-fail then try(after) else after ; fail end
end
```

Fig. 7. Template for advice weaving. Cursive identifiers are insertion sites for advice code. If a particular advice is not present in a joinpoint, it is replaced by an `id` (`after-fail` is replaced by `fail`).

failure, respectively. `after` advice may not change failure/success but may replace the current term.

While the built-in `log` strategy provides the ability to set the logging level at runtime (e.g. only errors, and no warning and debug messages), a program with explicit `log` calls inserted into its strategy and rule definitions will always take a slight performance hit. Stratego, where the coding style encourages many and small rules and strategies, is sensitive to any such overhead even with aggressive inlining. Consequently, it is desirable to have the ability to easily remove most or all `log` calls before final deployment.

The application of an aspect may open up for further adaptation, again using aspect weaving. For example, the strategy `invoked` in Fig. 5 may be the target for further aspects. Note that these second level — or "meta" — aspects pose a few potential problems with respect to weaving order that have not been solved in our implementation yet. In our current implementation, aspects are weaved in the order of declaration. Consider the following definition of `ext-invoked`:

```
aspects
 pointcut invoked = calls(invoked)
```

```
aspect ext-invoked =
 before : invoked = ...
```

If this aspect were to be weaved before `simple-logger`, it would have no effect, as `invoked` is not called anywhere at the time `ext-invoked` is weaved. As long as the user is aware of this, and manually linearizes the dependency chain between aspects by declaring `ext-invoked` after `simple-logger`, the result will match the intention of the user.

4.2 Type checking

Terms in Stratego are built with constructors from a signature, but the language does not enforce a typing discipline on the terms. With the signature in Fig. 1, the program may construct an invalid term, e.g. `!Plus(Int("0"), "0")`. As the normal mode of operation for Stratego is local and piecewise rewriting of terms, possibly from one signature to another, invalid intermediates cannot be forbidden. To debug such a problem, it is common to manually insert debug printing, or weave in a logger to generate a program trace for manual inspection, but manual verification is highly error-prone.

The Stratego/XT environment comes with format checking tools for this purpose. The tools can be applied to the resulting term of a Stratego program, checking the resulting term against a given signature. While all signature violations will be caught by these tools, they cannot help in telling where in your program the actual problem is present, as the check happens entirely after program execution. Using aspects, we can weave the format checker into the rules of our program at precisely the spots we would like the structural invariants dictated by the signature to hold. The `typechecker` aspect in Fig 8 makes use of the format checker functionality in Stratego/XT to pervasively weave format checking into all rules in a Stratego program. By modifying the `typecheck-rules` pointcut, the user can control the exact application of the type checker. Its usage is similar to the `simple-logger`: it must be imported, and a `typecheck` strategy for the relevant signature must be declared in a `strategies` section:

```
typecheck(|t) = format-check-Imp(|t)
```

The signature in Fig.1 is an excerpt of the signature for a small imperative language called `Imp`. Given a language grammar, the Stratego/XT format checker tools will generate a Stratego module containing a complete format checker for that grammar. The top level strategy for this format checker is named `format-check-<language-name>`. It may be applied to a term, to see if it is a valid (sub)term of that language.

As with logging, introducing this aspect provides the user with a quick and concise mechanism to decide which parts (if any) of a program should be type checked, and its usage can be toggled both at compile- and runtime (the latter will always incur a small performance hit, as previously discussed).

The argument *t* to `typecheck` is the pattern matched by the `typecheck-rules`

```
module typecheck-example
aspects
 pointcut typecheck-rules(n , t ) = rules(n ) ; rhs(t )
 aspect typechecker =
  around(n , t ) : typecheck-rules(n , t ) =
  proceed ; (typecheck(|t )
    <+ ( log(|incorrect-term(n ) ; fail ))
```

Fig. 8. An aspect for weaving simple dynamic type of terms into rules.

pointcut, and is therefore the right-hand side pattern of a rule. In the case that t is a term (no variables), it can in theory be entirely checked at compile time as both the signature and the term are completely known to the compiler. In the case that t contains variables, the static parts may be checked at compile time, but the variable part must be evaluated at runtime. If we make the assumption that only rules can construct terms, and only in their right-hand side pattern, we can further optimize the matching by only checking that the top term of a variable is a valid subterm of its enclosing "static" term. Currently, the aspect compiler does not perform such optimizations, but relies on the match optimizer in the back end of the Stratego compiler.

The type checking aspect is only a partial replacement for a built-in type system, particularly because it does no type inferencing, and can therefore not eliminate redundant checks. The topic of typed, strategic term rewriting is discussed in [16].

4.3 Extending algorithms

The algorithm in Fig. 1 is an instance of the more general data-flow problem of forward propagation, examples of which are common subexpression elimination, copy propagation, unreachable code elimination and bound variable renaming. The algorithm can be factored into a language-specific skeleton and problem-specific extensions. The skeleton needs to be implemented once for each language, as it is dependent only on the language constructs and scoping rules. A *variation point* is a concrete point in a program where variants of an entity may be inserted. By providing clearly defined variation points, the skeleton is made adaptable to the specific propagation problem at hand.

4.3.1 Expressing Adaptable Algorithms

There are many well-known techniques for expressing adaptable algorithms. When providing an algorithm which is supposed to be reused and adapted by other programmers (users), we are after techniques which offer:

- *adaptability*; we would like maximal freedom in which variation points we may expose to our users.
- *reuse*; the users of our algorithms should need to reimplement as little code

as possible. This is especially important in the face of maintenance.

- *traceability*; when errors (either design or implementation) are discovered in our algorithm, we want to offer users an easy upgrade path. Ideally, they should only need to replace the library file wherein our algorithm resides. This may not always be feasible, but at the very least, we want our users to know which parts of his system may be affected by the error.
- *evolution*; we must be able to change the internals of our algorithm without disturbing our users.

Boilerplates

One of the most popular, but least desirable techniques for adaptation is *boilerplate* adaptation. In this approach, a code template is manually copied then modified to fit the situation at hand. The approach suffers from high maintenance costs due to inherent code duplication. It is especially problematic if the original template is later found to contain grave (security) errors, as there is no traceability of where it has been used. On the other hand, it offers very high flexibility as all variation points may be reached. At its most extreme, boilerplate adaptation allows the applicant to gradually replace the entire algorithm.

Design Patterns

Another, popular technique for reuse are *design patterns* [6]. A design pattern is a piece of reusable engineering knowledge. For every case where a design pattern is applicable, it must be implemented from scratch by the programmer. In the recent years, much research has been into improving reuse of design patterns, either by providing direct language support [4,9] or by placing them in reusable libraries [1,8].

Hooks and Callbacks

Hooks and *callbacks* are well-known techniques for exposing variation points through *overridable* stubs the user of a library or algorithm can extend. By calling registration functions, the user may add callbacks and hooks which are called at pre-determined locations in the algorithm, or upon particular events in the program. As long as the contract between the algorithm and its callbacks is maintained, the algorithm internals may evolve separately from the adapted hooks, and therefore offers good maintenance properties. Its drawbacks include the fact that not all variation points may be expressed as hooks and that it is difficult to adapt an algorithm with different sets of hooks in multiple contexts within the same program. In Stratego, this can to some degree be solved using scoped dynamic rules. For other paradigms, function pointers, closures and/or objects allow multiple contexts to exist.

Higher-order Parameters

In functional languages, it is common to expose variation points through *higher-order parameters*. The paper [20] describes an adaptable skeleton for forward propagation using this approach. The technique provides a precise way for exposing variation points which is both easy to use and allows the user to adapt the algorithm on a per-context basis within the same program. One drawback is the issue of "parameter plethora": the number of parameters we want our users to deal with. In cases where the problem space is large, the algorithm will often have many variation points, yielding a long parameter list. A common solution to this problem is providing multiple entry points into the algorithm, each with an increasing number of parameters, or having parameters with default values, where the language supports this.

4.3.2 Limitations

Boilerplates and design patterns are not really desirable, given its poor support for code reuse and traceability. While the last two solutions offer both good reuse and traceability, they suffer from a few additional drawbacks. Over time, experience with the use of an algorithm may expose a need to extend it with further variation points, unanticipated by the original implementer. Exposing a new variation point frequently results in a change in the algorithm interface, either by adding new higher-order parameters, hooks or even new parameter to the existing hooks. Backwards compatibility can normally be handled by writing wrappers mimicking the old interface which forwards to the new, at the cost of maintaining multiple versions of the same interface.

Another consideration when extending an algorithm is how to propagate the new variation point through its internals. Suppose in `prop-const` in Fig. 1, we wanted to add the ability to transform the current term before recursively descending into the children. With a solution based on higher-order parameters, this transform parameter would have to be "threaded" through all `prop-*` strategies as a higher-order parameter, and thus result in an intrusive rewrite.

Yet another consideration is who should be able to perform adaptation and extension of existing algorithms. It is normally not possible for the user of the algorithm to extend it outside the exposed variation points, even if they can be clearly identified, unless the user has access to the source code, in which case the boilerplate technique may be resorted to.

4.3.3 Dealing with Evolution

We demonstrate a solution to the extensibility problem for handling *unanticipated* variation points that is complementary to hooks and higher-order parameters. It uses the declarative features of aspects to clearly identify and name the variation points in the algorithm. The code in Fig. 9 shows how some of the variation points already discussed have been exposed through pointcuts. Since we are the algorithm providers, we decided to add some

```
module forward-prop
strategies
 forward-prop =
   fail <+ prop-assign <+ prop-if <+ prop-while
     <+ all(forward-prop)
 prop-assign =
   Assign(?Var(x), forward-prop => e) ; id
 prop-if =
   If(forward-prop, id, id)
   ; (If(id,forward-prop,id) /\ If(id,id,forward-prop))
 prop-while =
   ?While(e1, e2)
   ; (/\* While(forward-prop, forward-prop)))
aspects
 pointcut prop-rule(r) =
   (calls(dr-fork-and-intersect) ; args(_, _, r))
   + (calls(dr-fix-and-intersect) ; args(_, r))
 pointcut prop-rule    = fails ; withincode(forward-prop)
 pointcut forward-prop = calls(all) ; withincode(forward-prop)
 pointcut prop-assign  = calls(id) ; withincode(prop-assign)
 pointcut prop-if =
   calls(dr-fork-and-intersect) ; withincode(prop-if)
 pointcut prop-while =
   calls(dr-fix-and-intersect) ; withincode(prop-while)
```

Fig. 9. Skeleton for forward propagation with variation points exposed as pointcuts. For a real language, the skeleton is often quite large and often difficult to construct. `s1 /Rule\ s2` is syntactic sugar for `dr-fork-and-intersect(s1, s2 | [ "Rule" ])`, and `/Rule\*` is sugar for `dr-fix-and-intersect`. In the above code, the `Rule` will be filled in later by aspects, thus the empty fork (`/\`) and fix (`/\*`) in `prop-if` and `prop-while`, respectively.

trivial points, `fail` in `forward-prop` and `id` in `prop-assign`, to allow the pointcuts and advice to be expressed more clearly, but they are not strictly necessary: the same joinpoints can be identified and used with only slightly more complicated pointcuts and advice, even entirely by a user of the skeleton without involving its provider nor changing code.

The `forward-prop` pointcut may be used to insert the transformation code before and after the propagator visits subterms of a given term. The `prop-*` pointcuts may be used similarly for inserting code before and after recursive descent into subterms of their respective language constructs. The `prop-rule` pointcuts are used for declaring which dynamic rule(s) to use for intersections and during traversal. Note that the pointcuts have the same names as the strategies they match inside. This makes it very clear to the user where his advice is applied. Admittedly, this is also a potential source of confusion, as the same identifier may refer to either an aspect or a strategy/rule, depending on context. We decided to keep the pointcut namespace separate from the other namespaces in Stratego (one for rules and strategies, another for constructor names) because the namespaces in Stratego are global.

By using these variation points, the code in Fig. 10 shows how the skeleton may be instantiated with advice to obtain a constant propagator. After weaving, the result will be the exact algorithm presented in Fig. 1. `around` advice is used instead of `after` advice to properly parenthesize the expressions and control operator precedence. Take the weaving of the `around` advice for `prop-while` pointcut. The pointcut matches the joinpoint code `/\* While(forward-prop,forward-prop)`. By using the around advice, we entirely replace this expression with `(While(forward-prop,id); EvalWhile <+ proceed)` and then substitute `proceed` with the original joinpoint code, ending up with the same code as found in `prop-const-while` in Fig. 1, modulo the fact that the driving strategy is now named `forward-prop`.

4.3.4 Evaluation

We now evaluate our solution based on the criteria set out above:

Adaptability

Exposing variation points through pointcuts is more adaptable than higher-order parameters and hooks because it can be done without changing the algorithm itself. As long as the variation point can be picked out using a pointcut, we may use an advice to insert a callback into the algorithm at that point. This is easier with AspectStratego than many other aspect extensions, as the data normally is passed through the algorithm as the *current term*. By using a pointcut, we may modify the current term before or after any strategy or rule invocation in the algorithm implementation. We can view the aspects as a complementary extension mechanism to callbacks/hooks since it may be used to add these. Similarly, the aspect technique is complementary to higher-order parameters. We may wrap the entry point to the algorithm in a reparametrized strategy.

We may expose different levels of adaptability using aspects, separate from the algorithm skeleton. By choosing between the available adaptation aspects, the user may choose which sets of variation points he intends to deal with. Assuming white-box reuse, the user himself may add new variation points to the algorithm in this fashion.

Reuse

Compared to design patterns and boilerplates, we get much better reuse. With a properly designed skeleton, the amount of code needed to adapt the algorthm is proportional to the extra functionality added.

```
module forward-prop-usage
imports forward-prop
aspects
 aspect prop-const =
  around : prop-rule(r) = proceed([ "PropConst" ])
  around : prop-rule = PropConst
  around : forward-prop = (proceed ; try(EvalBinOp))
  before : prop-assign-ext =
    ?Assign(Var(x), e)
    ; if <is-value> e
       then rules(PropConst: Var(x) -> e)
       else rules(PropConst:- Var(x)) end
  around : prop-if = EvalIf ; forward-prop <+ proceed
  around : prop-while =
    (While(forward-prop,id) ; EvalWhile <+ proceed)
```

Fig. 10. Instantiation of the forward-prop to make the constant propagator in Fig. 1, using aspects. Admittedly, the example is somewhat contrived, as these are variation points we normally would anticipate and explicitly parameterize easily.

Traceability

It is directly evident from the code both which version of the skeleton that has been used, and how it has been adapted (using which aspects). Traceability is therefore better than for boilerplates and patterns, and at the same level as parameters and callbacks.

Evolution

As time goes by, new callbacks and higher-order parameters may easily be added to the skeleton using aspects. Further, aspects may be used internally to propagate the parameters to all subparts of the algorithm implementation. Arguably, extra care must be taken to ensure that the semantics of the pointcuts are kept after an algorithm revision, since they now are declared separately. This problem is no different from variation points exposed through higher-order parameters or hooks as long as the pointcuts are known to the revising party.

In the case where users have identified and extended variation points through their own pointcuts, the situation is more precarious. This is a known drawback of white box reuse.

5 Implementation of the Weaver

The aspect weaver for AspectStratego is realized entirely inside the Stratego compiler as one additional step in the front end. It operates on the normalized AST where the module structure has been collapsed, so all definitions from all included modules are easily available for weaving. The weaver is implemented as traversals on the AST:

Pointcut collection and evaluation is a top down traversal that will pick out all pointcut declarations. All pointcuts encountered will be decomposed into conjunctive normal forms called *fragments*. A fragment contains one jointpoint and an arbitrary set of joinpoint context predicates all separated by logical *and*. For example, the pointcut `(rules(n) + strategies(n)) ; args(y)` is split into the two fragments `rules(n) ; args(y)` and `strategies(n) ; args(y)`. For each named pointcut, we generate a dynamic rule used as lookup from pointcut name to the fragment set for that pointcut.

Advice collection and evaluation is a top down traversal that will pick out all advice declarations. When an advice is encountered, its associated pointcut is looked up and we generate one dynamic rule for each fragment of that pointcut. In a generated rule, the left-hand side matches the term in the Stratego AST corresponding to the fragment's joinpoint predicate. For example, `rules` will match against the AST term for rule declaration, `RDefT`. The generated rule will evaluate all joinpoint context predicates in its condition. These rules will be applied later by the weaver. When a rule succeeds, it provides the weaver with the context information picked out by its pointcut fragment, and the associated advice body.

Weaving; The actual weaving is a bottom up traversal of the AST which exhaustively attempts to apply all generated rules from the previous step. On any term where one or more rules match, their associated advice bodies are collected and applied in place.

When evaluating the pointcuts in the aspect compiler, there is a need to do interpretation of the pointcut expressions. In the current implementation, this is realized as interpretive dynamic rules. Unfortunately, this leads to a rather rigid and tangled implementation where extending the language with new joinpoint (context) predicates becomes needlessly complex.

It is conceptually much more appealing to view each advice as a small meta program to be executed on the AST. This meta program must be constructed at compilation time, and can therefore not be a fixed part of the compiler. Our current implementation can be seen as a manual specialization of such a meta program, where the dynamic parts are captured by dynamic rules.

Instead of inventing and maintaining our own intepreter for such a meta program, it is desirable to generate a small Stratego program for every meta program. This would be possible in a rewriting language with an open compiler or a in a flexible, multi-staged language. The weaver would generate compiler extensions (meta programs), then execute these as part of the compilation process.

6 Discussion

There are several documented examples of cross-cutting concerns found in the domain of rule-based programming. The problem of origin tracking is documented in [24] and the problem of rewriting with layout in [23]. Both papers present interpreter extensions as the solution to their respective problems. In [14], it is argued that both the above cases are instances of the more general problem of propagating term annotations, a separate concern which should be adaptable by the programmer. The solution proposed is providing the programmer with declarative *progression methods* expressed as logic meta-programs. It is realized as a research prototype in Prolog.

Our aspect extension also provides a mechanism for adaptably specifying cross-cutting concerns in a declarative way, but our style is very similar to the popular AspectJ language, though recast for Stratego. The implementation is fully integrated with the Stratego/XT environment and is available for experimentation.

Many other aspect extensions have been documented. The AspectS system for Squeak dialect of Smalltalk [11] describes a weaver which works entirely at runtime, using the reflective features of the Smalltalk runtime environment. The Casear aspect extension for Java [18] brings runtime weaving to Java. In [17], the authors describe a small object-oriented language Jcore and its extension Cool for expressing coordination of threads. The two are composed using an aspect weaver. AspectC++ [22] is an aspect extension to the C++ language. An aspect extension for the functional language Caml is described in [10]. In [19], the authors document an aspect extension to the GAMMA transformation language for multiset rewriting and demonstrates its use to express timing constraints and distribution of data and processes. AspectStratego attempts to solve many of the same problems as the languages above, because the problems are cross-language, but we also motivate how problems specific to rule-based programming may have solutions based on aspects.

The implementation of aspect-weavers using rewriting has been documented in [3], for the context of graph rewriting and in [7] using term rewriting. In both cases, the subject languages were object-oriented. In [15], the authors detail an aspect language for declarative logic programs with formally described semantics, and a weaver based on functional meta-programs. We view reflective languages with meta programming facilities such as Maude [5] as alternative implementation vehicles for aspects. The appeal of aspects is their concise, declarative nature with their clearly defined goal: pick out joinpoints for inserting code. This contrasts with flexibility and complexity offered by general meta programming. The general-purposeness of meta programming may in fact often be a hindrance to users. We believe that distilling the power of general meta programming into a concise, declarative aspect language is useful.

While our paper also describes the implementation of an aspect weaver using a term-rewriting system, our subject language is not a declarative logic nor an object-oriented language, giving rise to a different set of joinpoints than considered by these papers.

The algorithm extension technique described in 4.3 is an example of compile-time code composition and is thus somewhat related to techniques such as templates in C++. But unlike C++ templates, our composition language is purely declarative and we can retroactively expose new variation points without reparameterizing.

Conclusion

We have presented an aspect extension for the Stratego term-rewriting language, combining the paradigms of aspect-oriented programming and strategic programming. We discussed its implementation and demonstrated a flexible dynamic type checker of terms as a practical example of aspects as an alternative to the interpreter extensions in [24], [23]. We also showed how aspects may be helpful in handling unanticipated algorithm extension through invasive software composition.

Acknowledgements: We thank Martin Bravenboer, Karina Olmos and Magne Haveraaen for providing insightful feedback, and the The Research Council of Norway (NFR) for funding. The article has benefited greatly from the detailed feedback provided by the anonymous reviewers.

References

[1] Agerbo, E. and A. Cornils, *How to preserve the benefits of design patterns*, in: *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1998), pp. 134–143.

[2] Assmann, U., "Invasive Software Composition," Springer-Verlag New York, Inc., 2003.

[3] Assmann, U. and A. Ludwig, *Aspect weaving with graph rewriting*, in: *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering* (2000), pp. 24–36.

[4] Bosch, J., *Design patterns as language constructs*, Journal of Object-Oriented Programming **11** (1998), pp. 18–32.
URL `citeseer.ist.psu.edu/bosch98design.html`

[5] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, *The Maude 2.0 system*, in: R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science (2003), pp. 76–87.

[6] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns," Addison-Wesley, 1995.

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

[8] Hannemann, J. and G. Kiczales, *Design pattern implementation in Java and AspectJ*, in: *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2002), pp. 161–173.

[9] Hedin, G., *Language support for design patterns using attribute extension*, in: *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology* (1998), pp. 137–140.

[10] Hideaki Tatsuzawa, A. Y., Hidehiko Masuhara, *Aspectual Caml: an aspect-oriented functional language*, in: C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL 2005 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2005*, Chicago, IL, 2005.

[11] Hirschfeld, R., *Aspects - aspect-oriented programming with Squeak*, in: *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World* (2003), pp. 216–232.

[12] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, Lecture Notes in Computer Science **2072** (2001), pp. 327–355.

[13] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science **1241**, Springer-Verlag, Berlin, Heidelberg, and New York, 1997 pp. 220–242.

[14] Kort, J. and R. Lämmel, *Parse-Tree Annotations Meet Re-Engineering Concerns*, in: *Proc. Source Code Analysis and Manipulation (SCAM'03)* (2003), 10 pages.

[15] Lämmel, R., *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*, 1999, pp. 131–146.

[16] Lämmel, R., *Typed Generic Traversal With Term Rewriting Strategies*, Journal of Logic and Algebraic Programming **54** (2003), also available as arXiv technical report cs.PL/0205018.

[17] Lopes, C. and G. Kiczales, *D: A language framework for distributed programming*, Technical Report SPL97-010, P9710047, Xerox Palo Alto Research Center (1997).

[18] Mezini, M. and K. Ostermann, *Conquering aspects with Caesar*, in: *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (2003), pp. 90–99.

[19] Mousavi, M., M. Reniers, T. Basten, M. Chaudron, G. Russello, A. Cursaro, S. Shukla, R. Gupta and D. C. Schmidt, *Using Aspect-GAMMA in the design of embedded systems*, in: *Proceedings of Seventh Annual IEEE International Workshop on High Level Design Validation and Test*, pp. 69–74.

[20] Olmos, K. and E. Visser, *Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules*, in: R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, Lecture Notes in Computer Science **3443** (2005), pp. 204–220.

[21] Parnas, D. L., *On the criteria to be used in decomposing systems into modules*, Commun. ACM **15** (1972), pp. 1053–1058.

[22] Spinczyk, O., A. Gal and W. Schröder-Preikschat, *AspectC++: an aspect-oriented extension to the C++ programming language*, in: *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific* (2002), pp. 53–60.

[23] van den Brand, M. and J. Vinju, *Rewriting with layout*, in: C. Kirchner and N. Dershowitz, editors, *Proceedings of RULE2000*, 2000.

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