# Strategic Programming Meets Adaptive Programming

Ralf Lämmel [1], Eelco Visser [2], and Joost Visser [3]

[1] Free University, Amsterdam, The Netherlands, ralf@cs.vu.nl
[2] University Utrecht, The Netherlands, visser@cs.uu.nl
[3] SIG, Amsterdam, The Netherlands, Joost.Visser@software-improvers.nl

## ABSTRACT

Strategic programming is a generic programming idiom for processing compound data such as terms or object structures. At the heart of the approach is the separation of two concerns: basic data-processing computations vs. traversal schemes. Actual traversals are composed by passing the former as arguments to the latter. Traversal schemes can be defined by the strategic programmer using a combinator style that relies on primitives for layered traversal.

In this paper, we take a look at strategic programming from an aspect-oriented programming perspective. Throughout the paper, we compare strategic programming with adaptive programming, which is a well-established aspectual approach to the traversal of object structures. We start from the observation that aspect-oriented programming terms, e.g., crosscutting, join point, and advice can be instantiated for aspectual traversal approaches.

## Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Strategic Programming, Adaptive Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

## General Terms

Languages, Design

## Keywords

Generic programming, Traversal, Strategic programming, Adaptive programming, Aspect-oriented programming, Strategy, Language design, Program transformation, Program analysis

## 1. INTRODUCTION

This paper is devoted to the advanced separation of two concerns in processing compound data such as many-sorted terms, object structures, XML documents, and others:

- *basic computations* for data-processing, and
- *traversal schemes* with rich variation points.

We will characterise a reference model for the separation of these concerns, namely the idiom of *strategic programming* (SP), which we developed over the last few years. We will compare SP with another aspectual approach to traversal, namely *adaptive programming* (AP) as developed by Lieberherr and collaborators [9, 18, 17, 11, 15]. We will work towards a marriage of the ideas underlying the two aspectual approaches to traversal.

*Adaptive programming (AP) at a glance.* Quoting [17]: "An adaptive program can be understood as an object-oriented program where the class graph is a parameter, and hence the class graph may be changed without changing the program. ... Adaptive programs consist of traversal specifications and code wrappers." Traversal specifications realise adaptiveness, say 'structure shyness' by only mentioning the milestone classes and relationships that are immediately relevant for the specific programming problem. The execution of an adaptive program applies the class-specific code wrappers to the objects that are identified by the adaptive traversal. AP employs predicates like the following to compose traversal specifications:

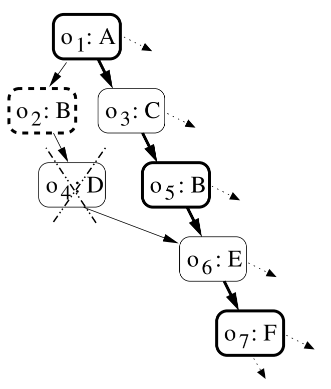
**from** Identify source nodes.

**to** Identify target nodes.

**through** Identify required intermediate nodes.

**bypassing** Identify disfavoured intermediate nodes.

In Fig. 1, we illustrate adaptive traversal.



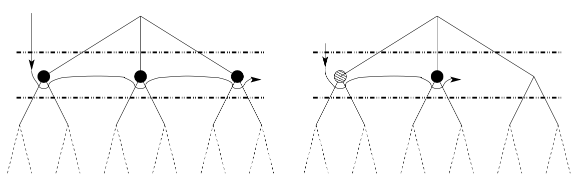
The heavy arrows build up a path through an object structure with objects $o_1, \ldots, o_7$. The classes in the object structure are denoted by $A, \ldots, F$. The shown path meets the adaptive traversal specification *from A through B to F bypassing D*. Notice how some objects on the path are not mentioned explicitly (cf. $o_3$ and $o_6$). The figure also indicates another path which meets all the requirements except for the *bypassing* predicate (cf. $o_4$).

**Figure 1: Adaptive traversal of an object structure**

*Strategic programming (SP) at a glance.* SP was initiated in the setting of term rewriting [22, 6], but has been transposed to other programming paradigms, most notably functional

**One-layer traversal**

Below we illustrate two forms of one-layer traversal: **all** to process all immediate subcomponents, and **one** to process the leftmost one for which the argument strategy succeeds. (Shaded vs. black nodes represent failure vs. success of processing.)



**Deep traversal**

Below we illustrate two recursive schemes. The left one corresponds to **all** applied at all levels. This results in a full top-down traversal (for short, *full_td*). The right one attempts **one** at all levels from bottom to top. This results in a single-hit bottom-up traversal (for short, *once_bu*).

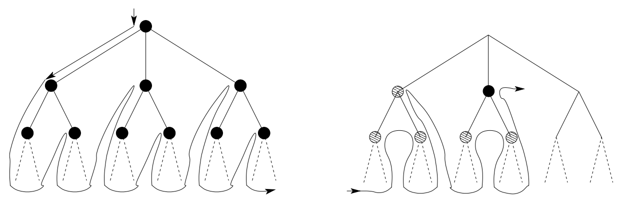


**Figure 2: Strategic traversal of a term or a tree**

programming [7, 5] and object-oriented programming [21]. The contribution of SP is to provide the programmer with full control in designing and implementing traversal functionality on the basis of programmer-definable traversal schemes. An actual traversal is synthesised by passing problem-specific basic computations as arguments to the appropriate traversal scheme. (The basic computations are like the code wrappers in AP.) The definition of schemes of traversal relies on traversal primitives that only process the immediate subcomponents of a datum, e.g.:

**all** Apply an argument strategy to all immediate subcomponents while preserving the overall shape of the datum.

**one** Apply an argument strategy to one 'fit' subcomponent while preserving the overall shape of the datum. Success and failure behaviour of the argument strategy determines fitness.

**reduce** Similar to `all` but the results of processing the immediate subcomponents are summed up with a given operator.

**select** Similar to `one` but the successfully processed immediate subcomponent is returned as the result.

By *not* anticipating any scheme of recursion, one-layer traversal can still be completed into deep traversal in different ways using ordinary recursion. In Fig. 2, we illustrate strategic traversal.

*Aspectual traversal.* Both adaptive and strategic programming are aspectual traversal approaches in the sense of aspect-oriented programming (AOP). The link between AP and AOP is discussed in [10]. The code wrappers in AP instantiate the AOP notion of *advice*. The traversal specifications in AP instantiate the AOP notion of *join points* in a somewhat unusual way. Instead of intercepting points along the execution of a given program, we first assume the execution of a traversal over an object structure. Then, traversal specifications are like join points or even point cuts along this traversal execution. This implies that AP addresses a form of *crosscutting* in the sense that a traditional implementation of a traversal results in scattering functionality and traversal control throughout several classes. This aspect-oriented view on AP carries over to SP, but SP addresses an additional form of separation of concerns. That is, one can capture reusable definitions of generic traversal schemes while exploring various variation points.

*Table of contents.* In Sec. 2, we will discuss the aspiration of SP in detail while we compare SP with AP and further related work. In Sec. 3, we will characterise the key notion used in SP, namely strategies, and we will provide a guideline suite of basic strategic combinators. In Sec. 4, we will present prime examples of SP-like traversal schemes, including approximative reconstructions of adaptive program patterns. In Sec. 5, we will provide an overview of SP incarnations in term rewriting, functional and object-oriented programming. In Sec. 6, we will pay special attention to the traversal of *object structures* since this is at the heart of AP. In Sec. 7, the paper is concluded.

## 2. THE CONTRIBUTION OF STRATEGIES

The aspiration of SP is to provide the programmer with *full traversal control.* This sets SP apart from all other approaches to traversal including AP. We will first explain the meaning of full traversal control as opposed to tangling of traversal actions and basic computations. We will then demonstrate the strategic style. We will also review other traversal approaches.

*Full traversal control.* We view a traversal as a program that performs basic data-processing actions on the appropriate data parts in the right order. *Data* is meant here in the sense of heterogeneously typed data such as many-sorted terms, object structures, and XML documents. *Control* can be classified as follows:

**i** the *order* of applying the basic actions,

**ii** the *side conditions* guarding the basic actions,

**iii** the propagation of *effects* caused by the actions, and

**iv** the *traversal* over the compound input data.

Control in the sense of (i) ordering, (ii) side conditions, and (iii) effects is reasonably understood. SP contributes to (iv) *traversal control*, and to its interaction with (i)–(iii). The resulting achievement is called *full traversal control.* The prime application domain of SP is program transformation, and indeed, full traversal control is crucial in this domain to guarantee correctness and termination of many transformations. The SP idiom is independent of a specific language or paradigm.

*Entangled traversal.* Traversals are often implemented in a way that the traversal logic and basic computations are entangled. This tangling can be observed in many areas of computing, for example, in object-oriented programming with visitors and functional programming. This is illustrated with a Haskell program in Fig. 3. The fact that the traversal logic is heavily entangled with the basic actions is a major problem because the size of the entangled traversal code is proportional to the number of data constructors regardless of the specific problem. Also, such entangled traversal is not robust in the view of changes to the traversed data structure. Furthermore, the tangling has to be repeated for every new piece of traversal functionality.

*The Haskell function implements a simple transformation by a traversal. It descends into terms representing regular expressions to perform simplifications according to the first two equations. The remaining equations serve traversal only.*

```
simplify :: RegExp → RegExp
simplify (Alt Epsilon exp) = Opt (simplify exp)      -- introduce '?'
simplify (Opt (Plus exp))  = Star (simplify exp)     -- introduce '*'
  -- recurse into compound regular expressions
simplify (Star exp)        = Star (simplify exp)
simplify (Plus exp)        = Plus (simplify exp)
simplify (Opt exp)         = Opt (simplify exp)
simplify (Seq exp1 exp2)   = Seq (simplify exp1) (simplify exp2)
simplify (Alt exp1 exp2)   = Alt (simplify exp1) (simplify exp2)
simplify exp               = exp
```

**Figure 3: Tangled traversal in functional programming**

*The contribution of AP.* Both AP and SP improve on the above tangling problem. With AP, the programmer can separate the traversal code (i.e., traversal specifications) from the basic computations (i.e., code wrappers). One also gains *adaptiveness*, that is, a traversal is only centred around the specific milestones of a traversal over a certain object graph.

*The contribution of SP.* Adaptiveness is also served by SP as we will demonstrate below. In addition, SP allows the programmer to capture reusable, potentially generic traversal schemes. Hence, control patterns for traversal become programmer-definable abstractions. As a result, both traversal schemes and problem-specific computations can be reused across applications and their components.

*The strategic method.* The SP idiom encompasses both *expressiveness* and *a method* for designing and implementing traversal functionality. The 'strategic' expressiveness is that traversal strategies can be defined in terms of appropriate strategy combinators (cf. Sec. 3). This expressiveness is sometimes hard, sometimes easy to achieve — depending on the targeted programming paradigm, the required strength of typing and programming convenience. The strategic method can be summarised in the following steps for implementing a piece of strategic traversal functionality:

1. identification of a reusable traversal scheme,
2. definition of the problem-specific ingredients, and
3. synthesis of the traversal by parameter passing.

The traversal schemes are usually generic, that is, problem-specific ingredients are anticipated via parameters. These problem-specific ingredients are type-specific actions or generic actions with type-specific branches. These actions are meant to describe how data of 'interesting' types is processed when encountered during traversal. The strategic method, although general, is in no way difficult.

*A strategic example.* The tangling in Fig. 3 is easily eliminated if we use a strategic traversal scheme to iterate the basic simplification rules all over the tree. The disentangled version is shown in Fig. 4. The function *simplify* is reconstructed in strategic style by passing a helper function *simplifyStep* to the traversal scheme *full_td* — read as 'full top-down'. To be precise, *simplifyStep* is wrapped with choice ... id to make sure that node processing always succeeds. The function *simplifyStep* captures the basic 'rewrite steps' for simplifying regular expressions. In the type of *simplifyStep*, we use the *Maybe* type constructor in order to express whether any simplification rule triggers or not.

```
simplify  =  full_td (choice simplifyStep id)

simplifyStep :: RegExp → Maybe RegExp
simplifyStep (Alt Epsilon exp)    =  Just (Opt exp)
simplifyStep (Opt (Plus exp))     =  Just (Star exp)
simplifyStep _                    =  Nothing
```

**Figure 4: Aspectual variation on Fig. 3**

*A variation.* Although the above definition of *simplify* is perfectly modular, and faithfully reconstructs the original tangled definition, a shortcoming becomes obvious. The definition does not enforce the exhaustive application of *simplifyStep*. This is because *full_td* applies its argument to the input datum before its immediate subcomponents were traversed. Here is a variation *simplify′* that eliminates this problem because it performs simplifications according to the folklore traversal scheme *innermost*:

$$simplify' \;=\; innermost\ simplifyStep$$

The scheme *innermost* operates bottom-up, and it loops until a fixpoint is reached. Hence, in this example, *innermost* is more appropriate than *full_td*. In another context, the opposite situation is possible, for example, if the use of *innermost* would cause a nonterminating strategy due to the nature of the given rewrite step. The schemes *full_td* and *innermost* are two beginner's favourites.

*Rich variation points.* SP enables and encourages the programmer to reflect on the variation points of traversals for each new problem. This makes it even easier to alter the design of a traversal when compared to the mere achievement of concise traversal implementations. These are typical *variation points* for traversals:

- transformation vs. query,
- single vs. cascaded traversal,
- top-down vs. bottom-up traversal,
- depth-first vs. breadth-first traversal,
- left-to-right traversal and vice versa,
- full vs. single-hit vs. cut-off traversal,
- types vs. general predicates as milestones,
- fixpoint by equality test vs. fixpoint by failure,
- local choice vs. full backtracking vs. explicit cut,
- traversal with effects (accumulation, cloning, etc.).

We have experienced these and other variation points in actual applications. AP and other generic programming idioms do not address this rich variety of variation points.

*Applications.* References to a few typical applications of SP are in place. In [1], a transformation system *CodeBoost* for *domain-specific optimisation* of C++ programs in the domain of numeric programming is described. It was implemented in *Stratego* making use of the *XT* bundle of tools for program transformation which includes packages for parsing and pretty printing. In [7], the use of functional strategies for the implementation of program refactoring for Java is demonstrated. The refactorings were implemented in Haskell using *Strafunski*. In [3], the *program understanding* tool *ControlCruiser* is described which reconstructs and visualises Cobol control flow. The *JJTraveler / JJForester* architecture has been used for the implementation.

*Levels of traversal control.* We conclude our discussion of traversal control by placing different styles of programming in a range of levels to measure the sophistication of traversal control:

**Entangled traversal** The folklore style of visitor programming or (non-generic) functional programming is placed at this level.

**Disentangled traversal** Approaches that separate traversal logic and the basic computations qualify for this level. The orthogonal example is the notion of generalised folds in functional programming [14], which provides a uniform traversal scheme for all datatypes. The basic computations per constructor can be passed as arguments to the fold combinator. An object-oriented approach to disentangled traversal is described in [16] (inspired by AP concepts). There, OOP is enriched with a domain-specific language for specifying reusable traversals of object structures.

**Adaptive traversal** If a traversal approach provides some means to abstract from the traversed data structure, then we call this an adaptive approach. This is obviously the case for AP. Polytypic programming [4] can also be used in a way to perform adaptive traversal. The above-mentioned notion of generalised folds can be refined to serve adaptiveness by considering generic, primitive fold algebras that only need to be updated for the data constructors relevant for a specific traversal problem [8]. Yet another adaptive approach is term rewriting with traversal functions [2].

**First-class traversal** SP inhabits this level. Traversal schemes are programmer-definable entities. Often, these schemes are completely generic.

*Absent variation points in AP.* Generic traversal schemes that make available various variation points are beyond the aspiration of AP. In fact, the semantics of adaptive programs fixes certain variation points of traversals, e.g.:

- Adaptive traversals are *depth-first* traversals.
- Milestones are searched in *top-down* manner.
- Milestones are constrained by *classes*.
- Traversal specifications denote *all* valid paths.

## 3. THE FOUNDATIONS OF STRATEGIES

Strategic programming is programming with the use of (traversal) strategies. Depending on the SP incarnation within a certain programming paradigm, strategies might correspond to objects, pure functions, impure functions, and others (cf. Sec. 5). Below, we will characterise an *abstract* notion of strategy that is not bound to any particular programming language or paradigm. We will also define a guideline suite of basic strategy combinators.

*Characteristics of strategies.* Strategies in the sense of SP are data-processing actions with the following characteristics:

**Genericity** Strategies are generic in the sense that they are applicable to data of any type (say, sort, or class).

**Specificity** Though generic, strategies provide access to the actual data structures by means of type-specific operations.

**Composability** There are means to express compound, conditional, and iterated strategy application.

**One-layer traversal** Strategies enable generic traversal into the immediate subcomponents of heterogeneous data structures.

**Partiality** The application of a strategy to a given datum may fail, and recovery from failure is possible.

**First-class** Strategies are first-class citizens in the sense that they can be named, can be passed as arguments, etc.

*SP vs. AP.* The abstract notion of strategy corresponds to a requirement specification for incarnating strategic programming in a given programming language or paradigm. It also provides a reference chart to assess other generic programming approaches. In the case of adaptive programming, we can pinpoint deviations of adaptive programs from our characterisation of strategies, in particular:

- Adaptive programs are not fully generic because their traversal specifications refer to class names and labels to describe milestones and and the relations between them. Using 'symbolic names' [11] instead of concrete names, traversal specifications become reusable.
- The traversal specifications of adaptive programs do not involve one-layer traversal on immediate subcomponents. AP favours instead operations on *sets of paths*.
- Adaptive programs do not involve a designated form of partiality. Traversals are performed by visiting all milestones. 'Around' wrappers control if the rest of the traversal is performed before or after the wrapper, or maybe not at all.
- Adaptive programs are normally not first-class citizens, although recent implementations [15] might admit the potential for first-class adaptive programs.

*A guideline combinator suite.* In the following, we specify a set of strategy combinators that must be supported by an incarnation of SP. Actual incarnations of strategic programming may include further combinators than those proposed below. Here is the syntax for strategy combinators $s$:

| | | | |
|---|---|---|---|
| $s$ | $::=$ | $\mathsf{id}$ | Identity strategy |
| | $\mid$ | $\mathsf{fail}$ | Failure strategy |
| | $\mid$ | $\mathsf{seq}(s, s)$ | Sequential composition |
| | $\mid$ | $\mathsf{choice}(s, s)$ | Left-biased choice |
| | $\mid$ | $\mathsf{adhoc}(s, a)$ | Type-based dispatch for a basic action $a$ |
| | $\mid$ | $\mathsf{all}(s)$ | Process all immediate subcomponents |
| | $\mid$ | $\mathsf{one}(s)$ | Process one immediate subcomponent |

The semantics of the combinators is shown in Fig. 5 while we suggest a semi-formal reading of the figure. The given semantics deliberately leaves open how to blend with the expressiveness offered by the host paradigm of an eventual incarnation. (Think of value semantics vs. reference semantics.) We refer to [22, 6] for the formal treatment of SP in a term-rewriting setting. We will now discuss the combinators in detail.

*Constants and composition.* The strategy $\mathsf{id}$ succeeds for any datum and returns its input without change. Dually, the strategy $\mathsf{fail}$ fails for any datum, indicated by the output $\uparrow$. There are two combinators for strategy composition. The sequence combinator $\mathsf{seq}$ applies its two argument strategies in succession. The left-biased $\mathsf{choice}$ combinator first attempts application of its first argument strategy. If and only if this application fails, the second argument is attempted. We assume that the definition of new named combinators can involve recursion.

*One-layer traversal.* The definitions of the combinators $\mathsf{all}$ and $\mathsf{one}$ formalise the intuitions from Fig. 2. They both push their argument strategy one level down into the input datum to process

| **Notation** | | | **Meaning** | | |
|---|---|---|---|---|---|
| $d$ | ... | data | $\mathsf{id}@d$ | $\Rightarrow$ | $d$ |
| $c$ | ... | data constructors | $\mathsf{fail}@d$ | $\Rightarrow$ | $\uparrow$ |
| $\overline{d}$ | ... | data or failure "$\uparrow$" | $\mathsf{seq}(s, s')@d$ | $\Rightarrow$ | $\overline{d}$ if $s@d \Rightarrow d' \wedge s'@d' \Rightarrow \overline{d}$ |
| $a$ | ... | type-specific actions | $\mathsf{seq}(s, s')@d$ | $\Rightarrow$ | $\uparrow$ if $s@d \Rightarrow \uparrow$ |
| $s$ | ... | strategies | $\mathsf{choice}(s_1, s_2)@d$ | $\Rightarrow$ | $d'$ if $s_1@d \Rightarrow d'$ |
| $a@d$ | ... | application of $a$ to $d$ | $\mathsf{choice}(s_1, s_2)@d$ | $\Rightarrow$ | $\overline{d}$ if $s_1@d \Rightarrow \uparrow \wedge s_2@d \Rightarrow \overline{d}$ |
| $s@d$ | ... | application of $s$ to $d$ | $\mathsf{all}(s)@[d]$ | $\Rightarrow$ | $[d]$ |
| $d \Rightarrow \overline{d}$ | ... | big-step semantics | $\mathsf{all}(s)@c(d_1 \cdots d_n)$ | $\Rightarrow$ | $c(d'_1 \cdots d'_n)$ if $s@d_1 \Rightarrow d'_1,\ldots,s@d_1 \Rightarrow d'_n$ |
| $a : t$ | ... | type handled by $a$ | $\mathsf{all}(s)@c(d_1 \cdots d_n)$ | $\Rightarrow$ | $\uparrow$ if $\exists i.\ s@d_i \Rightarrow \uparrow$ |
| $d : t$ | ... | type of a datum $d$ | $\mathsf{one}(s)@[d]$ | $\Rightarrow$ | $\uparrow$ |
| $[d]$ | ... | indivisible data | $\mathsf{one}(s)@c(d_1 \cdots d_n)$ | $\Rightarrow$ | $c(\cdots d'_i \cdots)$ if $\exists i.\ s@d_1 \Rightarrow \uparrow \wedge \cdots \wedge\ s@d_{i-1} \Rightarrow \uparrow \wedge\ s@d_i \Rightarrow d'_i$ |
| $c(d_1 \cdots d_n)$ | ... | compound data | $\mathsf{one}(s)@c(d_1 \cdots d_n)$ | $\Rightarrow$ | $\uparrow$ if $s@d_1 \Rightarrow \uparrow,\ldots,s@d_n \Rightarrow \uparrow$ |
| | | | $\mathsf{adhoc}(s, a)@d$ | $\Rightarrow$ | $a@d$ if $a : t$ and $d : t$ |
| | | | $\mathsf{adhoc}(s, a)@d$ | $\Rightarrow$ | $s@d$ if $a : t \wedge d : t' \wedge t \neq t'$ |

**Figure 5: Semantics of the strategy combinators**

all immediate components, or just the leftmost one for which the argument strategy succeeds, respectively. We use dedicated notation to differentiate between indivisible data and compound data. Note that all and one preserve the shape of the input datum because the constructor $c$ reappears in the result. We say that this kind of strategies is type-preserving, or that they perform a transformation. We omit the discussion of dual combinators that perform a query or an analysis with a fixed result type regardless of the input datum's type. (Recall select and reduce from the introduction.) To illustrate the definition of recursive traversal schemes in terms of one-layer combinators, we define *full_td* for full top-down traversal in terms of all. The following definition means that *full_td*$(s)$ applies its argument strategy $s$ at the root of the incoming datum, and then (cf. seq) it applies itself to all immediate components of the datum:

$$\mathit{full\_td}(s) \;=\; \mathsf{seq}(s, \mathsf{all}(\mathit{full\_td}(s)))$$

*Lifting type-specific actions.* In Fig. 5, we distinguish *type-specific* actions vs. *generic* data-processing actions — the latter being called strategies. There are means to mediate between the two categories. Obviously, a generic action $s$ can be applied immediately to a datum $d$ of any type. (The application operator $\ldots @d$ is overloaded for type-specific and generic actions.) Notably, a type-specific action can also be turned into a generic action by what we call 'type-based dispatch' or simply 'lifting'. This is necessary in order to enable the application of type-specific actions to subcomponents of different sorts in the course of traversal. Explicit lifting is accomplished by the adhoc combinator, which constructs a new strategy from a generic default $s$ and a type-specific action $a$. That is, the strategy $\mathsf{adhoc}(s, a)$ behaves like $s$ except for data of $a$'s input type; here it dispatches to $a$. An incarnation of strategic programming can omit the adhoc combinator, and favour *implicit lifting* instead. Then, a type-specific action $a$ is viewed as $\mathsf{adhoc}(\mathsf{fail}, a)$. We illustrate lifting by adding an application of adhoc to the strategic Haskell snippet from Fig. 4:

$$\mathit{simplify} \;=\; \mathit{full\_td}\ (\mathsf{choice}\ (\mathsf{adhoc}\ \mathsf{fail}\ \mathit{simplifyStep})\ \mathsf{id})$$

We use fail as default. We could have used id as well because we recover from failure anyway via choice ... id. Defaults other than id and fail are also sensible. One could, for example, consider recursive descent as default which is only meant to happen if the type of the basic action and the type of the given datum do not fit.

*Adaptive traversal primitives.* Let us clarify what it means that AP does not cater for access to immediate subcomponents of compound data. We recall that traversal specifications are composed in terms of the predicates *from*, *to*, *through*, and *bypassing* as sketched in the introduction. The semantics of AP usually refers to other primitives [18] that make clear that traversal specifications denote *sets of paths* in a graph with classes as nodes, and edges for subclassing and subobjects. These are the most fundamental primitives to compose traversal specifications $S$:

- $[A, B]$ — the set of paths from class $A$ to class $B$; this form corresponds to *from $A$ to $B$*.
- $S_1 \cdot S_2$ — the concatenation of the sets of paths $S_1$ and $S_2$ where the target class in $S_1$ must coincide with the source class in $S_2$; *through* predicates can be modelled via this form.
- $S_1 + S_2$ — the union of sets of paths where $S_1$ and $S_2$ must agree on the source and target classes; merging traversals can be modelled via this form.

There are further forms of composition, e.g., for intersection, and acyclic paths [11, 12]. The *bypassing* predicate can be viewed as a combination of a complement operation and intersection.

*Other forms of strategies.* The term *strategy* or related terms like *tactics* and *tacticals* are also used in other contexts of computing. Usually some sort of 'control' is associated to this use, but not the means to cater for generic access to components of heterogeneous data structures. For example, strategies are used to describe proof tactics and tacticals or programmable evaluation strategies in term rewriting [19]. In the newer AP literature, the term (traversal) strategy is also used as a generalisation of the 'traversal specifications' in the earlier literature. The generalisation concerns the way how traversal specifications are viewed. An AP-like strategy is viewed as a function on graphs preparing the actual traversal of milestones. At the programming level, still the same predicates *from*, *to*, etc. are used.

## 4. TRAVERSAL SCHEMES

The power of our strategy combinators can best be demonstrated with a few examples. Fig. 6 shows a list of combinators defined in terms of the basic ones. The first two control patterns *try* and *repeat* do not involve traversal whereas the remaining combinators define different traversal schemes. In fact, these are all general-purpose traversal schemes. We omit a discussion of domain-specific schemes, e.g., schemes for language processing. We will first explain all the schemes from the figure. Eventually, we will clarify how the adaptive style meets our strategic style.

*Non-traversal control.* The combinator *try* turns its argument strategy into an always succeeding strategy: $\mathit{try}(s)$ attempts $s$ but

$$
\begin{array}{rcl}
\mathit{try}(s) & = & \mathsf{choice}(s, \mathsf{id}) \\
\mathit{repeat}(s) & = & \mathit{try}(\mathsf{seq}(s, \mathit{repeat}(s))) \\
\mathit{full\_td}(s) & = & \mathsf{seq}(s, \mathsf{all}(\mathit{full\_td}(s))) \\
\mathit{full\_bu}(s) & = & \mathsf{seq}(\mathsf{all}(\mathit{full\_bu}(s)), s) \\
\mathit{once\_td}(s) & = & \mathsf{choice}(s, \mathsf{one}(\mathit{once\_td}(s))) \\
\mathit{once\_bu}(s) & = & \mathsf{choice}(\mathsf{one}(\mathit{once\_bu}(s)), s) \\
\mathit{stop\_td}(s) & = & \mathsf{choice}(s, \mathsf{all}(\mathit{stop\_td}(s))) \\
\mathit{naive\_innermost}(s) & = & \mathit{repeat}(\mathit{once\_bu}(s)) \\
\mathit{innermost}(s) & = & \mathsf{seq}(\mathsf{all}(\mathit{innermost}(s)), \mathit{try}(\mathsf{seq}(s, \mathit{innermost}(s))))
\end{array}
$$

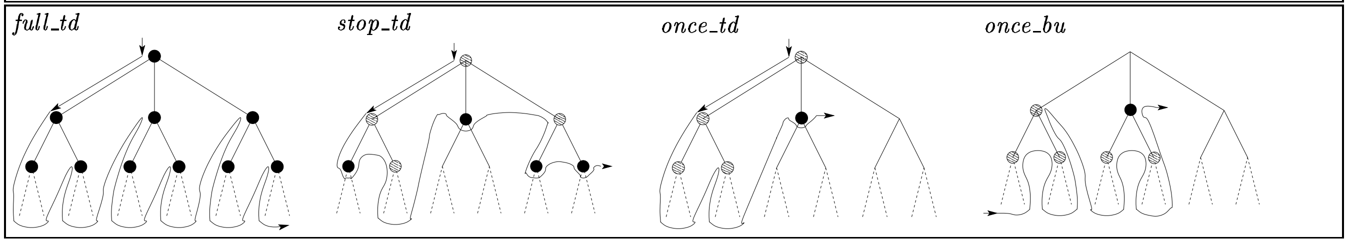


**Figure 6: Some defined strategy combinators**

resorts to id if $s$ fails. The *repeat* combinator serves for fixpoint computation: $\mathit{repeat}(s)$ applies $s$ repeatedly until $s$ fails. This control pattern is useful in the definition of traversal schemes like *innermost* where traversal involves exhaustive application of actions.

*Traversal schemes.* The combinators *full_td* and *full_bu* model a full top-down or bottom-up traversal, respectively. They apply their argument strategy at the root of the incoming datum, *and* at *all* its immediate and non-immediate components. The combinators *once_td* and *once_bu* are variations that apply the argument strategy only to the first component at which it succeeds. The combinator *stop_td* attempts the application of the argument strategy to components along all branches, and it stops in a given branch when an application succeeds. The *naive_innermost* and *innermost* combinators both implement the leftmost innermost evaluation strategy, but the second is more efficient than the first.

*SP meets AP.* We are now in the position to reconstruct adaptive program patterns as strategy combinators. This reconstruction is based on the following ideas:

**Milestones as strategies** Adaptive traversal specifications refer to class names as milestones. In SP, we call this a type guard — a strategy that succeeds if and only if faced with data of a given type. Hence, the combinator for an AP pattern receives strategy arguments for milestone identification.

**Code wrappers as strategy combinators** An 'around' wrapper in AP dictates how to superimpose the wrapper's functionality onto the traversal steps (with 'before' and 'after' as special cases). In SP, we model code wrappers as strategy combinators that take the 'rest of the traversal' as a strategy argument.

**Predicates as traversal schemes** The predicates used in the traversal specifications of AP are mapped to appropriate strategic traversal schemes. A traversal specification with several predicates maps then to a cascaded traversal strategy.

We start with the most simple example of a traversal specification, namely *from A to B* with associated code wrappers $W_A$ and $W_B$ for the two milestones. We define a strategy combinator *from_to* for this pattern with arguments for $A$, $B$, $W_A$, and $W_B$:

$$
\begin{array}{lcl}
\mathit{from\_to}(A, B, W_A, W_B) & = & \mathsf{seq}(A, W_A(\mathit{to}(B, W_B))) \\
\mathit{to}(B, W_B) & = & \mathsf{all}(\mathit{stop\_td}(\mathsf{seq}(B, \mathit{try}(W_B))))
\end{array}
$$

For short, it is the *stop_td* scheme which does all the work. We first test for $A$ to enforce that we are faced with a datum of type $A$. Then, we invoke the wrapper $W_A$ and pass the rest of the traversal to it as an argument. The rest is defined by a helper combinator *to*. The outermost all implies descending one level, and the $\mathit{stop\_td}(\mathsf{seq}(B, \ldots))$ means following all branches but stopping for $B$. When we found $B$, we invoke the wrapper $W_B$. This wrapper strategy takes no arguments because there is no remaining traversal. We enforce success via *try*. As a general remark regarding the above reconstruction: the semantics of SP implies that the reconstruction is more greedy than prescribed by the semantics of AP. That is, even subcomponents the types of which rule out nested $B$s are traversed. This concern will be addressed in Sec. 6. We continue with a combinator that includes a *through* predicate:

$$
\begin{array}{l}
\mathit{from\_through\_to}(A, \boxed{T}, B, W_A, \boxed{W_T}, W_B) \ = \\
\quad \mathit{from\_to}(A, T, W_A, W_T(\mathit{to}(B, W_B)))
\end{array}
$$

We boxed the added parameters $T$ for the *through* milestone, and the associated wrapper $W_T$. We first go from source nodes (cf. $A$) to intermediate nodes (cf. $T$) via the *from_to* combinator. The wrapper $W_T$ for the intermediate nodes receives the appropriate rest of the traversal, which is meant to eventually reach all target nodes (cf. $B$) via the *to* combinator. This idea works for any number of *through* predicates. Each new milestone is found by a *stop_td* traversal. We can also cope with *bypassing* predicates:

$$
\begin{array}{l}
\mathit{from\_to\_bypassing}(A, B, \boxed{N}, W_A, W_B) \ = \\
\quad \mathit{from\_to}(A, \mathsf{choice}(N, B), W_A, \mathsf{choice}(N, W_B))
\end{array}
$$

So we stop the *from_to* traversal at both target nodes (cf. $B$) as well as bypassing nodes (cf. $N$), and we make sure that the wrapper $W_B$ is only applied in case we are faced with a proper target node.

## 5. IMPLEMENTING STRATEGIES

The strategic programming idiom has been realised within several programming paradigms. There are fully worked-out and tool-supported incarnations for term rewriting based on *Stratego* [22,

| *Concept per paradigm* | **Term rewriting** | **Functional programming** | **OO programming** |
|---|---|---|---|
| *Datum* | many-sorted term | term of an algebraic datatype | object graph |
| *Immediate component* | subterm | subterm | referenced object |
| *Basic action* | rewrite rule | monomorphic function | specific visit method |
| *Strategy* | term rewriting strategy | 'strategically' polymorphic function | generic visitor object |
| *Strategy application* | dedicated operator | function application | visit method invocation |
| *Strategy combinators* | strategy definitions | higher-order functions | visitor classes |
| *Type-based dispatch* | implicit | type-safe cast | RTTI, dynamic binding |
| *Types* | liberal checks | rank-2 types, constrained "∀" | subtype polymorphism |
| *Partiality* | built-in | monadic effect | exceptions |
| *Host idioms* | 'DSL-like' extensions | monadic effects | graphs, side effects |

**Figure 7: Overview of strategic programming incarnations**

20], for functional programming in Haskell based on *Strafunski* [7, 5], and for object-oriented programming in Java based on *JJTraveler* / *JJForester* [21]. *Stratego* is a language that is devoted to the strategic programming idiom. The functional and object-oriented incarnations take a different road: they aim at making the SP idiom available in general-purpose programming languages. We will not revisit the incarnations in detail. Instead, we will discuss how to incarnate, in general, strategic programming. Furthermore, we will compare the incarnations at a higher level of abstraction. Implementational models for AP will also be sketched.

*Incarnation process.* An incarnation is designed by mapping the abstract notion of strategy onto the host paradigm. This involves the identification of an abstraction form for modelling strategies. In the case of functional programming, for example, strategies are modelled as a specific kind of polymorphic functions. One also has to instantiate subsidiary concepts such as *type-specific action*, *datum*, *component*, *partiality*. The incarnation process culminates in the implementation of the guideline set of basic strategy combinators id, fail, seq, choice, all, and one. In Fig. 7, we compare strategic programming in three paradigms based on the instantiation of the relevant concepts. The incarnations exhibit different trade-offs as we will pinpoint below. The incarnation process involves certain challenges. One is that all programming idioms that are 'native' to the host paradigm should remain available to the programmer when using strategies. In object-oriented programming, strategies should blend with reference semantics, and side-effects. Functional strategies should have value semantics, allow monadic effects, and be strict or lazy depending on the host language. Another challenge is the typing of strategies. In a strongly typed setting, a kind of 'strategic polymorphism' is needed [5]. This necessitates second-order polymorphism, and goes beyond parametric polymorphism and ad-hoc polymorphism.

*Term rewriting strategies.* The *Stratego* [22, 20] encoding of the running example is shown in Fig. 8. As one can see, basic computations are represented as ordinary (though labelled) rewrite rules. We use *Stratego*'s left-biased choice combinator <+ to combine the rewrite rules [S1] and [S2] into the helper strategy *simplifyStep*. *Stratego* uses implicit lifting, and hence, the scheme *full_td* is directly applied to the type-specific strategy *simplifyStep*. *Stratego* as of today only performs liberal type checks, namely a kind of arity checking for term constructors and strategy combinators. The well-formedness of terms according to a given signature can be checked at run-time.

*Stratego—a DSL for program transformation.* In the design of *Stratego*, the prime issue was to effectively support the

```
signature
 constructors
  Alt : RegExp * RegExp -> RegExp
  Opt : RegExp -> RegExp
  ...
rules
 S1 : Alt(Epsilon,exp) -> Opt(exp)
 S2 : Opt(Plus(exp))   -> Star(exp)
strategies
 simplify     = full_td(simplifyStep <+ id)
 simplifyStep = S1 <+ S2
```

**Figure 8: Stratego representation of Fig. 4**

development of program transformation systems. Hence, *Stratego* can be viewed as a domain-specific language (DSL). In fact, a number of domain-specific constructs are available in *Stratego*, e.g., the hygienic generation of fresh names as needed in transformations, and scoped dynamic rewrite rules to compute rules at run-time. *Stratego*'s DSL character is also reflected by other provisions. The language implementation performs specific traversal-aware optimisations. It further uses a designated run-time term representation that allows for sharing, constant time equality test, and hidden transportation of comments and layout.

*Functional strategic programming.* By modelling strategies as functions [7, 5], the first-class requirement for strategies can be met without further ado. All other incarnations of strategic programming are more problematic in this respect. We have investigated a variety of models for functional strategies. They differ regarding the selection of strategy primitives, and subtle details of typing and representation. The original expressiveness of strategies can be captured in just two special function combinators:

- The *adhoc* combinator as defined earlier.
- A highly parameterised one-layer traversal combinator.

The two special combinators can be made available in three ways:

- The programmer instantiates them for each new datatype.
- A generative tool supplies the datatype-specific code.
- A language extension covers the combinators.

The generic programming bundle *Strafunski* supports several models via a generative tool component.

*Object-oriented strategic programming.* This incarnation uses generalised *visitor* objects to model strategies [21]. A number of ideas are needed to make folklore visitors fit for SP, that is, to meet all defining characteristics of strategies:

```
class Seq implements Visitor {
  Visitor v1, v2;
  public Seq(Visitor v1, Visitor v2) {
    this.v1 = v1; this.v2 = v2;
  }
  public Visitable visit(Visitable x) {
    return v2.visit(v1.visit(x));
  }
}
public class FullTD extends Seq {
  public FullTD(Visitor v) {
    super(v,null);
    v2 = new All(this);
  }
}
```

**Figure 9: Strategy combinators as visitor combinators**

- While standard visitors are specific to a class hierarchy, strategic programming additionally necessitates completely generic visitors. These visitors implement a single *visit* method.
- To enable one-layer traversal, we need to cater for generic access to the immediate subobjects of objects. This is accomplished by a *Visitable* interface to get and set all 'children'.
- To cover partiality for visitors, failure is encoded by throwing a *VisitFailure* exception, and left-biased choice recovers from failure via exception handling.
- The double-dispatch protocol of ordinary visitors is complemented by a visitor combinator that forwards any class-specific visit method to a generic visit method. By subclassing a forwarding visitor, one achieves the effect of adhoc.
- While ordinary visitors are 'void' visitors, 'returning' visitors are preferred in the SP setting. That is, $v.visit(x)$ always returns an object — normally $x$. This makes it easier to replace objects by new ones (possibly of different subtypes).
- A combinator style for first-class visitors relies on parameterised constructors for visitors. Using the generic visitor interface, one can define generic visitor combinators. This is demonstrated for seq and *full_td* in Fig. 9.

The 'strategies as visitors' approach is naturally supported via a generative tool (*JJForester* in our case). This concerns the aforementioned *Visitable* interface to be introduced into a given class hierarchy, and the derivation of the ordinary visitor class, as well as the forwarding visitor combinator. The current typing model necessitates some casting. This problem can be remedied with generics.

*AP language implementation.* Previous approaches to the implementation of adaptive programs normally relied on compilation. In [18], an adaptive program is compiled into an object-oriented program where the class hierarchy contains a method for each adaptive traversal. The generated method definitions recurse into subobjects, and they invoke the code wrappers. A problem with this approach is that the generated code could be invoked incorrectly without starting at a proper source node. In [17], a more general compilation technique is described with several methods per traversal. The idea is here that the search through the object graph can be modelled as a (deterministic) finite automaton where the states are modelled by methods. In [11], a generic approach to the generation of traversal methods is described. In this approach, traversal specifications are compiled into road-maps that are used by the traversal methods at run-time. The main idea is to maintain a traversal graphs with tokens that represent the traversal history. In [15], an implementational model for AP is described which is reflection-based, that is, no preprocessing or compilation is needed.

## 6. TRAVERSING OBJECT STRUCTURES

While the notion of strategies was initiated in the declarative programming setting of term rewriting (i.e., referential transparency, no cycles in data structures, many-sorted data, no side effects), strategies are perfectly sound in other settings, too. The *Stratego* incarnation demonstrates how to cope with side effects. The object-oriented incarnation provides the prime platform for traversal strategies on mutable graphs, i.e., object structures. Since adaptive programming focuses on traversing object structures, we will investigate the issues that come up in this context.

*Mutable structures.* There are two overall options to deal with mutable structures, say, object structures, in SP:

1. Strategies mutate the (traversed) objects to compute a resulting object structure.
2. Strategies perform (deep or shallow) cloning, in particular the one-layer traversal combinators.

In our implementations, we adhere to the first option because the preservation of the original object structure is normally not required. Deep cloning can still be triggered explicitly by the programmer. This attitude is well in line with AP where the code wrappers are immediately executed whenever milestones are encountered in the course of a traversal, without even waiting for a proper target node to be reached [18]. These code wrappers typically accumulate some result, or they mutate the milestone objects, or both. In principle, they could also perform cloning. However, searching through the object structure by itself does not involve any cloning.

*General topologies.* In the primary application domain of SP — program transformation —, strategies are predominantly applied to tree-shaped data. Even in the object-oriented setting, the object structures normally resemble the context-free grammar of the language subject to program transformation and analysis. We have certainly encountered situations where other topologies need to be covered, e.g., the traversal of control-flow graphs [3]. There are the following issues regarding the potential of different topologies such as trees, directed acyclic graphs (DAG), or even cyclic graphs:

- If cyclic structures are traversed, we have to make sure that the traversal will always *terminate*.
- If objects can be referenced several times, we have to decide if the traversal should *differentiate* between these references.

Creating awareness of the different references to an object as in a DAG is just a refinement of cycle detection as discussed below. These are the possible attacks for coping with cycles:

1. We restrict ourselves to tree-shaped object structures for strategic traversal. A corresponding check can be automated.
2. We make sure that a traversal only sees tree-shaped slices of object structures. This can be done via bypassing conditions.
3. We keep track of traversed objects, and we let the traversal fail for the second encounter of a given object.

In Fig. 10, we list a visitor combinator *FailIfVisited* that can be used as a guard in strategic programs to detect a cycle in the sense of (3.) above. One can prefix node processors by *FailIfVisited* so that a traversal will not descend into an object for a second time. As an orthogonal example, here is a full top-down traversal that merely tests for tree-shape:

```
public class FailIfVisited extends Visitor {
  Set visited = new HashSet();
  public FailIfVisited() {}
  public Visitable visit(Visitable x)
            throws VisitFailure
  {
    if visited.contains(x) {
      throw new VisitFailure();
    } else {
      visited.add(x); return x;
    }
  }
}
```

**Figure 10: A provision for cycle detection**

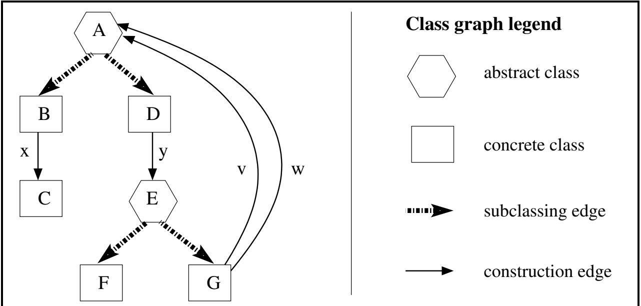
```
public class IsTree extends FullTD {
  public IsTree() {
    super(new FailIfVisited());
  }
}
```

Again, this approach is well in line with AP. In [11], a similar technique for invoking traversal methods is described to cut off traversal whenever an object is encountered for the second time. In [12], a predicate for traversal specifications is considered that specifically rules out cyclic paths. In fact, cyclic structures are otherwise not too much of an issue in the AP literature. Instead of using a combinator like *FailIfVisited* and a collection of visited objects, one might also think of other means that could be served by a native SP implementation, e.g., marking objects with visit flags.

*Compatibility.* In [18], compatibility is described as a desirable property for AP. Given a traversal specification and a concrete class graph, compatibility means that all the milestones of the traversal can be *possibly* established for some object graph. This is illustrated in Fig. 11. Note that one can not *guarantee* the reachability of the milestones for all possible object graphs. (Think of NIL references, and non-instantiated subclasses.) Compatibility can be transposed to SP as follows:

- Compatibility of a strategy $s$, a root class $c$, and a class graph $g$ means that each type-specific branch in $s$ should be possibly encountered for some object graph rooted by an object of class $c$. This deviation reflects that strategies are not centred around the idea of milestones but we rather assume that type-specific computations interact with the concrete data structure (recall explicit lifting via adhoc or implicit lifting).
- Compatibility checking in AP can be reduced to a graph-theoretical reachability analysis. Compatibility in SP is, in general, undecidable. This is because all ingredients of a strategic program are defined in the same Turing-complete language as opposed to the separation of traversal specifications and code wrappers in AP.
- Compatibility is biased towards adaptive traversal. In SP, we also consider completely generic traversal schemes. One might also want to check these schemes for properties, e.g., if a traversal scheme can succeed at all, if it might tend to fail too often, or if it might succeed too easily with a trivial result. Here are examples of problematic strategies:
  - *full_td* (adhoc fail ...) — will fail too often
  - *once_td* (adhoc id ...) — will succeed too easily

We consider the study of such properties as a topic for future work.



*Properties of traversal specifications:*

- from $A$ through $E$ to $C$ *is compatible with the class graph.*
- from $A$ through $C$ to $E$ *is not compatible with the class graph.*

*To see this, one can observe:*

- *A $C$ object is reachable from an $E$ object for some object graphs.*
- *An $E$ object is not reachable from a $C$ object for any object graph.*

**Figure 11: Compatibility of adaptive programs**

*Premature termination.* The semantics of adaptive programs employs static meta-information on the class graph to limit the traversal to those branches in the given object graph that can possibly lead to a target node [13]. Intuitively, a reachability analysis as in the case of compatibility determines the optimised traversal sequences which cut off hopeless branches. Recall Fig. 11:

> *Traversing an object graph according to "from $A$ through $E$ to $C$", traversal ends prematurely when we hit on a $B$ node before we saw an $E$ node because $E$ is not reachable from $B$ in the class graph.*

The benefit of premature termination is foremost efficiency since less nodes need to be visited. In addition, code wrappers for *through* nodes will not be executed unnecessarily if a target node cannot be reached anyway. If the cut-off analyses in AP are done at compile-time, then a closed-world assumption is necessary (i.e., no further subclassing). The reflection-based approach in [15] avoids this problem. Premature termination of strategies in SP on the basis of static analyses has not been an issue so far. We consider the transposition of the AP techniques as a prime topic for future work on SP, and as the crucial element of a completed marriage of SP and AP. Note that cutting off branches in a strategic traversal can be very well achieved with stop conditions. This is common practice in SP — not necessarily related to efficiency, but often also to correctness, just as for *bypassing* predicates.

*Surprising paths.* The structure shyness in AP is sometimes considered harmful [16]. The problem is that traversal strategies might go along surprising paths especially when the class hierarchy changes. One can call this a robustness issue. This under-specification problem is perceived differently in the SP context. In the application context of programming transformation and analysis, the programmer is usually very well aware of the given language syntax — so few surprises are to be expected. Adaptiveness is mainly employed for *conciseness*. In fact, languages do not change so dramatically as arbitrary object models.

## 7. CONCLUSION

Traversal programming in SP and AP are prime examples of general-purpose aspect-oriented programming techniques. Strategic and adaptive programming share several benefits: conciseness of traversal specification, support for reuse of basic computations, and isolation from changes in the data structures. Both idioms are well founded and well supported by corresponding tools and programming environments. We will conclude the paper by highlighting the respective strengths and weaknesses of AP and SP.

*Adaptive programming*

+ Use of static meta-information to optimise traversals.
+ Compatibility checking of traversal specifications and class graphs.
+ Clear separation of traversal specifications and code wrappers.
− Absence of a number of variation points for traversal.
− Bias towards object-oriented programming.

*Strategic programming*

+ Programmer-definable, reusable, generic traversal schemes.
+ Access to the full range of variation points for traversal.
+ Language-independent generic programming idiom.
− Premature termination of traversals not automated.
− Freewheeling combinator style blurs focus.

The ultimate, unified aspectual traversal approach shall combine the strengths to dissolve the weaknesses. To us, this seems feasible.

*Acknowledgement.* We are grateful for the helpful comments and suggestions by the anonymous AOSD 2003 referees. We also appreciate the feedback from the Demeter team, in particular, Karl Lieberherr, Doug Orleans, Yi Qian, Therapon Skotiniotis.

*Availability.* The rewriting, the functional and the object-oriented incarnations of SP are supported by corresponding programming environments Stratego/XT, Strafunski, JJForester/JJTraveler. These software bundles together with documentation, and related research papers are freely available from the following locations:

- `http://www.stratego-language.org/`
- `http://www.cs.vu.nl/Strafunski/`
- `http://www.jjforester.org/`

## 8. REFERENCES

[1] O. S. Bagge, M. Haveraaen, and E. Visser. CodeBoost: A Framework for the Transformation of C++ Programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, 2001.

[2] M. Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, July 2001.

[3] A. v. Deursen and J. Visser. Building Program Understanding Tools using Visitor Combinators. In *Proc. of 10th Int. Workshop on Program Comprehension, IWPC 2002*. IEEE Computer Society, 2002.

[4] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.

[5] R. Lämmel. The Sketch of a Polymorphic Symphony. In B. Gramlich and S. Lucas, editors, *Proc. of Inter. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *ENTCS*. Elsevier Science, 2002. 21 pages.

[6] R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54:1–64, September 2002.

[7] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In S. Krishnamurthi and C. Ramakrishnan, editors, *Proc. of PADL 2002, Portland, OR, USA*, volume 2257 of *LNCS*. Springer-Verlag, Jan. 2002.

[8] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proc. of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[9] K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[10] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *CACM*, 44(10):39–41, Oct. 2001.

[11] K. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, Northeastern University, Boston, 1997.

[12] K. Lieberherr, B. Patt-Shamir, and S. Pradhan. An Efficient Compiler for Adaptive Programs. Technical Report NU-CCS-97-03, Northeastern University, Boston, 1997.

[13] K. Lieberherr and M. Wand. Navigating through Object Graphs Using Local Meta-Information. Submitted for publication, June 2002.

[14] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. of FPCA'91*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[15] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 73–80, Kyoto, Japan, Sept. 2001. Springer-Verlag.

[16] J. Ovlinger and M. Wand. A Language for Specifying Recursive Traversals of Object Structures. In *Proc. of the 1999 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 70–81, 1999.

[17] J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A new approach to compiling adaptive programs. *Science of Computer Programming*, 29(3):303–326, Sept. 1997.

[18] J. Palsberg, C. Xiao, and K. J. Lieberherr. Efficient implementation of adaptive software. *ACM Trans. Prog. Lang. Syst.*, 17(2):264–292, Mar. 1995.

[19] L. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, Aug. 1983.

[20] E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, May 2001.

[21] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices, OOPSLA 2001 Conf. Proc.*, 36(11):270–282, Nov. 2001.

[22] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proc. of the 3rd ACM SIGPLAN Inter. Conf. on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, Sept. 1998.