|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | |  | |  | |
| Logo Clearsy -4,3x2,9cm HD | | Edition | | 0 | |
| Revision | | 1 | |
| Page number | | 42 | |
| State | | To be checked | |

***BXML***

***-***

***Format description***

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Redaction** | **Checked** | **Approved** |
| **Function** | Developer |  |  |
| **Name** | Lilian BURDY |  |  |
| **Date** |  |  |  |
| **Signature** |  |  |  |

# Revisions

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Author** | **Comment** |
| 0.1 | 6/10/2016 | L. Burdy |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Contents

[Revisions 2](#_Toc463523641)

[Contents 3](#_Toc463523642)

[I Introduction 5](#_Toc463523643)

[I.1 Purpose 5](#_Toc463523644)

[I.2 Convention 5](#_Toc463523645)

[II Documents 6](#_Toc463523646)

[II.1 Applicable documents 6](#_Toc463523647)

[II.2 Reference documents 6](#_Toc463523648)

[III Global BXML structure 7](#_Toc463523649)

[III.1 Attribute 7](#_Toc463523650)

[III.2 Machine 8](#_Toc463523651)

[III.2.1 Refines 8](#_Toc463523652)

[III.2.2 Parameters and Constraints 9](#_Toc463523653)

[III.2.3 Visibility clauses 10](#_Toc463523654)

[III.2.4 Values 11](#_Toc463523655)

[III.2.5 Sets 12](#_Toc463523656)

[III.2.6 Constants and variables 13](#_Toc463523657)

[III.2.7 Properties, invariant and assertions 14](#_Toc463523658)

[III.2.8 Initialisation and operations 15](#_Toc463523659)

[III.2.9 Type information 17](#_Toc463523660)

[III.3 Substitution 18](#_Toc463523661)

[III.3.1 BEGIN 19](#_Toc463523662)

[III.3.2 skip 19](#_Toc463523663)

[III.3.3 Assignment 19](#_Toc463523664)

[III.3.4 ASSERT 20](#_Toc463523665)

[III.3.5 IF 20](#_Toc463523666)

[III.3.6 SELECT 22](#_Toc463523667)

[III.3.7 CASE 23](#_Toc463523668)

[III.3.8 ANY 25](#_Toc463523669)

[III.3.9 LET 26](#_Toc463523670)

[III.3.10 Becomes in 27](#_Toc463523671)

[III.3.11 Becomes such that 28](#_Toc463523672)

[III.3.12 Local variable 29](#_Toc463523673)

[III.3.13 Sequence, Parallel and Choice 29](#_Toc463523674)

[III.3.14 Operation call 31](#_Toc463523675)

[III.3.15 While loop 32](#_Toc463523676)

[III.4 Predicate 33](#_Toc463523677)

[III.4.1 Binary predicate 33](#_Toc463523678)

[III.4.2 N-ary predicate 33](#_Toc463523679)

[III.4.3 Unary predicate 34](#_Toc463523680)

[III.4.4 Quantified predicate 34](#_Toc463523681)

[III.4.5 Expression comparison 34](#_Toc463523682)

[III.5 Expression 34](#_Toc463523683)

[III.5.1 Binary expression 35](#_Toc463523684)

[III.5.2 N-ary expression 36](#_Toc463523685)

[III.5.3 Unary expression 36](#_Toc463523686)

[III.5.4 Quantified expression 37](#_Toc463523687)

[III.5.5 Quantified Set 38](#_Toc463523688)

[III.5.6 Struct and record 38](#_Toc463523689)

[III.5.7 Terminal expression 39](#_Toc463523690)

[III.6 Identifier 40](#_Toc463523691)

[Index 42](#_Toc463523692)

# Introduction

## Purpose

The purpose of this document is to describe and illustrate the BXML Format.

BXML is an XML representation of a B Component.

## Convention

The format is described using the Relax NG compact syntax. Relax NG description are given in boxes

element XXX {

attribute XXX { text },

element XXX { XXX },

…

}

In textual description element name appears in **blue**, attribute name in **red**, and attribute value in **orange**.

# Documents

## Applicable documents

|  |  |  |
| --- | --- | --- |
| ***Reference*** | ***Version*** | ***Title*** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Reference documents

|  |  |  |
| --- | --- | --- |
| ***Reference*** | ***Version*** | ***Title*** |
| http://relaxng.org/compact-20021121.html |  | RELAX NG Compact Syntax |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Global BXML structure

A BXML file contains 6 different kind of structure:

* Component
* Substitution
* Predicate
* Expression
* Identifier
* Attribute

Each structure is described in Relax NG format in a specific file and corresponds to a pattern.

The main *bxml.rng* file includes the others, it structure is

# A RELAX NG compact syntax pattern for a bxml document.

grammar {

include "substitution.rng"

include "predicate.rng"

include "expression.rng"

include "identifier.rng"

include "attribute.rng"

start = element Machine { … }

Predicate |= …

ReferencedMachine = …

Operation = …

TypeInfo = …

Type =…

}

## Attribute

The BXML is a kind of “open” xml formal. That means that a BXML document can contain additional information. That information can be associated to every element of the BXML document. It is placed in an **Attr** element that can contain any well formed xml.

Attribute =

element Attr { anyElement\* }

anyElement =

element \* {

(attribute \* { text }

| text

| anyElement)\*

}

This element is always optional, if it exists it must be the first child of an element, see examples below.

## Machine

A BXML document main element is **Machine**.

element Machine {

attribute name { text },

attribute type { "abstraction" | "refinement" | "implementation" },

Attribute?,

…

}

The **name** attribute contains the name of component. The **type** attribute indicates if the component is an **abstraction**: MACHINE or SYSTEM, a **refinement** for a REFINEMENT or an **implementation** for an IMPLEMENTATION

Example:

The component:

MACHINE

M1

END

is described:

<Machine name = ‘M1’ type = ‘abstraction’>

<Attr>

<Pos l='4' c='1' s='18'/>

</Attr>

</Machine>

The element **Attr** can be present. In this example, it contains position information.

### Refines

A **Machine** element can contain an optional **Abstraction** element describing the REFINES clause.

element Machine {

…

element Abstraction { Attribute?, text }?,

…

}

This element contains a text that defines the refined component.

Example:

The component:

REFINEMENT

M1\_r

REFINES

M1

END

is described:

<Machine name = ‘M1\_r’ type = ‘refinement’>

<Abstraction>

M1

</Abstraction>

</Machine>

### Parameters and Constraints

Parameters and CONSTRAINTS in a component are described as optional **Parameters** and **Constraints** element.

element Machine {

…

element Parameters {

Attribute?,

element Id { Identifier }+

}?,

element Constraints { Attribute?, Predicate }?,

…

}

Predicate |= element Set { element Id { Identifier } }

A **Parameters** element contains a non-empty list of **Id** element containing identifier description (see II.6).

A **Constraints** element contains a predicate description (see II.4). This predicate contains also specific **Set** elements for sets declared in parameters.

Example:

The component:

MACHINE

MA(p1, ENS1)

CONSTRAINTS

p1 : ENS1

END

is described:

<Machine name = ‘M1’ type = ‘abstraction’>

<Parameters>

<Id value = ‘p1’ />

<Id value = ‘ENS1’ />

</Parameters>

<Constraints>

<Set><Id value = ‘ENS1’/></Set>

<Exp\_Comparison op = ‘:’ >

<Id value = ‘p1’ />

<Id value = ‘ENS1’ />

</Exp\_Comparison>

</Constraints>

</Machine>

### Visibility clauses

INCLUDES, IMPORTS, SEES, EXTENDS, USES are described by optional elements containing a non-empty list of referenced component.

element Machine {

…

element Includes { Attribute?, ReferencedMachine+ }?,

element Imports { Attribute?, ReferencedMachine+ }?,

element Sees { Attribute?, ReferencedMachine+ }?,

element Extends { Attribute?, ReferencedMachine+ }?,

element Uses { Attribute?, ReferencedMachine+ }?,

element Promotes {

Attribute?,

element Promoted\_Operation { Attribute?, text }+

}?,

…

}

ReferencedMachine =

element Referenced\_Machine {

Attribute?,

element Name { text },

element Instance {text}?,

element Parameters { Expression+ }?

}

A referenced machine contains the name of the machine and optionally the name of the instance and instantiated parameters.

The optional **Promotes** element contains a non-empty list of promoted operation described by their name.

Example:

The component:

MACHINE

MA

SEES

I1.Msees

IMPORTS

MB(10)

PROMOTES

op1

END

is described:

<Machine name = ‘MA’ type = ‘abstraction’>

<Imports>

<Referenced\_Machine>

<Name>MB</Name>

<Parameters>

<Integer\_Literal value=’10’/>

</Parameters>

</Referenced\_Machine>

</Imports>

<Sees>

<Referenced\_Machine>

<Name>Msees</Name>

<Instance>I1</Instance>

</Referenced\_Machine>

</Sees>

<Promotes>

<Promoted\_Operation>op1</Promoted\_Operation>

</Promotes>

</Machine>

### Values

The VALUES clause is represented by an optional **Values** element that contain a non-empty list of **Valuation** element describing a valuation by giving an identifier and an expression.

element Machine {

…

element Values {

Attribute?,

element Valuation {

Attribute?,

attribute ident { text },

Expression

}+

}?,

…

}

Example:

The component:

IMPLEMENTATION

MA\_i

REFINES

MA

VALUES

AbsSet = 0..100

END

is described:

<Machine name = ‘MA\_i’ type = ‘implementation’>

<Abstraction>MA</Abstraction>

<Values>

<Valuation ident=’AbsSet’>

<Binary\_Exp op=’..’>

<Integer\_Literal value=’0’/>

<Integer\_Literal value=’100’/>

</Binary\_Exp>

</Valuation>

</Values>

</Machine>

### Sets

Abstract sets and enumerated sets are described in the **Sets** element

element Machine {

…

element Sets {

Attribute?,

element Set {

Attribute?,

element Id { Identifier },

element Enumerated\_Values {

element Id { Identifier }+

}?

}+

}?,

…

}

Abstract set has just an **Id** element, enumerated sets have also an non-empty list of **Id** element contained in an **Enumared\_Values** element.

Example:

The component:

MACHINE

MA

SETS

POSITION;

STATE = {On, Off}

END

is described:

<Machine name = ‘MA’ type = ‘abstraction>

<Sets>

<Set>

<Id value=’POSITION’/>

</Set>

<Set>

<Id value=’STATE’/>

<Enumerated\_Values>

<Id value = ‘On’/>

<Id valie = ‘Off’/>

</Enumerated\_Values>

</Set>

</Sets>

</Machine>

### Constants and variables

Constants and variables are represented in optional elements containing a non-empty list of identifier.

element Machine {

…

element Abstract\_Constants {

Attribute?,

element Id { Identifier }+

}?,

element Concrete\_Constants {

Attribute?,

element Id { Identifier }+

}?,

element Abstract\_Variables {

Attribute?,

element Id { Identifier }+

}?,

element Concrete\_Variables {

Attribute?,

element Id { Identifier }+

}?,

…

}

Example:

The component:

MACHINE

MA

ABSTRACT\_CONSTANTS

MaxNumber,

MinNumber

CONCRETE\_CONSTANTS

PosMin

ABSTRACT\_VARIABLES

Abscissa

CONCRETE\_VARIABLES

Position

END

is described:

<Machine name = ‘MA’ type = ‘abstraction’>

<Abstract\_Constants>

<Id value=’MaxNumber’/>

<Id value=’MinNumber’/>

</Abstract\_Constants>

<Concrete\_Constants>

<Id value=’PosMin’/>

</Concrete\_Constants>

<Abstract\_Variables>

<Id value=’Abscissa’/>

</Abstract\_Variables>

<Concrete\_Variables>

<Id value=’Position’/>

</Concrete\_Variables>

</Machine>

### Properties, invariant and assertions

PROPERTIES, INVARIANT and ASSERTIONS are represented in optional elements containing predicates.

element Machine {

…

element Properties { Attribute?, Predicate }?,

element Invariant { Attribute?, Predicate }?,

…

element Assertions { Attribute?, Predicate+ }?,

…

}

**Properties** and **Invariant** elements contain one predicate, **Assertions** contains a non-empty list of predicate.

Example:

The component:

MACHINE

MA

PROPERTIES

Cte1 : INT &

Cte2 : NAT

INVARIANT

V1 : BOOL

ASSERTIONS

Cte1 < Cte2 ;

Cte2 >= 0 &

Cte2 <= MAXINT

END

is described:

<Machine name = ‘MA’ type = ‘abstraction’>

<Properties>

<Nary\_Pred op =’&’>

<Exp\_Comparison op = ‘:’>

<Id value=’Cte1’/>

<Id value=’INT’/>

</Exp\_Comparison>

<Exp\_Comparison op = ‘:’>

<Id value=’Cte2’/>

<Id value=’NAT’/>

</Exp\_Comparison>

</Nary\_Pred>

</Properties>

<Invariant>

<Exp\_Comparison op = ‘:’>

<Id value=’V1’/>

<Id value=’BOOL’/>

</Exp\_Comparison>

</Invariant >

<Assertions>

<Exp\_Comparison op = ‘<’>

<Id value=’Cte1’/>

<Id value=’Cte2’/>

</Exp\_Comparison>

<Nary\_Pred op =’&’>

<Exp\_Comparison op = ‘>=’>

<Id value=’Cte2’/>

<Integer\_Literal value=’0’/>

</Exp\_Comparison>

<Exp\_Comparison op = ‘<=’>

<Id value=’Cte2’/>

<Id value=’MAXINT’/>

</Exp\_Comparison>

</Nary\_Pred>

</Assertions>

</Machine>

### Initialisation and operations

INITIALISATION is described by an **Initialisation** element containing a substitution. OPERATIONS and LOCAL\_OPERATIONS contain a non-empty list of operations. EVENTS are also described in an **Operations** element.

element Machine {

…

element Initialisation { Attribute?, Substitution }?,

…

element Local\_Operations { Attribute?, Operation+ }?,

element Operations { Attribute?, Operation+ }?, …

}

Operation =

element Operation {

Attribute?,

attribute name { text },

element Refines { Attribute?, element Id { Identifier }+ }?,

element Output\_Parameters { element Id { Identifier }+ }?,

element Input\_Parameters { element Id { Identifier }+ }?,

element Precondition { Predicate }?,

element Body { Substitution }

}

An operation is described by a name, an optional **Refines** element for system models, input parameters, output parameters and precondition for software models and a substitution **Body**.

Example:

The component:

MACHINE

MA

OPERATIONS

Res <-- Compare(X1, X2) =

PRE

…

THEN

…

END

END

is described:

<Machine name = ‘MA’ type = ‘abstraction’>

<Operations>

<Operation name =’Compare’>

<Output\_Parameters>

<Id value=’Res’/>

</Output\_Parameters>

<Input\_Parameters>

<Id value=’X1’/>

<Id value=’X2’/>

</Input\_Parameters>

<Precondition>

…

</Precondition>

<Body>

…

</Body>

</Operation>

</Operations>

</Machine>

### Type information

BXML format can contain type information.

This information can be associated to each expression. In order to get a “relative” compact description for a component, types are described in a specific TypeInfos element, then types are referenced by an id in the expressions of the component.

element Machine {

…

element TypeInfos { element Type { TypeInfo }\* }

}

TypeInfo =

attribute id { xsd:integer },

Type

Type =

( element Id {attribute value { text } }

| element Unary\_Exp { attribute op { "POW" }, Type }

| element Binary\_Exp { attribute op { "\*" }, Type, Type }

| Struct

| element Generic\_Type { text }

)

A **TypeInfos** element contains **Type** elements that are described by an **id** and a type.

A type can be either an **Id** (INTEGER, BOOL, …), a specific **Generic\_Type** element for describing untyped expression (as empty set), a subset of another type, the Cartesian product of two types or a struct.

Example:

The component:

MACHINE

MA

VARIABLES

mySet

INVARIANT

mySet <: c1..c2

END

is described:

<Machine name = ‘MA’ type = ‘abstraction’>

<Abstract\_Variables>

<Id value=’mySet’ typref='3692107306'/>

</Abstract\_Variables>

<Invariant>

<Exp\_Comparison op = ‘<:’>

<Id value=’mySet’ typref='3692107306'/>

<Binary\_Exp op=’..’>

<Integer\_Literal value=’0’ typref = ‘2701647786‘/>

<Integer\_Literal value=’100’ typref = ‘2701647786‘/>

</Binary\_Exp>

</Exp\_Comparison>

</Invariant>

<TypeInfos>

<Type id=‘2701647786’>

<Id value=’INTEGER’/>

</Type>

<Type id = '3692107306'>

<Unary\_Exp op=’POW’>

<Id value=’INTEGER’/>

</Unary\_Exp>

</Type>

</TypeInfos>

</Machine>

## Substitution

A substitution pattern is a choice between all kinds of substitution.

The *substitution.rng* file structure is:

Substitution =

Attribute?,

( element Bloc\_Sub { … }

| element Skip { … }

| element Assert\_Sub { … }

| element If\_Sub { … }

| element Becomes\_Such\_That { … }

| element Assignement\_Sub { … }

| element Select { … }

| element Case\_Sub { … }

| element ANY\_Sub { … }

| element LET\_Sub { … }

| element Becomes\_In { … }

| element VAR\_IN { … }

| element Nary\_Sub { … }

| element Operation\_Call { … }

| element While { … }

)

Variables =

element Variables {

Attribute?,

element Id { Identifier }+

}

### BEGIN

A BEGIN substitution is described by a **Bloc\_Sub** element.

element Bloc\_Sub { Attribute?, Substitution }

Example:

The substitution:

BEGIN

skip

END

is described:

<Bloc\_Sub>

<Skip/>

</Bloc\_Sub>

### skip

A skip substitution is represented by a empty **Skip** element.

### Assignment

An assignment can be a multiple assignment, so the **Assignement\_Sub** element contains a **Variables** element with a non-empty list of expression and a **Values** element with also a non-empty list of expression.

element Assignement\_Sub {

Attribute?,

element Variables { Attribute?, Expression+ },

element Values { Attribute?, Expression+ }

}

Example:

The substitution:

xx, ff(yy) := yy,xx

is described:

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

<Binary\_Exp op=’(‘>

<Id value=’ff”/>

<Id value=’yy’/>

</Binary\_Exp>

</Variables>

<Values>

<Id value=’yy’/>

<Id value=’xx’/>

</Values>

</Assignement\_Sub>

### ASSERT

ASSERT substitution is described by a **Assert\_Sub** element containing two elements: **Guard** and **Body**.

element Assert\_Sub {

Attribute?,

element Guard { Attribute?, Predicate },

element Body { Attribute?, Substitution }

}

Example:

The substitution:

ASSERT

xx=yy

THEN

skip

END

is described:

<Assert\_Sub>

<Guard>

<Exp\_Comparison op=’=‘>

<Id value=’xx”/>

<Id value=’yy’/>

</Exp\_Comparison>

</Guard>

<Body>

<Skip/>

</Body>

</Assert\_Sub>

### IF

IF substitution is described in an **If\_Sub** element that contains a **Condition** and a **Then** element, an optional **Else** element and an attribute **elseif** with possible value **yes** or **no** that indicates if the IF corresponds to an ELSIF structure.

element If\_Sub {

attribute elseif { "yes" | "no" },

Attribute?,

element Condition { Attribute?, Predicate },

element Then { Attribute?, Substitution },

element Else { Attribute?, Substitution }?

}

Example:

The substitution:

IF xx=2

THEN

xx := 1

ELSIF xx = 3

THEN

xx := 2

ELSE

xx := 3

END

is described:

<If\_Sub elseif=’no’>

<Condition>

<Exp\_Comparison op=’=‘>

<Id value=’xx”/>

<Integer\_Literal value=’2’/>

</Exp\_Comparison>

</Condition>

<Then>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’1’/>

</Values>

</Assignement\_Sub>

</Then>

<Else>

<If\_Sub elseif=’yes’>

<Condition>

<Exp\_Comparison op=’=‘>

<Id value=’xx”/>

<Integer\_Literal value=’3’/>

</Exp\_Comparison>

</Condition>

<Then>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’2’/>

</Values>

</Assignement\_Sub>

</Then>

<Else>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’3’/>

</Values>

</Assignement\_Sub>

</Else>

</If\_Sub>

</Else>

</If\_Sub>

### SELECT

The SELECT substitution contains a **When\_Clauses** element and an optional **Else** element. A **When\_Clauses** element contains a non-empty list of **When** elements that contain a **Condition** and a **Then**.

element Select {

Attribute?,

element When\_Clauses {

Attribute?,

element When {

Attribute?,

element Condition { Attribute?, Predicate },

element Then { Attribute?, Substitution }

}+

},

element Else { Attribute?, Substitution }?

}

Example:

The substitution:

SELECT xx=2

THEN

xx := 1

WHEN xx = 3

THEN

xx := 2

ELSE

xx := 3

END

is described:

<Select>

<When\_Clauses>

<When>

<Condition>

<Exp\_Comparison op=’=‘>

<Id value=’xx”/>

<Integer\_Literal value=’2’/>

</Exp\_Comparison>

</Condition>

<Then>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’1’/>

</Values>

</Assignement\_Sub>

</Then>

</When>

<When>

<Condition>

<Exp\_Comparison op=’=‘>

<Id value=’xx”/>

<Integer\_Literal value=’3’/>

</Exp\_Comparison>

</Condition>

<Then>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’2’/>

</Values>

</Assignement\_Sub>

</Then>

</When>

</When\_Clauses>

<Else>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’3’/>

</Values>

</Assignement\_Sub>

</Else>

</Select>

### CASE

The CASE substitution is described with a **Case\_Sub** element, this element contains a **Value** element and a **Choices** element and an optional **Else** element. The **Choices** element contains a non-empty list of **Choice** element that contains a non-empty list of **Value** and a **Then**.

element Case\_Sub {

Attribute?,

element Value { Attribute?, Expression },

element Choices {

Attribute?,

element Choice {

Attribute?,

element Value { Attribute?, Expression }+,

element Then { Attribute?, Substitution }

}+

},

element Else {

Attribute?,

element Choice {

Attribute?,

element Then { Attribute?, Substitution }

}

}?

}

Example:

The substitution:

CASE xx OF

EITHER 0,2 THEN

xx := 1

OR 3 THEN

xx := 2

ELSE

xx := 3

END

END

is described:

<Case\_Sub>

<Value>

<Id value=’xx’/>

</Value>

<Choices>

<Choice>

<Value>

<Integer\_Literal value=’0’/>

</Value>

<Value>

<Integer\_Literal value=’2’/>

</Value>

<Then>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’1’/>

</Values>

</Assignement\_Sub>

</Then>

</Choice>

<Choice>

<Value>

<Integer\_Literal value=’3’/>

</Value>

<Then>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’2’/>

</Values>

</Assignement\_Sub>

</Then>

</Choice>

</Choices>

<Else>

<Assignement\_Sub>

<Variables>

<Id value=’xx’/>

</Variables>

<Values>

< Integer\_Literal value=’3’/>

</Values>

</Assignement\_Sub>

</Else>

</Case\_Sub>

### ANY

An ANY substitution is described by a **ANY\_Sub** element that contains a **Variables** element, a **Pred** element and a **Then** element.

element ANY\_Sub {

Attribute?,

Variables,

element Pred { Attribute?, Predicate},

element Then { Attribute?, Substitution }

}

Example:

The substitution:

ANY r1,r2 WHERE

r1 : NAT &

r2 : NAT

THEN

skip

END

is described:

<ANY\_Sub>

<Variables>

<Id value=’r1’/>

<Id value=’r2’/>

</Variables>

<Pred>

<Nary\_Pred op=’&’>

<Exp\_Comparison op =’:’>

<Id value=’r1’/>

<Id value=’NAT’/>

</ Exp\_Comparison >

<Exp\_Comparison op =’:’>

<Id value=’r2’/>

<Id value=’NAT’/>

</ Exp\_Comparison >

</Nary\_Pred>

</Pred>

<Then>

</Skip>

</Then>

</ANY\_Sub>

### LET

The LET Substitution is described in a **LET\_Sub** element that contains an element **Variables** with a non-empty list of identifier, an element **Values** with a non-empty list of valuation and an element **Then**.

element LET\_Sub {

Attribute?,

Variables,

element Values {

Attribute?,

element Valuation {

Attribute?,

Expression,

attribute ident { text }

}+

},

element Then { Attribute?, Substitution }

}

Example:

The substitution:

LET r1,r2 BE

r1 = 1 &

r2 =2

IN

skip

END

is described:

<LET\_Sub>

<Variables>

<Id value=’r1’/>

<Id value=’r2’/>

</Variables>

<Values>

<Valuation ident=’r1’>

<Integer\_Literal value=’1’/>

</Valuation>

<Valuation ident=’r2’>

<Integer\_Literal value=’2’/>

</Valuation>

</Values>

<Then>

</Skip>

</Then>

</ANY\_Sub>

### Becomes in

The “becomes in” substitution is described in a **Becomes\_In** element with a **Variables** element and a **Value** element containing an expression

element Becomes\_In {

Attribute?,

Variables,

element Value { Attribute?, Expression }

}

Example:

The substitution:

y1, y2 :: INT\*NAT

is described:

<Becomes\_In>

<Variables>

<Id value=’y1’/>

<Id value=’y2’/>

</Variables>

<Value>

<Binary\_Exp op=’\*’>

<Id value=’INT’/>

<Id value=’NAT’/>

</Binary\_Exp>

</Value>

</Becomes\_In>

### Becomes such that

The “becomes such that” substitution is described in a **Becomes\_Such\_That** element with a **Variables** element and a **Pred** element containing a predicate.

element Becomes\_Such\_That {

Attribute?,

Variables,

element Pred { Attribute?, Predicate }

}

Example:

The substitution:

y1, y2 :(

y1 : INT &

y2 : NAT

)

is described:

<Becomes\_Such\_That>

<Variables>

<Id value=’y1’/>

<Id value=’y2’/>

</Variables>

<Pred>

<Nary\_Pred op=’&’>

<Exp\_Comparison op =’:’>

<Id value=’y1’/>

<Id value=’INT’/>

</ Exp\_Comparison >

<Exp\_Comparison op =’:’>

<Id value=’y2’/>

<Id value=’NAT’/>

</ Exp\_Comparison >

</Nary\_Pred>

</Pred>

</Becomes\_Such\_That>

### Local variable

Local variables are described in a **VAR\_IN** element containing a **Variables** element and a **Body** element with a substitution.

element VAR\_IN {

Attribute?,

Variables,

element Body { Attribute?, Substitution }

}

Example:

The substitution:

VAR l1, l2

IN

skip

END

is described:

<VAR\_IN>

<Variables>

<Id value=’l1’/>

<Id value=’l2’/>

</Variables>

<Body>

<Skip/>

</Body>

</VAR\_IN>

### Sequence, Parallel and Choice

The CHOICE substitution is described in a **Nary\_Sub** element with attribute **op** equal to ‘**CHOICE**’.

Sequential substitutions are described in the same element with **op**=’**;**’ and parallel substitution with **op**=’**||**’. Those binary substitutions are described as N-ary.

element Nary\_Sub {

Attribute?,

attribute op { "||" | ";" | "CHOICE" },

Substitution+

}

Example:

The substitution:

CHOICE

x1 := 1 ||

x2 := 2

OR

x1 := 1 ||

x2 := 2 ;

x3 := 3

END

is described:

<Nary\_SUB op=’CHOICE’>

<Nary\_SUB op=’||’>

<Assignement\_Sub>

<Variables>

<Id value=’xx1’/>

</Variables>

<Values>

<Integer\_Literal value=’1’/>

</Values>

</Assignement\_Sub>

<Assignement\_Sub>

<Variables>

<Id value=’xx2’/>

</Variables>

<Values>

<Integer\_Literal value=’2’/>

</Values>

</Assignement\_Sub>

</Nary\_SUB>

<Nary\_Sub op='||'>

<Assignement\_Sub>

<Variables>

<Id value=’xx1’/>

</Variables>

<Values>

<Integer\_Literal value=’1’/>

</Values>

</Assignement\_Sub>

<Nary\_Sub op=';'>

<Assignement\_Sub>

<Variables>

<Id value=’xx2’/>

</Variables>

<Values>

<Integer\_Literal value=’2’/>

</Values>

</Assignement\_Sub>

<Assignement\_Sub>

<Variables>

<Id value=’xx3’/>

</Variables>

<Values>

<Integer\_Literal value=’3’/>

</Values>

</Assignement\_Sub>

</Nary\_Sub>

</Nary\_Sub>

</Nary\_Sub>

### Operation call

An operation call is described in an **Operation\_Call** element containing a **Name** element, two optional elements for input and ouput parameters.

element Operation\_Call {

Attribute?,

element Name {

Attribute?,

element Id { Identifier }

},

element Input\_Parameters { Attribute?, Expression+ }?,

element Output\_Parameters { Attribute?, Expression+ }?

}

Example:

The substitution:

BEGIN

opa;

res, flag <-- opb(xx)

END

is described:

<Bloc\_Sub>

<Nary\_SUB op=’;’>

<Operation\_Call>

<Name>

<Id value=’opa’/>

</Name>

</Operation\_Call>

<Operation\_Call>

<Name>

<Id value=’opb’/>

</Name>

<Input\_Parameters>

<Id value=’xx’/>

</Input\_Parameters>

<Output\_Parameters>

<Id value=’res’/>

<Id value=’flag’/>

</Output\_Parameters>

</Operation\_Call>

</Nary\_SUB>

</Bloc\_Sub>

### While loop

The while loop substitution is described in an element **While** that contains elements **Condition**, **Body**, **Invariant** and **Variant**.

element While {

Attribute?,

element Condition { Attribute?, Predicate },

element Body { Attribute?, Substitution },

element Invariant { Attribute?, Predicate },

element Variant { Attribute?, Expression }

}

Example:

The substitution:

WHILE counter < 5 DO

counter := counter + 1

INVARIANT

counter : NAT

VARIANT

10 – counter

END

is described:

<While>

<Condition>

<Exp\_Comparison op=’<’>

<Id value=’counter’/>

<Integer\_Literal value=’5’/>

</Exp\_Comparison>

</Condition>

<Body>

<Assignement\_Sub>

<Variables>

<Id value=’counter’/>

</Variables>

<Values>

<Binary\_Exp op=’+’>

<Id value=’counter’/>

<Integer\_Literal value=’1’/>

</Binary\_Exp>

</Values>

</Assignement\_Sub>

</Body>

<Invariant>

<Exp\_Comparison op =’:’>

<Id value=’counter’/>

<Id value=’NAT’/>

</ Exp\_Comparison >

</Invariant>

<Variant>

<Binary\_Exp op=’-’>

<Integer\_Literal value=’10’/>

<Id value=’counter’/>

</Binary\_Exp>

</Variant>

</While>

## Predicate

A predicate pattern is a choice between 5 kinds of predicate.

The *predicate.rng* file structure is:

# A RELAX NG compact syntax pattern for a predicate.

Predicate =

element Binary\_Pred { … }

| element Nary\_Pred {… }

| element Unary\_Pred {… }

| element Quantified\_Pred {… }

| element Exp\_Comparison {… }

### Binary predicate

Conjonction, disjonction, implication, equivalence and conditional conjonction are described with a **Binary\_Pred** element with an attribute op corresponding to the operator and with two predicates.

element Binary\_Pred {

Attribute?,

attribute op { "&" | "or" | "=>" | "<=>" | "cand" },

Predicate,

Predicate

}

### N-ary predicate

Conjonction can also be described with a **Nary\_Pred** element in order to obtain a more compact xml file.

element Nary\_Pred {

Attribute?,

attribute op { "&" },

Predicate\*

}

### Unary predicate

Negation is described with an **Unary\_Pred** element.

element Unary\_Pred {

Attribute?,

attribute op { "not" },

Predicate

}

### Quantified predicate

Universal and existential predicate are described with a **Quantified\_Pred** element containing an attribute type, variables and a Body element.

element Quantified\_Pred {

Attribute?,

attribute type { "!" | "#" },

Variables,

element Body { Attribute?, Predicate }

}

### Expression comparison

Equalities, belonging, inclusion and comparison predicates are described in an **Exp\_Comparison** element with an attribute corresponding two the operator and two expressions.

element Exp\_Comparison {

Attribute?,

attribute op { ":" | "/:" | "<:" | "/<:" | "<<:" | "/<<:" | "=" | "/=" | "<="

| "<" | ">=" | ">" | "rgt" | "rlt" | "rle" | "rge" },

Expression,

Expression

}

## Expression

An expression pattern is a choice between the different kinds of expression. An optional **typeref** attribute can be associated to every expression.

The *expression.rng* file structure is:

# A RELAX NG compact syntax pattern for an expression.

Expression =

element Binary\_Exp {… }

| element Nary\_Exp {… }

| element Unary\_Exp {…}

| element Quantified\_Exp {… }

| element Quantified\_Set { … }

| Struct

| element Record {…}

| element Boolean\_Exp {… }

| element EmptySet { … }

| element EmptySeq { … }

| element Id { … }

| element Integer\_Literal { … }

| element Boolean\_Literal {… }

| element STRING\_Literal {… }

Struct =

element Struct {…}

### Binary expression

Binary expressions are described in a **Binary\_Exp** element with the corresponding operator.

Function application and relation image are binary expressions

element Binary\_Exp {

Attribute?,

attribute typref { xsd:integer }?,

attribute op { "," | "\*" | "\*\*" | "+" | "+->" | "+->>" | "-" | "-->" | "-->>"

| "->" | ".." | "/" | "/\" | "/|\" | ";" | "<+" | "<->" | "<-"

| "<<|" | "<|" | ">+>" | ">->" | ">+>>" | ">->>" | "><"

| "||" | "\/" | "\|/" | "^" | "mod" | "|->" | "|>" | "|>>" | "["

| "(" | "'" | "<'" | "rplus" | "rminus" | "rmul" | "rdiv" | "rpow"

},

Expression,

Expression

}

Example:

The expression:

ff(xx)

is described:

<Binary\_Exp op=’(‘>

<Id value=’ff/>

<Id value=’xx’/>

</Binary\_Exp>

### N-ary expression

Extensive sets and extensive sequences are described with a **Nary\_Exp** element.

element Nary\_Exp {

Attribute?,

attribute typref { xsd:integer }?,

attribute op { "[" | "{" },

Expression+

}

Example:

The expression:

{1,3,5,7}

is described:

<Nary\_Exp op=’{‘>

<Integer\_Literal value=’1’/>

<Integer\_Literal value=’3’/>

<Integer\_Literal value=’5’/>

<Integer\_Literal value=’7’/>

</Binary\_Exp>

### Unary expression

Unary expressions are described in an **Unary\_Exp** element.

Projections, iteration expressions are described as unary expressions

element Unary\_Exp {

Attribute?,

attribute typref { xsd:integer }?,

attribute op { "max" | "min" | "card" | "dom" | "ran" | "POW" | "POW1"

| "FIN" | "FIN1" | "union" | "inter" | "seq" | "seq1" | "iseq"

| "iseq1" | "-" | "~" | "size" | "prj1" | "prj2" | "perm"

| "first" | "last" | "id" | "iterate" | "closure" | "closure1"

| "tail" | "front" | "rev" | "conc" | "succ" | "pred" | "rel" | "fnc"

},

Expression

}

Example:

The expression:

prj1(NAT,NAT)(x1 |-> x2)

is described:

<Binary\_Exp op=’(‘>

<Unary\_Exp op=’prj1’>

<Binary\_Exp op=’,’>

<Id value=’NAT’/>

<Id value=’NAT’/>

</Binary\_Exp>

</Unary\_Exp>

<Binary\_Exp op=’|->’>

<Id value=’x1’/>

<Id value=’x2’/>

</Binary\_Exp>

</Binary\_Exp>

### Quantified expression

Lambda expression, quantified sum and product, quantified union and intersection are described with a **Quantified\_Exp** element containing the corresponding operators, a Variables pattern, a predicate and an expression

element Quantified\_Exp {

Attribute?,

attribute typref { xsd:integer }?,

attribute type { "%" | "SIGMA" | "PI" | "RSIGMA" | "RPI" | "INTER"

| "UNION" },

Variables,

element Pred { Attribute?, Predicate },

element Body { Attribute?, Expression }

}

Example:

The expression:

%xx.(xx : NATURAL | xx \* 2)

is described:

<Quantified\_Exp type=’%‘>

<Variables>

<Id value=’xx’/>

</Variables>

<Pred>

<Exp\_Comparison op=’:’>

<Id value=xx’/>

<Id value=’NATURAL’/>

</Exp\_Comparison>

</Pred>

<Body>

<Binary\_Exp op=’\*’>

<Id value=’xx’/>

<Integer\_Literal value=’2’/>

</Binary\_Exp>

</Body>

</Quantified\_Exp>

### Quantified Set

Comprehension sets are described in a **Quantified\_Set** element containing variables and a predicate

element Quantified\_Set {

Attribute?,

attribute typref { xsd:integer }?,

Variables,

element Body { Attribute?, Predicate }

}

Example:

The expression:

{xx | xx : NATURAL & xx mod 2 = 0}

is described:

<Quantified\_Set>

<Variables>

<Id value=’xx’/>

</Variables>

<Body>

<Binary\_Pred op=’&’>

<Exp\_Comparison op=’:’>

<Id value=xx’/>

<Id value=’NATURAL’/>

</Exp\_Comparison>

<Exp\_Comparison op=’=’>

<Binary\_Exp op=’mod’>

<Id value=xx’/>

<Integer\_Literal value=’2/>

</Binary\_Exp>

</Exp\_Comparison>

</Binary\_Pred>

</Body>

</Quantified\_Set>

### Struct and record

Set of records are described in a **Struct** element containing a non-empty list of **Record\_Item** elements with a **label** attribute and an expression.

Records in extension are described in a **Record** element containing a non-empty list of **Record\_Item** elements with a **label** attribute and an expression.

Struct =

element Struct {

Attribute?,

attribute typref { xsd:integer }?,

element Record\_Item {

Attribute?,

attribute label { text },

Expression

}+

}

element Record {

Attribute?,

attribute typref { xsd:integer }?,

element Record\_Item {

Attribute?,

attribute label { text },

Expression

}+

}

Example:

The expression:

struct(l1 : NAT, l2 : BOOL)

is described:

<Struct>

<Record\_Item label=’l1’>

<Id value=’NAT’/>

</Record\_Item>

<Record\_Item label=’l2’>

<Id value=’BOOL’/>

</Record\_Item>

</Struct>

### Terminal expression

The bool operator is described with a **Boolean\_Exp** element containing a predicate. All terminal expressions are described with a specific element.

| element Boolean\_Exp {

Attribute?,

attribute typref { xsd:integer }?,

Predicate

}

|element EmptySet {

Attribute?,

attribute typref { xsd:integer }?

}

| element EmptySeq {

Attribute?,

attribute typref { xsd:integer }?

}

| element Id { Identifier }

| element Integer\_Literal {

Attribute?,

attribute typref { xsd:integer }?,

attribute value { xsd:integer }

}

| element Boolean\_Literal {

Attribute?,

attribute typref { xsd:integer }?,

attribute value { "TRUE" | "FALSE" }

}

| element STRING\_Literal {

Attribute?,

attribute value { text },

attribute typref { xsd:integer }?

}

Example:

The expression:

bool([] = {}) = bool(“empty seq” = “empty set”)

is described:

<Exp\_Comparison op=’=’>

<Boolean\_Exp>

<Exp\_Comparison op=’=’>

<Empty\_Seq/>

<Empty\_Set/>

</Exp\_Comparison>

</Boolean\_Exp>

<Boolean\_Exp>

<Exp\_Comparison op=’=’>

<STRING\_Literal value=’empty seq’/>

<STRING\_Literal value=’empty set’/>

</Exp\_Comparison>

</Boolean\_Exp>

</Exp\_Comparison>

## Identifier

An identifier pattern described the attributes that can be associated to an identifier.

# A RELAX NG compact syntax pattern for an identifier.

Identifier =

Attribute?,

attribute value { text },

attribute suffix { text }?,

attribute typref { xsd:integer }?,

(attribute instance { text },

attribute component { text })?

The **value** attribute contains the name of the identifier. The **suffix** attribute is used for $0 variable. For renamed variable, the attributes **instance** and **component** are used.

Example:

The expression:

xx$0 =Msees.var

is described:

<Exp\_Comparison op=’=’>

<Id value=’xx’ suffix=’0’/>

<Id value=’Msees.var’ instance=’Msees’ component=’var’/>

</Exp\_Comparison>

# Index

abstract set 12

ANY 25

ASSERT 20

ASSERTIONS 14

becomes in 27

becomes such that 28

BEGIN 19

belonging 34

bool operator 39

CASE 23

CHOICE 29

comparison 34

comprehension set 37

conditional conjonction 33

conjonction 33

constant 13

CONSTRAINTS 9

disjonction 33

enumerated set 12

equality 34

equivalence 33

EVENTS 15

existential quantification 34

EXTENDS 10

extensive sequence 35

extensive set 35

function application 35

IF 20

IMPLEMENTATION 8

implication 33

IMPORTS 10

INCLUDES 10

inclusion 34

INITIALISATION 15

input parameter 16

instance 10

instantiated parameter 10

INVARIANT 14

iteration expressions 36

lambda expression 37

LET 26

local variable 28

LOCAL\_OPERATIONS 15

MACHINE 8

negation 33

operation call 31

OPERATIONS 15

output parameter 16

parallel substitution 29

parameter 9

projection 36

promoted operation 10

PROPERTIES 14

quantified intersection 37

quantified product 37

quantified sum 37

quantified union 37

records in extension 38

REFINEMENT 8

REFINES 8

relation image 35

SEES 10

SELECT 22

sequential substitution 29

set of records 38

skip 19

SYSTEM 8

universal quantification 34

USES 10

VALUES 11

variable 13

while loop substitution 31