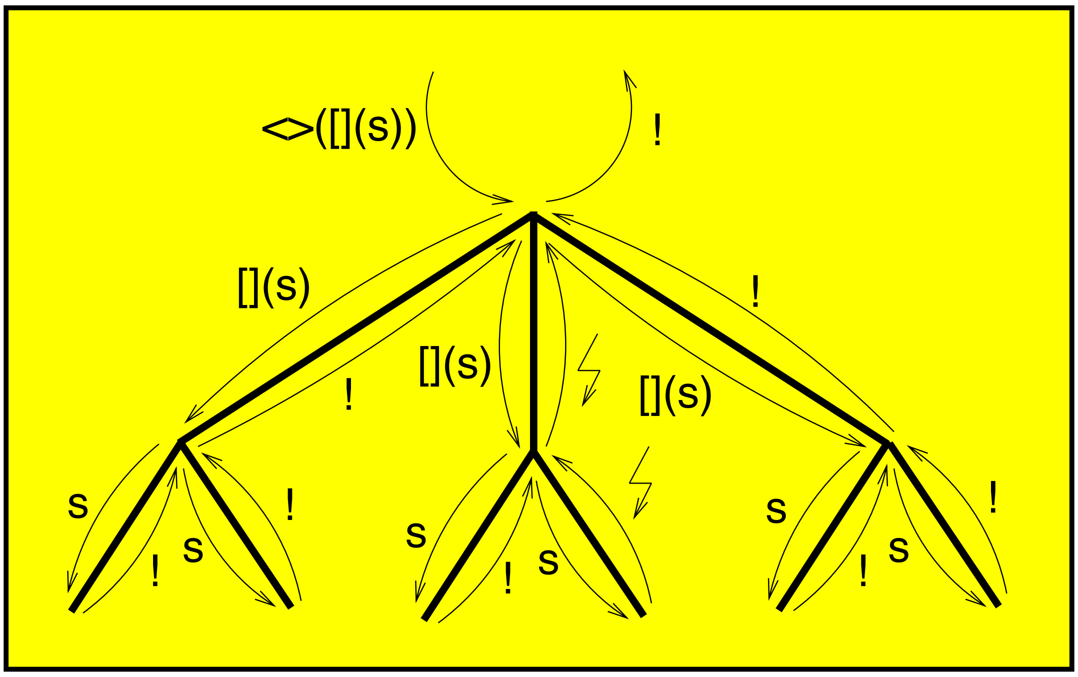
University of Amsterdam

Programming Research Group



Specification of Rewriting Strategies

Bas Luttik
Eelco Visser

Report P9710 June 1997

University of Amsterdam

Department of Computer Science

Programming Research Group

Specification of rewriting strategies

Bas Luttik

Eelco Visser

Report P9710

June 1997

Bas Luttik

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

CWI

P.O. Box 94079
NL-1090 GB Amsterdam
The Netherlands

tel. +31 20 592 4247
e-mail: luttik@cwi.nl

E. Visser

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7590
e-mail: visser@wins.uva.nl

Universiteit van Amsterdam, 1997

Specification of Rewriting Strategies

Bas Luttik[1,2] Eelco Visser[2]

1: CWI, PO Box 94079, 1090 GB Amsterdam, The Netherlands.
2: Programming Research Group, University of Amsterdam,
Kruislaan 403, 1098 SJ, Amsterdam, The Netherlands.
email: luttik@cwi.nl, visser@wins.uva.nl

June 17, 1997

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, non-deterministic choice, sequential choice, fixed point recursion, and a primitive for expressing term traversal. The traversal primitive called 'push-down' applies a strategy to the direct descendants of a term. Several complex term traversal strategies such as bottom-up and top-down application and innermost and outermost reduction can be defined in terms of push-down. 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 redex selection scheme, and the possibilities the user has to control the order in which rules are tried are rather limited. For example, ASF+SDF 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 performing transformations in the right order. This gives a great overhead in the specification and distracts from the 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 (Van Deursen *et al.*, 1996). 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 a setting of process algebra with conditionals. Strategies are necessary there, because one of the transformation rules is non-terminating. 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 does not terminate with standard innermost rewriting. Using the strategy language a terminating strategy is specified.

Overview In §2 we introduce the basic strategy operators: labels indicating axioms, the identity strategy, sequential composition of strategies, non-deterministic choice and sequential choice between strategies. In §3 we extend this basic language with a fixed point operator for the expression of recursive strategies. As an application an iteration operator for strategies is defined using the fixed point operator. In §4 several strategies traversing terms are defined using a generic primitive called push-down that applies a strategy to the direct descendants of a term. The traversal strategies include bottom-up application of a strategy and innermost and outermost normalization using a strategy. In §5 the strategy language is applied to the transformation of process expressions. In §6 an application to transformation of box expressions is discussed.

2 Basic Strategies

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

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

is 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 succesful application of the strategy, or the original application, 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 $\pi_{\mathrm{t}}(r)$ gives the term part of this pair and the function $\pi_{\mathrm{b}}(r)$ determines the Boolean value.

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

Term	$\to$ Reduct
"[" Strategy "]" Term	$\to$ Reduct
$\pi_{\rm t}$(Reduct)	$\to$ Term
$\pi_{\rm b}$(Reduct)	$\to$ Bool

  variables
    "t"$[0\text{-}9']*$ $\to$ Term
    "r"$[0\text{-}9']*$ $\to$ Reduct
equations

[1]	$\pi_{\rm t}(t) = t$	[2]	$\pi_{\rm b}(t) =$ true
[3]	$\pi_{\rm t}([s]\ t) = t$	[4]	$\pi_{\rm b}([s]\ t) =$ false

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

2.1 Syntax

Labels are atomic strategies. A label is an identifier starting with an uppercase letter. Their interpretation is defined by the user by means of labeled rewrite rules. Basic strategies are composed from labels by means of the identity composition (ϵ), sequential composition ($s_1 \cdot s_2$), non-deterministic choice ($s_1 + s_2$) and sequential choice ($s_1 \rhd s_2$).

module Basic-Strategies
imports Layout
exports
  sorts Label Strategy
  lexical syntax
    [A-Z][A-Za-z0-9\–]$*$ $\to$ Label
  context-free syntax

Label	$\to$ Strategy	
"ϵ"	$\to$ Strategy	
Strategy "." Strategy	$\to$ Strategy	{**left**}
Strategy "+" Strategy	$\to$ Strategy	{**left**}
Strategy "$\rhd$" Strategy	$\to$ Strategy	{**right**}
"(" Strategy ")"	$\to$ Strategy	{**bracket**}

  priorities
    Strategy "."Strategy $\to$ Strategy $>$ Strategy "$\rhd$"Strategy $\to$ Strategy $>$
    Strategy "+"Strategy $\to$ Strategy
  variables
    "l"$[0\text{-}9']*$ $\to$ Label
    "s"$[0\text{-}9']*$ $\to$ 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 following rules 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 $[l]\ t$ is in normal form. 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.

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 non-deterministic 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 sequential choice $s_1 \triangleright s_2$ 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] $$\frac{[s_1]\ t = t'}{[s_1 \triangleright s_2]\ t\ =\ t'}$$

[6] $$\frac{[s_2]\ t = t'}{[s_1 \triangleright s_2]\ t\ =\ t'} \quad \textbf{otherwise}$$

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 instiantiated to 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 in order to repeatedly apply a strategy. At the end of this section we show how the fixed point operator 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 occurences 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 all free occurences of v in s by s'. Substitution is defined in §A. Because this equation leads to a non-terminating innermost rewrite system, we interpret the fixed point operator lazily as defined 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 "$*$" $\to$ Strategy
    Strategy "$+$" $\to$ Strategy
  priorities

    {Strategy "$*$" $\to$ Strategy, Strategy "$+$" $\to$ Strategy} $>$
    Strategy "."Strategy $\to$ 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. This entails that the strategy always succeeds. The function get-fresh is used to create a binding variable v that does not occur freely in the strategy s. Observe how the sequential choice operator $\triangleright$ is used to enforce as many applications as possible. If s can not be applied the sequential composition $s \cdot v$ fails and ϵ is applied and succeeds. The $s+$ operator is expressed in terms of the $*$ operator. A strategy $s+$ succeeds if s succeeds at least once.

module CS-Interpretation
imports Complex-Strategies[3.3] Strategy-Substitution[A]
equations

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

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

4 Traversal Strategies

The strategies we have discussed until now apply to the root of a term. Now we introduce a strategy operator that can be used to define a wide range of term traversals. The 'push-down' of a strategy applies it to all arguments of the root function symbol. Using this operator we can define bottom-up and top-down traversals of terms, as well as innermost and outermost redex selection schemes.

4.1 Preliminaries

Before we introduce the traversal strategies we need the following definition of Boolean operators generalized to arbitrary many arguments, i.e., lists 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 parameters of the function 'pd' defined below.

module Booleans-Generalized
imports Booleans
exports
  sorts BoolOp
  context-free syntax

"$\bigwedge$"	$\rightarrow$ BoolOp
"$\bigvee$"	$\rightarrow$ BoolOp
BoolOp "{" Bool$*$ "}"	$\rightarrow$ Bool

  variables
    "$\oplus$"$[\mathit{0}\text{-}\mathit{9}']*$ $\rightarrow$ BoolOp
hiddens
  variables

"b"$[\mathit{0}\text{-}\mathit{9}']*$	$\rightarrow$ Bool
"b" "$*$"$[\mathit{0}\text{-}\mathit{9}']*$	$\rightarrow$ Bool$*$

equations

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

4.2 Syntax

We define the following strategy operators: the parameterized push-down operator ($\mathrm{pd}_{\oplus}$), the conjunctive ($\Box$) and disjunctive ($\Diamond$) push-down operators, the bottom-up (bu) and top-down (td) traversals, and the 'once', 'innermost' and 'outermost' selection schemes.

module Traversal-Strategies
imports Recursive-Strategies$^{3.1}$ Booleans-Generalized$^{4.1}$
exports
  context-free syntax

pd "_" BoolOp "(" Strategy ")"	$\rightarrow$ Strategy
$\Box$(Strategy)	$\rightarrow$ Strategy
$\Diamond$(Strategy)	$\rightarrow$ Strategy
bu(Strategy)	$\rightarrow$ Strategy
td(Strategy)	$\rightarrow$ Strategy
once(Strategy)	$\rightarrow$ Strategy
innermost(Strategy)	$\rightarrow$ Strategy
outermost(Strategy)	$\rightarrow$ Strategy

4.3 Interpretation

The interpretation of these strategies is presented in two parts. First we define the generic schema for the parameterized push-down operator in module Term-TS. Then we define the other strategies in terms of this push-down.

Push-Down The push-down $\mathrm{pd}_\oplus(s)$ applies s to each argument of the root symbol of term t and substitutes the result for each of the arguments for which application s succeeds. The strategy is parameterized with a Boolean operator $\oplus$ that determines the 'succes behaviour'.

The push-down strategy 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. Strictly speaking we only need to instiantiate it for the constructors of the language. 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$^{3.2}$ Traversal-Strategies$^{4.2}$ Booleans-Generalized$^{4.1}$
equations
For each function $f : s_1 \times \cdots \times s_n \to s_0$ in the signature[1] of a specification define a rule

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

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 $t_i = \pi_\mathrm{t}(r_i)$.

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

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

This entails that a push-down on a constant succeeds depending on the default value of the boolean operator $\oplus$. In case it is a conjunction it succeeds and in case of 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.

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

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.

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

[1]In the schemata in this section we abstract from the grammatical (mix-fix function) aspect of SDF signatures.

Traversal Strategies Given the push-down strategy we can define several term traversal strategies.

module TS-Interpretation
imports Traversal-Strategies$^{4.2}$ CS-Interpretation$^{3.3}$ Strategy-Substitution^{A}
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 descendants and that $\Diamond(s)$ succeeds if s succeeds for at least one direct descendant.

[1] $$\Box(s) = \mathrm{pd}_{\bigwedge}(s)$$
[2] $$\Diamond(s) = \mathrm{pd}_{\bigvee}(s)$$

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 descendants 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.

[3] $$\frac{\text{get-fresh}(\mathtt{x},\ s) = v}{\mathrm{bu}(s) \ = \ \mu\ v\ .\ \Box(v) \cdot s}$$

[4] $$\frac{\text{get-fresh}(\mathtt{x},\ s) = v}{\mathrm{td}(s) \ = \ \mu\ v\ .\ s \cdot \Box(v)}$$

The strategy once(s) applies a strategy s once to each node in a term. In case s does not apply to a node the strategy succeeds with the identity strategy.

[5] $$\mathrm{once}(s) = \mathrm{bu}(s \triangleright \epsilon)$$

The strategy innermost(s) applies a strategy s bottom-up to a term until it is no longer applicable. The strategy outermost(s) applies a strategy top-down until it no longer applies.

[6] $$\frac{\text{get-fresh}(\mathtt{x},\ s) = v}{\mathrm{innermost}(s) \ = \ \mu\ v\ .\ \mathrm{bu}(s \cdot v \triangleright \epsilon)}$$

[7] $$\frac{\text{get-fresh}(\mathtt{x},\ s) = v}{\mathrm{outermost}(s) \ = \ (\mu\ v\ .\ s \triangleright \Diamond(v))\ *}$$

The innermost strategy works by two recursion steps: the bottom-up loop applies the strategy s to each node of a term starting at the leafs. If that succeeds the entire innermost strategy is recursively applied again to the term, otherwise the term is in normal form. This entails that when s is applied to a term, its direct descendants are in normal form.

The outermost strategy works also by means of two loops. The outer loop is an iteration that applies the inner loop as long as it is applicable. The inner loop is a disjunctive variant of the top-down strategy. It tries to apply the strategy s. If that succeeds the strategy terminates succesfully. If it fails the recursive application $\Diamond(v)$ tries to apply s to at least one of the descendants of the term. In this manner all outermost redexes of s are rewritten one step.

5 Process Expressions

In process algebra (see Baeten and Weijland, 1990) it is common to prove a lemma stating that every term is equal to some term with a simpler inductive structure, a so-called *basic term.* Usually, such a lemma is proved by defining of 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 process terms with a rule that is non-terminating, 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 $\mathcal{A}$.

```
module BoolTerms
exports
  sorts 𝔹
  context-free syntax
    "⊤"                      → 𝔹
    "⊥"                      → 𝔹
    "and" "(" 𝔹 "," 𝔹 ")"    → 𝔹
    "not" "(" 𝔹 ")"          → 𝔹
  variables
    "b"[0-9′]∗ → 𝔹
```

```
module NcrlTerms
imports BoolTerms$^{5.1}$
exports
  sorts 𝒜 ℙ
  lexical syntax
    [abc][0-9′]∗ → 𝒜
  context-free syntax
    "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
    "p"[0-9′]∗ → ℙ
```

5.2 Transformation Rules

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

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

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

$$
\begin{array}{ll}
[\texttt{R1}]\ x \lhd b \rhd y & = x \lhd b \rhd \delta + y \lhd \mathrm{not}(b) \rhd \delta \\
[\texttt{R2}]\ (x + y) \cdot z & = x \cdot z + y \cdot z \\
[\texttt{R3}]\ (x \cdot y) \cdot z & = x \cdot y \cdot z \\
[\texttt{R4}]\ \delta \cdot x & = \delta \\
[\texttt{R5}]\ \delta \lhd b \rhd \delta & = \delta \\
[\texttt{R6}]\ (x \lhd b \rhd \delta) \cdot y & = x \cdot y \lhd b \rhd \delta \\
[\texttt{R7}]\ (x + y) \lhd b \rhd \delta & = x \lhd b \rhd \delta + y \lhd b \rhd \delta \\
[\texttt{R8}]\ (x \lhd b_1 \rhd \delta) \lhd b_2 \rhd \delta & = x \lhd \mathrm{and}(b_1, b_2) \rhd \delta
\end{array}
$$

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 q 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 non-terminating. 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 respectively 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 §4.3.

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

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

normalizes to the basic term

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

module ToBasic
imports NcrlAxioms$^{5.2}$ NcrlTerm-TS$^{B.1}$ TS-Interpretation$^{4.3}$
exports
  context-free syntax
    "to-basic" "(" $\mathbb{P}$ ")" $\rightarrow$ $\mathbb{P}$
equations

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

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 non-termination with 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 as much as possible. 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). The rules 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$^{B.2}$
equations

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

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

Each of the sets of rules `Flatten` and `Comment` is a terminating rewrite system, but their combination is non-terminating. 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 done

```
(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, leading to a non-terminating rewriting system.

6.3 Normalization Strategy

To overcome this termination problem 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$^{B.2}$ Box-Laws$^{6.2}$ TS-Interpretation$^{4.3}$
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] $$\frac{[\mathrm{bu}(\texttt{Comment}\ *)\cdot\mathrm{innermost}(\texttt{Flatten})]\ B^* = C^*}{\mathrm{normalize}(B^*)\ =\ C^*}$$

7 Conclusion

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 expression given a set of labeled rewrite rules. The main result of this paper is the definition of the push-down strategy as a primitive to define a wide range of term traversal strategies.

7.1 Future Work

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 non-greedy variants of these operators, say $\blacksquare$ and $\blacklozenge$, other schemes may be defined. For instance,

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

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

Parameterized Strategies We have considered strategies without data. By parameterizing labels with data 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 occurences 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 over the term t' applying the replacement everywhere, i.e., $[\text{once}(x := t)]\ t'$. Here we directly reuse the generated traversal. There are many similar applications of strategies parameterized with data.

It is thus straightforward to distribute data over a term. Accumulation of data is not so straightforward 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. Once more experience with controlled rewriting has been obtained it might be interesting to consider the integration of strategies 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, merging of nested loops. Such optimizations require an algebra of strategies.

Correspondences At a first glance it seems that there is a correspondence between the operators of process algebra (Baeten and Weijland, 1990) and of strategies. However, if we consider their semantics the correspondence is not so clear. For instance, consider the distribution laws for sequential composition over choice.

$$(x + y) \cdot z = (x \cdot z) + (y \cdot z) \tag{1}$$
$$x \cdot (y + z) = (x \cdot y) + (x \cdot z) \tag{2}$$

In BPA the first one holds and not the second, whereas in our strategies, the second one holds and not the first. In BPA the success or failure (deadlock) of the second element in a sequential composition does not affect the choice in the first element (equation (1)).

Another correspondence that comes to mind is the relation to modal logic. Clearly 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.

The strategies defined in this paper form a mixture between the actions of process algebras—transformation of a term—and the thruth of modal logic—success or failure of a strategy. Further research is needed to find the algebraic and logic laws of the interaction of process and logic in strategies.

References

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

Bergstra, J. and Klop, J. (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. 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. 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). Reusing ASF+SDF specifications by means of renaming. Technical report, Programming Research Group, University of Amsterdam. (Submitted for publication).

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$^{3.1}$ Complex-Strategies$^{3.3}$
    Traversal-Strategies$^{4.2}$ TS-Interpretation$^{4.3}$
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] $v \in \mathrm{FV}(l) = \mathrm{false}$
[2] $v \in \mathrm{FV}(v) = \mathrm{true}$
[3] $v \in \mathrm{FV}(v') = \mathrm{false}$ **when** $v \neq v'$
[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] $v \in \mathrm{FV}(s\,*) = v \in \mathrm{FV}(s)$
[8] $v \in \mathrm{FV}(s\,+) = v \in \mathrm{FV}(s)$

[9] $v \in \mathrm{FV}(s_1 \triangleright s_2) = v \in \mathrm{FV}(s_1) \vee v \in \mathrm{FV}(s_2)$

[10] $v \in \mathrm{FV}(\mathrm{pd}_{\oplus}(s)) = v \in \mathrm{FV}(s)$

[11] $v \in \mathrm{FV}(\mu\ v\ .\ s) = \text{false}$

[12] $v \in \mathrm{FV}(\mu\ v'\ .\ s) = v \in \mathrm{FV}(s) \quad \textbf{when}\ \ v \neq v'$

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

[13] $$\text{prime}(\text{var}(c^+)) = \text{var}(c^+\ \texttt{"'"})$$

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

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

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

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.

[16] $l\,[v := s] = l$

[17] $v\,[v := s] = s$

[18] $v\,[v' := s] = v \quad \textbf{when}\ \ v \neq v'$

[19] $\epsilon\,[v := s] = \epsilon$

[20] $(s_1 + s_2)\,[v := s] = s_1\,[v := s] + s_2\,[v := s]$

[21] $(s_1 \cdot s_2)\,[v := s] = s_1\,[v := s] \cdot s_2\,[v := s]$

[22] $s' *\,[v := s] = s'\,[v := s] *$

[23] $s' +\,[v := s] = s'\,[v := s] +$

[24] $(s_1 \triangleright s_2)\,[v := s] = s_1\,[v := s] \triangleright s_2\,[v := s]$

[25] $\mathrm{pd}_{\oplus}(s')\,[v := s] = \mathrm{pd}_{\oplus}(s'\,[v := s])$

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

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

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.

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

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

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

B Instantiated Modules

In this appendix we list the instantiations of the generic modules Term-* for the Process and Box case studies.

B.1 Processes

module NcrlTerm-SA
imports Basic-Strategies$^{2.1}$ Booleans NcrlTerms$^{5.1}$
exports
  sorts $\mathbb{P}$Reduct
  context-free syntax
    $\mathbb{P}$ $\rightarrow$ $\mathbb{P}$Reduct
    "[" Strategy "]" $\mathbb{P}$ $\rightarrow$ $\mathbb{P}$Reduct

    $\pi_\mathrm{t}(\mathbb{P}\mathrm{Reduct})$ $\rightarrow$ $\mathbb{P}$
    $\pi_\mathrm{b}(\mathbb{P}\mathrm{Reduct})$ $\rightarrow$ Bool
  variables
    "t"$[0\text{-}9']*$ $\rightarrow$ $\mathbb{P}$
    "r"$[0\text{-}9']*$ $\rightarrow$ $\mathbb{P}$Reduct
equations

[1]	$\pi_\mathrm{t}(t) \;=\; t$	[2]	$\pi_\mathrm{b}(t) \;=\; \mathrm{true}$
[3]	$\pi_\mathrm{t}([s]\ t) \;=\; t$	[4]	$\pi_\mathrm{b}([s]\ t) \;=\; \mathrm{false}$

module NcrlTerm-BS
imports NcrlTerm-SA$^{B.1}$
equations

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

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

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

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

[5] $$\frac{[s_1]\ t = t'}{[s_1 \rhd s_2]\ t \;=\; t'}$$

[6] $$\frac{[s_2]\ t = t'}{[s_1 \rhd s_2]\ t \;=\; t'} \quad \textbf{otherwise}$$

module NcrlTerm-RS
imports NcrlTerm-BS$^{B.1}$ Recursive-Strategies$^{3.1}$ Strategy-Substitution^{A}
equations

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

module NcrlTerm-TS
imports NcrlTerm-RS$^{B.1}$ Traversal-Strategies$^{4.2}$
equations

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

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

[3] $$\frac{[s]\ p_1 = r_1,\ [s]\ p_2 = r_2,\ \oplus\{\pi_{\text{b}}(r_1)\ \pi_{\text{b}}(r_2)\} = \text{true}}{[\text{pd}_{\oplus}(s)]\ p_1 \cdot p_2\ =\ \pi_{\text{t}}(r_1) \cdot \pi_{\text{t}}(r_2)}$$

Note: we implement first-order, single-sorted rewriting for terms of the sort NcrlTerm. Therefore, we consider $\lhd b \rhd$ as a binary operator taking two arguments of the sort NcrlTerm.

[4] $$\frac{[s]\ p_1 = r_1,\ [s]\ p_2 = r_2,\ \oplus\{\pi_{\text{b}}(r_1)\ \pi_{\text{b}}(r_2)\} = \text{true}}{[\text{pd}_{\oplus}(s)]\ p_1 \lhd b \rhd p_2\ =\ \pi_{\text{t}}(r_1) \lhd b \rhd \pi_{\text{t}}(r_2)}$$

[5] $$\frac{[s]\ p_1 = r_1,\ [s]\ p_2 = r_2,\ \oplus\{\pi_{\text{b}}(r_1)\ \pi_{\text{b}}(r_2)\} = \text{true}}{[\text{pd}_{\oplus}(s)]\ p_1 + p_2\ =\ \pi_{\text{t}}(r_1) + \pi_{\text{t}}(r_2)}$$

B.2 Box

module Box-SA
imports Basic-Strategies$^{2.1}$ Booleans Box$^{6.1}$
exports
  sorts BoxReduct BoxListReduct
  context-free syntax

Box	→ BoxReduct
"[" Strategy "]" Box	→ BoxReduct
BoxList	→ BoxListReduct
"[" Strategy "]" BoxList	→ BoxListReduct
π_{t}(BoxReduct)	→ Box
π_{b}(BoxReduct)	→ Bool
π_{t}(BoxListReduct)	→ BoxList
π_{b}(BoxListReduct)	→ Bool

variables

"r"[*0-9'*]$*$ $\to$ BoxReduct

"$r*$"[*0-9'*]$*$ $\to$ BoxListReduct

equations

[1]	$\pi_{\mathrm{t}}(A) \ = \ A$	[2]	$\pi_{\mathrm{b}}(A) \ = \ \mathrm{true}$
[3]	$\pi_{\mathrm{t}}([s]\ A) \ = \ A$	[4]	$\pi_{\mathrm{b}}([s]\ A) \ = \ \mathrm{false}$
[5]	$\pi_{\mathrm{t}}(A^*) \ = \ A^*$	[6]	$\pi_{\mathrm{b}}(A^*) \ = \ \mathrm{true}$
[7]	$\pi_{\mathrm{t}}([s]\ A^*) \ = \ A^*$	[8]	$\pi_{\mathrm{b}}([s]\ A^*) \ = \ \mathrm{false}$

module Box-BS
imports Box-SA$^{B.2}$
equations
Boxes

[1] $$[\epsilon]\ B \ = \ B$$

[2] $$\frac{[s_1]\ B = B',\ \ [s_2]\ B' = B''}{[s_1 \cdot s_2]\ B \ = \ B''}$$

[3] $$\frac{[s_1]\ B = B'}{[s_1 + s_2]\ B \ = \ B'}$$

[4] $$\frac{[s_2]\ B = B'}{[s_1 + s_2]\ B \ = \ B'}$$

[5] $$\frac{[s_1]\ B = B'}{[s_1 \rhd s_2]\ B \ = \ B'}$$

[6] $$\frac{[s_2]\ B = B'}{[s_1 \rhd s_2]\ B \ = \ B'} \quad \textbf{otherwise}$$

Lists

[7] $$[\epsilon]\ B^* \ = \ B^*$$

[8] $$\frac{[s_1]\ B^* = B'^*,\ \ [s_2]\ B'^* = B''^*}{[s_1 \cdot s_2]\ B^* \ = \ B''^*}$$

[9] $$\frac{[s_1]\ B^* = B'^*}{[s_1 + s_2]\ B^* \ = \ B'^*}$$

[10] $$\frac{[s_2]\ B^* = B'^*}{[s_1 + s_2]\ B^* \ = \ B'^*}$$

[11] $$\frac{[s_1]\ B^* = B'^*}{[s_1 \rhd s_2]\ B^* \ = \ B'^*}$$

[12] $$\frac{[s_2]\ B^* = B'^*}{[s_1 \rhd s_2]\ B^* \ = \ B'^*} \quad \textbf{otherwise}$$

module Box-RS
imports Box-BS$^{B.2}$ Recursive-Strategies$^{3.1}$ Strategy-Substitution^{A}
equations

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

[2] $$[\mu\ v\ .\ s]\ B^*\ =\ [s\,[v := \mu\ v\ .\ s]]\ B^*$$

module Box-TS
imports Box-RS$^{B.2}$ Traversal-Strategies$^{4.2}$
equations
Strings

[1] $$\frac{\oplus\{\} = \mathrm{true}}{[\mathrm{pd}_{\oplus}(s)]\ a\ =\ a}$$

Box lists

[2] $$\frac{\oplus\{\} = \mathrm{true}}{[\mathrm{pd}_{\oplus}(s)]\ =\ }$$

[3] $$\frac{[s]\ B = r,\ [\mathrm{pd}_{\oplus}(s)]\ B^* = r^*,\ \oplus\{\pi_{\mathrm{b}}(r)\ \pi_{\mathrm{b}}(r^*)\} = \mathrm{true},\ \pi_{\mathrm{t}}(r^*) = C^*}{[\mathrm{pd}_{\oplus}(s)]\ B\ B^*\ =\ \pi_{\mathrm{t}}(r)\ C^*}$$

Box operators

[4] $$\frac{[s]\ B^* = r^*,\ \oplus\{\pi_{\mathrm{b}}(r^*)\} = \mathrm{true}}{[\mathrm{pd}_{\oplus}(s)]\ \mathrm{H}[B^*]\ =\ \mathrm{H}[\pi_{\mathrm{t}}(r^*)]}$$

[5] $$\frac{[s]\ B^* = r^*,\ \oplus\{\pi_{\mathrm{b}}(r^*)\} = \mathrm{true}}{[\mathrm{pd}_{\oplus}(s)]\ \mathrm{V}[B^*]\ =\ \mathrm{V}[\pi_{\mathrm{t}}(r^*)]}$$

[6] $$\frac{[s]\ B^* = r^*,\ \oplus\{\pi_{\mathrm{b}}(r^*)\} = \mathrm{true}}{[\mathrm{pd}_{\oplus}(s)]\ \mathrm{HV}[B^*]\ =\ \mathrm{HV}[\pi_{\mathrm{t}}(r^*)]}$$

[7] $$\frac{[s]\ B^* = r^*,\ \oplus\{\pi_{\mathrm{b}}(r^*)\} = \mathrm{true}}{[\mathrm{pd}_{\oplus}(s)]\ \mathrm{VPAR}[B^*]\ =\ \mathrm{VPAR}[\pi_{\mathrm{t}}(r^*)]}$$

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (URL `http://www.wins.uva.nl/research/prog/reports/`) or using anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*

[P9504] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (editors). ASF+SDF*'95: a workshop on Generating Tools from Algebraic Specifications, May 11&12, 1995, CWI Amsterdam.*

[P9503] J.A. Bergstra and A. Ponse. *Frame-based process logics.*

[P9208c] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208b).*