ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

M P A Sellink (Ed)

2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997

Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997

Specification of Rewriting Strategies

S.P. Luttik and E. Visser

Published in collaboration with the British Computer Society

BCS

Springer

ISBN: 3-540-76228-0

Specification of Rewriting Strategies*

Sebastiaan P. Luttik[1,3,†] Eelco Visser[2,3,‡]

luttik@cwi.nl visser@acm.org

[1]CWI, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

[2]OGI, P.O. Box 91000, Portland, Oregon 97291-1000, USA

[3]Programming Research Group, University of Amsterdam
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

Abstract

User-definable strategies for the application of rewrite rules provide a means to construct transformation systems that apply rewrite rules in a controlled way. This paper describes a strategy language and its interpretation. The language is used to control the rewriting of terms using labeled rewrite rules. Rule labels are atomic strategies. Compound strategies are formed by means of sequential composition, nondeterministic choice, left choice, fixed point recursion, and two primitives for expressing term traversal. Several complex strategies such as bottom-up and top-down application and (parallel) innermost and (parallel) outermost reduction can be defined in terms of these primitives. The paper contains two case studies of the application of strategies.

1 Introduction

Term rewriting is an ideal technique for program transformation where the transformation of one construct into another is defined by means of rewrite rules. Usually, the rewrite engine contracts redexes according to some fixed selection scheme, and the possibilities the user has to control the order in which rules are tried are rather limited. For example, ASF+SDF (Van Deursen *et al.*, 1996) implements a leftmost innermost redex selection scheme and the only way the user can explicitly control the order in which rules are tried is by means of 'default equations'.

Often it is desirable to have more control over the reduction process. For instance, some rewriting systems exhibit better termination behaviour under a leftmost innermost scheme, while others behave better under a parallel outermost scheme. Also, it often yields more efficient normalization if rules are tried according to some specific priority ordering.

The usual solution to get more control over the strategy used to apply transformation rules is to write an explicit transformation function that traverses an expression and performs transformations in a fixed order. This gives great overhead in the specification and often distracts from conceptually simple transformation rules.

In this paper we show how the definition of a transformation in terms of rewrite rules can be separated from a specification of the order in which these rules should be applied. We define a language of strategies inspired by the strategy mechanism of the rewriting language ELAN (Vittek, 1994; Borovanský *et al.*, 1996) and demonstrate its use in ASF+SDF. This leads to a small set of generic modules that can easily be instantiated for any target language.

We illustrate this approach in two case studies. In the first example, we give an implementation of the proof of a (basic term) lemma in the setting of process algebra with conditionals. Strategies are necessary there, because one of the transformation rules is nonterminating. In the second example we discuss the normalization of box expressions in a typesetting language. The transformation rules form a weakly terminating rewrite system that shows infinite reductions under innermost rewriting. Using the strategy language a terminating strategy is specified.

*This research was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO).

†Project 612-33-008: A Modular Toolset for μCRL developed using μCRL.

‡Project 612-317-420: Incremental parser generation and context-sensitive disambiguation: a multi-disciplinary perspective.

Overview In Section 2 we introduce the basic strategy operators: *labels* referring to axioms, the *identity strategy*, *sequential composition* of strategies, *nondeterministic choice* and *left choice* between strategies. In Section 3 we extend this basic language with a fixed point operator for the expression of recursive strategies. As an application an iteration operator is defined using this fixed point operator. In Section 4 two primitives are added that allow for a general definition of a range of term traversing strategies, including bottom-up application of a strategy, and innermost and outermost normalization according to a strategy. We show how these traversals can be interpreted by means of a generic 'push-down' function, that applies a strategy to all direct proper subterms of a term. In Sections 5 and 6 the strategy language is applied in transformations of process expressions and box expressions. In Section 7 we compare our strategy language with that of ELAN and in Section 8 we give some indications for further research.

2 Basic Strategies

Labels, which refer to rewrite rules, are the basic building blocks of strategies. For example, the expression

$$[\texttt{RAssoc}]\,(x_1+x_2)+x_3 = x_1+(x_2+x_3)$$

denotes a rewrite rule labeled RAssoc that transforms a left-associative application of $+$ into a right-associative one.

A strategy defines a transformation function on the terms of a language. For instance, RAssoc defines the function [RAssoc]. According to the rule above, applying this function to $(a+b)+c$ gives $a+(b+c)$. Such an application does not have to succeed; for instance, [RAssoc] applied to $a+(b+c)$ is not defined here.

Module Term-SA below defines the syntax of the application of strategies to terms. The application of a strategy s to a term t, written as $[s]\,t$, results in a 'reduct'. This is either a term, denoting the successful application of the strategy, or $t\uparrow$, denoting failure of the application of the strategy. Consequently, a reduct can be seen as a pair of a term and a Boolean value indicating success or failure. Given a reduct r the function $\pi_t(r)$ gives the term part of this pair and the function $\pi_b(r)$ determines the Boolean value.

module Term-SA
imports Basic-Strategies$^{2.1}$ Booleans
exports
  sorts Term Reduct
  context-free syntax

Term	$\rightarrow$	Reduct
Term $\uparrow$	$\rightarrow$	Reduct
"[" Strategy "]" Term	$\rightarrow$	Reduct
π_t(Reduct)	$\rightarrow$	Term
π_b(Reduct)	$\rightarrow$	Bool

  variables
    "t"[0-9′]∗ $\rightarrow$ Term
    "r"[0-9′]∗ $\rightarrow$ Reduct
equations

[1]	$\pi_t(t) = t$	[2]	$\pi_b(t) = \text{true}$
[3]	$\pi_t(t\uparrow) = t$	[4]	$\pi_b(t\uparrow) = \text{false}$

This section and the next two will be concerned with the definition of operators for the composition of strategies and their interpretation as transformation functions.

2.1 Syntax

Labels (here defined as identifiers starting with an uppercase letter) are atomic strategies. Their interpretation is provided by the user by means of labeled rewrite rules. Apart from the labels, our basic strategy language contains a special constant denoting the *identity strategy* (ϵ), and binary operators for *sequential composition* ($s_1 \cdot s_2$), *nondeterministic choice* ($s_1 + s_2$) and *left choice* ($s_1 \lessdot\!\!+ s_2$).

module Basic-Strategies
imports Layout
exports
  sorts Label Strategy
  lexical syntax
    [A-Z][A-Za-z0-9\−]∗ → Label
  context-free syntax

Label	→ Strategy	
“ϵ”	→ Strategy	
Strategy “·” Strategy	→ Strategy	{**left**}
Strategy “+” Strategy	→ Strategy	{**left**}
Strategy “$\lessdot\!\!+$” Strategy	→ Strategy	{**right**}
“(” Strategy “)”	→ Strategy	{**bracket**}

  priorities
    Strategy “·”Strategy → Strategy > Strategy “$\lessdot\!\!+$”Strategy → Strategy > Strategy “+”Strategy → Strategy
  variables
    “*l*”[*0-9′*]∗ → Label
    “*s*”[*0-9′*]∗ → Strategy

2.2 Interpretation

Given a set of labeled rewrite rules, the application of a strategy expression to a term is interpreted according to the rules below and the user-defined labeled rewrite rules. A label that is undefined, or undefined for some term, fails, i.e., if label l is undefined for t, then $[l]t$ yields $t\uparrow$. In the rules below, success of an application is tested in a condition $[s]t = t'$, where the right-hand side is a term t' injected into Reduct. This test fails if the application results in failure, i.e., $[s]t = t\uparrow$.

module Term-BS
imports Basic-Strategies$^{2.1}$ Term-SA^{2}
equations
The identity strategy ϵ always succeeds and yields the term t itself.

[1] $$[\epsilon]\ t = t$$

The sequential composition $s_1 \cdot s_2$ succeeds if s_1 applied to t succeeds and yields a term t' and s_2 applied to t' succeeds and yields t''.

[2] $$\frac{[s_1]\ t = t',\ \ [s_2]\ t' = t''}{[s_1 \cdot s_2]\ t\ =\ t''}$$

The nondeterministic choice $s_1 + s_2$ succeeds if either s_1 or s_2 succeeds.

[3] $$\frac{[s_1]\ t = t'}{[s_1\ +\ s_2]\ t\ =\ t'}$$

[4] $$\frac{[s_2]\ t = t'}{[s_1\ +\ s_2]\ t\ =\ t'}$$

The left choice $s_1 \leftarrow\!\!\!\!+ s_2$ succeeds if either s_1 or s_2 succeeds, with a preference for s_1. That is, if s_1 succeeds, then it will be applied and s_2 is only tried when s_1 fails.

[5] $$\frac{[s_1]\ t = t'}{[s_1 \leftarrow\!\!\!\!+ s_2]\ t\ =\ t'}$$

[6] $$\frac{[s_1]\ t = t\uparrow,\ \ [s_2]\ t = t'}{[s_1 \leftarrow\!\!\!\!+ s_2]\ t\ =\ t'}$$

[7] $$[s]\ t = t\uparrow \quad \textbf{otherwise}$$

Generic Modules The Term-* modules defined above and in the next sections are *generic* modules that define the application and interpretation of strategies to some language of terms. These modules are intended to be instantiated for each sort under consideration by renaming the sorts Term and Reduct. The tool of De Jonge (1997) can be used to automate this instantiation.

3 Recursive Strategies

In this section we provide a fixed point operator for the definition of recursive strategies. At the end of this section we show how it can be used to define iterative strategies.

3.1 Syntax

The fixed point operator $\mu v.\, s$ denotes a recursive strategy with recursion point v. The variable v is bound by the fixed point operator.

module Recursive-Strategies
imports Basic-Strategies$^{2.1}$
exports
  sorts Var
  lexical syntax
    [a-z][A-Za-z0-9\−′]∗ → Var
  context-free syntax
    Var → Strategy
    "μ" Var "." Strategy → Strategy
  priorities
    Strategy "+"Strategy → Strategy > "μ"Var "."Strategy → Strategy
  variables
    "v"[*0-9′*]∗ → Var

3.2 Interpretation

A fixed point $\mu v.\, s$ denotes the infinite strategy expression obtained by recursively replacing the entire expression for the free occurrences of the variable v in s, i.e., we have

$$\mu v.\, s = s\ [v := \mu v.\, s]$$

where $s\ [v := s']$ denotes the substitution of s' for all free occurrences of v in s. Substitution is defined in Section A. Because this equation would lead to an innermost non-terminating rewrite system, we interpret the fixed point operator lazily in the following module.

module Term-RS
imports Term-BS$^{2.2}$ Recursive-Strategies$^{3.1}$ Strategy-Substitution^{A}
equations
The application of a fixed point strategy to a term is interpreted by unfolding the fixed point one step. In this way unfoldings are performed *by need*.

[1] $$[\mu v.\ s]\ t\ =\ [s\ [v := \mu v.\ s]]\ t$$

3.3 Example: Iteration of Strategies

As a first example application of recursive strategies we introduce the operators $*$ and $+$ to express the iteration of a strategy zero or more and one or more times, respectively.

module Complex-Strategies
imports Basic-Strategies$^{2.1}$
exports
  context-free syntax
    Strategy "$*$" $\rightarrow$ Strategy
    Strategy "$+$" $\rightarrow$ Strategy
  priorities

$$\{\text{Strategy “}*\text{”} \rightarrow \text{Strategy, Strategy “}+\text{”} \rightarrow \text{Strategy}\}\ >\ \text{Strategy “.”Strategy} \rightarrow \text{Strategy}$$

The strategy $s*$ is defined by means of a recursive strategy that applies the strategy s as long as it succeeds and then terminates successfully. The function get-fresh is used to create a binding variable v that does not occur freely in the strategy s. Observe how the left choice operator $\Leftarrow\!\!\!+$ is used to enforce as many applications as possible. If s cannot be applied, then the sequential composition $s \cdot v$ fails, so ϵ is applied and succeeds. The unary operator $+$ is defined in terms of $*$. A strategy $s+$ succeeds if s succeeds at least once.

module CS-Interpretation
imports Complex-Strategies$^{3.3}$ Strategy-Substitution^{A}
equations

[1] $$\frac{\text{get-fresh}(\mathsf{x}, s) = v}{s\ *\ =\ \mu v.\ s \cdot v \Leftarrow\!\!\!+\ \epsilon}$$

[2] $$s\ +\ =\ s \cdot s\ *$$

4 Traversal Strategies

The strategies we have discussed so far are applications of a strategy at the root of a term. In order to allow for a general definition of a wide range of term traversing strategies, we introduce two unary operators. Let $t = f(t_1, \ldots, t_n)$, we will call the arguments $t_1, \ldots, t_n$ of the leading function symbol f of t the *direct proper subterms of* t.

1. The strategy $\Box(s)$ (*conjunctive push-down of* s) applies s to all direct proper subterms of a term, provided that these applications will all succeed.

2. The strategy $\Diamond(s)$ (*disjunctive push-down of* s) applies s to all direct proper subterms of a term for which application is successful, provided that at least one of these applications will succeed.

In the sequel, we will show how both operators can be interpreted by means of one generic push-down function, and how general term traversing strategies can be defined in terms of these operators.

4.1 Interpretation

Clearly, $\Box(s)$ and $\Diamond(s)$ only differ with respect to their 'success behaviour'. Therefore, we interpret both operators by means of one generic primitive $\text{pd}_{\oplus}(s)$, parameterized with a boolean operator $\oplus$ that determines the 'success behaviour'.

module Push-Down
imports Booleans-Generalized$^{4.1}$ Basic-Strategies$^{2.1}$
exports
  context-free syntax
    pd "_" BoolOp "(" Strategy ")" $\rightarrow$ Strategy

A generalized boolean operator operates on a list of booleans. The application $\bigwedge\{b_1, \ldots, b_n\}$ denotes $b_1 \wedge \ldots \wedge b_n$ and $\bigvee\{b_1, \ldots, b_n\}$ denotes $b_1 \vee \ldots \vee b_n$. The operators are defined as a separate sort such that they can be used as 'higher-order' parameters of the function pd.

module Booleans-Generalized
imports Booleans
exports
  sorts BoolOp
  context-free syntax
    "$\bigwedge$" $\rightarrow$ BoolOp
    "$\bigvee$" $\rightarrow$ BoolOp
    BoolOp "{" Bool$*$ "}" $\rightarrow$ Bool
  variables
    "$\oplus$"[*0-9′*]$*$ $\rightarrow$ BoolOp
hiddens
  variables
    "*b*"[*0-9′*]$*$ $\rightarrow$ Bool
    "*b*$*$"[*0-9′*]$*$ $\rightarrow$ Bool$*$
equations

[1]	$\bigwedge\{\} = \text{true}$	[2]	$\bigvee\{\} = \text{false}$
[3]	$\bigwedge\{b\ b^*\} = b \wedge \bigwedge\{b^*\}$	[4]	$\bigvee\{b\ b^*\} = b \vee \bigvee\{b^*\}$

Generic Push-Down The function pd is defined by a schema that should be instantiated for each function symbol f of arity n in the signature of the language under consideration. This schema can be instantiated for a given SDF definition using the specification generation techniques described in Van den Brand and Visser (1996).

module Term-TS
imports Term-RS$^{3.2}$ Push-Down$^{4.1}$
equations
For each function $f : s_1 \times \cdots \times s_n \rightarrow s_0$ in the signature (in the schemata in this section we abstract from the grammatical (mix-fix function) aspect of SDF signatures) of a specification define a rule

$$\frac{[s]\,t_1 = r_1,\ \ldots,\ [s]\,t_n = r_n,\ \oplus\{\pi_{\text{b}}(r_1), \ldots, \pi_{\text{b}}(r_n)\} = \text{true}}{[\text{pd}_{\oplus}(s)]\ f(t_1, \ldots, t_n) = f(\pi_{\text{t}}(r_1), \ldots, \pi_{\text{t}}(r_n))}$$

The first n conditions apply the strategy s to the respective arguments t_i. These conditions always succeed because the r_i are variables of sort Reduct. The application of the boolean operator $\oplus$ then determines the success or failure of the strategy from the list of Boolean values denoting the success or failure of the argument applications. If the outcome is true, the resulting term is the application of f to the term components of the reducts. Note that if the push-down succeeds, but s fails on t_i, then $\pi_{\text{t}}(r_i) = t_i$.

In the case of a constant c ($n = 0$; an application without arguments) we thus have

$$\frac{\oplus\{\} = \text{true}}{[\text{pd}_{\oplus}(s)]\, c = c}$$

This entails that the success of a push-down on a constant depends on the default value of the boolean operator $\oplus$; if it is a conjunction it succeeds, and if it is a disjunction it fails.

Associative lists can be considered as functions with an arbitrary number of arguments. A push-down on such a list entails the application of the strategy to each of the elements of the list. This is expressed by means of the following equation schemata that should be instantiated for each list sort in the specification.

The empty list is treated as a constant.

$$\frac{\oplus\{\} = \text{true}}{[\text{pd}_{\oplus}(s)] \, =}$$

For a non-empty list the strategy is applied to the head t of the list and the push-down to the tail t^* of the list.

$$\frac{[s]\,t = r,\; [\text{pd}_{\oplus}(s)]\,t^* = r^*,\; \oplus\{\pi_{\text{b}}(r), \pi_{\text{b}}(r^*)\} = \text{true}}{[\text{pd}_{\oplus}(s)]\,t\;t^* = \pi_{\text{t}}(r)\;\pi_{\text{t}}(r^*)}$$

Conjunctive and Disjunctive Push-Down We can now define $\Box(s)$ and $\Diamond(s)$ as the conjunctive and disjunctive instantiations of pd:

module Traversal-Strategies
imports Push-Down$^{4.1}$ Strategy-Substitution^{A}
exports
  context-free syntax
    $\Box(\text{Strategy}) \rightarrow \text{Strategy}$
    $\Diamond(\text{Strategy}) \rightarrow \text{Strategy}$
equations
The strategies $\Box(s)$ and $\Diamond(s)$ are the conjunctive and disjunctive instantiations of push-down. This entails that $\Box(s)$ succeeds if s succeeds on all direct proper subterms and that $\Diamond(s)$ succeeds if s succeeds for at least one direct proper subterm.

[1] $$\Box(s) = \text{pd}_{\wedge}(s)$$
[2] $$\Diamond(s) = \text{pd}_{\vee}(s)$$

4.2 Defining Traversal Strategies

Given the push-down operators we can define several term traversal strategies.

module Defined-Traversals
imports Basic-Strategies$^{2.1}$ Recursive-Strategies$^{3.1}$ CS-Interpretation$^{3.3}$ Traversal-Strategies$^{4.1}$
exports
  context-free syntax

bu(Strategy)	$\rightarrow$ Strategy
td(Strategy)	$\rightarrow$ Strategy
once(Strategy)	$\rightarrow$ Strategy
innermost(Strategy)	$\rightarrow$ Strategy
outermost(Strategy)	$\rightarrow$ Strategy
innermost-eff(Strategy)	$\rightarrow$ Strategy

equations

The strategies bu(s) and td(s) apply s bottom-up and top-down to a term. In the case of bu(s), $\Box(v)$ is used to recursively apply the strategy to all proper subterms of the term, after which the strategy s is applied to the result. Top-down is the dual of bottom-up obtained by reversing the order of the sequential composition.

[1] $$\frac{\text{get-fresh}(\mathsf{x},\, s) = v}{\text{bu}(s) \;=\; \mu v.\, \Box(v) \cdot s}$$

[2] $$\frac{\text{get-fresh}(\mathsf{x},\, s) = v}{\text{td}(s) \;=\; \mu v.\, s \cdot \Box(v)}$$

The strategy once(s) applies s once at each position in a term where application is possible.

[3] $$\text{once}(s) = \text{bu}(s \lessdot\!\!+ \epsilon)$$

The strategy innermost(s) works by means of two loops. The inner loop makes sure that s is pushed down in the term until it finds the innermost redexes. Note that $\Diamond(v)$ fails only on terms whose direct proper subterms are in normal form with respect to s. So s is applied to all innermost redexes. This is iterated until the term is in normal form with respect to s.

[4] $$\frac{\text{get-fresh}(\mathsf{x},\, s) = v}{\text{innermost}(s) \;=\; (\mu v.\, \Diamond(v) \lessdot\!\!+ s) *}$$

The strategy outermost(s) is be defined as the dual of innermost(s); it exhibits a preference for applying s at the root over pushing s down in the term. Thus, repeatedly all outermost redexes are contracted.

[5] $$\frac{\text{get-fresh}(\mathsf{x},\, s) = v}{\text{outermost}(s) \;=\; (\mu v.\, s \lessdot\!\!+ \Diamond(v)) *}$$

The strategy innermost-eff is a more efficient implementation of parallel innermost reduction, because it makes use of information about the locations in the term of the previous applications of s. The strategy consists of a single bottom-up traversal that applies s to each node; if such an application succeeds, the result is reduced innermost, using s.

[6] $$\frac{\text{get-fresh}(\mathsf{x},\, s) = v}{\text{innermost-eff}(s) \;=\; \mu v.\, \text{once}(s \cdot v)}$$

5 Process Expressions

In process algebra (see Baeten and Weijland, 1990) it is common to prove a lemma stating that every closed term is equal to a closed term with a simple inductive structure, a so-called *basic term*. Usually, such a lemma is proved by defining a terminating rewriting system consisting of rules that are sound with respect to the process algebraic theory, such that the set of basic terms coincides with the set of normal forms of this system.

A transformation of arbitrary process terms to basic terms is part of a tool that takes μCRL process specifications to linear format (we refer to Groote and Ponse (1994b) for μCRL and to Bosscher and Ponse (1995) for an informal description of the tool). In this particular setting it is a natural approach to first preprocess terms with a rule that is nonterminating, and then reduce the result in another rewriting system. Below we discuss an implementation of this using the strategy language just defined.

5.1 Syntax

We consider a small fragment of μCRL allowing only one datatype $\mathbb{B}$ of booleans and a restricted syntax for the actions $\mathbb{A}$.

```
module NcrlTerms
exports
  sorts 𝔸 𝔹 ℙ
  lexical syntax
    [abc][0-9′]∗ → 𝔸
  context-free syntax
    "⊤"                    → 𝔹
    "⊥"                    → 𝔹
    "and" "(" 𝔹 "," 𝔹 ")"  → 𝔹
    "not" "(" 𝔹 ")"        → 𝔹
    "δ"                    → ℙ
    𝔸                      → ℙ
    ℙ "." ℙ                → ℙ {right}
    ℙ "◁" 𝔹 "▷" ℙ          → ℙ {right}
    ℙ "+" ℙ                → ℙ {assoc}
    "(" ℙ ")"              → ℙ {bracket}
  priorities
    ℙ "."ℙ → ℙ  >  ℙ "◁"𝔹 "▷"ℙ → ℙ  >  ℙ "+"ℙ → ℙ
  variables
    "α"[0-9′]∗ → 𝔸
    "b"[0-9′]∗ → 𝔹
    "p"[0-9′]∗ → ℙ
```

5.2 Transformation Rules

The set of basic terms that we are interested in is inductively defined as follows:

1. δ, α and $\alpha \lhd b \rhd \delta$ are basic terms ($\alpha \in \mathbb{A},\ b \in \mathbb{B}$);
2. if p_1 and p_2 are basic terms, then so are $\alpha \cdot p_1$, $\alpha \cdot p_1 \lhd b \rhd \delta$, and $p_1 + p_2$ ($\alpha \in \mathbb{A},\ b \in \mathbb{B}$).

The following rules are all derivable in the proof theory for μCRL (see Groote and Ponse, 1994a); in fact, the rules [R2], [R3] and [R4] respectively correspond to axioms A4, A5 and A7 of process algebra.

module NcrlAxioms
equations

[R1] $x \lhd b \rhd y = x \lhd b \rhd \delta + y \lhd \text{not}(b) \rhd \delta$
[R2] $(x + y) \cdot z = x \cdot z + y \cdot z$
[R3] $(x \cdot y) \cdot z = x \cdot y \cdot z$
[R4] $\delta \cdot x = \delta$
[R5] $\delta \lhd b \rhd \delta = \delta$
[R6] $(x \lhd b \rhd \delta) \cdot y = x \cdot y \lhd b \rhd \delta$
[R7] $(x + y) \lhd b \rhd \delta = x \lhd b \rhd \delta + y \lhd b \rhd \delta$
[R8] $(x \lhd b_1 \rhd \delta) \lhd b_2 \rhd \delta = x \lhd \text{and}(b_1, b_2) \rhd \delta$

It follows by means of the method of lexicographic path orderings (see Bergstra and Klop, 1985) that the rewriting system consisting of [R2], ..., [R8] is terminating. Process terms not containing conditionals of the form $p_1 \lhd b \rhd p_2$ with p_2 not equal to δ are basic terms iff they are in normal form with respect to these rules.

The rule [R1] can be used to remove conditionals with a rightmost argument not equal to δ, but it is clearly nonterminating. Notice, however, that it is enough to apply [R1] only once at every subterm to arrive at a term that does not contain conditionals with a rightmost argument not equal to δ.

5.3 Normalization Strategy

We obtain the modules NcrlTerm-SA, NcrlTerm-BS and NcrlTerm-RS by applying the tool of De Jonge (1997) to Term-SA, Term-BS and Term-RS with renaming [Term $\Rightarrow \mathbb{P}$ Reduct $\Rightarrow \mathbb{P}$Reduct]. The module NcrlTerm-TS can be generated according to the scheme of Section 4.1.

The transformation 'to-basic' below transforms a process term into an equivalent basic term. For instance,

$$\text{to-basic}(((a_1 + a_2) \cdot b_1) \cdot c_1 \lhd \text{and}(\top, \bot) \rhd \delta \cdot a_3)$$

normalizes to the basic term

$$a_1 \cdot b_1 \cdot c_1 \lhd \text{and}(\top, \bot) \rhd \delta + a_2 \cdot b_1 \cdot c_1 \lhd \text{and}(\top, \bot) \rhd \delta + \delta.$$

module ToBasic
imports NcrlAxioms$^{5.2}$ NcrlTerm-TS Defined-Traversals$^{4.2}$
exports
  context-free syntax
    "to-basic" "(" $\mathbb{P}$ ")" $\rightarrow \mathbb{P}$
equations

[1] $$\frac{[\text{once}(\mathtt{R1}) \cdot \text{outermost}(\mathtt{R2} + \mathtt{R3} + \mathtt{R4} + \mathtt{R5} + \mathtt{R6} + \mathtt{R7} + \mathtt{R8})]\ p = p'}{\text{to-basic}(p)\ =\ p'}$$

6 Box Expressions

Box expressions are used in typesetting languages to indicate the layout structure of a piece of text. Horizontal boxes are used for horizontal composition, vertical boxes for vertical composition, etc. The language of box expressions Box (Van den Brand and Visser, 1994, 1995, 1996) is a target independent intermediate language for pretty-printing and typesetting programs. Typically, a pretty-printer for some programming language translates the abstract syntax tree of a program to a box expression, which is then translated to the input format for the displaying device desired. A further discussion of this application can be found in Van den Brand and Visser (1996).

One of the target languages of the Box language is the typesetting language TEX. In order to express box expressions in TEX, box expressions are flattened by means of a (large) number of transformation rules (Van den Brand and Visser, 1994, 1995). One of the problems that we encountered was the following. The combination of the rules for repositioning comment boxes and the rules for flattening box expressions leads to a weakly terminating rewrite system that causes an infinite reduction under innermost rewriting. The solution chosen in the implementation was to first apply the rewrite rules for comments and then apply the flattening rules. This could only be achieved outside ASF+SDF by consecutively applying the rewrite rules in two different modules. In this section we discuss the fragment of the language and the transformation rules that contain the problem and show how it is solved inside ASF+SDF using strategies.

6.1 Syntax

Boxes are either strings or expressions composed by means of one of the operators H, V, HV, and VPAR. (In fact, there are more operators in the full language, but these are not of interest for the current paper.)

module Box
imports Layout Strings
exports
  sorts Box BoxList
  context-free syntax

String	$\rightarrow$ Box
"H" "[" BoxList "]"	$\rightarrow$ Box
"V" "[" BoxList "]"	$\rightarrow$ Box
"HV" "[" BoxList "]"	$\rightarrow$ Box
"VPAR" "[" BoxList "]"	$\rightarrow$ Box
Box$*$	$\rightarrow$ BoxList

variables

$[A\text{-}E][0\text{-}9']*$	$\rightarrow$ Box
$[A\text{-}E]$"$*$"$[0\text{-}9']*$	$\rightarrow$ Box$*$
$[A\text{-}E]$"$+$"$[0\text{-}9']*$	$\rightarrow$ Box$+$

6.2 Transformation Rules

The specification of the Box language contains a large number of transformation rules on box expressions that are used to flatten expressions. Here we show a few of these to discuss the transformation problem, but they are not sufficient to get the desired normal forms. The first two rules are in the set of rules that flatten terms by moving horizontal compositions (H) inside vertical compositions (V and HV). They take the part of a horizontal composition after a vertical composition and attach it horizontally to the last box in the vertical composition. The last two rules move a comment box (VPAR) outside horizontal and horizontal-vertical boxes until they are in a vertical environment.

module Box-Laws
imports Box$^{6.1}$ Box-SA
equations

[`H-V`] $\text{H}[A^* \ \text{V}[B^* \ B] \ C^+] = \text{H}[A^* \ \text{V}[B^* \ \text{H}[B \ C^+]]]$
[`H-HV`] $\text{H}[A^* \ \text{HV}[B^* \ B] \ C^+] = \text{H}[A^* \ \text{HV}[B^* \ \text{H}[B \ C^+]]]$

[`H-VPAR`] $B^* \ \text{H}[C^* \ \text{VPAR}[D^*]] \ E^* = B^* \ \text{H}[C^*] \ \text{VPAR}[D^*] \ E^*$
[`HV-VPAR`] $B^* \ \text{HV}[C^* \ \text{VPAR}[D^*]] \ E^* = B^* \ \text{HV}[C^*] \ \text{VPAR}[D^*] \ E^*$

The labels `Flatten` and `Comment` are abbreviations for the systems for flattening and normalizing comments, repectively.

`Flatten` = `H-V` + `H-HV`
`Comment` = `H-VPAR` + `HV-VPAR`

Both `Flatten` and `Comment` are terminating rewrite systems, but their union is not. Consider for instance the following sequence of transformations:

```
(1)      V[H["a" HV["b" H["c" VPAR["d"]]]]]
     =   {H-VPAR}
(2)      V[H["a" HV["b" H["c"] VPAR["d"]]]]
     =   {HV-VPAR}
(3)      V[H["a" HV["b" H["c"]] VPAR["d"]]]
     =   {H-VPAR}
(4)      V[H["a" HV["b" H["c"]]] VPAR["d"]]
```

The last box expression is in normal form with respect to the rules above. However, after step (3) we could also have chosen the alternative sequence:

```
(3)       V[H["a" HV["b" H["c"]] VPAR["d"]]]
      =   {H-HV}
(4')      V[H["a" HV["b" H["c" VPAR["d"]]]]]
```

to get back our original box expression. In fact, an innermost strategy will do exactly this, giving rise to an infinite reduction.

6.3 Normalization Strategy

To overcome the termination problem sketched above we define a strategy that applies the comment rules and the flattening rules in sequence. In fact for the rules discussed above it suffices to apply the comment rules in a bottom-up fashion to each operator in a box-expression.

module Box-Normalization
imports Box-TS Box-Laws$^{6.2}$ Defined-Traversals$^{4.2}$
exports
  context-free syntax
    normalize(BoxList) $\rightarrow$ BoxListReduct
equations
The normal form of a box list is obtained by first applying comment rules in a bottom-up traversal and then applying the flattening rules with an innermost strategy.

[1]
$$\frac{[\text{bu}(\texttt{Comment}\ *)\cdot\text{innermost}(\texttt{Flatten})]\ B^* = C^*}{\text{normalize}(B^*)\ =\ C^*}$$

7 Comparison with ELAN

Our strategy language shows much resemblance with that of ELAN. Both languages contain operators for sequential composition, nondeteministic choice, and left choice and allow for the definition of strategies using rewrite rules.

ELAN does not have a recursion operator. Recursive strategies must also be expressed by means of rewrite rules in combination with an extra built-in strategy that ensures the lazy evaluation of such rules. The problem with this built-in strategy is that it is not formally defined, as is our fixed-point recursion.

Furthermore, ELAN has a different way of expressing that a strategy must be applied at a proper subterm of a term, instead of at the root. ELAN does not contain general push-down operators, such as our $\Box$ and $\Diamond$. Instead, the notion of a *congruence strategy* is used. Below, we will discuss the relation between these two approaches.

7.1 Congruence Strategies

In ELAN, for each n-ary function f in the signature a 'congruence strategy' $f(s_1,\ldots,s_n)$ is defined according to the following schema:

$$\frac{[s_1]\,t_1 = t_1',\ \ldots,\ [s_n]\,t_n = t_n'}{[f(s_1,\ldots,s_n)]\,f(t_1,\ldots,t_n) = f(t_1',\ldots,t_n')}$$

For a given signature our conjunctive push-down can be defined as a sum of congruences, i.e., if $f_1,\ldots,f_m$ are the functions of the signature, then $\Box(s)$ is defined by

$$\Box(s) = f_1(s,\ldots,s) + \cdots + f_m(s,\ldots,s)$$

Conversely, the congruence strategies of ELAN can be expressed in our strategy language via the following construction. For each function f in the signature define a strategy label $\texttt{f}$ that succeeds if it is applied to a term with f as root symbol.

$$[\texttt{f}]\,f(x_1,\ldots,x_n) = f(x_1,\ldots,x_n)$$

This rule does not transform the term; it only tests its leading function symbol. Now the congruence strategy $f(s,\ldots,s)$ can be expressed as $\mathtt{f}\cdot\Box(s)$, i.e., `f` is a *guard* for the application of $\Box(s)$.

To express congruence strategies $f(s_1,\ldots,s_n)$ in which the substrategies $s_1,\ldots,s_n$ are different, we need a family of push-down operators. Define for each arity n an operator $\mathrm{pd}^n_\oplus$ by means of the following schema:

$$\frac{[s_1]\,t_1 = r_1,\ \ldots,\ [s_n]\,t_n = r_n,\ \oplus\{\pi_{\mathrm{b}}(r_1),\ldots,\pi_{\mathrm{b}}(r_n)\} = \mathrm{true}}{[\mathrm{pd}^n_\oplus(s_1,\ldots,s_n)]\,f(t_1,\ldots,t_n) = f(\pi_{\mathrm{t}}(r_1),\ldots,\pi_{\mathrm{t}}(r_n))}$$

Clearly, the schema for $\mathrm{pd}^n_\oplus$ can be instantiated in the same way as that for $\mathrm{pd}_\oplus$. As in the previous section, we obtain for every arity n the conjunctive and disjunctive push-down operators $\Box^n$ and $\Diamond^n$ as instantiations of this generic operator $\mathrm{pd}^n_\oplus$. Now, ELAN's strategy $f(s_1,\ldots,s_n)$ is expressed in our language by $\mathtt{f}\cdot\Box^n(s_1,\ldots,s_n)$. Also, by means of such constructions we can define strategies such as map(s) that applies s to all elements of a cons/nil list.

$$\begin{array}{lll} [\mathtt{nil}] & \mathrm{nil} & = \mathrm{nil} \\ [\mathtt{cons}] & \mathrm{cons}(x,l) & = \mathrm{cons}(x,l) \end{array}$$

$$\mathrm{map}(s) = \mu\mathtt{x}.\mathtt{nil} + \mathtt{cons}\cdot\Box^2(s,\mathtt{x})$$

8 Concluding Remarks

We have described the setup of a language of term rewriting strategies and its interpretation in ASF+SDF. This approach gives us the possibility to control the transformation of expressions, given a set of labeled rewrite rules. The main result of this paper is the definition of the push-down operators as primitives to define term traversals in a general way.

The work in this paper opens up a range of further research issues.

Generic Specification In this paper we have made use of generic specifications to express the semantics of strategies. This genericity is of two kinds. The first kind requires the instantiation of a generic module, achieved by renaming its sorts. A tool for this purpose is discussed by De Jonge (1997). The second kind requires the generation of equations for the push-down operator for each constructor in the signature of the language under consideration. This can be achieved by a variant of the pretty-printer generation techniques described by Van den Brand and Visser (1996).

Strategy Operators We have defined parallel innermost and parallel outermost in terms of the push-down operators $\Box$ and $\Diamond$. These operators could be called *greedy*; they apply to as many arguments as possible. Using the nongreedy variants of these operators, say ■ and ⧫, other schemes may be defined. For instance,

$$\mathrm{reduce}(s) = (\mu v.(s + \blacklozenge(v)))*$$

defines arbitrary reduction; the operator 'reduce' repeatedly ($*$) selects redexes nondeterministically ($+$).

Parameterized Strategies By parameterizing labels with extra information we can define still more powerful transformation systems. Consider for example the substitution of terms for variables. The substitution of a term t for variable x in a term t' consists of a traversal of the term t' replacing all occurrences of x by t. Using strategies this is concisely specified by defining the label strategy $x := t$ as $[x := t]\,x = t$ to express the replacement of a variable. Substitution is then expressed as a traversal in the term t' applying the replacement everywhere, i.e., $[\mathrm{once}(x := t)]\,t'$. Here we directly reuse the generated traversal. There are many similar applications of strategies parameterized with data.

While it is straightforward to distribute information over a term, accumulation is not and is an issue to be addressed in future research.

Optimization In this paper we have discussed the interpretation of strategies in ASF+SDF using the built-in innermost rewrite engine. We have focussed on the definition of the strategy operators that we consider fundamental; we tried to keep this definition as simple as possible. However, if efficiency is at stake, one could for instance consider a lazy interpretation of the sequential choice (this would require an auxiliary operator).

Once more experience with controlled rewriting has been gained, it might be interesting to consider the integration of the strategy language in the rewrite engine of ASF+SDF itself to get a more efficient interpretation.

Another approach to the optimization of strategies is to consider transformation of strategy expressions to more efficient expressions. For example, nested loops could be merged. Such optimizations require an algebra of strategies.

Correspondence with Trace Semantics At a first glance it seems that there is a natural correspondence between trace semantics and the way strategy expressions are interpreted in our system. A strategy expression could be viewed as a specification of a set of paths in the reduction graph of a term. However, while the law

$$(x+y)\cdot z = x\cdot z + y\cdot z \tag{1}$$

is sound for trace semantics, it does not hold for the interpreter defined here. Suppose that $[x]\,t = t'$, $[y]\,t = t''$, $[z]\,t' = t'''$, but $[z]\,t'' = t''\uparrow$. Then, on the one hand, $[x\cdot z + y\cdot z]t$ will always yield t'''. On the other hand, $[(x+y)\cdot z]t$ has two possible outcomes: t''' and $t\uparrow$. That is, backtracking is only performed over choice and not over sequential composition.

The easiest way to resolve this, is to literally add the above law as an equation over strategy expressions. Given the fact that ASF+SDF implements a left-most innermost reduction strategy, this will yield the desired result. However, a drawback of this solution is that interpretation is only defined for strategies in normal form with respect to (1).

Correspondence with Modal Logic Another correspondence that comes to mind is the relation to modal logic. The operators $\cdot$ and $+$ correspond to conjunction and disjunction of success; the push-down operators $\Box$ and $\Diamond$ are so chosen because of their similarity to the modalities in modal logic. (Blackburn *et al.* (1993) discuss a modal logic of trees that contains modalities to traverse a tree.) However, the influence of the outcome of x on the outcome of y in $x\cdot y$ is not expressible by a logical conjunction only.

In the strategy language as defined in this paper there is an interesting interaction between operational behaviour—transformation of a term—and the success or failure of the application of a strategy to a term. Further research is needed to find algebraic and logical laws that formalize this interaction.

References

Baeten, J. C. M. and Weijland, W. P. (1990). *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Bergstra, J. A. and Klop, J. W. (1985). Algebra of communicating processes with abstraction. *Theoretical Computer Science*, **37**(1), 77–121.

Blackburn, P., Gardent, C., and Meyer-Viol, W. (1993). Talking about trees. In *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics*, pages 21–29.

Borovanský, P., Kirchner, C., and Kirchner, H. (1996). Controlling rewriting by rewriting. *Electronic Notes in Theoretical Computer Science*, **4**. In J. Meseguer (ed.) *First International Workshop on Rewriting Logic and its Applications.* Asilomar Conference Center, Pacific Grove, CA, September 3-6, 1996.

Bosscher, D. J. B. and Ponse, A. (1995). Translating a process algebra with symbolic data values to linear format. In U. H. Engberg, K. G. Larsen, and A. Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number NS-95-2 in BRICS Notes Series. BRICS.

Van den Brand, M. G. J. and Visser, E. (1994). From Box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam.

Van den Brand, M. G. J. and Visser, E. (1995). Box: Language, laws and formatters (version 1.4). Technical documentation, Programming Research Group, University of Amsterdam.

Van den Brand, M. G. J. and Visser, E. (1996). Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, **5**(1), 1–41.

Van Deursen, A., Heering, J., and Klint, P., editors (1996). *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore.

Groote, J. F. and Ponse, A. (1994a). Proof theory for μCRL: A language for processes with data. In D. Andrews, J. Groote, and C. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computing, pages 232–251, Utrecht, The Netherlands. Springer-Verlag. An extended version appeared as chapter 4 of Ponse (1992).

Groote, J. F. and Ponse, A. (1994b). The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62, Utrecht, The Netherlands. Springer-Verlag.

De Jonge, M. (1997). Reuse of ASF+SDF specifications by means of renaming. Technical Report P9718, Programming Research Group, University of Amsterdam.

Ponse, A. (1992). *Process Algebras with Data*. Ph.D. thesis, University of Amsterdam.

Vittek, M. (1994). *ELAN: Un cadre logique pour le prototypage de language de programmation avec contraintes*. Ph.D. thesis, Université Henri Poincaré – Nancy I, Nancy, France.

A Substitution

In this section we define the substitution of strategies for strategy variables. This operation is used to define the semantics of the fixed point operator $\mu v.s$. Because this operator is a binding operator we have to take care that free variables do not become bound.

module Strategy-Substitution
imports Basic-Strategies$^{2.1}$ Recursive-Strategies$^{3.1}$ Push-Down$^{4.1}$
exports
  sorts StratSubst
  context-free syntax

Var "∈" "FV" "(" Strategy ")"	$\rightarrow$ Bool
"prime" "(" Var ")"	$\rightarrow$ Var
"get-fresh" "(" Var "," Strategy ")"	$\rightarrow$ Var
"[" Var ":=" Strategy "]"	$\rightarrow$ StratSubst
Strategy StratSubst	$\rightarrow$ Strategy

  priorities
    Strategy StratSubst $\rightarrow$ Strategy $>$ Strategy "."Strategy $\rightarrow$ Strategy
  variables
    "*c*""+"[*0-9′*]$*$ $\rightarrow$ CHAR+
equations
Free variables. The predicate $v \in \mathrm{FV}(s)$ determines whether the variable v has a free occurence in s.

[1] $$v \in \mathrm{FV}(l) = \text{false}$$
[2] $$v \in \mathrm{FV}(v) = \text{true}$$
[3] $$v \in \mathrm{FV}(v') = \text{false} \quad \textbf{when} \quad v \neq v'$$

[4] $$v \in \mathrm{FV}(\epsilon) = \mathrm{false}$$
[5] $$v \in \mathrm{FV}(s_1 + s_2) = v \in \mathrm{FV}(s_1) \vee v \in \mathrm{FV}(s_2)$$
[6] $$v \in \mathrm{FV}(s_1 \cdot s_2) = v \in \mathrm{FV}(s_1) \vee v \in \mathrm{FV}(s_2)$$
[7] $$v \in \mathrm{FV}(s_1 \mathrel{\triangleleft\!\!\!+} s_2) = v \in \mathrm{FV}(s_1) \vee v \in \mathrm{FV}(s_2)$$
[8] $$v \in \mathrm{FV}(\mathrm{pd}_{\oplus}(s)) = v \in \mathrm{FV}(s)$$
[9] $$v \in \mathrm{FV}(\mu v.\, s) = \mathrm{false}$$
[10] $$v \in \mathrm{FV}(\mu v'.\, s) = v \in \mathrm{FV}(s) \quad \textbf{when } v \neq v'$$

The function ‘prime’ renames a variable by adding a prime as its last character.

[11] $$\mathrm{prime}(\mathrm{var}(c^{+})) = \mathrm{var}(c^{+}\ \texttt{"}\,'\,\texttt{"})$$

The function ‘get-fresh’ primes a variable as long as it occurs freely in a strategy s.

[12] $$\frac{v \in \mathrm{FV}(s) = \mathrm{true}}{\text{get-fresh}(v, s) \ = \ \text{get-fresh}(\mathrm{prime}(v), s)}$$

[13] $$\frac{v \in \mathrm{FV}(s) = \mathrm{false}}{\text{get-fresh}(v, s) \ = \ v}$$

Substitution. Replace all free occurences of a variable v by a strategy s. For all operators except the fixed point operator this entails a simple traversal of the strategy term replacing v everywhere.

[14] $$l\ [v := s] = l$$
[15] $$v\ [v := s] = s$$
[16] $$v\ [v' := s] = v \quad \textbf{when } v \neq v'$$
[17] $$\epsilon\ [v := s] = \epsilon$$
[18] $$(s_1 + s_2)\ [v := s] = s_1\ [v := s] + s_2\ [v := s]$$
[19] $$(s_1 \cdot s_2)\ [v := s] = s_1\ [v := s] \cdot s_2\ [v := s]$$
[20] $$(s_1 \mathrel{\triangleleft\!\!\!+} s_2)\ [v := s] = s_1\ [v := s] \mathrel{\triangleleft\!\!\!+} s_2\ [v := s]$$
[21] $$\mathrm{pd}_{\oplus}(s')\ [v := s] = \mathrm{pd}_{\oplus}(s'\ [v := s])$$

If v is bound by a fixed point operator it is not replaced.

[22] $$(\mu v.\, s)\ [v := s'] \ = \ \mu v.\, s$$

If the bound variable v' does not occur free in s' then the substitution can be applied to the body of the fixed point operator.

[23] $$\frac{v \neq v',\ \ v' \in \mathrm{FV}(s') = \mathrm{false}}{(\mu v'.\, s)\ [v := s'] \ = \ \mu v'.\, s\ [v := s']}$$

If v' occurs free in s', then we must apply α-conversion. The occurence of v' in s' must not become bound.

[24] $$\frac{v \neq v',\ \ v' \in \mathrm{FV}(s') = \mathrm{true},\ \ \text{get-fresh}(v', s') = v''}{(\mu v'.\, s)\ [v := s'] \ = \ \mu v''.\, s\ [v' := v'']\ [v := s']}$$