Effective Strategic Programming for Java Developers

Emilie Balland, Pierre-Etienne Moreau, Antoine Reilles

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–33
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Effective Strategic Programming for Java Developers

Emilie Balland[1], Pierre-Etienne Moreau[2,3] and Antoine Reilles[4]

[1] *INRIA, Bordeaux (E-mail: Emilie.Balland@inria.fr)*
[2] *Université de Lorraine, Loria, Villers-lès-Nancy, F-54600, France (E-mail: Pierre-Etienne.Moreau@loria.fr)*
[3] *Inria, Villers-lès-Nancy, F-54600, France*
[4] *Dassault Systemes, Paris (E-mail: Antoine.Reilles@3ds.com)*

SUMMARY

In object programming languages, the *Visitor* design pattern allows separation of algorithms and data-structures. When applying this pattern to tree-like structures, programmers are always confronted with the difficulty of making their code evolve. One reason is that the code implementing the algorithm is interwound with the code implementing the traversal inside the Visitor.
When implementing algorithms such as data analyses or transformations, encoding the traversal directly into the algorithm turns out to be cumbersome as this type of algorithm only focuses on a small part of the data-structure model (*e.g.*, program optimization). Unfortunately, typed programming languages like Java do not offer simple solutions for expressing generic traversals.
Rewrite-based languages like ELAN or Stratego have introduced the notion of *strategies* to express both generic traversal and rule application control in a declarative way. Starting from this approach, our goal was to make the notion of *strategic programming* available in a widely used language such as Java and thus to offer generic traversals in typed Java structures. In this paper, we present the strategy language SL that provides programming support for strategies in Java. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: generic programming, term rewriting, tree traversal, strategies, object-oriented programming

1. INTRODUCTION

In object-oriented programming languages, the *Visitor* design pattern [1] allows separation of algorithms and data-structures, making it possible to add new functions to a hierarchy of classes without changing their implementation. This approach is suitable when new operations need to be added frequently like in a compiler or an interpreter. The main drawback of this design pattern is that if the underlying data-structure is extended, the visitor has to be adapted to deal with the new elements of the hierarchy. This tension between datatype extension and function extension is well-known as the expression problem [2].

Another difficulty when implementing visitors it to deal with transformations on deep hierarchical structures such as abstract syntax trees or XML documents. Indeed there is no separation of the traversal of the structures and the logic of the function, requiring the user to define the recursive call by hand. One improvement of the Visitor pattern is the *Hierarchical Visitor* pattern [3] that allows one to separate the traversal from the logic. This pattern is used in association with the Composite pattern [1]. The Composite pattern is used to describe tree-structured data such that individual objects and tree-structured objects are treated in an uniform way. The traversal behaviour can thus be defined at the level of the *accept* method of the composite. It means that the way of traversing the structure is directly encoded in the underlying composite hierarchy. Thus, offering several traversal types for the same hierarchy is still cumbersome.

In rule-based languages, the concept of strategies has been introduced as a way to control rule application and thus offer a clear separation between traversal, function logic and data-structure. In this paper, we show how this notion of strategic transformations can be embedded in a general-purpose language like Java. Contrary to the visitor and hierarchical visitor patterns, strategies allow one to clearly separate the traversal from both the visitors and the composite hierarchy. Moreover, embedding strategies into Java goes beyond the control of transformations. For example, it is possible to specify complex code analyses in a clear and declarative manner. Using examples in a broad range of application domains, we demonstrate the interest of *strategic programming* for Java developers in term of usability, code quality and maintenance.

Contributions

The main contributions of this paper can be summarized as follows:

- the formal description of the SL language enabling strategic programming in Java. This includes the definition of its operational semantics.
- the design and implementation of a Java library that implements the SL language. This implementation is inspired by the JJTraveler [4] library but offers several enhancements such as environments and introspection.
- the extension of the rule-based language TOM [5] with strategy declaration constructs based on SL. This allows one to embed declarative constructs for SL in Java while still relying on its Java library implementation.
- the validation of the SL language in a wide range of areas including program optimization, XML manipulation and object-relational model transformation for Java persistence.

This paper goes beyond the previous publications [5, 6] by giving a comprehensive and formal description of the SL language and by discussing implementation details that were never published before. In particular, this paper shows how SL is independent from the TOM language and thus can be used in a plain Java application. Most of the examples used in this paper to illustrate the benefits of SL are also different from the ones used in the previous publications (in particular, the examples taken from the domain of program optimization in Section 4).

Outline

Section 2 presents the background of the SL strategy language, mainly rule-based languages and datatype-generic programming. In Section 3, we present the syntax and the semantics of SL. Section 4 shows how to program using TOM extended with SL constructs. The examples are taken from the domain of program optimization. In Section 5, we detail the design of the implementation of SL in Java (*i.e,* as a Java library). In particular, we explain how the library has been designed to enable extensions of the language to more complex data-structures like graphs. Before concluding, Section 6 gives an overview of existing applications written in TOM and SL in a wide range of application domains. This illustrates how such an approach could improve the quality and reduce the maintenance cost of software. Related work is discussed in Section 7 and conclusions are given in Section 8.

2. BACKGROUND

The work presented in this paper is inspired by different research areas. Originally, it comes from the domain of rule-based languages, but it has also been influenced by datatype-generic programming and languages dedicated to XML manipulation.

2.1. Control in rule-based languages

The notion of rewrite rule is an abstraction that can be used to model various processes. It has been intensively used to model, study, and analyze different parts of complex systems, from algorithms

DOI: 10.1002/spe

to software [7]. In addition to modeling and analysis, the notion of rewrite rule has also been used as a programming paradigm for tree-structure transformation. Indeed, the concept of *tree* is a fundamental concept in data-structures, largely used in computer science to represent structured documents such as programs, symbolic formulae, parse trees, abstract syntax trees (AST), XML documents, *etc.* In the context of rule-based programming, first-order terms are used to represent tree data-structures and there exist several languages based on term rewriting such as ASF+SDF [8], Clean [9], ELAN [10], MAUDE [11], Stratego [12] and TOM [5].

Programming with rewrite rules consists of decomposing a complex transformation into elementary transformations, encoded by rewrite rules. Given this set of rules and an input term, the rewrite engine attempts to fire a rule whenever it is possible. Most of the time, users are interested in getting a result whose computation is deterministic and thus need to control how the rules are applied. A first solution is to encode some form of control directly in the rules. From a software engineering point of view, this minimizes the elegance of term rewriting and makes the system much more complex and difficult to maintain. Another solution is to assign a priority to each rule and to consider an execution mechanism that encodes a fixed order of reduction, such as *innermost* or *outermost*, also called *call by value* or *call by name* in functional programming languages. Another solution is to separate the control from the rules. Instead of encoding the control into the rules themselves, it is described in a distinct expression or language. For instance, the expression *TopDown(R)* denotes that the rewrite system R is applied in a *top-down* way. The expressions that specify how the rules should be applied are called *strategies*. The design of such a strategy language is not easy and several attempts and proposals have been made.

OBJ [13] is one of the first languages that introduced an explicit form of strategy, called an *evaluation strategy*. To each operator a list of integers can be attached to specify in which order the arguments should be evaluated. For instance, the declaration `if_then_else_ (1 0 2 3)` stands for a ternary operator where the first argument is evaluated first. Then, depending on its value, the second or the third argument is reduced.

ASF+SDF also has a strategy language [14] similar to OBJ's. To each operator, one can attach an annotation that specifies its behavior. For example, the combination of `traversal` and `bottom-up` indicates that a given set of rules should be applied in a bottom-up way. This approach is of course less general than the previous one, but it is an interesting trade-off between expressiveness and simplicity of use.

MAUDE [15, 11] followed this approach and added the notion of *meta-level*. In this setting, a rewrite rule has a name, considered as a constant, and can be explicitly applied via the `meta-apply` operator. The application of a set of rules can be controlled by another program, defined by rewriting and using `meta-apply`. This new program can also be controlled by another program from the meta-meta-level and so on. This tower of reflection is elegant and expressive, but quite difficult to use. In addition, MAUDE offers a mechanism of evaluation strategies inspired by OBJ's.

ELAN [16] had its origins in OBJ, but it followed another approach. Instead of having a meta-level, ELAN was the first language to introduce an explicit *strategy language* [17] to control the application of the rules. Each rule has a name that corresponds to an elementary strategy. A strategy can then be combined with another one using operators such as `;` (sequence), `repeat`, `dont-care`, and `dont-know`. This strategy language was both expressive and easy to use.

Stratego [12] has been inspired by ELAN, MAUDE, and the functional programming style. It introduces an elegant and simple strategy language. As in ELAN, an elementary strategy can be a rule, the identity function (`Identity`), or a failure (`Fail`). They can be combined with strategy operators such as `;` (sequence), `<+` (left-choice), *etc.* The main contribution comes from the introduction of a recursion operator μ and two generic congruence operators `All` and `One` that can be used to describe higher-level strategies such as *top-down*, *innermost* or *outermost*. In this setting, we have $TopDown(s) = \mu x\,.\,s\ ;\ All(x)$, which applies s to a term t, and then recursively applies the $TopDown(s)$ strategy to the immediate subterms of t.

DOI: 10.1002/spe

2.2. XML Programming

An XML document is a tree and as such has been a natural target of rule-based languages such as Stratego or TOM. There also exist domain-specific languages dedicated to the manipulation of XML documents. For example, XPath [18] is a language providing a concise syntax and an expressive semantics for selecting parts of an XML document. Built on top of XPath, XSLT [19] is a transformation language for rewriting XML documents. Both of these languages are recommendations of the W3C and the Java standard library features an implementation of XPath 1.0 (`javax.xml.xpath`), which enables the Java programmer to evaluate XPath expressions over DOM documents.

In comparison with traversal strategies in rule-based languages, a noticeable characteristic of XPath is to propose several path axes such as sibling, parent and descendant axes. This specificity has influenced standard strategy languages like SYB [20] or SL (presented in this paper). These strategy languages have both introduced a notion of application context to take into account other axes than the descendant axis.

2.3. Datatype-generic programming

In generalist programming languages, datatype-generic programming consists in defining a function that does not depend on a specific datatype but only on the structure of the datatype. This enables the definition of functions reusable with several datatypes (*e.g.*, the definition of print or equal functions). Several approaches have been proposed for both functional languages (*e.g.*, generic HASKELL [21], "Scrap Your Boilerplate" [22] (SYB), Kure [23]) and object-oriented languages (*e.g.*, Pizza [24], Kiama [25], JJTraveler [4]). In statically typed languages, the main challenge is to ensure the type safety of the strategy combinators, which depends a lot on the expressiveness of the underlying type system. For example, Java's type system is less expressive than HASKELL's and thus a lot of invariants of strategies that can be enforced in HASKELL can only be checked at runtime in Java. As a matter of fact, HASKELL is the language where datatype-generic programming has been the most studied, resulting in a profusion of language extensions and libraries. Some papers have attempted to compare all these approaches with regard to criteria such as extensibility, performance or ease of use [26, 27].

In the context of Java, the work closest to ours is JJTraveler. JJTraveler is a framework that provides generic visitor combinators for Java. Contrary to the original visitor design pattern, this framework makes the composition of visitors possible by providing a set of generic combinators similar to the ones in Stratego. This separation between generic combinators and data-structure specific visitors leads to better code reuse.

2.4. The TOM language

TOM* is a rule-based language embedded in Java. This work is behind the development of the SL library as the initial goal was to enrich TOM with strategies.

TOM provides two main features: a "`%match`" construct to match objects, and a "`` ` ``" construct to build objects. The `%match` construct adds pattern matching facilities (*i.e,* rules of the form `lhs -> rhs`) similar to the ones that exist in functional programming languages. A pattern is built upon an algebraic signature that describes the structure of the objects being matched. Informally, a signature is a list of *sorts* that correspond to types, and each sort is associated with a list of *constructors*, described by their name and their domain. For instance, the notion of arithmetic expression can be described by the following algebraic signature:

```
Expression = Plus(e1:Expression,e2:Expression)
           | Mult(e1:Expression,e2:Expression)
           | Var(name:String)
           | Cst(n:int)
```

*http://tom.loria.fr

DOI: 10.1002/spe

In a `%match` construct the right-hand side of a rule is a list of instructions written in Java. This code, called the *action*, is executed each time the pattern matches. In this code, the "`" construct can be used to build a term. More generally, the "`" construct can be used anywhere a Java expression can be used.

The following example is a TOM program. First, it builds an expression (*i.e,* a tree) that corresponds to the addition of the constants 3 and 2. The lexical scope of "`" corresponds to a well-formed term. In this example, it ends just before the ";". In a second step, the term referenced by the Java variable `t` is matched:

```
public static void main(String[] args) {
  Expression t = `Plus(Cst(3),Cst(2));
  %match(t) {
    Plus(Cst(x),Cst(y)) -> {
      System.out.println( `x + `y );
    }
    Mult(Cst(x),Cst(y)) -> {
      System.out.println( `x * `y );
    }
  }
}
```

In this example, the first pattern will match `t`, and the variables `x` and `y` will be respectively instantiated by 3 and 2, resulting in the printing of 5. The second rule does not fire since the pattern does not match. The semantics of TOM is intuitive for Java developers. As in the `switch/case` construct of Java, the action part is executed for all the patterns that match the subject. The patterns are evaluated from top to bottom.

A main originality of TOM is to not enforce any particular tree representation for the objects being matched. To make this possible, TOM provides a *mapping definition formalism* to describe the relationship between the concrete implementation and the algebraic data-structure [28, 29]. A mapping is a piece of code that gives TOM the information about the algebraic structure of the objects that we intend to match on. For example, TOM needs to know how to test the type of the object and the equality between two objects of the same type as well as how to decompose it. This idea, related to P. Wadler's views [30], allows TOM to rewrite any kind of data-structure, as long as a mapping is provided. Similarly to views, a TOM mapping defines both the decomposition and the construction functions (denoted respectively *in* and *out* in views). A view is well-defined if these two functions are the inverses of each other. In TOM, there is no specific verification to check whether a mapping is well-defined or not.

In addition to standard matching, TOM provides a more powerful form of matching called *associative matching with neutral element* [7]. This gives more expressiveness to search for elements in an array or a list data-structure. In particular, this matching technique is well suited to implementing XML transformations. For a detailed presentation of the TOM language, the reader may refer to [5, 29].

2.5. Design requirements for the SL language

By adding pattern matching to Java, TOM identifies the need for a new kind of object: the notion of *transformation rule*. In this paper, we present a strategy language called SL designed to describe in a high-level way how to apply transformations in Java. The design and the implementation of this language have been strongly influenced by more than 10 years of research in this area, particularly by ELAN, ASF+SDF, Stratego, and JJTraveler. The requirements that served as design guidelines were the following:

- SL should be expressive enough to describe classical traversals such as *innermost* or *outermost*, as well as non-deterministic exploration strategies like *breadth-first search*,
- SL should be fully implementable in Java and thus usable in a Java program just like any other library (*i.e,* no dependency with the TOM language),

DOI: 10.1002/spe

- SL should be easy to use and smoothly integrated in a Java programming environment (*i.e,* extending the TOM language with dedicated declarative statements),
- SL should be easy to extend and applicable to a large number of application domains (*i.e,* usable with arbitrary data-structures).

3. A STRATEGY LANGUAGE FOR JAVA

In this section, we give a detailed presentation of the different constructs of the SL language and their semantics. SL is a framework written in Java where a strategy is an object of sort `Strategy`. Such a strategy `s` can be applied to a term `t` by evaluating `t2 = s.visit(t)`. The application of a strategy may *fail*, or returns a unique result `t2` corresponding to the application of `s` to the *root* of `t`. Taking the same terminology as the one proposed by ELAN and Stratego, a transformation rule is considered to be an *elementary strategy* and a *strategy* is an expression built over a *strategy language*. Strategies allow one to specify how rules should be applied. We call the term to which the strategy is applied the *subject*. To make the SL presentation lighter, we will present SL examples using TOM syntax.

3.1. Elementary strategies

The simplest strategies we can imagine are *identity* and *fail*. *Identity* is a strategy that can be applied to any term, and never fails. Conversely, the strategy *fail* always fails when applied to a term. In SL, a strategy is implemented by a Java class. The application of a strategy is fired by the `visit` method. To invoke a strategy, we first have to build it using a creation function, `new Identity()` for instance, but we can also use the "`" mechanism. Given an object `t`, the expression `` `Identity().visit(t) `` evaluates to `t`, whereas the evaluation of `` `Fail().visit(t) `` always fails. In SL, failure is represented by the exception `VisitFailure`. Therefore, the previous expression throws the exception `VisitFailure`.

In SL, a set of *transformation rules* (possibly reduced to a singleton) is also a strategy. It can be applied to a term to perform an elementary transformation step, returning the resulting term. In TOM, the definition of a set of rules is introduced by the keyword `%strategy`:

```
%strategy EvalPlus() {
  Plus(Cst(c),Cst(0)) -> Cst(c)
  Plus(Cst(0),Cst(c)) -> Cst(c)
}
```

The construct above declares a set of rules whose name is `EvalPlus`. Its application to an object `t` can be computed by `` `EvalPlus().visit(t) ``. In this example, we have considered two rewrite rules that simplify symbolic expressions. The expressions are built upon the algebraic signature described in Figure 1. In the following, we will consider this signature, which corresponds to abstract syntax trees (AST) of a toy programming language, to illustrate the SL constructs.

When applying the `EvalPlus()` strategy to a term `t`, the root symbol of `t` is inspected. When it is a `Plus`, the subterms are matched to see whether they correspond to a constant (*i.e,* a term whose root symbol is `Cst`). If one of them is `Cst(0)`, the expression is simplified. If none of these two rules match `t`, by default the application fails.

The `%strategy` construct has two syntaxes: the simple one presented above, and a generalized form where the name of a strategy can be parameterized by a list of arguments. The default behavior can be specified. For instance, we can decide that a strategy does nothing (*i.e,* the identity) when no rule can be applied. As in a `%match` construct, the right-hand side of a rule can be an arbitrary Java block of statements. Finally, the type of the patterns can be specified. Indeed, the strategies have to be *type preserving* to ensure that their applications always lead to a well-formed term. By specifying the type of a rule, we help the compiler to check that a strategy is type preserving. For instance, let us consider:

DOI: 10.1002/spe

```
Instruction = Sequence(i1:Instruction,i2:Instruction)
            | Declare(varname:String,e:Expression,i:Instruction)
            | Assign(varname:String,e:Expression)
            | Print(e:Expression)

Expression = Plus(e1:Expression,e2:Expression)
           | Mult(e1:Expression,e2:Expression)
           | Var(name:String)
           | Cst(n:int)
```

Figure 1. Abstract syntax of a simple imperative language that will be used in the rest of the paper to illustrate each concept. — We consider four instructions: a sequence, a declaration, an assignment, and a print statement. The `Declare(v,e,i)` instruction defines a variable `v` whose scope is restricted to the evaluation of the instruction `i`. A variable can be reassigned in its scope with the `Assign` instruction. A program is said to be *well-formed* when `Assign` is only used in the scope of a variable with the same name. In addition, `Declare(v,e,i)` should not be used if `v` is not fresh. *i.e,* in the scope of an already defined variable `v`. For example, the program `Declare("x",Cst(1),Declare("x",Cst(2),Print(Var("x"))))` is not well-formed, but `Declare("x",Cst(1),Declare("y",Cst(2),Print(Var("x"))))` is.

```
%strategy CollectVar(set:Set) extends Identity() {
  visit Expression {
    Var(v) -> { set.add(`v); }
  }
}
```

This construct defines a strategy called `CollectVar` that is parameterized by an argument `set` of sort `Set` (here `Set` is the interface in the Java Collections framework). The strategy extends `Identity`, meaning that the default behavior is to leave the subject unchanged when no Java `return` statement is performed. `return` or `break` statements can appear in the Java right-hand side of the rules (*i.e,* in the Java block that follows the arrow operator `->`). In contrast to functional rules, but similarly to the `switch/case` construct, several right-hand sides may be fired as long as no `return` or `break` is executed. The construction `visit Expression` specifies that the rule is only fired on an object of sort `Expression`. In this example, the right-hand side of the rule (`{ set.add(`v); }`) performs a side-effect on the object `set`, but no `return` is performed, leaving the subject unchanged. When the pattern `Var(v)` matches the subject, the name of the variable, which is stored in `v`, is added to the `set` given as an argument. This elementary strategy becomes more interesting when combined with *traversal* strategies. For instance, consider the example below, where the application of `CollectVar` in a *top-down* way collects the set of variable names that appear in the subject `e`.

```
Set bag = new HashSet();
Expression e = `Plus(Var("x"),Mult(Cst(2),Var("a")));
`TopDown(CollectVar(bag)).visit(e);
```

First, the node `Plus` is traversed, the pattern `Var(v)` does not match, so the `Identity` strategy is applied (nothing is done); the left subtree is traversed, `Var("x")` is matched (`v` is assigned to `"x"`), so `"x"` is added to the set `bag`. The second subterm is then traversed, recursively nothing is done until `Var("a")` is traversed, and then `"a"` is added to `bag`. This results in `bag = {"x", "a"}`.

3.2. Combinators

By combining elementary strategies, more complex strategies can be built. In SL we have considered the list of combinators presented in Figure 2. These combinators correspond to the ones that already

Prepared using speauth.cls
DOI: 10.1002/spe

Signature	Description
`Identity()`	always succeeds and returns the subject unchanged.
`Fail()`	always fails.
`Sequence(s1,s2)`	applies `s2` to the result of the application of `s1`. Fails when `s1` fails.
`Choice(s1,s2)`	first tries to apply `s1`. If `s1` fails, it applies `s2`.
`Not(s)`	fails when `s` succeeds. It performs the identity when `s` fails.
`IfThenElse(s1,s2,s3)`	applies `s2` to the original subject if `s1` succeeds. Otherwise it applies `s3` to the original subject.
`One(s)`	looks for an immediate subterm of the subject on which `s` succeeds and performs the application. When no such subterm exists or when applied to a constant, the strategy fails.
`All(s)`	applies `s` to *all* the subterms of the subject. It fails when `s` fails on a subterm. `All(s)` always succeeds when applied to a constant.
`Mu(vname,s)`	is an abstractor used to encode recursion.
`MuVar(vname)`	is a reference to the abstracted variable.
`Omega(i,s)`	applies `s` to the `i`-th subterm of the subject. If this position does not exist, the strategy fails.
`Up(s)`	applies `s` to the parent traversed to reach the subject.

Figure 2. SL core strategies. In the signature definition, the parameters named `s`, `s1`, `s2` and `s3` are of type `Strategy`, the parameter named `vname` is of type `String` and the parameter named `i` is of type `int`.

exist in Stratego and JJTraveler, except `Omega` and `Up` which are new, and `Mu` which does not exist in JJTraveler.

The strategy `Omega(i,s)` applies `s` to the `i`-th subterm of the subject. If this position does not exist, the strategy fails. For example, applying `Omega(2,Identity())` to `Cst(0)` fails because there is no 2nd subterm. The strategy `Up(s)` applies `s` to the immediate parent of the subject. These kinds of combinators are key for defining compiler-like analyses that need contextual information (*e.g.*, name binding that associates each identifier to the corresponding object declaration). Contrary to the other combinators, `Up(s)` depends on the global subject and can potentially change its structure. For example, applying `Up(EvalPlus())` to the first subterm of `Plus(Cst(1),Cst(0))` (*i.e,* `Cst(1)`) results in `Cst(1)`. The position of application can also potentially disappear in the resulting global subject. In this case, the strategy always fails. For example, applying `Up(EvalPlus())` to the second subterm of `Plus(Cst(1),Cst(0))` (*i.e,* `Cst(0)`) fails because there is no 2nd subterm in the resulting global subject. Similarly to the `All` combinator that always succeeds when applied to a constant, the `Up` combinator always succeeds when applied to the root of the global subject.

3.3. Composed strategies

With the combinators given in Figure 2 we can define a strategy named `Try`, parameterized by a strategy `s`, which tries to apply `s` and applies the identity when `s` fails: `Try(s) = Choice(s,Identity())`.

Using recursion the `Repeat(s)` strategy can be defined. It applies `s` as many times as possible, until a failure occurs (`Repeat(s)` does not terminate when `s` never fails). The last result before the failure is returned. `Repeat(s)` could be defined by `Repeat(s) = Try(Sequence(s,Repeat(s)))`. In the following we use another presentation that introduces a recursive operator: `Repeat(s) = Mu("x",Try(Sequence(s,MuVar("x"))))`. The idea is the same, `MuVar("x")` means that the strategy labeled by `"x"` (the current one in this example) is called recursively.

DOI: 10.1002/spe

Most of the classical reduction strategies can also be defined using the generic traversal operators `One` and `All`:

```
OnceBottomUp(s) = Mu("x",Choice(One(MuVar("x")),s))
BottomUp(s) = Mu("x",Sequence(All(MuVar("x")),s))
OnceTopDown(s) = Mu("x",Choice(s,One(MuVar("x"))))
TopDown(s) = Mu("x",Sequence(s,All(MuVar("x"))))
Innermost(s) = Repeat(OnceBottomUp(s))
Innermost'(s) = Mu("x",Sequence(All(MuVar("x")),
                                Try(Sequence(s,MuVar("x")))))
```

The strategy `OnceBottomUp` tries to apply the strategy `s` once, starting from the leftmost-innermost leaves. `BottomUp` behaves almost like `OnceBottomUp` except that `s` is applied to all nodes, starting from the leaves. Note that in the latter case the application of `s` should not fail, otherwise the whole strategy also fails. The strategy `Innermost` applies `s` as many times as possible, starting from the leaves. This construct is useful to compute normal forms (*i.e,* terms that cannot be further rewritten). `Innermost'` is equivalent to `Innermost` but is more efficient [12].

With these definitions, we can combine `OnceBottomUp` with `EvalPlus`. For example, `` `OnceBottomUp(EvalPlus()).visit(`Plus(Cst(0),Plus(Cst(1),Cst(0)))) `` evaluates to `Plus(Cst(0),Cst(1))`. To obtain a normal form, we consider the strategy `Innermost(EvalPlus())`, whose evaluation results in `Cst(1)`.

3.4. Operational semantics

In this section we give the operational semantics of the strategy combinators. We briefly recall the concepts and notations needed for the definition of the semantics. For a more formal presentation of these concepts, the reader can refer to [7]. A signature $\mathcal{F}$ is a set of function symbols, each one having a fixed arity. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of *terms* built from a signature $\mathcal{F}$ and a set $\mathcal{X}$ of variables. If no variable occurs in a term t, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ denotes the set of ground terms. A *position* ω in a term t is a finite sequence of natural numbers that allows one to identify a subterm in t. We use ϵ for the empty sequence, denoting the empty path to the root. The subterm *u* of a term *t* at position ω is denoted $t_{|\omega}$, where ω describes the path from the root of *t* to the root of *u*. For example, consider the term $t =$ `Plus(Mult(Cst(1),Cst(2)),Cst(3))`. The expression $t_{|1.2}$ is the `Cst(2)` expression. $symb(t)$ denotes the root symbol of t. For example, $symb(t_{|1.2})$ is the `Cst` symbol. By $t[u]_\omega$, we express the term *t* where the subterm at position ω has been replaced by *u*. σ denotes a *substitution*: $\sigma(t)$ corresponds to the term t where each occurrence of its variables (for instance x) are replaced by their instance (for example $\sigma(x)$).

The presentation of the SL semantics differs from the presentation of the Stratego semantics by making explicit the application context. Indeed a reduction step is indicated by $\langle s, t, \omega\rangle \rightarrow r$ where s is the global strategy to apply, t is the global subject, ω is the position in the subject where the current strategy is applied, and r is the resulting term or `Fail`. Making explicit the application position allows one to properly define the evaluation of the `Up` operator that depends on the global subject. Moreover, as SL strategies are terms (this feature is detailed in paragraph 3.6.2), the semantics indicates explicitly at which position of the global strategy the application is realized. This position is denoted by underlining, in the global strategy s, the corresponding subterm that is under evaluation (*i.e,* the current strategy). Maintaining the global strategy during the evaluation allows one to properly define the evaluation of the `MuVar` operator that depends on the upper `Mu` binder operator. In each inference rule, context variables (denoted $S\{\ldots\}$) are used in the strategy term to indicate that the current strategy is not necessarily the global strategy. The application of a strategy `s` on a subject `t` starts with the following reduction step $\langle \underline{s}, t, \epsilon\rangle \rightarrow r$ where both the application position in the subject and the position in the global strategy are the root position (denoted respectively by ϵ and $\underline{s}$).

To present the SL operational semantics in Figure 3, we separate the language into three sets of operators:

DOI: 10.1002/spe

- *elementary strategies* that are applied to the current application position. For example, the inference rule of the `Identity` strategy (rule id in Figure 3) consists of returning the global term t unchanged. This rule is applied when the current strategy equals `Identity` (denoted by $S\{\overline{Identity}\}$).
- *control combinators* that compose strategies but do not modify the current application position. For example, the inference rule of the `MuVar` combinator (rule $muvar$) consists of evaluating the corresponding upper `Mu` binder, allowing recursion. The evaluation of a `MuVar` operator does not change the current application position but only the current strategy. In this rule, a second context variable named K indicates the context between the `Mu` binder and the `MuVar` variable. As this rule can be applied for any strategy position where a `MuVar` combinator appears, the resulting global strategy term is $S\{Mu(\text{“}x\text{”}, K\{\overline{MuVar(\text{“}x\text{”})}\})\}$. To simplify the presentation of the semantics, we follow Barendregt's variable convention [31] (to avoid unexpected variable binding): given a strategy of the form $Mu(\text{“}x\text{”}, K\{MuVar(\text{“}x\text{”})\})$, there is no other $Mu(\text{“}x\text{”}, \ldots)$ in the path between the root position and the variable $MuVar(\text{“}x\text{”})$ in the context K. It is always possible to ensure this convention by renaming bound variables.
- *traversal combinators* that modify the current application position. These operators are also known as *position transformation operators*, as in [32] for instance. For example, the `Up` combinator is defined by four inference rules. The rule up_1 defines the base case where the current strategy is successfully applied at the upper position (denoted ω) resulting in a new term t' where the starting position $(\omega \mathbin{.} i)$ still exists (enforced by the condition $(\omega \mathbin{.} i) \in Pos(t')$). If the application of `Up` is at the root position, the strategy always succeeds (rule up_2). In the other cases (rules up_3 and up_4), the strategy fails.

Notice that only the rewriting rules and the `One` strategy operator have a non deterministic semantics. For a rewriting rule, only rules with equational operators can lead to several substitutions. In practice, the system chooses in a deterministic way one of the substitutions. Similarly, the implementation of `One` is made deterministic by evaluating subterms from left to right.

3.5. Congruence and construction strategies

TOM features a specific statement for describing typed tree-structures [33]. Given a signature, it generates a Java implementation and also provides support for strategic programming. In particular, congruence and construction strategy operators are automatically generated. Congruence and construction strategies are respectively used to discriminate constructors and to construct terms at the strategy level. Consider the following signature:

```
List = Cons(head:Element,tail:List) | Empty()
```

The strategies `_Cons` and `_Empty` (prefixed by `_`) denote congruence operators associated to `Cons` and `Empty`. They are automatically generated from the signature. The arity of a congruence operator is the same as its corresponding construction operator and all its arguments are strategies. Their semantics are defined in Figure 4 (informally, when `_Cons(s1,s2)` is applied on a term rooted by `Cons`, it applies `s1` to the first child, and `s2` to the second one).

For example, *map*, that is a classical higher-order function, can be encoded by the following composed strategy:

```
Map(s) = Mu("x",Choice(_Cons(s,MuVar("x")),_Empty()))
```

where `s` is the strategy to be applied to each element of the list. When the list is empty, the first strategy of the choice fails, but the second one succeeds, performing the identity. When the list is not empty, it means that its root symbol is `Cons`. In that case, the strategy `s` is applied to the head of the list and the current strategy (*i.e,* $Map(s)$, denoted by `MuVar("x")`) is applied recursively to the tail of the list.

Congruence strategies are commonly used to construct strategies whose behavior depends on the context (*i.e,* the shape of the term they are applied to). Such situations are frequent in a compiler.

DOI: 10.1002/spe

Elementary strategies

$$\frac{\exists \sigma, \sigma(lhs) = t_{|\omega}}{\langle S\{\underline{lhs \to rhs}\}, t, \omega\rangle \to t[\sigma(rhs)]_\omega} \ (\mathbf{r_1})$$

$$\frac{\nexists \sigma, \sigma(lhs) = t_{|\omega}}{\langle S\{\underline{lhs \to rhs}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{r_2})$$

$$\frac{}{\langle S\{\underline{Identity}\}, t, \omega\rangle \to t} \ (\mathbf{id})$$

$$\frac{}{\langle S\{\underline{Fail}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{fail})$$

Control combinators

$$\frac{\langle S\{Choice(\underline{s_1}, s_2)\}, t, \omega\rangle \to t'}{\langle S\{\underline{Choice(s_1, s_2)}\}, t, \omega\rangle \to t'} \ (\mathbf{choice_1})$$

$$\frac{\langle S\{Choice(\underline{s_1}, s_2)\}, t, \omega\rangle \to \texttt{Fail} \quad \langle S\{Choice(s_1, \underline{s_2})\}, t, \omega\rangle \to r}{\langle S\{\underline{Choice(s_1, s_2)}\}, t, \omega\rangle \to r} \ (\mathbf{choice_2})$$

$$\frac{\langle S\{Sequence(\underline{s_1}, s_2)\}, t, \omega\rangle \to t' \quad \langle S\{Sequence(s_1, \underline{s_2})\}, t', \omega\rangle \to r}{\langle S\{\underline{Sequence(s_1, s_2)}\}, t, \omega\rangle \to r} \ (\mathbf{seq_1})$$

$$\frac{\langle S\{Sequence(\underline{s_1}, s_2)\}, t, \omega\rangle \to \texttt{Fail}}{\langle S\{\underline{Sequence(s_1, s_2)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{seq_2})$$

$$\frac{\langle S\{ITE(\underline{s1}, s2, s3)\}, t, \omega\rangle \to \texttt{Fail} \quad \langle S\{ITE(s1, s2, \underline{s3})\}, t, \omega\rangle \to r}{\langle S\{\underline{ITE(s1, s2, s3)}\}, t, \omega\rangle \to r} \ (\mathbf{if_1})$$

$$\frac{\langle S\{ITE(\underline{s1}, s2, s3)\}, t, \omega\rangle \to t' \quad \langle S\{ITE(s1, \underline{s2}, s3)\}, t, \omega\rangle \to r}{\langle S\{\underline{ITE(s1, s2, s3)}\}, t, \omega\rangle \to r} \ (\mathbf{if_2})$$

$$\frac{\langle S\{Mu(\text{“}x\text{”}, \underline{s})\}, t, \omega\rangle \to r}{\langle S\{\underline{Mu(\text{“}x\text{”}, s)}\}, t, \omega\rangle \to r} \ (\mathbf{mu})$$

$$\frac{\langle S\{\underline{Mu(\text{“}x\text{”}, K\{MuVar(\text{“}x\text{”})\})}\}, t, \omega\rangle \to r}{\langle S\{Mu(\text{“}x\text{”}, K\{\underline{MuVar(\text{“}x\text{”})}\})\}, t, \omega\rangle \to r} \ (\mathbf{muvar})$$

Traversal combinators

$$\frac{\exists i \in [1,n]\ \langle S\{One(\underline{s})\}, t, \omega \centerdot i\rangle \to t'}{\langle S\{\underline{One(s)}\}, t, \omega\rangle \to t'} \ (\mathbf{one_1})$$

$$\frac{\forall i \in [1,n] \langle S\{One(\underline{s})\}, t, \omega \centerdot i\rangle \to \texttt{Fail}}{\langle S\{\underline{One(s)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{one_2})$$

$$\frac{\forall i \in [1,n]\ \exists t_i \in \mathcal{T}(\mathcal{F}),\ \langle S\{All(\underline{s})\}, t, \omega \centerdot i\rangle \to t[t_i]_{\omega \centerdot i}}{\langle S\{\underline{All(s)}\}, t, \omega\rangle \to t[t_1]_{\omega \centerdot 1} \ldots [t_n]_{\omega \centerdot n}} \ (\mathbf{all_1})$$

$$\frac{\exists i \in [1,n],\ \langle S\{All(\underline{s})\}, t, \omega \centerdot i\rangle \to \texttt{Fail}}{\langle S\{\underline{All(s)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{all_2})$$

$$\frac{\langle S\{Omega(i, \underline{s})\}, t, \omega \centerdot i\rangle \to r \quad i \in [1,n]}{\langle S\{\underline{Omega(i, s)}\}, t, \omega\rangle \to r} \ (\mathbf{omega_1})$$

$$\frac{i \notin [1,n]}{\langle S\{\underline{Omega(i, s)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{omega_2})$$

$$\frac{\langle S\{Up(\underline{s})\}, t, \omega\rangle \to t' \quad (\omega \centerdot i) \in Pos(t')}{\langle S\{\underline{Up(s)}\}, t, \omega \centerdot i\rangle \to t'} \ (\mathbf{up_1})$$

$$\frac{}{\langle S\{\underline{Up(s)}\}, t, \epsilon\rangle \to t} \ (\mathbf{up_2})$$

$$\frac{\langle S\{Up(\underline{s})\}, t, \omega\rangle \to \texttt{Fail}}{\langle S\{\underline{Up(s)}\}, t, \omega \centerdot i\rangle \to \texttt{Fail}} \ (\mathbf{up_3})$$

$$\frac{\langle S\{Up(\underline{s})\}, t, \omega\rangle \to t' \quad (\omega \centerdot i) \notin Pos(t')}{\langle S\{\underline{Up(s)}\}, t, \omega \centerdot i\rangle \to \texttt{Fail}} \ (\mathbf{up_4})$$

Figure 3. Semantics of SL. In the semantics rules above, we have $n = ar(\, symb(t_{|\omega}))$. The meta variable t' denotes a term (that cannot be `Fail`), whereas the meta variable r denotes a result that can be either a well-formed term, or `Fail`. ITE is a shortcut for the $IfThenElse$ combinator.

For example, some of the standard optimizations are dedicated to specific statements (*e.g.*, loop optimizations). This restriction can be directly encoded in the strategy using congruence operators.

TOM also generates construction strategy operators (prefixed by `Make_`), which allow one to construct terms at the strategy level. Therefore the result of such strategies does not depend on the subject. For instance, `Make_Empty`, applied to any term, returns the empty list denoted `Empty`. Their semantics is defined in Figure 5.

These construction strategies, combined with congruence strategies can be used to implement rewrite rules as strategies. For instance, a rule $f(a, b) \to g(a, b)$ can be implemented by the following strategy:

```
Strategy rule = `Sequence(
  _f(_a(),_b()),
```


DOI: 10.1002/spe

Congruence combinators

$$\frac{\langle S\{\_Cons(\underline{s_1}, s_2)\}, t, \omega \cdot 1\rangle \to t[t_1]_{\omega \cdot 1} \quad \langle S\{\_Cons(s_1, \underline{s_2})\}, t, \omega \cdot 2\rangle \to t[t_2]_{\omega \cdot 2} \quad symb(t_{|\omega}) = \texttt{Cons}}{\langle S\{\underline{\_Cons(s_1, s_2)}\}, t, \omega\rangle \to t[\texttt{Cons}(t_1, t_2)]_\omega} \ (\mathbf{cons_1})$$

$$\frac{\langle S\{\_Cons(\underline{s_1}, s_2)\}, t, \omega \cdot 1\rangle \to \texttt{Fail} \quad symb(t_{|\omega}) = \texttt{Cons}}{\langle S\{\underline{\_Cons(s_1, s_2)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{cons_2})$$

$$\frac{\langle S\{\_Cons(s_1, \underline{s_2})\}, t, \omega \cdot 2\rangle \to \texttt{Fail} \quad symb(t_{|\omega}) = \texttt{Cons}}{\langle S\{\underline{\_Cons(s_1, s_2)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{cons_3})$$

$$\frac{symb(t_{|\omega}) \neq \texttt{Cons}}{\langle S\{\underline{\_Cons(s_1, s_2)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{cons_4})$$

$$\frac{symb(t_{|\omega}) = \texttt{Empty}}{\langle S\{\underline{\_Empty()}\}, t, \omega\rangle \to t} \ (\mathbf{empty_1})$$

$$\frac{symb(t_{|\omega}) \neq \texttt{Empty}}{\langle S\{\underline{\_Empty()}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{empty_2})$$

Figure 4. Semantics of the congruence combinators $\_Cons$ and $\_Empty$.

Construction combinators

$$\frac{\langle S\{Make\_Cons(\underline{s_1}, s_2)\}, t, \omega\rangle \to t[t_1]_\omega \quad \langle S\{Make\_Cons(s_1, \underline{s_2})\}, t, \omega\rangle \to t[t_2]_\omega}{\langle S\{\underline{Make\_Cons(s_1, s_2)}\}, t, \omega\rangle \to t[Cons(t_1, t_2)]_\omega} \ (\mathbf{make\_cons_1})$$

$$\frac{\langle S\{Make\_Cons(\underline{s_1}, s_2)\}, t, \omega\rangle \to \texttt{Fail}}{\langle S\{\underline{Make\_Cons(s_1, s_2)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{make\_cons_2})$$

$$\frac{\langle S\{Make\_Cons(s_1, \underline{s_2})\}, t, \omega\rangle \to \texttt{Fail}}{\langle S\{\underline{Make\_Cons(s_1, s_2)}\}, t, \omega\rangle \to \texttt{Fail}} \ (\mathbf{make\_cons_3})$$

$$\frac{}{\langle S\{\underline{Make\_Empty()}\}, t, \omega\rangle \to t[Empty()]_\omega} \ (\mathbf{make\_empty})$$

Figure 5. Semantics of the construction combinators $Make\_Cons$ and $Make\_Empty$.

```
  Make_g(Make_a(),Make_b())
);
```

The interest of such encoding is to allow dynamic transformation of rules, as shown in the subsection 3.6.2. For example, the behavior of a self-adaptive system can be specified with a set of dynamic rules [34]. The rules specify the basic behavior of the system and then can be dynamically adapted in reaction to a change of the external environment. This dynamic adaption is encoded using reflexivity mechanisms.

3.6. Extensions

In this section, we present several extensions of the SL language, improving its expressiveness and time efficiency:

- term positions as first order objects,
- strategy reflection,
- efficient strategy combinators based on the behaviour of the `Identity` strategy.

DOI: 10.1002/spe

3.6.1. Explicit positions. In comparison with ELAN or Stratego, a notable innovation of the TOM strategy language is the ability for each strategy, including elementary ones that correspond to transformation rules, to know where they are applied, *i.e,* the *position*, *wrt.* the global term. These positions are represented by first order objects, *i.e.* objects of the class `Position` from the SL library, and can be used to build strategies such as "*replace at position* ω" or "*get subterm at position* ω".

For example, the `Position` class contains the `Strategy getOmega(Strategy v)` method. This method returns a strategy that applies `v` to the subterm referenced by the `Position` object. Using `getOmega`, it is possible to write in a concise way the replacement of a subterm (given by a position) by another term t. One only has to get the position `p` of the subterm to replace, and use the strategy `p.getOmega(s)` where `s` is a strategy implementing the rewrite system $x \to t$ (where x is a variable and t is a term).

In a `%strategy` statement, the `getPosition()` method allows one to get the current application position. For instance, the strategy below collects, in a Java set, the positions where the variable whose name is `nameVar` appears.

```
%strategy CollectVarPos(set:Set, nameVar:String) {
  visit Expression {
    Var(name) -> {
      if ('name.equals(nameVar)) {
        set.add(getPosition());
      }
    }
  }
}
```

Suppose we apply `TopDown(CollectVarPos(bag,"x"))` (where `bag` is an empty set) on the term `Plus(Mult(Var("x"),Var("x")),Var("y"))`. It results in $bag = \{1.1, 1.2\}$.

When dealing with transformation systems producing several results, a typical analysis that is performed is reachability analysis [35]. This analysis consists in computing the set of all the possible successors and then verifying whether a specific set of terms (representing in general the bad states of the system) is accessible. Given a rewrite rule $l \to r$ and a subject t, t' is a successor of t *wrt.* $l \to r$ if there exists a position ω and a substitution σ such that $\sigma(l) = t_{|\omega}$ and $t' = t[\sigma(r)]_\omega$.

To solve this problem, we have to find all the positions where the rule can be applied, and for each of these compute all the substitutions that solve the matching problem. Contrary to other rule-based languages, the reification of the notion of position makes the implementation of this task possible in TOM. Indeed, to compute the set of all successors of t, we consider the top-down application of a strategy parameterized by the term t itself and a collection `bag`. For each position ω where the rule can be applied, we store $t[\sigma(r)]_\omega$ in `bag`, using `getPosition()` to obtain ω and `replace(t,p,u)` to perform the replacement of the subterm in `t` at the position `p` by `u`:

```
%strategy Collect(t:Expression, bag:Collection) extends Identity() {
  visit Expression {
    Plus(Cst(c1),Cst(c2)) ->
      { bag.add(replace(t, getPosition(), `Cst(c1 + c2))); }
    Mult(Cst(c1),Cst(c2)) ->
      { bag.add(replace(t, getPosition(), `Cst(c1 * c2))); }
  }
}
Expression e = `Plus(Mult(Cst(1),Cst(2)),Plus(Cst(3),Cst(4)));
`TopDown(Collect(e,bag)).visit(e);
```

The resulting `bag` contains all the intermediate results after one rewriting step: `Plus(Cst(2), Plus(Cst(3),Cst(4)))` and `Plus(Mult(Cst(1),Cst(2)),Cst(7))`. This reification of the notion of position to the object level is new in the domain of rewrite-based languages like

DOI: 10.1002/spe

MAUDE or Stratego, and it adds expressiveness while keeping programs close to their specification. Moreover, this feature is key to describe tools and analyzers that deal with reachability or control flow analysis problems.

3.6.2. Reflective strategies. The mapping mechanism of TOM allows one to perform matching against any kind of term implementation, and in particular against the objects that represent a strategy. Therefore, strategy expressions, including the recursion operator, are first-class objects. TOM can be used to match, transform, or dynamically create any strategy expression. A strategy is also traversable, in such a way that rules and strategies can be applied on a strategy itself. In generic-programming libraries for generalist languages such as HASKELL [27], strategies are usually implemented by functions and thus cannot be decomposed or traversed as any term.

These reflective capabilities can be used to perform optimizations or on the fly compilation of dynamically created strategies. For example, suppose that we want to normalize composed strategies into more efficient forms. For example, given a strategy `s`, `Sequence(Identity(), s)` is equivalent to `s` but is less efficient so we should normalize `Sequence(Identity(), s)` into `s`. As TOM strategies are first-class citizens, such normalization function can be naturally encoded by rewrite rules:

```
%strategy Optimize() extends Identity() {
  Sequence(Identity(), x) -> x
  Choice(Fail(), x)       -> x
  Not(One(Not(x)))        -> All(x)
  Not(All(Not(x)))        -> One(x)
}
```

Using construction and congruence strategies, this naturally allows for the definition of dynamic rules. By defining two parameterized strategies `Get` and `Set` (which manage a Java `HashMap`), we can easily implement the rewrite rule $f(x) \to g(x)$ by the strategy `Sequence(_f(Set("x")),Make_g(Get("x")))`.

Using such encoding, we can now formulate meta rules such as $f(X) \to g(Y) \Rightarrow g(X) \to f(Y)$. This meta rule transforms for example the rule $f(g(x)) \to g(x)$ (encoded by `Sequence(_f(_g(Set("x"))),Make_g(Get("x")))`) into the rule $g(g(x)) \to f(x)$. This meta rule can be encoded by the following strategy based on strategy reflection:

```
%strategy MetaRule() extends Identity() {
  visit Strategy {
    Sequence(_f(X),Make_g(Y)) -> Sequence(_g(X),Make_f(Y))
  }
}
```

Applying `MetaRule()` on `Sequence(_f(_g(Set("x"))),Make_g(Get("x")))`, we obtain `Sequence(_g(_g(Set("x"))),Make_f(Get("x")))`, The X and Y variables are respectively instantiated by `_g(Set("x"))` and `Get("x")`. Such encoding is useful when a set of rules needs to be modified dynamically according to the execution context.

Reflexivity is useful for example to specify the adaptation level of a self-adaptive system, as shown in [34]. In this work, the authors have used rewriting rules to specify the basic behavior of a robot and then have applied the reflexivity mechanism of MAUDE to encode dynamic adaptation in reaction to a change of the external environment. A similar encoding could be realized in TOM using meta rules.

3.6.3. Identity as failure. In Figure 2, we have introduced several combinators whose semantics depends on the failure of the strategy given as an argument. For example, `Choice(s1,s2)` applies `s2` when the application of `s1` fails. In SL (and in other implementations such as JJTraveler or Stratego), failure is implemented using exceptions (or `setjmp/longjmp` in C). In some strategies, failure is used only as a control mechanism [36]. For example, when evaluating the

DOI: 10.1002/spe

$$\frac{\langle S\{SequenceId(\underline{s_1}, s_2)\}, t, \omega\rangle \to t}{\langle S\{\underline{SequenceId(s_1, s_2)}\}, t, \omega\rangle \to t}\ (\mathbf{seqid_1})$$

$$\frac{\langle S\{SequenceId(\underline{s_1}, s_2)\}, t, \omega\rangle \to \texttt{Fail}}{\langle S\{\underline{SequenceId(s_1, s_2)}\}, t, \omega\rangle \to \texttt{Fail}}\ (\mathbf{seqid_2})$$

$$\frac{\langle S\{SequenceId(\underline{s_1}, s_2)\}, t, \omega\rangle \to t' \quad \langle S\{SequenceId(s_1, \underline{s_2})\}, t', \omega\rangle \to r \quad t \neq t'}{\langle S\{\underline{SequenceId(s_1, s_2)}\}, t, \omega\rangle \to r}\ (\mathbf{seqid_3})$$

Figure 6. Semantics of `SequenceId`, an example of identity-based combinators

`Innermost` strategy, a failure occurs each time a subterm is no longer reducible. In this case, the exception encoding can be too costly [37].

In order to improve runtime performance, SL has been extended with several original combinators whose semantics is based on *identity* (*i.e,* when no modification occurs) instead of failure. This considerably reduces the need for exceptions, and thus improves runtime performance [37]. For instance, let us consider `SequenceId`, which is like `Sequence` but behaves like `Identity` when the first strategy does not modify the subject. Its semantics is defined in Figure 6.

Using this new combinator, we can define a more efficient leftmost-innermost normalization strategy:

```
InnermostId(s) =
  Mu("x",Sequence(All(MuVar("x")),SequenceId(s,MuVar("x"))))
```

In this case, the strategy `s` must be an unfailing strategy, otherwise, the `InnermostId` strategy would stop just after the first failure.

3.7. Summary

In this section, we have presented the SL strategy language. This language can be used in a plain Java program or using specific declarative statements of the TOM language.

The main originality of TOM extended with SL statements is to mix Java (the right hand side of the rules can be any Java block of statements) and algebraic features (*i.e,* pattern matching and strategies). Unlike usual program transformation systems, strategies can be mixed with any Java feature, allowing more flexibility in the control of the transformations.

In comparison with usual strategy languages such as Stratego or JJTraveler, another originality of SL is to explicitly offer the concept of position so often used in rule-based formalisms. This provides a better understanding of strategy application by making explicit the context of application. Context-aware strategies have been investigated in the domain of graph rewriting [38] and XML transformation [20] but to our knowledge, never in term-rewriting languages such as ELAN or Stratego. Another consequence is the knowledge of the current application position's parent. This concept of parent traversal axes has already been investigated in generic programming libraries such as SYB on the basis of XPath-like combinators [20]. However, to preserve the purity of functional programming, this context information was read-only. On the contrary, SL supplies an original strategy combinator that applies the strategy to the parent of the current application position (inversely to `All` and `One` combinators) and thus can have side-effect. Such combinators are key for defining compiler-like analyses that need contextual information (*e.g.*, name binding that associates each identifier to the corresponding object declaration).

4. PROGRAMMING WITH STRATEGIES AND JAVA

In this section, we demonstrate how SL helps to produce clear and maintainable code. To illustrate the capabilities of the SL language, we show how to implement program optimizations such as

DOI: 10.1002/spe

constant propagation and inlining of variables for the tiny language presented in Figure 1. These examples demonstrate how mixing strategies and Java helps the gathering of information in complex data-structures. The resulting code is more declarative and the set of generic strategy combinators avoids code duplication. Indeed the user has only to write the code part depending on the data-structure as all the traversal and control of the transformations are expressed with generic traversal combinators.

4.1. Propagating constant expressions

A commonplace optimization consists in propagating constants and then eliminating dead code corresponding to unused variable definitions. To simplify the problem, we only consider programs without the `Assign` statement. For example, the program

```
Declare("x",Cst(3),
  Declare("y",Plus(Var("x"),Cst(1)),
    Print(Plus(Var("x"),Var("y")))
  )
)
```

should be optimized into `Print(Plus(Cst(3),Plus(Cst(3),Cst(1))))`. In general, this optimization is followed by *constant folding* that simplifies constant expressions at compilation time. In this example, after constant folding, we obtain `Print(Cst(7))`.

One step of constant propagation corresponds to the strategy `TopDown(PropagateCst(m))`. As we can give Java data-structures as parameters of a `%strategy`, `PropagateCst` can keep the association between the variables and their expressions and propagate them during the top-down traversal. In this way, several variables can be inlined in one step. The strategy `PropagateCst` detects instructions of type `Declare` by pattern matching. Then if the assignment expression is only composed of constants (corresponding to the result of the Java method named `isConstant`), the couple composed of the variable and the expression is added to the `HashMap` given as parameter and the declaration is replaced by its body.

```
%strategy PropagateCst(m:HashMap) extends Identity() {
  visit Instruction {
    Declare(v,e,b) -> {
      if (isConstant(`e)) {
        m.put(`v, `e);
        return `b;
      }
    }
  }

  visit Expression {
    Var(v) -> {
      if(m.containsKey(`v)) {
        return (Expression) m.get(`v);
      }
    }
  }
}

public Instruction propagateCst(Instruction i) {
  HashMap m = new HashMap();
  return `InnermostId(PropagateCst(m)).visit(i);
}
```


DOI: 10.1002/spe

Using the combinator `InnermostId` presented in Section 3.6.3, the propagation of constants is applied on the whole program until a fix point is obtained. In the example, the fix point is reached in two steps. First, we obtain:

```
Declare("y",Plus(Cst(3),Cst(1)),
  Print(Plus(Cst(3),Var("y")))
```

and finally `Print(Plus(Cst(3),Plus(Cst(3),Cst(1))))`.

In this small example, we showed how practical it is to use Java data-structures for carrying information. Further on, we will apply variable declaration inlining in more complex cases.

4.2. Inlining

If a variable is used only once, the *inlining* optimization consists of replacing the variable by the expression to which it is assigned. This improves the runtime performance of the program by avoiding the creation of intermediate variables. This also contributes to reducing the code size.

For example, the program

```
Declare("y",Plus(Var("x"),Cst(1)),
  Print(Var("y"))
)
```

should be optimized into `Print(Plus(Var("x"),Cst(1)))`.

To implement this optimization, we introduce a strategy named `Inlining`. First, this strategy detects `Declare(v,e,i)` instructions and computes the number of occurrences of the variable `v` in the instruction `i`. Then, if the variable is used only once, the declaration is replaced by `i` where the unique use of `v` has been substituted by its value `e`. This substitution is implemented by the strategy `Replace`.

```
%strategy Inlining() extends Identity() {
  visit Instruction {
    Declare(Var(name),value,inst) -> {
      if (computeOccurrences(`inst,`name) == 1) {
        return `TopDown(Replace(name,value)).visit(`inst);
      }
    }
  }
}

%strategy Replace(varName:String,value:Expression) extends Identity() {
  visit Expression {
    Var(name) -> {
      if (`name.equals(varName)) {
        return value;
      }
    }
  }
}
```

The `Inlining` strategy depends on the `computeOccurrences(v,i)` function that computes the number of occurrences of `v` in the block `i`. We first describe a naive version of the `computeOccurrences` function. The number of occurrences of the variables is computed using a simple strategy named `Count`. This strategy is applied in a top-down way using the `TopDown` strategy. `Count` is parameterized by a Java object of type `Info`. The class `Info` contains two public attributes: `varName` that corresponds to the name of the variable and `readCount`, the number of occurrences.

DOI: 10.1002/spe

```
public int computeOccurrences(Instruction i, String varName) {
  Info info = new Info(varName);
  `TopDown(Count(info)).visit(i);
  return info.readCount;
}

%strategy Count(info:Info) extends Identity() {
  visit Expression {
    Var(name) -> {
      if(info.varName.equals(`name)) {
        info.readCount++;
      }
    }
  }
}
```

Note that, in general, computing the number of occurrences is not sufficient to implement the inlining optimization. For instance, even if a variable `v` is used only once, the inlining of `v` by its assignment expression `e` is not possible anymore if `v` is modified in `i` (by `Assign`) or if `e` contains variables modified in `i`. Moreover, as this strategy `Count` is used only for inlining, we are not interested in computing numbers of occurrences greater than 1. Thus, to improve the strategy `Count`, failures can be thrown in two cases:

1. `v` or the variables contained in the expression `e` are modified,
2. the number of uses is greater than 1. As the inlining is no longer possible, it makes no sense to continue to compute the number of occurrences.

For this purpose, the class `Info` contains a new attribute `assignmentVars`, corresponding to the list of the variables used to define `v`. The strategy `Count` is modified as follows:

```
%strategy Count(info:Info) extends Identity() {
  visit Instruction {
    Assign(name,e) -> {
     if(info.assignmentVars.contains(`name) ||
        info.varName.equals(`name)) {
        throw new VisitFailure();
     }
    }
  }

  visit Expression {
    Var(name) -> {
     if(info.varName.equals(`name)) {
        info.readCount++;
        if (info.readCount > 1) { throw new VisitFailure(); }
      }
    }
  }

}
```

As the `Count` strategy fails if the inlining is not possible, we can directly use it in the `Inlining` strategy instead of the intermediate `computeOccurrences` function. In fact, if the `TopDown(Count(name))` strategy succeeds, it means that the variable is used only once in the program. Thus we can apply the `TopDown(Replace(name,value))` strategy on

DOI: 10.1002/spe

the `inst` subject. Otherwise, we use the construction strategy named `Make_declare` that allows us to return the original declaration. To finish, we discriminate these two cases by using the `IfThenElse` strategy combinator. This composed strategy named `CountAndReplace` is defined as follows:

```
CountAndReplace(info:Info, value:Expression, inst:Instruction) =
  IfThenElse(TopDown(Count(info)), //condition
             TopDown(Replace(info.varName,value)), //success
             Make_Declare(Var(info.varName),value,inst)) //failure
```

In the `Inlining` strategy, the call to the `computeOccurrences` function and the Java `IfThenElse` statement can now be replaced by the `CountAndReplace` strategy:

```
%strategy Inlining() extends Identity() {
  visit Instruction {
    Declare(Var(name),value,inst) -> {
      Info info = new Info(`name);
      return `CountAndReplace(info,value,inst).visit(`inst);
    }
  }
}
```

Notice that with this version of the inlining optimization, some valid programs cannot be inlined. Indeed it would be sufficient to verify that the variables are not modified from the beginning of the declaration to the unique use of the variable. In this version, we verify that the variables are not modified in the whole body of the declaration which is too strong. For example, the following declaration cannot be inlined:

```
Declare(Var("y"),Var("x"),
  Sequence( Print(Var("y")),
            Assign(Var("x"),Cst(2))
  )
)
```

We can refine this program using temporal conditions. Informally, we check that from the declaration, the variables are not modified until the declared variable is used and that from this point this variable is not used anymore. These kinds of complex conditions can be expressed in an elegant way using temporal logic [39]. The reader is invited to refer to [40] for details about encoding such conditions in SL.

With this example, we have seen how SL can be used to implement optimizers. A concrete illustration of this idea is the TOM optimizer itself [37], which is written in TOM and where SL strategies have been intensively used. This approach can be applied not only to implement optimizers but for any tasks that require transformations or the collection of information in complex data-structures. Therefore, using strategies instead of a direct implementation in Java avoids code duplication. Moreover, thanks to the piggybacking approach of the TOM language, the code is more declarative and thus more concise and easily maintainable.

5. DESIGN OF THE SL LIBRARY

The design of the SL library is largely inspired from the JJTraveler library. Even if their designs are roughly similar, the main originality of the SL library is to explicitly provide the `Mu` combinator (as in Stratego) and thus classical strategies like top-down are not built-in but defined by combining basic operators. The other major contribution is a notion of position and environment. Generally, strategy applications are black boxes and there is no way to know the current state of the term during the traversal, in particular the current position. In SL, a strategy is applied in an environment

DOI: 10.1002/spe

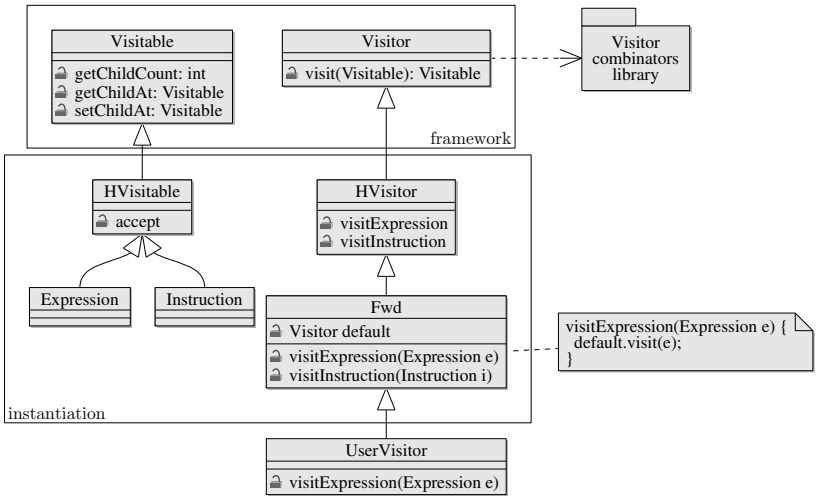

Figure 7. Class diagram of the JJTraveler library

representing the current state of the global subject (including the current application position), as defined in the semantics.

In this section, we compare the design of JJTraveler and SL and we detail the design of SL features such as introspection or reification of positions.

5.1. Background: JJTraveler

When using object-oriented languages, tree traversal is usually performed using a variant of the *Visitor* design pattern [1], the manipulated tree being an instance of the *Composite* pattern. While useful, this method suffers some limitations which the concept of visitor combinators, introduced by Joost Visser [4], bypasses. A first limitation is that visitors resist combination, and can only be specialized. A second is that the way the tree is traversed by the visitor is either chosen in the `accept` methods, and then fixed once and for all in the tree implementation, or programmed in the visitor itself, making visitor reuse harder. Visitor combinators provided by JJTraveler or SL are a solution to these problems.

JJTraveler consists of a framework that provides a minimal interface for nodes to be visited, `Visitable`, and a minimal interface for writing visitors, `Visitor`. As illustrated in Figure 7, using the JJTraveler system requires instantiating the framework for the class hierarchy to be traversed. This hierarchy is made visitable by implementing the `Visitable` interface. The `Visitor` interface is extended with visit methods for each class in the hierarchy, like in the `HVisitor` interface with the `visitA` and `visitB` methods.

To ease reuse of the library, a default implementation of the extended `Visitor` interface is provided, which forwards for each specific visit method the call to a generic default visitor, selected at construction time. Users' visitors are built using this forwarding visitor, which we call `Fwd`, by providing a default generic visitor, and then overriding some of the specific visit methods. The most common use is to provide `Fwd` with the `Identity` generic visitor, if on most nodes there is nothing to do, and then override the visit methods for the nodes requiring an action to be performed.

Using `Fwd`, the basic visitor combinators can be given a behavior specific to a node type. This can be done without depending on the whole tree-structure. Indeed adding a new node type only requires an extension of `Fwd`, not to each implementation of `Visitor`. This design allows description of visitor behavior using a *double dispatch* mechanism, since the code to execute is selected using both the visited object type and the visitor.

However, instantiating this framework can be difficult and error prone. The `Visitor` and `Visitable` interfaces, the accept methods and `Fwd` depend directly on the tree-structure. The

Prepared using speauth.cls
DOI: 10.1002/spe

JJForester [41] tool was used to generate such a tree implementation and the corresponding methods and classes from a syntax definition. In the case of SL, the use of visitors has been simplified. Indeed, as the data-structure is described by mappings, double-dispatch is not required. Unlike JJTraveler, the SL library does not need `Fwd` classes to be extensional.

The JJTraveler library provides a few primitive combinators (Identity, Fail, Sequence, Choice, All and One), and some more complex combinators defined using these primitive operators, which may be recursively defined. However, there is no support in the interface for building cyclic strategies. The choice was for the JJTraveler library to provide a wide range of complex visitor combinators that implement this recursion. Thus, instead of letting the user create the object corresponding to `Mu("x",Sequence(All(MuVar("x")),v))`, the strategy `BottomUp(v)` was directly offered as a built-in visitor combinator. This allowed the library implementors to implement the built-in visitor in a way slightly different from the original description, but had the drawback that it is difficult, and sometimes impossible, for the user of the library to implement a recursive strategy defined this way.

For instance, the `BottomUp(v)` operator is defined as the class:

```
class BottomUp extends Sequence {
  public BottomUp(Visitor v) {
    super(null,v);
    first = new All(this);
  }
}
```

With JJTraveler, in order for the user to build a recursive strategy, he has to rely on the fact that some visitor combinators do expose their sub-strategies. In our example, the `BottomUp` definition relies on the *sequence* operator exposing two sub-strategies, `first` being the first one. The *choice* operator, for instance, does not expose its `first` argument (it is package protected), making difficult the definition of `Mu("x",Choice(Sequence(v,All(MuVar("x")))),Identity())` (called `TopDownCollect`) without modifying the JJTraveler library. Assuming that fields corresponding to sub-strategies are public, the user has still to explicitly create the recursive strategy with an empty argument where the recursion has to take place, and then replace this argument with a reference to the recursive object. For the considered example, this would lead to the following code (not so easy to implement, introducing a cast expression):

```
class TopDownCollect extends Choice {
  public TopDownCollect(Visitor v) {
    super(new Sequence(v,null), new Identity());
    ((Sequence)first).then = new All(this);
  }
}
```

This procedure can only be done for the library operators that provide support for this, and lets the user create inconsistent strategies. Moreover, it is laborious to create complex compound strategies this way, while keeping the strategy constructor readable: the link between the formal description of the strategy and its implementation is lost.

Inspired by the Stratego language, the SL library implements the `Mu` operator as a `Visitor` class. Thus recursive strategies are naturally constructed by composition instead of being built-in.

5.2. Global architecture of SL

As in JJTraveler, the SL library is composed of two main interfaces: `Visitable` and `Strategy`. The `Strategy` interface corresponds to the JJTraveler `Visitor` interface. As illustrated in Figure 8, `Strategy` also extends `Visitable`, allowing strategy reflection.

Any SL strategy must implement the `Strategy` interface. The library offers a partial implementation of this interface named `AbstractStrategy`. Every SL combinator extends this abstract class.

DOI: 10.1002/spe

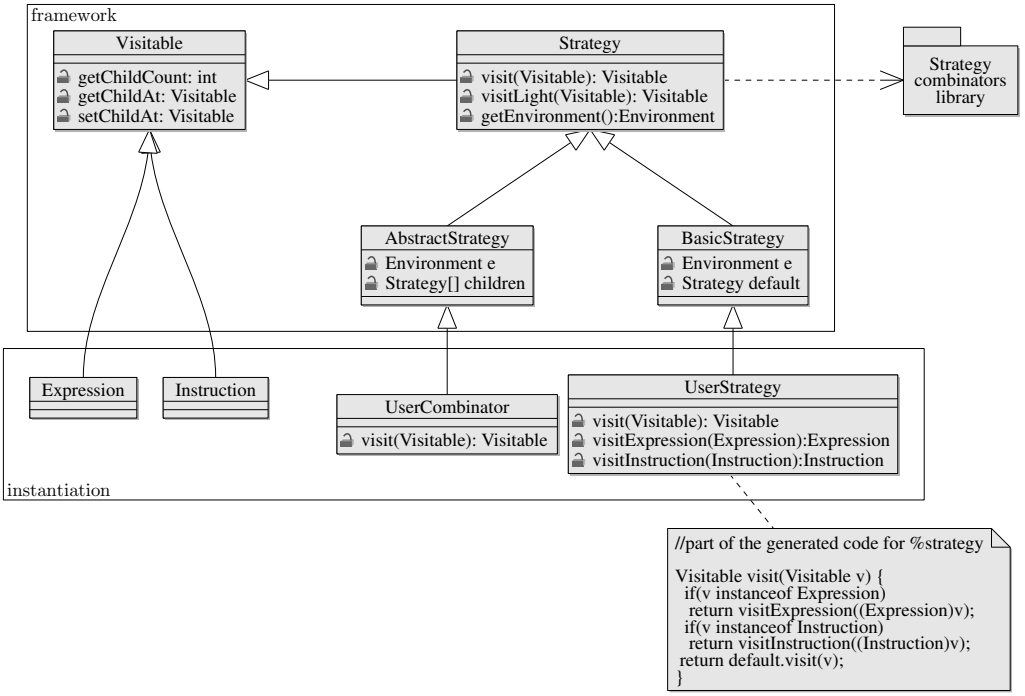

Figure 8. Class diagram of the SL library

To apply a strategy on a term (generally called the subject), this interface offers two methods:

- `<T extends Visitable> T visit(T any)` that visits the subject and updates the environment at each step. The environment is represented by an object (of type `Environment`) that represents the current position and the current state of the global subject. Inside a `%strategy`, the current environment can be accessed with the method `getEnvironment()`. This method is used in particular when the user needs to know the current application position during the traversal.
- `<T extends Visitable> T visitLight(T any)` that visits the subject in a light way (without maintaining an evaluation environment) and thus more efficiently.

As strategies are type preserving, these two methods return a `Visitable` object of the same type as the subject and in case of failures, they throw a `VisitFailure` exception. Contrary to JJTraveler, the SL library can be used either in a pure Java application or take advantage of the TOM constructions. In particular, the compilation of a `%strategy` construction corresponds to a static inner class that implements the `Strategy` interface. If the data-structure hierarchy has been automatically generated by TOM [33], the produced classes directly implement the `Visitable` interface, as shown in Figure 8. If the data-structure hierarchy is user defined and thus does not necessarily implement the `Visitable` interface, the SL library provides an introspection mechanism based on the mappings. The design of this mechanism is detailed below.

5.3. Visiting arbitrary data-structures by introspection

Until now, the strategy library was not restricted to a given term representation but the condition was that the manipulated data-structure implements the `Visitable` interface.

As this condition sometimes cannot be satisfied (*e.g.*, the data is defined in an external library, the source code is not available), the library offers a mechanism to support strategies in a general setting. The `Strategy` interface contains `visit` methods for objects that do not implement the `Visitable` interface:

DOI: 10.1002/spe

```
<T> T visit(T any, Introspector i)
<T> T visitLight(T any, Introspector i)
```

These two methods take an `Introspector` as argument. This object behaves like a proxy that renders any Java object visitable, by providing the following methods:

```
<T> T setChildren(T o, Object[] children)
Object[] getChildren(Object o)
<T> T setChildAt(T o, int i, Object child)
Object getChildAt(Object o, int i)
int getChildCount(Object o)
```

When programming in pure Java, users have to define their own implementation of `Introspector` that depends on their data-structures. For example, we define an implementation of `Introspector` that allows one to visit a hierarchy of Swing components:

```
public class SwingIntrospector implements Introspector {
  public <T> T setChildren(T o, Object[] children) {
    JComponent c = (JComponent) o;
    for(int i=0; i<children.length; i++) {
      c.add((JComponent)children[i]);
    }
    return (T) c;
  }

  public Object[] getChildren(Object o) {
    return ((JComponent)o).getComponents();
  }

  public <T> T setChildAt(T o, int i, Object child) {
    JComponent c = (JComponent) o;
    c.remove(i);
    c.setComponentZOrder((JComponent) child, i);
    return (T) c;
  }

  public Object getChildAt(Object o, int i) {
    return ((JComponent)o).getComponent(i);
  }

  public int getChildCount(Object o) {
    return  ((JComponent)o).getComponentCount();
  }
}
```

We can then easily use this introspector to change the background of all the components of a frame:

```
Strategy s = new AbstractStrategyBasic(new Identity()) {
  public <T> T visitLight(T any,Introspector i) {
    ((JComponent)any).setBackground(Color.CYAN);
    return any;
  }
};

`TopDown(s).visitLight(getContentPane(), new SwingIntrospector());
```


DOI: 10.1002/spe

The basic strategy `s` defines a simple transformation that sets the background of a Swing component into cyan. We then apply this basic strategy in a top-down way using the `TopDown` composed strategy and an instance of the `SwingIntrospector` we define above.

When programming in TOM with user-defined structures, the TOM compiler automatically generates an inner class that implements the `Introspector` interface. This uses information from the mappings to know how to visit the corresponding classes, enabling the users to program with strategies directly.

5.4. Implementation of the `Mu` operator

Contrary to JJTraveler, the definitions of the SL strategies with explicit recursive operators are close to the ones in Stratego [12]. For example, we can create the `BottomUp` strategy by writing a Java expression as close as possible to the original description:

```
Strategy BottomUp(Strategy s) {
  return new Mu( new MuVar("x"),
                 new Sequence(new All(new MuVar("x")),s) );
}
```

In this example, the `Mu` strategy traverses the tree represented by the Java object `new Sequence(new All(new MuVar("x")),s)`, and replaces the occurrence of `new MuVar("x")` by the root of the tree, in order to build a cyclic graph. This process is called *μ-expansion*. It requires that all strategy combinators implement the `Visitable` interface. To achieve this behavior, all strategy combinators have to implement the `Strategy` interface that inherits from the `Visitable` interface. This will let us use strategy combinators to describe the process of building a graph from the tree representation containing variables such as `MuVar("x")`. `MuVar` is itself a strategy combinator. Its role is to represent the recursive call in a recursively defined strategy. This strategy has no child in the `Visitable` sense, hence it is considered as a leaf by all strategy combinators. To achieve the recursive behavior, this `MuVar` strategy combinator has two fields, one being the name of the variable it represents, and the other being the recursive strategy itself. It is not possible to create such an object in a single step.

Thus, we create the `MuVar` with a `null` instance for the recursive strategy. Then, the μ-expansion phase has to walk through the strategy and replace all the `null` instances of `MuVar` with the recursive strategy.

All the other strategies are built using these core strategies (see Figure 8). Thus, the library is easily extensible, since only the core operators need to be changed to extend the behavior of all strategies. Our contribution is to promote the strategy recursion operator to a user primitive in order to allow the definition of recursive strategies by composition.

5.5. Extending the library

Since all strategy combinators are described using the core strategy combinators, it is easy to extend the behavior of all strategies in the library, including the user-defined strategies (by composing the library ones). In the SL library, all the *core* strategy combinators derive from the `AbstractStrategy` class, which implements the common behavior of all strategies described in the `Strategy` interface.

Suppose we want to add a new probabilistic choice operator `Pselect(p:int,q:int, s1:Strategy,s2:Strategy)` that applies `s1` with a probability of $\frac{p}{q}$ and `s2` with a probability of $\frac{q-p}{q}$. For that, we just have to write a single class that extends `AbstractStrategy`. Using the Java `Random` utility, this can be written in less than 30 lines of code, without any re-compilation of the system. This extensibility makes TOM an ideal platform to experiment with new paradigms. From this combinator, it is now possible to define a strategy that performs non deterministic choice:

```
NonDeterministicChoice(s1,s2) = Pselect(1,2,s1,s2)
```


DOI: 10.1002/spe

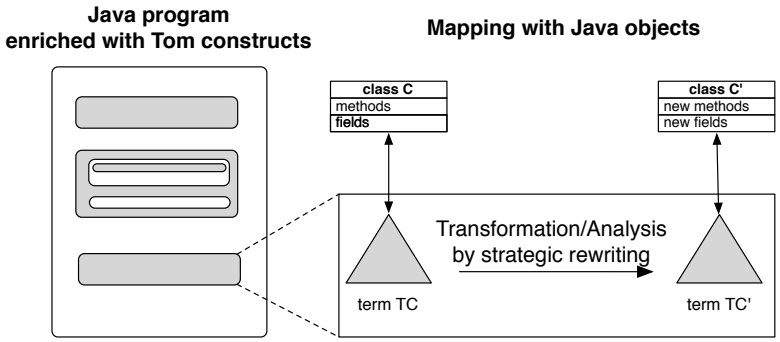

Figure 9. Development of a Java application enriched with TOM constructs. The grey color represents the TOM statements embedded in the Java code. As the two languages are deeply mixed, TOM constructs are only used in relevant situations and the concept of mappings [28, 29] ensures the link with Java objects.

Suppose that we want to test the compiler of the tiny language presented in Figure 1. In order to do this, we wish to generate random well-formed ASTs from the grammar. In the following we consider a generalization of `NonDeterministicChoice` that can take an arbitrary number of parameter strategies. This generator can be written using only SL strategies:

```
GenerateInstruction =
  Mu("x", NonDeterministicChoice(
      Make_Sequence(MuVar("x"),MuVar("x")),
      Make_Declare(GenerateName,GenerateExpression,MuVar("x")),
      Make_Assign(GenerateName,GenerateExpression),
      Make_Print(GenerateExpression)
  ))
```

where `GenerateName` is a basic strategy that randomly generates a name and `GenerateExpression` is a composed strategy similar to `GenerateInstruction` that randomly generates terms of type `Expression`. Each call to the `GenerateInstruction` strategy will construct a new random program. This example illustrates again the interest of the `Mu` constructor: it is for instance difficult to define the recursive `GenerateInstruction` strategy using the JJTraveler library.

6. APPLICATIONS

This section describes applications of the SL language coupled with TOM in a broad range of domains: from databases to model- driven engineering. In each example, the mappings and the introspectors respectively used in TOM and SL allow one to work directly on the user-defined data-structures without any adaptation.

With respect to the other rule-based languages, the main interest of the SL/TOM approach is that it piggybacks on Java, allowing smooth integration of declarative transformation code in existing Java applications. Figure 9 outlines the TOM and SL development approach. Even if the SL language can be used directly without TOM (as a Java API), the combination with TOM improves the readability of the code thanks to declarative constructs for pattern matching and strategic rules.

6.1. Persistence APIs

Persistence APIs like JPA [42] (Java Persistence API) are now ubiquitous. In JPA, an entity class represents a database table and each entity instance corresponds to a row in that table. Data analysis and manipulation is performed directly in Java. The resulting code is composed of nested *if-then-else*

Prepared using speauth.cls
DOI: 10.1002/spe

statements and getter calls, rendering the code quite illegible. As shown in [43], such applications can greatly benefit from a more declarative approach based on pattern matching and strategies. The code quality is improved, reducing the maintenance cost.

For example, suppose we develop an E-commerce application and that we want to access the list of favored clients. From a database schema describing client information, we get a class hierarchy (JPA offers several strategies to encode the inheritance between entities in the relational database model). In particular, this class hierarchy is composed of `Account` (with a field owner of type `Owner`), inherited by `FavoredAccount` (for favored clients) with a field of type Integer for loyalty points. Given an object named `site` describing all the application, we would like to collect all the owners' names of `FavoredAccount`s. The corresponding TOM code describes how to locally collect a name using pattern matching:

```
%strategy getFavored(set:Collection) extends Identity() {
  visit Account {
    FavoredAccount(Owner(name)) -> { set.add(`name); }
  }
}
```

Given a Java object `bag` of type `Collection`, we can apply this strategy in a *top-down* manner using the `TopDown(getFavored(bag))` strategy and thus collect all the information in `site`. The equivalent Java code (based on *for* and *if-then-else* statements and getter calls) would be:

```
for(Account account: site.getAccounts()) {
 if (account instanceof FavoredAccount) {
  set.add(account.getOwner().getName());
 }
}
```

If the database schema evolves (*e.g.*, the site is divided by departments and accounts are directly associated to a specific department), the TOM code needs no modification. On the contrary, the Java code would have to be adapted to the new schema:

```
for(Department department: site.getDepartments()) {
  for(Account account: department.getAccounts()) {
    if (account instanceof FavoredAccount) {
      set.add(account.getOwner().getName());
    }
  }
}
```

This example shows that the use of pattern matching and strategies improves the code quality and reduces the maintenance cost by clearly separating algorithms and data-structures. Such separation is critical in the context of persistence APIs where the evolution of the schemas can have a huge impact on the application code.

In [43], we show how this kind of TOM application can be directly integrated into a Java application using an Eclipse plugin that, among other things, automates the generation of TOM mappings from any Java class hierarchy.

6.2. Program transformation

As shown in Section 4, the SL language is suitable for program transformation, such as program optimization, and thus has been used for various applications requiring program transformation.

TOM compiler. Strategies were intensively used in the TOM compiler, particularly in the optimization phase. In [37], we describe the TOM optimizer in detail. The optimization rules were designed independently from any application strategy. After ensuring these rules were

DOI: 10.1002/spe

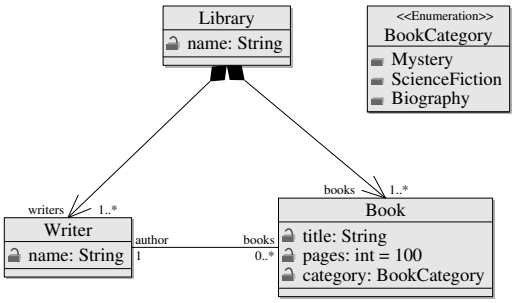

Figure 10. EMF example of a library description

semantics preserving (*i.e,* optimizations do not change the behavior of programs), we designed an efficient application strategy for combining them. It would have been difficult to directly prove the correctness of this optimization phase if we had reasoned directly on the program where the strategy and the rules were mixed.

Java Bytecode manipulation. Another application of SL in the domain of program transformation is an *on-the-fly* strategy compiler presented in [6]. This compilation method based on Bytecode specialization is expressed in TOM itself, using rules and strategies. Bytecode mappings have also been used for an application of bytecode rewriting for secure class loading. In [40], we show how to implement defensive class loaders that redirect method invocations to new targets (*e.g.* safe IO API) by rewriting Bytecode just before loading. This application uses elaborate strategies to check conditions on the control flow of the Bytecode.

Translation of database queries (from MDX to SQL). TOM has been used by BusinessObjects/SAP to implement a translator from MDX queries (Multidimensional Expressions) to SQL queries. The Crystal Reports software development team first benchmarked TOM to be sure that it would be efficient enough to support on-the-fly translations. In a second step, they used TOM to define their translation. The notion of rule and strategy was essential in their approach: this was the guarantee that the transformation would be expressed in a high-level fashion. The benefit of the approach is its extensibility combined with a better confidence in the implementation due to code readability. A very important point for the project leader, in order to increase discussions and code review, was that each translation step could be presented on a single slide. This work is integrated in the current version of the Crystal Reports software, but for confidentiality reasons, there is unfortunately no research report. This industrial experience spotted the need, for industrials, to have formal methods that help to characterize the shape of a term resulting from the application of a strategy. For instance, when applying a simplifying strategy on an arithmetic expression, can we ensure (statically or dynamically) that the normal form will contain at most two levels of addition?

6.3. Model-driven engineering

TOM provides automatic generation of TOM mappings from any EMF (Eclipse Modelling Framework [44]) meta models. Then it is possible to use the TOM language for directly expressing complex transformations on EMF models without extra translation.

Let us consider the example given in Figure 10. We can write the following strategy that collects all the book titles in a Java collection:

```
%strategy CollectBookName(set:Collection) extends Identity() {
    visit Book {
      Book[title = name] -> { set.add(`name); }
    }
  }
```


DOI: 10.1002/spe

```
}
```

Note the use of the syntax `Book[title = name]` that is a shortcut for `Book(name,_,_,_)`, allowing us to specify only the slots in which we are interested. This strategy can be composed using SL combinators to obtain a more complex strategy. For example, we can apply `CollectBookName` in a *top-down* manner on a library model as in the following code:

```
public Set collectAllBookNames(Library model) {
  Set set = new HashSet();
  Strategy s = TopDown(CollectBookName(s));
  s.visit(model, new EcoreContainmentIntrospector());
  return set;
}
```

This example shows the use of SL *Introspectors*, allowing the application of strategies on Java instances of user-defined classes (here, classes generated by the Eclipse Modelling Framework). Indeed TOM automatically generates the `EcoreContainmentIntrospector` class by using the mapping definitions. See Section 5 for more details.

As explained in [45], we can consider three different architectural approaches for defining model transformations: (1) direct model manipulation, in Java using EMF for instance; (2) intermediate representations such as XML, using XSLT as a transformation engine; and (3) using higher-level transformation language support such as ATL, QVT or Xtend for example. Our approach fits the gap between approaches 1 and 3 by embedding high level constructs *a la* ATL or QVT (*resolveTemp* or *resolveIn* for instance) in a general-purpose language such as Java. This combination allows a declarative description of the transformation, while enabling the use of Java and EMF to perform computations in an efficient way. Our approach being compiled, this also offers performance improvements, in particular on pattern matching operations.

6.4. XML handling

TOM offers concrete syntax for XML documents [46]. For example, consider the following XML document:

```
<Bank name="BNP">
 <Branch name="Etoile">
  <CCAccount id="12">
   <Owner gender="M">Bob</Owner>
   <Balance>10000</Balance>
  </CCAccount>
  <SAccount rate="4">
   <Owner gender="M">Bob</Owner>
   <Owner gender="F">Alice</Owner>
   <Balance>100000</Balance>
  </SAccount>
 </Branch>
 <Branch name="Lafayette">
  <CCAccount id="23">
   <Owner gender="M">John</Owner>
   <Balance>10000</Balance>
  </CCAccount>
  <CCAccount id="6">
   <Owner gender="M">Bob</Owner>
   <Balance>6000</Balance>
  </CCAccount>
 </Branch>
</Bank>
```

A *bank* is composed of *branches*, each of them containing different types of *accounts*. Whatever the representation the XML document is (DOM for instance), it can be *seen* as a tree built out of (XML) nodes. The TOM mapping of XML documents is based on the DOM representation offered by the `w3c.dom` package.

Thanks to DOM mapping, complex XML transformations can be accomplished by strategies. For example, if we want to give a 15% bonus to all account owners that have opened a savings account in the same branch, then the following strategy can be used:

```
%strategy Bonus() extends Identity() {
 visit TNode {
```


DOI: 10.1002/spe

```
  <Branch>
   (A1*,
    <CCAccount>
      (X1*,owner,X2*,<Balance>#TEXT(bal)</Balance>,X3*)
    </CCAccount>,
    A2*, sa@<SAccount>owner</SAccount>, A3*)
   </Branch> -> {
    double newbal = Double.parseDouble(bal)*1.15;
    TNode node = `xml(<Balance>#TEXT(newbal)</Balance>);
    return `xml(<Branch>
                  A1*
                  <CCAccount>X1* owner X2* node X3*</CCAccount>
                  A2* sa A3*
                </Branch>);
  }
 }
}
```

Note that the pattern syntax is a mix of XML and plain term-based syntax. TOM desugars the XML syntax into the plain one using the DOM mapping. This example also uses elaborate non-linear patterns involving list matching of the form `(X1*,...,X2*,...,X3*)` to retrieve the context information (*i.e,* the other XML nodes) needed in order to build the XML tree in the right-hand side of the rule. The TOM construct `` `xml(...) `` can be used to build a tree (a DOM object in our case) using a concrete XML notation, making the code more legible. Applying `TopDown(Bonus())` on the previous document leads to the expected result. In the *Etoile* branch, Bob's credit card account has been increased with the bonus.

This example shows that high-level constructs *a la* XSLT can be natively integrated into a programming language such as Java. The interest of using TOM for performing XML manipulations, compared to XSLT for instance, is multiple: the proposed pattern matching is semantically well defined and can be compiled in an efficient way[†]; this avoids the usage of external tools or libraries when XML manipulations are needed in a project already written in TOM; the integration into Java is an advantage when complex computations are needed in addition to XML manipulation. This extension has been used in an industrial context by Cril Technology to interconnect model checkers for timed automata expressed in XML [47].

6.5. Summary

In this section, we have shown how SL is used in a variety of applications for specifying declarative transformations or analyses. The view mechanism used in both TOM (using mappings) and SL (using introspectors) enables programmers to take advantage of the strategic programming approach in a large scope of application domains. Note that SL introspectors can be automatically generated from TOM mappings so when using TOM and SL in combination, only mappings have to be specified by hand. Moreover, TOM already offers a large set of mappings:

- DOM representation for XML,
- EMF model used in model-driven engineering,
- Java Bytecode program based on the ASM library [48],
- Java class hierarchy based on the *getters/setters* convention (*e.g*. Java beans, Java persistence API).

[†]the current implementation of TOM does not yet support deep matching, expressed by `//` in XSLT. This can be simulated using nested strategy calls, but some work is still needed to get a proper integration.

DOI: 10.1002/spe

7. RELATED WORK

SL has been inspired by work coming from different research areas. In this section, we compare SL with languages and libraries from the domains of rule-based languages, XML programming and datatype-generic programming.

7.1. Control in rule-based languages

The design of SL has been inspired mostly by the ELAN [10] and Stratego [12] rule-based languages.

Compared to the strategy language of ELAN, SL does not support implicitly non-deterministic strategies using back-tracking semantics. Although, as positions are first class in SL, it allows for the implementation of explicitly non-deterministic computations, as shown in Section 3.6.1 (`Collect` strategy).

Stratego is the language to which SL is the closest, the main differences being strategies as terms and explicit evaluation context. In particular, this latter feature makes it possible to introduce original traversal combinators such as `Up`, which moves the focus to the current node's parent. Another main difference is that Stratego is weakly typed. On the contrary, SL relies on the Java type system, which enables some static guarantees, like the type safety of base strategies and of some combinators. However there is some limitation of what can be statically checked in Java. For example, traversal combinators such as `All` do not enjoy type safe definitions in Java (to the extent of our knowledge). Indeed their implementations invariably involve downcasts.

A shown in Section 2, ASF+SDF [8] does not have a strategy language but provides traversal functions that can be used to control how a set of rules is applied. By raising the notion of strategy to the object level, SL offers meta-programming capabilities that may remind one of the meta-level of MAUDE [11]. In the context of graph rewriting, the use of strategy languages has also been investigated to gain precise control over rules application. For example, the graph rewriting engine PORGY features a strategy language that makes the concept of positions explicit [38], similarly to SL. This allows one, among other things, to explicitly change the application position.

7.2. XML programming

There exist several languages to manipulate XML documents. For example, the W3C has defined XPath, an expression language that allows one to easily define path queries on an XML document. We have shown in Section 6 that TOM provides concrete syntax to manipulate XML documents. Combining with SL, it is possible to encode any XPath queries. In particular, the notion of XPath parent and sibling axes can be encoded using the original SL strategy combinators `Up` and `Omega`. This is made possible by maintaining an execution context during the evaluation of the strategy. On the contrary, it is not possible with XPath to have full control over the way the document is explored, as it is with SL.

The Java standard library provides a DOM implementation that enables manipulation of XML documents through the DOM API. Additionally, the `javax.xml.xpath` package provides an XPath implementation (version 1.0) that enables the Java programmer to evaluate XPath expressions over DOM documents. Contrary to TOM, this approach is purely interpreted, and does not provide any guarantee on the transformations.

XSLT is a XML transformation language relying on XPath [19]. In XSLT, XPath is used to select part of the original document and query it, thus one can only loop over the results of an XPath query. The result of the application of an XSLT template on a document may only be another document. Contrary to SL, it is not possible to execute arbitrary actions when examining the initial documents.

The OCamlDuce system [49] is an extension of the OCAML functional language that integrates XML expressions, regular expression types and patterns. OCamlDuce provides static insurance that a program will produce well-typed XML values by leveraging the OCAML type system. The integration of XML manipulation in TOM cannot provide such a guarantee. On the other side, a main advantage of SL coupled with TOM is to be fully embedded in Java, allowing one to reuse existing Java representations of XML documents such as the DOM API.

DOI: 10.1002/spe

7.3. Datatype-generic programming

In generalist programming languages, datatype-generic programming consists of defining a function that does not depend on a specific datatype but only on the structure of the datatype. Thus the strategy combinators of rule-based languages can be encoded by specific generic functions relying on the tree-structure of datatypes.

In functional languages such as HASKELL, there exist numerous languages and libraries such as generic HASKELL [21], SYB [22] and Kure [23]. In particular, many research efforts have been made to extend the set of types a generic function can be used on while preserving type safety. For a detailed comparison of the HASKELL libraries, the reader can refer to [27]. In these libraries, generic functions are directly expressed as HASKELL functions and any usual strategy combinator can be easily defined as a higher-order generic function. For example, in [20], the authors showed how to encode the XPath combinators in SYB. In particular, they discuss how to introduce a notion of application context similar to SL's, enabling the combinators to follow other axes than the descendant axis.

In object-oriented languages, the work closest to ours are JJTraveler [4] and Kiama [25]. JJTraveler is a framework that provides generic visitor combinators for Java. In Section 5, we have made a detailed comparison between JJTraveler and SL, the main differences being explicit recursive combinators, execution environment and introspection. The Kiama library is a domain-specific language dedicated to language processing, which is embedded in Scala. Among other things, Kiama features specific tree rewriting statements and a strategy language very similar to Stratego. Similarly to SL that is embedded in Java, Kiama relies on the underlying Scala programming language for both data representation and type checking. The main differences include the ones between SL and Stratego, mainly reflexivity and explicit evaluation context. The main benefit of Kiama is the embedding in Scala that allows for smooth integration. This is made possible through Scala features such as higher-order functions and infix operators, which allow Kiama to provide an embedded strategy language with a syntax very similar to Stratego's one. Contrary to TOM, Kiama is interpreted and thus some optimizations such as pattern-matching optimizations [37] cannot be realized.

8. CONCLUSION

Contributions. In this paper, we show how *strategic programming* can be integrated into a general-purpose language like Java. Thanks to examples in the area of program optimization, we demonstrate the interest of such a design pattern for Java developers who need easily extensible visitors for specifying data-structure transformation or analysis. In combination with TOM, a DSL for declarative transformations embedded in Java, SL has been applied in a wide range of applications: from Bytecode manipulation to model-driven engineering.

In comparison with other strategy languages, the main innovation of SL is that it is data-structure agnostic, which allows non-intrusive visitors. This property avoids extra translation phases by enabling the use of external libraries (*e.g.* the `w3c.dom` package used for XML manipulation) to represent tree-structured data.

Ongoing and future work. We are currently working on the integration of graph rewriting capabilities into the TOM language [50]. This extension is based on a generalisation of the concept of position, which offers new perspectives for the design of the SL language. For example, as studied in [51], we have started to add new strategy combinators dedicated to cyclic structures. These combinators rely on the evaluation environment of the SL library to control the cycles.

Another perspective is a development environment dedicated to the SL language. Lämmel *et al.* give a taxonomy of the programming errors in strategic programs and propose a set of static analyses, which offers a first step towards more guidance in strategic programming [36]. In particular, when manipulating large terms such as ASTs and complex strategic rewriting programs such as optimizers, it is essential to propose dedicated analysis and debugging features.

DOI: 10.1002/spe

Acknowledgments. We sincerely thank the anonymous referees for their valuable and detailed remarks that led to a substantial improvement of the paper.

REFERENCES

1. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Addison-Wesley: Boston, MA, 1995.
2. Wadler P. The Expression Problem. Mailing list Nov 1998. URL `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.
3. Hierarchical Visitor Pattern. URL `http://c2.com/cgi/wiki?HierarchicalVisitorPattern`.
4. Visser J. Visitor combination and traversal control. *OOPSLA'01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, ACM Press: New York, NY, USA, 2001; 270–282.
5. Balland E, Brauner P, Kopetz R, Moreau PE, Reilles A. Tom: Piggybacking rewriting on Java. *RTA'07: Proceedings of the 18th International Conference on Rewriting Techniques and Applications*, *Lecture Notes in Computer Science*, vol. 4533, Springer-Verlag, 2007; 36–47.
6. Balland E, Moreau PE, Reilles A. Rewriting strategies in Java. *RULE'07: 8th International Workshop on Rule-Based Programming*, *Electronic Notes in Theoretical Computer Science*, vol. 219, Elsevier Science Publishers, 2007; 97–111.
7. Baader F, Nipkow T. *Term Rewriting and all That*. Cambridge University Press, 1998.
8. van den Brand M, Heering J, Klint P, Olivier P. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems* 2002; **24**(4):334–368.
9. Plasmeijer MJ, van Eekelen MCJD. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.
10. Moreau PE, Kirchner H. A compiler for rewrite programs in associative-commutative theories. *PLILP'98: Proceedings of the 10th International Symposium on Principles of Declarative Programming*, no. 1490 in Lecture Notes in Computer Science, Springer-Verlag, 1998; 230–249.
11. Clavel M, Meseguer J. Reflection and strategies in rewriting logic. *RWLW'96: Proceedings of the 1st International Workshop on Rewriting Logic and its Applications*, *Electronic Notes in Theoretical Computer Science*, vol. 4, Meseguer J (ed.), Elsevier Science Publishers, 2000; 126–148.
12. Visser E, Benaissa ZeA, Tolmach A. Building program optimizers with rewriting strategies. *SIGPLAN Notices* 1999; **34**(1):13–26.
13. Futatsugi K, Goguen JA, Jouannaud JP, Meseguer J. Principles of OBJ-2. *Proceedings 12th ACM Symposium on Principles of Programming Languages*, Reid B (ed.), ACM Press, 1985; 52–66.
14. van den Brand M, Klint P, Vinju J. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology* 2003; **12**(2):152–190.
15. Clavel M. Reflection in general logics, rewriting logic, and Maude. PhD Thesis, University of Navarre, Spain 1998.
16. Borovanský P, Kirchner C, Kirchner H, Moreau PE, Vittek M. ELAN: A logical framework based on computational systems. *WRLA'96: Proceedings of the 1st International Workshop on Rewriting Logic and its Applications*, vol. 4, Meseguer J (ed.), Electronic Notes in Theoretical Computer Science, 1996.
17. Borovanský P, Kirchner C, Kirchner H. Controlling rewriting by rewriting. *Proceedings of the 1st International Workshop on Rewriting Logic*, *Electronic Notes in Theoretical Computer Science*, vol. 4, Meseguer J (ed.), Elsevier Science Publishers: Asilomar (California), 1996.
18. XML Path Language (XPath) Version 2.0 - W3C Recommendation. URL `http://www.w3.org/TR/xpath20/`.
19. XSL Transformations (XSLT) Version 2.0 - W3C Recommendation. URL `http://www.w3.org/TR/xslt20/`.
20. Lämmel R. Scrap your boilerplate with XPath-like combinators. *POPL'07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM: New York, NY, USA, 2007; 137–142.
21. Hinze R, Jeuring J. Chapter 1. Generic Haskell: Practice and Theory. *Generic Programming*, *Lecture Notes in Computer Science*, vol. 2793, Backhouse R, Gibbons J (eds.). Springer-Verlag, 2003; 1–56.
22. Lämmel R, Peyton Jones S. Scrap your boilerplate: a practical design pattern for generic programming. *TLDI'03: Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation*, vol. 38, ACM Press, 2003; 26–37.
23. Gill A. A Haskell hosted DSL for writing transformation systems. *DSL'09: Proceedings of the IFIP Working Conference on Domain Specific Languages*, *Lecture Notes in Computer Science*, vol. 5658, Taha W (ed.), Springer-Verlag, 2009; 285–309.
24. Odersky M, Wadler P. Pizza into Java: Translating theory into practice. *POPL'97: Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, ACM Press, 1997; 146–159.
25. Sloane A. Lightweight language processing in Kiama. *Generative and Transformational Techniques in Software Engineering III*, *Lecture Notes in Computer Science*, vol. 6491, Fernandes J, Lämmel R, Visser J, Saraiva J (eds.), Springer-Verlag, 2011; 408–425.
26. Hinze R, Jeuring J, Löh A. Comparing approaches to generic programming in Haskell. *Datatype-Generic Programming*, *Lecture Notes in Computer Science*, vol. 4719, Backhouse R, Gibbons J, Hinze R, Jeuring J (eds.), Springer-Verlag, 2007; 72–149, doi:10.1007/978-3-540-76786-2_2.
27. Rodriguez A, Jeuring J, Jansson P, Gerdes A, Kiselyov O, Oliveira BCdS. Comparing libraries for generic programming in Haskell. *Haskell'08: Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, ACM Press, 2008; 111–122.

DOI: 10.1002/spe

28. Moreau PE, Ringeissen C, Vittek M. A pattern matching compiler for multiple target languages. *CC'03: Proceedings of the 12th Conference on Compiler Construction*, *Lecture Notes in Computer Science*, vol. 2622, Hedin G (ed.), Springer-Verlag, 2003; 61–76.
29. Balland E, Brauner P, Kopetz R, Moreau PE, Reilles A. Tom Manual. LORIA, Nancy (France), version 2.6 edn. Apr 2008. URL `http://tom.loria.fr/soft/release-2.6/manual-2.6/`.
30. Wadler P. Views: a way for pattern matching to cohabit with data abstraction. *POPL'87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM Press, 1987; 307–313.
31. Barendregt HP. *The Lambda Calculus – Its Syntax and Semantics*, *Studies in Logic and the Foundations of Mathematics*, vol. 103. North-Holland, 1984.
32. Fernández M, Namet O. Strategic programming on graph rewriting systems. *IWS'10: Proceedings of the 1st International Workshop on Strategies in Rewriting, Proving, and Programming*, 2010; 1–20.
33. Reilles A. Canonical abstract syntax trees. *WRLA'06: Proceedings of the 6th International Workshop on Rewriting Logic and its Applications*, vol. 176, Denker G, Talcott C (eds.), Electronic Notes in Theoretical Computer Science: Vienna, Austria, 2006; 165–179.
34. Bruni R, Corradini A, Gadducci F, Lafuente AL, Vandin A. Modelling and analyzing adaptive self-assembling strategies with Maude. *WRLA'12: Proceedings of the 9th International Workshop on Rewriting Logic and its Applications*, 2012. To be published.
35. Feuillade G, Genet T, Viet Triem Tong V. Reachability analysis over term rewriting systems. *J. Autom. Reasoning* 2004; **33**(3-4):341–383.
36. Lämmel R, Thompson SJ, Kaiser M. Programming errors in traversal programs over structured data. *LDTA'08: Proceedings of the 8th Workshop on Language Descriptions, Tools, and Applications*, *Electronic Notes in Theoretical Computer Science*, vol. 238, Elsevier Science Publishers, 2009; 135–153.
37. Balland E, Moreau PE. Optimizing pattern matching compilation by program transformation. *SeTra'06: Proceedings of the 3rd Workshop on Software Evolution through Transformations*, Electronic Communications of EASST, 2006.
38. Andrei O, Fernández M, Kirchner H, Melançon G, Namet O, Pinaud B. PORGY: Strategy-driven interactive transformation of graphs. *TERMGRAPH'11: Proceedings of the 6th International Workshop on Computing with Terms and Graphs*, *Electronic Proceedings in Theoretical Computer Science*, vol. 48, Echahed R (ed.), 2011; 54–68.
39. Lacey D, de Moor O. Imperative program transformation by rewriting. *CC'01: Proceedings of the 10th International Conference on Compiler Construction*, no. 2027 in Lecture Notes in Computer Science, Springer-Verlag: London, UK, 2001; 52–68.
40. Balland E, Moreau PE, Reilles A. Bytecode rewriting in Tom. *BYTECODE'07: Proceedings of the 2nd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, vol. 190, Elsevier Science Publishers, 2007; 19–33.
41. Kuipers T, Visser J. Object-oriented tree traversal with JJForester. *Technical Report*, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands 2000.
42. Keith M, Schincariol M. *Pro EJB 3: Java Persistence API (Pro)*. Apress: Berkely, CA, USA, 2006.
43. Kopetz R, Moreau PE. Software quality improvement via pattern matching. *FASE'08: Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, vol. 4961, Fiadeiro J, Inverardi P (eds.), Springer-Verlag, 2008; 296–300.
44. Steinberg D, Budinsky F, Paternostro M, Merks E. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.
45. Sendall S, Kozaczynski W. Model transformation: the heart and soul of model-driven software development. *Software* 2003; **20**(5):42–45.
46. Cirstea H, Moreau PE, Reilles A. TomML: A rule language for structured data. *RuleML'09: Proceedings of the International Symposium on Rule Interchange and Applications*, *Lecture Notes in Computer Science*, vol. 5858, Springer-Verlag: Las Vegas, USA, 2009; 262–271.
47. Tavernier B. Calife: A generic graphical user interface for automata tools. *LDTA'04: Proceedings of the 4th Workshop on Language Descriptions, Tools, and Applications*, *Electronic Notes in Theoretical Computer Science*, vol. 110, Elsevier Science Publishers, 2004; 169 – 172.
48. Bruneton É, Lenglet R, Coupaye T. ASM: a code manipulation tool to implement adaptable systems. *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, 2002.
49. Frisch A. OCaml + XDuce. *PLAN-X*, Castagna G, Raghavachari M (eds.), BRICS, Department of Computer Science, University of Aarhus, 2006; 36–48.
50. Balland E, Moreau PE. Term-graph rewriting via explicit paths. *RTA'08: Proceedings of the 19th International Conference on Rewriting Techniques and Applications*, *Lecture Notes in Computer Science*, vol. 5117, Springer-Verlag, 2008; 32–47.
51. Fernández M, Mackie I. A calculus for interaction nets. *PPDP'99: Proceedings of the International Conference on Principles and Practice of Declarative Programming*, *Lecture Notes in Computer Science*, vol. 1702, Springer-Verlag, 1999; 170–187.

DOI: 10.1002/spe