

# CML-DEVS - Technical Report

Diego Hollmann

May 22, 2015

This technical report presents, formally, CML-DEVS, a language to describe DEVS models in its abstract or conceptual form. The syntax is presented via its EBNF (Extended Backus Naur Form) and its grammar via the rules to translate a CML-DEVS model into PowerDEVS and DEVS-Suite models.

## 1 CML-DEVS Grammar (BNF)

As in almost all BNF, terms between angle brackets,  $\langle \rangle$ , are nonterminals, and there are production rules for replace them by terminal or other nonterminals. Terms in quotes, “ ”, are terminals and represent *tokens* of the language, values or identifiers. Expressions enclosed in braces,  $\{ \}$ , means 0 or more repetitions of that expression. Finally, expressions between square brackets,  $[ ]$ , are optional expressions.

```
 $\langle CML - DEVS \rangle ::= \langle atomic \rangle$   
     $[\langle functions \rangle]$   
     $[\langle simulate \rangle]$   
 $\langle atomic \rangle ::= \text{atomic } \langle id \rangle [(\langle params \rangle) < [S,] [X,] [Y,] [\delta int,] [\delta int,] [\lambda,] [ta] > \text{is}$   
     $[\langle params \rangle]$   
     $[\langle S \rangle]$   
     $[\langle X \rangle]$   
     $[\langle Y \rangle]$   
     $[\langle \delta int \rangle]$   
     $[\langle \delta ext \rangle]$   
     $[\langle \lambda \rangle]$   
     $[\langle ta \rangle]$   
    end atomic  
 $\langle id \rangle ::= \langle letter \rangle \{ \langle letter \rangle \mid \langle digit \rangle \mid \_ \} \mid \sigma$   
 $\langle letter \rangle ::= a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p$   
     $\mid q \mid r \mid s \mid t \mid uv \mid w \mid x \mid y \mid z \mid A \mid B \mid C \mid D \mid E \mid F$   
     $\mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid NO \mid P \mid Q \mid R \mid S \mid T \mid U$   
     $\mid V \mid W \mid X \mid Y \mid Z$   
 $\langle digit \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$   
 $\langle S \rangle ::= S \text{ is}$   
     $\langle definitions \rangle$   
     $[\langle synonyms \rangle]$   
    end S  
 $\langle definitions \rangle ::= \langle id \rangle \{ , \langle id \rangle \} : \langle type \rangle ;$   
     $[\langle definitions \rangle]$   
 $\langle synonyms \rangle ::= \langle id \rangle = \langle Type \rangle ;$   
     $[\langle synonyms \rangle]$ 
```

```

⟨type⟩ ::= ℕ | ℤ | ℝ | Time | Text | Boolean | ⟨enum⟩
          | ℙ ⟨type⟩
          | List ⟨type⟩
          | ⟨type⟩ ∪ ⟨type⟩ { ∪ ⟨type⟩ }
          | ⟨type⟩ × ⟨type⟩ { × ⟨type⟩ }
          | ⟨type⟩ → ⟨type⟩
          | ⟨id⟩
⟨enum⟩ ::= “{”⟨id⟩ | ⟨symbol⟩{, ⟨id⟩ | ⟨symbol⟩}“}”
⟨symbol⟩ ::= ∞ | ∅
⟨X⟩ ::= X is
        ⟨definitions⟩
        [⟨synonyms⟩]
        end X
⟨Y⟩ ::= Y is
        ⟨definitions⟩
        [⟨synonyms⟩]
        end Y
⟨deltint⟩ ::= δint [⟨ids⟩] is
            ⟨sentence⟩
            {⟨sentence⟩}
            end δint
⟨deltext⟩ ::= δext [⟨ids⟩] is
            ⟨sentence⟩
            {⟨sentence⟩}
            end δext
⟨ids⟩ ::= ((⟨id⟩ | ⟨ids⟩){, (⟨id⟩ | ⟨ids⟩)})
⟨sentence⟩ ::= ⟨assignment⟩
            | ⟨expr⟩
            | ⟨foreach⟩
            | ⟨caseblock⟩
            | ⟨whereblock⟩
⟨assignment⟩ ::= ⟨var⟩ = ⟨expr⟩;
⟨var⟩ ::= ⟨id⟩ | ⟨idComp⟩ | value | port | e
⟨idComp⟩ ::= ⟨id⟩.⟨digit⟩{.⟨digit⟩}
⟨expr⟩ ::= ⟨var⟩ | ⟨val⟩ | ⟨vals⟩ | ⟨set⟩ | ⟨list⟩ | ⟨mathFun⟩ | ⟨defFun⟩ | ⟨pFun⟩ | ⟨operation⟩
⟨val⟩ ::= ⟨number⟩ | ⟨string⟩ | ‘{⟨digit⟩ | ⟨letter⟩}’
⟨vals⟩ ::= (⟨expr⟩,⟨expr⟩){,⟨expr⟩}
⟨set⟩ ::= “{”⟨expr⟩{“,”⟨expr⟩}“}”
⟨list⟩ ::= “<”⟨expr⟩{“,”⟨expr⟩}“>”
⟨operation⟩ ::= [⟨unOp⟩]⟨expr⟩[⟨binOp⟩⟨expr⟩]
⟨mathFun⟩ ::= ⟨funId⟩(⟨expr⟩)
⟨funId⟩ ::= sin | cos | tan | arcsin | arccos | arctan | log | sign | min
          | max | sqrt
⟨defFun⟩ ::= ⟨id⟩(⟨expr⟩{,⟨expr⟩});
⟨unOp⟩ ::= – | rev | head | last | tail | front | #
⟨binOp⟩ ::= + | – | \ | * | ∩ | ∪
⟨pFun⟩ ::= ⟨id⟩ ⟨expr⟩
⟨foreach⟩ ::= foreach ⟨id⟩ in ⟨expr⟩ do
            ⟨sentence⟩
            {⟨sentence⟩}
            end foreach

```

```

⟨caseblock⟩ ::= defcases
    (case ⟨sentence⟩; {⟨sentence⟩;} if ⟨conditions⟩)
    | (if ⟨conditions⟩ ⇒ ⟨sentence⟩; {⟨sentence⟩;})
    (⟨case ⟨sentence⟩; {⟨sentence⟩;} if ⟨conditions⟩})
    | (⟨if ⟨conditions⟩ ⇒ ⟨sentence⟩; {⟨sentence⟩;}⟩)
    [default ⟨sentence⟩; {⟨sentence⟩;}]
end defcases

⟨conditions⟩ ::= ⟨condition⟩
    | ¬ (⟨conditions⟩)
    | (⟨conditions⟩) ∧ (⟨conditions⟩)
    | (⟨conditions⟩) ∨ (⟨conditions⟩)

⟨condition⟩ ::= ⟨expr⟩ ⟨comparison⟩ ⟨expr⟩
⟨comparison⟩ ::= < | > | ≤ | ≥ | ≠ | = | ∈ | ∉ | ⊂ | ⊆ | ⊄ | ⊈
⟨whereblock⟩ ::= defwhere
    ⟨sentence⟩
    {⟨sentence⟩}
    where
    [⟨definition⟩]
    [⟨synonyms⟩]
    {⟨sentence⟩}
    end defwhere

⟨λ⟩ ::= λ [⟨ids⟩] is
    ⟨sentence⟩
    {⟨sentence⟩}
    end λ

⟨ta⟩ ::= ta [⟨ids⟩] is
    ⟨sentence⟩
    {⟨sentence⟩}
    end ta

⟨params⟩ ::= params is
    ⟨id⟩ = ⟨val⟩;
    [⟨id⟩ = ⟨val⟩;]
    end params

⟨functions⟩ ::= functions ⟨id⟩ is
    ⟨function⟩
    {⟨function⟩}
    end functions

⟨function⟩ ::= function ⟨id⟩ is
    ⟨id⟩ : ⟨type⟩ {, ⟨id⟩ : ⟨type⟩} → ⟨id⟩ : ⟨Type⟩;
    [⟨definitions⟩]
    [⟨synonyms⟩]
    ⟨sentence⟩;
    {⟨sentence⟩;}
    end function

⟨simulate⟩ ::= simulate ⟨id⟩ from
    ⟨assignment⟩
    {⟨assignment⟩}
    end simulate

```

## 2 Preprocessing

### 2.1 Union Type Normalization

In this preprocessing phase, an union type  $A_1 \cup \dots \cup A_N$ , is normalized into the following form:  $B_1 \cup \dots \cup B_M$ , where  $M \leq N$  and:

- $A_i = \{x_1^{A_i}, \dots, x_{N_{A_i}}^{A_i}\} \wedge A_j = \{x_1^{A_j}, \dots, x_{N_{A_j}}^{A_j}\} \Rightarrow \exists k : B_k = \{x_1^{A_i}, \dots, x_{N_{A_i}}^{A_i}, x_1^{A_j}, \dots, x_{N_{A_j}}^{A_j}\}$
- $A_i = \mathbb{N} \wedge A_j = \mathbb{Z} \Rightarrow \exists k : B_k = \mathbb{Z} \wedge \nexists l : B_l = \mathbb{N}$
- $A_i = \mathbb{N} \wedge A_j = \mathbb{R} \Rightarrow \exists k : B_k = \mathbb{R} \wedge \nexists l : B_l = \mathbb{N}$
- $A_i = \mathbb{Z} \wedge A_j = \mathbb{R} \Rightarrow \exists k : B_k = \mathbb{R} \wedge \nexists l : B_l = \mathbb{Z}$
- $k \neq l \Rightarrow B_k \neq B_l$

with  $i, j \in 1 \dots N$  and  $k, l \in 1 \dots M$

The above means that, in a normalized union, there is (at most) only one  $\langle enum \rangle$  type containing all the elements of the  $\langle enum \rangle$  types from the not normalized union and only remains the “largest” mathematical type,  $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$ .

On the other hand, after this normalization, does not matter the order of types in an union. The type  $B_k \cup B_l$  is the same as  $B_l \cup B_k$  and thus, only one of those equivalent unions remains.

### 2.2 Types Redefinition and Types Normalization

The other phase of the preprocessing process is known as *flattening* of the type definitions. In such phase, a name (synonym) is assigned to each union, tuple or partial function that is part of a larger type involving a type constructor. Later, that union, tuple or partial function is replaced by the synonym assigned. Let us see this in a short example. The expression:

$$varName : \mathbb{P}((\mathbb{N} \cup \{\emptyset\}) \times (\mathbb{R} \cup \{\emptyset\}));$$

Is transformed into:

$$\begin{aligned} varNameType1 &== (\mathbb{N} \cup \{\emptyset\}); \\ varNameType2 &== (\mathbb{R} \cup \{\emptyset\}); \\ varNameType3 &== varNameType1 \times varNameType2; \\ varName &: \mathbb{P} varNameType3; \end{aligned}$$

Recall that the redefinition is “inside out”, i.e. at first the inner tuples, unions or partial functions are redefined and later the outer ones.

Another remark is that if  $Expr_i = Expr_j$  with  $i \neq j$ , then  $SynonName_i = SynonName_j$ . In other words, the same synonyms are assigned to equivalent expressions.

Then, once this redefinition process is done, each variable definition has the following form:

$$varName : Type;$$

where *Type* is one of the following:

- $(BT \mid SN)$
- $List(BT \mid SN)$
- $\mathbb{P}(BT \mid SN)$
- $((BT \mid SN) \mid (List(BT \mid SN)) \mid (\mathbb{P}(BT \mid SN))) \times \dots \times ((BT \mid SN) \mid (List(BT \mid SN)) \mid (\mathbb{P}(BT \mid SN)))$
- $((BT \mid SN) \mid (List(BT \mid SN)) \mid (\mathbb{P}(BT \mid SN))) \cup \dots \cup ((BT \mid SN) \mid (List(BT \mid SN)) \mid (\mathbb{P}(BT \mid SN)))$

with  $BT = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Text} \mid \text{Bool} \mid \text{Time} \mid EnumType$  and *SN* is the name that has been assigned to some synonym.

On the other hand, each synonym definition is of the following form:

$$SynonName == Type;$$

where *Type* is also one of the mentioned above.

### 3 Translation Rules

This section shows in detail how to translate CML-DEVS models into models that can be simulated in DEVS-Suite and in PowerDEVS. It begins by describing the general structure of the model and files that must be generated for the simulation. Then, the rules for the translation of each structure or component of a CML-DEVS model are given. The rules are presented by a general schema, described in Section 3.2. Moreover, in each section, the portion of the EBNF referring to the corresponding translation rule(s) is shown.

#### 3.1 General Structure

---

```

<atomic> ::= atomic <id>[(params)] <[<S>,<X>,<Y>,<δint>,<δext>,<λ>,<ta>]> is
    [(params)]
    [<S>]
    [<X>]
    [<Y>]
    [<δint>]
    [<δext>]
    [<λ>]
    [<ta>]
end atomic

```

---

##### 3.1.1 General Structure of a DEVS-Suite atomic model

```

import view.modeling.ViewableAtomic;
import model.modeling.content;
import model.modeling.message;
import ... ;

public class ModelName extends ViewableAtomic{
    /* State variables declaration */
    /* Input types declaration */
    ...
    public ModelName(){
        super("ModelName");
        /* Ports declaration */
        addInput("in");
        addOutput("out");
        /* Parameters configuration */
        ...
    }
    public void initialize(){
        /* Instantiate the corresponding variables */
        var = new ... ();
        /* Initialize state variables (if applicable) */
        ...
    }
    public void deltint(){
        ...
    }
    public void deltext(double e,message x){
        ...
    }
    public message out(){
        ...
    }
    public double ta(){
        ...
    }
}

```

##### 3.1.2 General Structure of a PowerDEVS atomic model

###### Structure of the file modelname.pds

```

Root-Coordinator
{
    Simulator
    {
        Path = "path/modelname.h"
        Parameters =
    }
    EIC
    {
    }
    EOC
    {
    }
    IC
}

```

```

{
}
}

```

### Structure of the file modelname.h

```

#ifndef modelname_h
#define modelname_h
#include "simulator.h"
#include "path/modelname.h"
#include "event.h"
#include "stdarg.h"
#include "..."

class modelname: public Simulator {
#define INFINITY 1e10
public:
    modelname(const char *n): Simulator(n) {};
    void init(double, ...);
    double ta(double t);
    void dint(double);
    void dext(Event , double );
    Event lambda(double);
    void exit();
};
#endif

```

### Structure of the file modelname.cpp:

```

#include "modelname.h"
void modelname::init(double t,...) {
    ...
    /* Initialize variables (if applicable) */
}
void modelname::dint(double t){
    ...
}
void modelname::dext(Event x, double t){
    ...
}
Event modelname::lambda(double t) {
    ...
}
double modelname::ta(double t){
    ...
}
void modelname::exit() {
    ...
}

```

## 3.2 Definitions

From now on, to represent a translation it will be used the structure whose scheme is described in Figure 1. Each rule consists in: the code to be translated; a context, i.e. the conditions under which the rule applies; the translated code (Java code and C++ code); and, eventually, a comment about the rule. The expressions between double square brackets,  $\llbracket \dots \rrbracket$ , are expressions that still have to be translated. The subscript indicates the target language, Java or C++; and the superscript, the rule that must be applied (like in the header of the rule schema).

RULE'S NAME
$\llbracket \text{code to be translated} \rrbracket$
CONTEXT:
Conditions under which the rule applies
CODE:
<ul style="list-style-type: none"> <li>• DEVS-Suite: Resulting Java code</li> <li>• PowerDEVS: Resulting C++ code</li> </ul>
COMMENT:
(Optional comment)

Figure 1: Schema of a translation rule

---

```

    <S> ::= S is
        <definitions>
        [<synonyms>]
    end S
<definitions> ::= <id>{,<id>}:<type>;
        [<definitions>]
<type> ::= ℕ | ℤ | ℝ | Time | Text | Boolean | <enum>
        | ℙ <type>
        | List <type>
        | <type> ∪ <type> { ∪ <type> }
        | <type> × <type> { × <type> }
        | <type> → <type>
        | <id>
<synonyms> ::= <id> = <Type>;
        [<synonyms>]

```

---



---

## DEFINITION

$\llbracket \langle id \rangle : \langle type \rangle ; \rrbracket^D$

---

## CONTEXT:

$\langle id \rangle = \text{"varName"};$

---

## CODE:

Case  $\langle type \rangle$  of:

- ℕ:
  - DEVS-Suite:
 

```
Integer varName;
```
  - PowerDEVS:
 

```
unsigned int varName;
```
- ℤ:
  - DEVS-Suite:
 

```
Integer varName;
```
  - PowerDEVS:
 

```
int varName;
```
- ℝ:
  - DEVS-Suite:
 

```
Double varName;
```
  - PowerDEVS:
 

```
double varName;
```
- Time:
  - DEVS-Suite:
 

```
Double varName;
```
  - PowerDEVS:
 

```
double varName;
```
- Boolean:
  - DEVS-Suite:
 

```
Boolean varName;
```
  - PowerDEVS:
 

```
bool varName;
```
- Text:
  - DEVS-Suite:
 

```
String varName;
```
  - PowerDEVS:
 

```
const char* varName;
```
- $\langle enum \rangle$ 
  - DEVS-Suite:
 

```
String varName;
```
  - PowerDEVS:
 

```
const char* varName;
```

- $Type_1 \cup \dots \cup Type_n$

– DEVS-Suite:

```
static class T_varName extends Object implements Comparable<Object>{
    public [[Type1]]J v_[[Type1]]N;
    :
    public [[Typen]]J v_[[Typen]]N;
    public String type;
    public boolean equals(T_varName other){
        :
    }
    public int compareTo(T_varName other){
        :
    }
    T_varName(){
    }
    T_varName([[Type1]]J v){
        v_[[Type1]]N = v;
        type="[[Type1]]N";
    }
    :
    T_varName([[Typen]]J v){
        v_[[Typen]]N = v;
        type="[[Typen]]N";
    }

    T_varName(T_varName other){
        v_[[Type1]]N = other.v_[[Type1]]N;
        :
        v_[[Typen]]N = other.v_[[Typen]]N;
        type=other.type;
    }
}
T_varName varName;
```

– PowerDEVS:

```
class T_varName{
    [[Type1]]C v_[[Type1]]N;
    :
    [[Typen]]C v_[[Typen]]N;
    const char* type;
    bool operator<(const T_varName& other) const{
        :
    }
    bool operator==(const T_varName& other) const{
        :
    }
    T_varName& operator=(const T_varName other){
        :
    }
    T_varName(){
    };
    T_varName([[Type1]]C v){
        v_[[Type1]]N = v;
        type="[[Type1]]N";
    }
    :
    T_varName([[Typen]]C v){
        v_[[Typen]]N = v;
        type="[[Typen]]N";
    }
}
varName;
```



- $\text{Type}_1 \times \dots \times \text{Type}_n$

- DEVS-Suite:

```
static class T_varName extends Object implements Comparable<Object>{
    public [[Type1]]J v1;
    :
    public [[Typen]]J vn;
    public boolean equals(T_varName other){
        :
    }
    public int compareTo(T_varName other){
        :
    }
    T_varName(){
    }
    T_varName([[Type1]]J v1, ..., [[Typen]]J vn){
        this.v1 = v1;
        :
        this.vn = vn;
    }
    T_varName(T_varName other){
        v_[[Type1]]N = other.v_[[Type1]]N;
        :
        v_[[Typen]]N = other.v_[[Typen]]N;
    }
}
T_varName varName;
```

- PowerDEVS:

```
class T_varName{
    [[Type1]]C v1;
    :
    [[Typen]]C vn;
    bool operator<(const T_varName& other) const{
        :
    }
    bool operator==(const T_varName& other) const{
        :
    }
    T_varName& operator=(const T_varName other){
        :
    }
    T_varName(){
    };
    T_varName([[Type1]]C v1, ..., [[Typen]]C vn){
        this.v1 = v1;
        :
        this.vn = vn;
    }
}
varName;
```

- $\mathbb{P}$  Type

- DEVS-Suite:

```
Set<[[Type]]J> varName;
```

- PowerDEVS:

```
std::set<[[Type]]C> varName;
```

- List Type

- DEVS-Suite:

```
List<[[Type]]J> varName;
```

- PowerDEVS:

```
std::list<[[Type]]C> varName;
```

- $\text{Type}_1 \rightarrow \text{Type}_2$

- DEVS-Suite:

```
Map<[[Type1]]J, [[Type2]]J> varName;
```

- PowerDEVS:

```
std::map<[[Type1]]C, [[Type2]]C> varName;
```

### 3.3 Input ports

---

```
 $\langle X \rangle ::= X$  is  
     $\langle definitions \rangle$   
end X
```

---

#### CML-DEVS:

```
portName1 : Type1;  
...  
portNamen : Typen;
```

The following CML-DEVS variable is created:

```
Tin : Type1  $\cup$  ...  $\cup$  Typen;
```

and is translated into DEVS-Suite and PowerDEVS according the *definition* rule (described above). Further:

#### DEVS-Suite:

Inside the class ModelName constructor:

```
addImport("portName1");  
...  
addImport("portNameN");
```

#### PowerDEVS:

If the PowerDEVS IDE is used, the number of ports is specified in the file modelName.pd.

### 3.4 Output ports

---

```
 $\langle Y \rangle ::= Y$  is  
     $\langle definitions \rangle$   
end Y
```

---

#### CML-DEVS:

```
portName1 : Type1;  
...  
portNamen : Typen;
```

#### DEVS-Java:

Inside the class ModelName constructor:

```
addOutputport("portName1");  
...  
addInputport("portNameN");
```

Further, a class for each port is created, following the *definition* translation rules described before:

```
public class T_ModelName_Typei extends entity implements Comparable<Object>{  
    ...  
}
```

#### PowerDEVS:

If the PowerDEVS IDE is used, the number of ports is specified in the file modelName.pd.

Further, a class for each port is created, following the *definition* translation rules described before:

```
public class T_ModelName_Typei{  
    ...  
}
```

### 3.5 Transition, output and time advance functions

---

```

 $\langle \delta int \rangle ::= \delta int[\langle ids \rangle]$  is
    { $\langle sentence \rangle$ }
end  $\delta int$ 
 $\langle \delta ext \rangle ::= \delta ext[\langle ids \rangle]$  is
    { $\langle sentence \rangle$ }
end  $\delta ext$ 
 $\langle \lambda \rangle ::= \lambda[\langle ids \rangle]$  is
    { $\langle sentence \rangle$ }
end  $\lambda$ 
 $\langle ta \rangle ::= ta[\langle ids \rangle]$  is
    { $\langle sentence \rangle$ }
end ta
 $\langle ids \rangle ::= ((\langle id \rangle \mid \langle ids \rangle)\{,(\langle id \rangle \mid \langle ids \rangle)\})$ 
 $\langle sentence \rangle ::= \langle assignment \rangle$ 
    |  $\langle foreach \rangle$ 
    |  $\langle caseblock \rangle$ 
    |  $\langle whereblock \rangle$ 

```

---

#### DEVS-Java:

```

public void deltint(){
    /*sentences...*/
}
public deltext(double e,message x){
    /*sentences...*/
}
public message out(){
    message m=new message();
    content con;
    /*sentences...*/
}
public double ta(){
    /*sentences...*/
    return sigma;
}

```

#### PowerDEVS:

```

void modelName::dint(double t){
    /*sentences...*/
}
void mdoelName::dext(Event x, double t){
    /*sentences...*/
}
Event modelName::lambda(double t){
    /*sentences*/
    return Event(..., ...);
}
double modeName::ta(double t){
    /*sentences...*/
    return sigma;
}

```

### 3.6 Assignments

---

```

 $\langle assignment \rangle ::= \langle var \rangle = \langle expr \rangle;$ 
 $\langle var \rangle ::= \langle id \rangle \mid \langle idComp \rangle \mid \text{value} \mid \text{port} \mid e$ 
 $\langle idComp \rangle ::= \langle id \rangle . \langle digit \rangle \{ . \langle digit \rangle \}$ 
 $\langle expr \rangle ::= \langle var \rangle \mid \langle val \rangle \mid \langle vals \rangle \mid \langle operation \rangle$ 
 $\langle vals \rangle ::= (\langle expr \rangle, \langle expr \rangle \{, \langle expr \rangle \})$ 
 $\langle set \rangle ::= \{ \langle expr \rangle \{, \langle expr \rangle \}$ 
 $\langle list \rangle ::= \langle \langle expr \rangle \{, \langle expr \rangle \}$ 

```

---

---

**ASSIGNMENT Basic Numbers**
$$\llbracket \langle var \rangle = \langle expr \rangle \rrbracket^A$$

---

**CONTEXT:**
$$\text{Type}(\langle var \rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$$
$$\text{Type}(\langle expr \rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$$

---

**CODE:**

case  $\text{Type}(\langle var \rangle)$  of:

- $\mathbb{N} \mid \mathbb{Z}$ :
    - DEVS-Suite:  
$$\llbracket \langle var \rangle \rrbracket_J = \text{toInteger}(\llbracket \langle expr \rangle \rrbracket_J);$$
    - PowerDEVS:  
$$\llbracket \langle var \rangle \rrbracket_C = (\text{int})(\llbracket \langle expr \rangle \rrbracket_C);$$
  - $\mathbb{R} \mid \text{Time}$ :
    - DEVS-Suite:  
$$\llbracket \langle var \rangle \rrbracket_J = \text{toDouble}(\llbracket \langle expr \rangle \rrbracket_J);$$
    - PowerDEVS:  
$$\llbracket \langle var \rangle \rrbracket_C = (\text{double})(\llbracket \langle expr \rangle \rrbracket_C);$$
- 

---

**ASSIGNMENT Basic Not Numbers**
$$\llbracket \langle var \rangle = \langle expr \rangle \rrbracket^A$$

---

**CONTEXT:**
$$\text{Type}(\text{varId}) = \text{Type}(\langle expr \rangle) = \text{Text} \mid \text{Bool} \mid \langle enum \rangle$$

---

**CODE:**

- DEVS-Suite:  
$$\llbracket \langle var \rangle \rrbracket_J = \llbracket \langle expr \rangle \rrbracket_J;$$
  - PowerDEVS:  
$$\llbracket \langle var \rangle \rrbracket_C = \llbracket \langle expr \rangle \rrbracket_C;$$
- 

---

**ASSIGNMENT Tuple**
$$\llbracket \langle var \rangle = \langle expr \rangle \rrbracket^A$$

---

**CONTEXT:**
$$\langle var \rangle = \text{varId}$$
$$\text{Type}(\text{varId}) = \text{Type}_1 \times \dots \times \text{Type}_n$$

---

**CODE:**

Case  $\langle expr \rangle$  of:

- $\langle expr \rangle = (\text{expr}_1, \dots, \text{expr}_n)$   
$$\llbracket \text{varId}.1 = \text{expr}_1 \rrbracket^A$$
  
$$\vdots$$
  
$$\llbracket \text{varId}.n = \text{expr}_n \rrbracket^A$$
  - $\langle id \rangle = \text{varName}$   
$$\llbracket \text{varId}.1 = \text{varName}.1 \rrbracket^A$$
  
$$\vdots$$
  
$$\llbracket \text{varId}.n = \text{varName}.n \rrbracket^A$$
  - $\langle funAppl \rangle = \text{funId}(\text{var1}, \dots, \text{varn})$   
$$\llbracket \text{varTmp} : \text{TypeVarId} \rrbracket^D$$
  
$$\text{varTmp} = \text{funId}(\text{var1}, \dots, \text{varn});$$
  
$$\llbracket \text{varId}.1 = \text{varTmp}.1 \rrbracket^A$$
  
$$\vdots$$
  
$$\llbracket \text{varId}.n = \text{varTmp}.n \rrbracket^A$$
-

---

**ASSIGNMENT Union and Number**
$$\llbracket \langle var \rangle = \langle expr \rangle \rrbracket^A$$
**CONTEXT:**
$$\text{Type}(\langle var \rangle) = \text{Type}_1 \cup \dots \cup \text{Type}_n$$
$$\text{Type}(\langle expr \rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$$
$$\text{Type}_i \in \{\text{Type}_1, \dots, \text{Type}_n\}$$
$$\text{Type}_i = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$$
**CODE:**case  $\text{Type}_i$  of:•  $\mathbb{N} \mid \mathbb{Z}$ :

– DEVS-Suite:

$$\llbracket \langle var \rangle \rrbracket_J.v_{\llbracket \text{Type}_i \rrbracket^N} = \text{toInteger}(\llbracket \langle expr \rangle \rrbracket_J);$$
$$\llbracket \langle var \rangle \rrbracket_J.type = "\llbracket \text{Type}_i \rrbracket^N";$$

– PowerDEVS:

$$\llbracket \langle var \rangle \rrbracket_C.v_{\llbracket \text{Type}_i \rrbracket^N} = (\text{int})(\llbracket \langle expr \rangle \rrbracket_C);$$
$$\llbracket \langle var \rangle \rrbracket_C.type = "\llbracket \text{Type}_i \rrbracket^N";$$
•  $\mathbb{R} \mid \text{Time}$ :

– DEVS-Suite:

$$\llbracket \langle var \rangle \rrbracket_J.v_{\llbracket \text{Type}_i \rrbracket^N} = \text{toDouble}(\llbracket \langle expr \rangle \rrbracket_J);$$
$$\llbracket \langle var \rangle \rrbracket_J.type = "\llbracket \text{Type}_i \rrbracket^N";$$

– PowerDEVS:

$$\llbracket \langle var \rangle \rrbracket_C.v_{\llbracket \text{Type}_i \rrbracket^N} = (\text{double})(\llbracket \langle expr \rangle \rrbracket_C);$$
$$\llbracket \langle var \rangle \rrbracket_C.type = "\llbracket \text{Type}_i \rrbracket^N";$$

---

**ASSIGNMENT Union and Not a Number**
$$\llbracket \langle var \rangle = \langle expr \rangle \rrbracket^A$$
**CONTEXT:**
$$\text{Type}(\langle var \rangle) = \text{Type}_1 \cup \dots \cup \text{Type}_n$$
$$\text{Type}(\langle expr \rangle) = \text{Type}_i \in \{\text{Type}_1, \dots, \text{Type}_n\}$$
$$\text{Type}_i \neq \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$$
**CODE:**

• DEVS-Suite:

$$\llbracket \langle var \rangle \rrbracket_J.v_{\llbracket \text{Type}_i \rrbracket^N} = \langle expr \rangle^A$$
$$\llbracket \langle var \rangle \rrbracket_J.type = "\llbracket \text{Type}_i \rrbracket^N";$$

• PowerDEVS:

$$\llbracket \langle var \rangle \rrbracket_C.v_{\llbracket \text{Type}_i \rrbracket^N} = \langle expr \rangle^A$$
$$\llbracket \langle var \rangle \rrbracket_C.type = "\llbracket \text{Type}_i \rrbracket^N";$$

---

**ASSIGNMENT Union and Number in Union**
$$\llbracket \langle var \rangle = \langle expr \rangle \rrbracket^A$$
**CONTEXT:**
$$\text{Type}(\langle var \rangle) = \text{Type}_1 \cup \dots \cup \text{Type}_n$$
$$\text{Type}(\langle expr \rangle) = \text{Type}_{n+1} \cup \dots \cup \text{Type}_m$$
$$\text{Type}_i = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$$
$$\langle var \rangle.type = "\llbracket \text{Type}_i \rrbracket^N"$$
$$\text{Type}_j = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$$
$$\langle expr \rangle.type = "\llbracket \text{Type}_j \rrbracket^N"$$

---

**CODE:**

case  $\text{Type}_i$  of:

- $\mathbb{N} \mid \mathbb{Z}$ :
    - DEVS-Suite:  
 $\llbracket \langle \text{var} \rangle \rrbracket_J.v\_ \llbracket \text{Type}_i \rrbracket^N = \text{toInteger}(\llbracket \langle \text{expr} \rangle \rrbracket_J.v\_ \llbracket \text{Type}_j \rrbracket^N);$   
 $\llbracket \langle \text{var} \rangle \rrbracket_J.\text{type} = "\llbracket \text{Type}_i \rrbracket^N";$
    - PowerDEVS:  
 $\llbracket \langle \text{var} \rangle \rrbracket_C.v\_ \llbracket \text{Type}_i \rrbracket^N = (\text{int})(\llbracket \langle \text{expr} \rangle \rrbracket_C.v\_ \llbracket \text{Type}_j \rrbracket^N);$   
 $\llbracket \langle \text{var} \rangle \rrbracket_C.\text{type} = "\llbracket \text{Type}_i \rrbracket^N";$
  - $\mathbb{R} \mid \text{Time}$ :
    - DEVS-Suite:  
 $\llbracket \langle \text{var} \rangle \rrbracket_J.v\_ \llbracket \text{Type}_i \rrbracket^N = \text{toDouble}(\llbracket \langle \text{expr} \rangle \rrbracket_J.v\_ \llbracket \text{Type}_j \rrbracket^N);$   
 $\llbracket \langle \text{var} \rangle \rrbracket_J.\text{type} = "\llbracket \text{Type}_i \rrbracket^N";$
    - PowerDEVS:  
 $\llbracket \langle \text{var} \rangle \rrbracket_C.v\_ \llbracket \text{Type}_i \rrbracket^N = (\text{double})(\llbracket \langle \text{expr} \rangle \rrbracket_C.v\_ \llbracket \text{Type}_j \rrbracket^N);$   
 $\llbracket \langle \text{var} \rangle \rrbracket_C.\text{type} = "\llbracket \text{Type}_i \rrbracket^N";$
- 

---

**ASSIGNMENT Union and Not a Number in Union**

$\llbracket \langle \text{var} \rangle = \langle \text{expr} \rangle \rrbracket^A$

---

**CONTEXT:**

$\text{Type}(\langle \text{var} \rangle) = \text{Type}_1 \cup \dots \cup \text{Type}_n$   
 $\text{Type}(\langle \text{expr} \rangle) = \text{Type}_{n+1} \cup \dots \cup \text{Type}_m$   
 $\text{Type}_i \in \{\text{Type}_1, \dots, \text{Type}_n\}$   
 $\text{Type}_j \neq \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$   
 $\langle \text{expr} \rangle.\text{type} = "\llbracket \text{Type}_j \rrbracket^N"$   
 $\text{Type}_i = \text{Type}_j$

---

**CODE:**

- DEVS-Suite:  
 $\llbracket \langle \text{var} \rangle \rrbracket_J.v\_ \llbracket \text{Type}_j \rrbracket^N = \langle \text{expr} \rangle.v\_ \llbracket \text{Type}_j \rrbracket^N^A$   
 $\llbracket \langle \text{var} \rangle \rrbracket_J.\text{type} = "\llbracket \text{Type}_i \rrbracket^N";$
  - PowerDEVS:  
 $\llbracket \langle \text{var} \rangle \rrbracket_C.v\_ \llbracket \text{Type}_j \rrbracket^N = \langle \text{expr} \rangle.v\_ \llbracket \text{Type}_j \rrbracket^N^A$   
 $\llbracket \langle \text{var} \rangle \rrbracket_C.\text{type} = "\llbracket \text{Type}_i \rrbracket^N";$
- 

---

**ASSIGNMENT Set**

$\llbracket \text{varId} = \langle \text{expr} \rangle \rrbracket^A$

---

**CONTEXT:**

$\text{Type}(\text{varId}) = \text{Type}(\langle \text{expr} \rangle) = \mathbb{P} \text{Type}$

---

**CODE:**

Case  $\langle \text{expr} \rangle$  of:

- $\langle \text{set} \rangle = \{\text{expr}_1, \dots, \text{expr}_n\}$ 
  - DEVS-Suite:  
 $\text{varId} = \text{buildSet}(\text{new } \llbracket \text{Type} \rrbracket_J(\llbracket \text{expr}_1 \rrbracket_J), \dots, \text{new } \llbracket \text{Type} \rrbracket_J(\llbracket \text{expr}_n \rrbracket_J));$
  - PowerDEVS:  
 $\text{varId} = \text{buildSet} \langle \llbracket \text{Type} \rrbracket_C \rangle(n, \llbracket \text{expr}_1 \rrbracket_C, \dots, \llbracket \text{expr}_n \rrbracket_C);$
- $\langle \text{var} \rangle \mid \langle \text{operation} \rangle$ 
  - DEVS-Suite:  
 $\text{varId}.\text{clear}();$   
 $\text{varId}.\text{addAll}(\llbracket \langle \text{expr} \rangle \rrbracket_J);$
  - PowerDEVS:  
 $\text{varId} = \llbracket \langle \text{expr} \rangle \rrbracket_C;$

---

---

**ASSIGNMENT List**
$$\llbracket \text{varId} = \langle \text{expr} \rangle \rrbracket^A$$

---

**CONTEXT:**
$$\text{Type}(\text{varId}) = \text{Type}(\langle \text{expr} \rangle) = \text{List Type}$$

---

**CODE:***Case*  $\langle \text{expr} \rangle$  *of:*

- $\langle \text{list} \rangle = \langle \text{expr}_1, \dots, \text{expr}_n \rangle$ 
    - DEVS-Suite:  
 $\text{varId} = \text{buildList}(\text{new } \llbracket \text{Type} \rrbracket_J(\llbracket \text{expr}_1 \rrbracket_J), \dots, \text{new } \llbracket \text{Type} \rrbracket_J(\llbracket \text{expr}_n \rrbracket_J));$
    - PowerDEVS:  
 $\text{varId} = \text{buildList}(\llbracket \text{Type} \rrbracket_C(n, \llbracket \text{expr}_1 \rrbracket_C, \dots, \llbracket \text{expr}_n \rrbracket_C);$
  - $\langle \text{var} \rangle \mid \langle \text{operation} \rangle$ 
    - DEVS-Suite:  
 $\text{varId.clear()};$   
 $\text{varId.addAll}(\llbracket \langle \text{expr} \rangle \rrbracket_J);$
    - PowerDEVS:  
 $\text{varId} = \llbracket \langle \text{expr} \rangle \rrbracket_C;$
- 

---

**ASSIGNMENT Partial Function**
$$\llbracket \text{varId} = \langle \text{expr} \rangle \rrbracket^A$$

---

**CONTEXT:**
$$\text{Type}(\text{varId}) = \text{Type}(\langle \text{expr} \rangle) = \text{Type1} \rightarrow \text{Type2}$$

---

**CODE:***Case*  $\langle \text{expr} \rangle$  *of:*

- $\langle \text{set} \rangle = \{(\langle \text{expr} \rangle_l^1, \langle \text{expr} \rangle_r^1), \dots, (\langle \text{expr} \rangle_l^n, \langle \text{expr} \rangle_r^n)\}$ 
    - DEVS-Suite:  
 $\text{varId} = \text{buildMap}(\text{new Pair}(\llbracket \text{Type1} \rrbracket_J, \llbracket \text{Type2} \rrbracket_J) < \llbracket \langle \text{expr} \rangle_l^1 \rrbracket_J, \llbracket \langle \text{expr} \rangle_r^1 \rrbracket_J) \dots,$   
 $\text{new Pair}(\llbracket \text{Type1} \rrbracket_J, \llbracket \text{Type2} \rrbracket_J) < \llbracket \langle \text{expr} \rangle_l^n \rrbracket_J, \llbracket \langle \text{expr} \rangle_r^n \rrbracket_J));$
    - PowerDEVS:  
 $\text{varId} = \text{buildMap}(\llbracket \text{Type1} \rrbracket_C, \llbracket \text{Type2} \rrbracket_C) < (n, \text{Pair}(\llbracket \text{Type1} \rrbracket_C, \llbracket \text{Type2} \rrbracket_C) < (\llbracket \langle \text{expr} \rangle_l^1 \rrbracket_C, \llbracket \langle \text{expr} \rangle_r^1 \rrbracket_C) \dots,$   
 $\text{Pair}(\llbracket \text{Type1} \rrbracket_C, \llbracket \text{Type2} \rrbracket_C) < (\llbracket \langle \text{expr} \rangle_l^n \rrbracket_C, \llbracket \langle \text{expr} \rangle_r^n \rrbracket_C));$
  - $\langle \text{var} \rangle \mid \langle \text{operation} \rangle$ 
    - DEVS-Suite:  
 $\text{varId.clear()};$   
 $\text{varId.putAll}(\llbracket \langle \text{expr} \rangle \rrbracket_J);$
    - PowerDEVS:  
 $\text{varId} = \llbracket \langle \text{expr} \rangle \rrbracket_C;$
- 

### 3.7 Partial Function Application

$$\langle pFun \rangle ::= \text{id } \langle \text{expr} \rangle$$

---

---

**Partial Function Application**
$$\llbracket \langle \text{id} \rangle \langle \text{expr} \rangle \rrbracket$$

---

**CONTEXT:**
$$\langle \text{id} \rangle = \text{varName}$$
$$\text{Type}(\langle \text{id} \rangle) = \text{TypeL} \rightarrow \text{TypeR}$$
$$\text{Type}(\langle \text{expr} \rangle) = \text{TypeL} \text{ (or } \text{Type}_1 \cup \dots \text{Type}_n \text{ with } \text{Type}_i = \text{TypeL})$$

---

**CODE:**

- DEVS-Suite:  
varName.get( $\llbracket \langle expr \rangle \rrbracket_J$ )
- PowerDEVS:  
varName( $\llbracket \langle expr \rangle \rrbracket_C$ )

### 3.8 Sentences “For Each”

```

 $\langle foreach \rangle ::= \text{foreach } \langle id \rangle \text{ in } \langle expr \rangle \text{ do}$ 
     $\langle sentence \rangle$ 
    {  $\langle sentence \rangle$  }
    end foreach

```

#### FOR EACH Set

```

 $\llbracket \text{foreach } \langle id \rangle \text{ in } \langle expr \rangle \{$ 
     $\langle sentences \rangle$ 
 $\} \rrbracket$ 

```

#### CONTEXT:

```

 $\langle id \rangle = \text{varName}$ 
Type( $\langle id \rangle$ ) = TypeId
Type( $\langle expr \rangle$ ) =  $\mathbb{P}$  TypeId

```

#### CODE:

```

Case  $\langle expr \rangle$  of:
•  $\langle set \rangle \mid \langle operation \rangle$ 
    – DEVS-Suite:
        Set<TypeId> setTmp =  $\llbracket \langle expr \rangle \rrbracket_J$ ;
        for(TypeId varName: setTmp){
             $\llbracket \langle sentences \rangle \rrbracket_J$ 
        }
    – PowerDEVS:
        std::set<TypeId> setTmp= $\llbracket \langle expr \rangle \rrbracket_C$ ;
        for (std::set<TypeId>::iterator it = setTmp.begin(); it != setTmp.end(); it++){
            TypeId varName = *it;
             $\llbracket \langle sentences \rangle \rrbracket_C$ 
        }
•  $\langle var \rangle = \text{varName2}$ ;
    – DEVS-Suite:
        Set<TypeId> setTmp = new TreeSet<TypeId>(varName2);
        for(TypeId varName: setTmp){
             $\llbracket \langle sentences \rangle \rrbracket_J$ 
        }
    – PowerDEVS:
        std::set<TypeId> setTmp = varName2;
        for(std::set<TypeId>::iterator it = setTmp.begin(); it!=setTmp.end(); it++){
            TypeId varName = *it;
             $\llbracket \langle sentences \rangle \rrbracket_C$ 
        }

```

### 3.9 Definition be cases

```

 $\langle caseblock \rangle ::= \text{defcases}$ 
    (case  $\langle sentence \rangle$ ; { $\langle sentence \rangle$ ;} if  $\langle conditions \rangle$ )
    | (if  $\langle conditions \rangle \Rightarrow \langle sentence \rangle$ ; { $\langle sentence \rangle$ ;} )
    | ({case  $\langle sentence \rangle$ ; { $\langle sentence \rangle$ ;} if  $\langle conditions \rangle$ })
    | ({if  $\langle conditions \rangle \Rightarrow \langle sentence \rangle$ ; { $\langle sentence \rangle$ ;} })
    [default  $\langle sentence \rangle$ ; { $\langle sentence \rangle$ ;} ]
    end defcases

```



---

**CASE BLOCK**

```
[[defcases
  ⋮
defcases]]
```

---

**CONTEXT:****CODE:**

- DEVS-Suite:

```
if (/conditions*){
  /sentences*/
}
else if (/conditions*){
  /sentences*/
}
⋮
else{ /*si es que está definido '\default'*/
  /sentences*/
}
```
  - PowerDEVS:

```
if (/conditions*){
  /sentences*/
}
else if (/conditions*){
  /sentences*/
}
⋮
else{ /*si es que está definido '\default'*/
  /sentences*/
}
```
- 

### 3.10 Conditions

---

$$\begin{aligned} \langle conditions \rangle ::= & \langle condition \rangle \\ & | \neg (\langle conditions \rangle) \\ & | (\langle conditions \rangle \wedge \langle conditions \rangle) \\ & | (\langle conditions \rangle \vee \langle conditions \rangle) \end{aligned}$$

---

---

**CONDITIONS**
$$\llbracket \langle conditions \rangle \rrbracket^{\text{Cond}}$$

---

**CONTEXT:****CODE:**

- Case*  $\langle conditions \rangle$  *of*:
- $\langle condition \rangle$ :
$$\llbracket \langle condition \rangle \rrbracket^{\text{Cond}}$$
  - $\neg (\langle condition \rangle)$ :
    - DEVS-Suite:
$$! (\llbracket \langle condition \rangle \rrbracket^{\text{Cond}})$$
    - PowerDEVS:
$$! (\llbracket \langle condition \rangle \rrbracket^{\text{Cond}})$$
  - $(\langle condition1 \rangle \wedge \langle condition2 \rangle)$ :
    - DEVS-Suite:
$$((\llbracket \langle condition1 \rangle \rrbracket^{\text{Cond}}) \ \&\& \ (\llbracket \langle condition2 \rangle \rrbracket^{\text{Cond}}))$$
    - PowerDEVS:

$$((\llbracket \langle condition1 \rangle \rrbracket^{Cond}) \ \&\& \ (\llbracket \langle condition2 \rangle \rrbracket^{Cond}))$$

- $(\langle condition1 \rangle \vee \langle condition2 \rangle)$ :
  - DEVS-Suite:
 
$$((\llbracket \langle condition1 \rangle \rrbracket^{Cond}) \ || \ (\llbracket \langle condition2 \rangle \rrbracket^{Cond}))$$
  - PowerDEVS:
 
$$((\llbracket \langle condition1 \rangle \rrbracket^{Cond}) \ || \ (\llbracket \langle condition2 \rangle \rrbracket^{Cond}))$$

### 3.10.1 Condition

$$\langle condition \rangle ::= (\langle var \rangle \mid \langle val \rangle \mid \langle operation \rangle) \langle comparison \rangle (\langle var \rangle \mid \langle val \rangle \mid \langle operation \rangle)$$

$$\langle comparison \rangle ::= < \mid > \mid \leq \mid \geq \mid = \mid \neq \mid \in \mid \notin \mid \subset \mid \subseteq$$

#### CONDITION

$$\llbracket \langle condition \rangle \rrbracket^{Cond}$$

#### CONTEXT:

$$\langle condition \rangle = \langle expr1 \rangle \langle comparison \rangle \langle expr2 \rangle$$

#### CODE:

Case  $\langle comparison \rangle$  of:

- =:
 
$$\llbracket \langle expr1 \rangle \rrbracket = \langle expr2 \rangle \rrbracket^E$$
- $\neq$ :
 
$$!(\llbracket \langle expr1 \rangle \rrbracket = \langle expr2 \rangle \rrbracket^E)$$
- <:
 
$$\llbracket \langle expr1 \rangle \rrbracket < \langle expr2 \rangle \rrbracket^L$$
- >:
 
$$\llbracket \langle expr1 \rangle \rrbracket > \langle expr2 \rangle \rrbracket^G$$
- $\leq$ :
 
$$\llbracket \langle expr1 \rangle \rrbracket \leq \langle expr2 \rangle \rrbracket^{LE}$$
- $\geq$ :
 
$$\llbracket \langle expr1 \rangle \rrbracket \geq \langle expr2 \rangle \rrbracket^{GE}$$
- $\in$ :
 
$$\llbracket \langle expr2 \rangle \rrbracket \in \langle expr1 \rangle \rrbracket^B$$
- $\notin$ :
  - DEVS-Suite:
 
$$!(\llbracket \langle expr2 \rangle \rrbracket \in \langle expr1 \rangle \rrbracket^B)$$
  - PowerDEVS:
 
$$!(\llbracket \langle expr2 \rangle \rrbracket \in \langle expr1 \rangle \rrbracket^B)$$
- $\subset$ :
  - DEVS-Suite:
 
$$((\llbracket \langle expr2 \rangle \rrbracket_J).containsAll(\llbracket \langle expr1 \rangle \rrbracket_J) \ \&\& \ ((\llbracket \langle expr1 \rangle \rrbracket_J).size() < (\llbracket \langle expr2 \rangle \rrbracket_J).size()))$$
  - PowerDEVS:
 
$$(std::includes((\llbracket \langle expr1 \rangle \rrbracket_C).begin(), (\llbracket \langle expr2 \rangle \rrbracket_C).end(), (\llbracket \langle expr2 \rangle \rrbracket_C).begin(), (\llbracket \langle expr2 \rangle \rrbracket_C).end()) \ \&\& \ ((\llbracket \langle expr1 \rangle \rrbracket_C).size() < (\llbracket \langle expr2 \rangle \rrbracket_C).size()))$$
- $\subseteq$ :
  - DEVS-Suite:
 
$$((\llbracket \langle expr2 \rangle \rrbracket_J).containsAll(\llbracket \langle expr1 \rangle \rrbracket_J))$$
  - PowerDEVS:
 
$$(std::includes((\llbracket \langle expr1 \rangle \rrbracket_C).begin(), (\llbracket \langle expr2 \rangle \rrbracket_C).end(), (\llbracket \langle expr2 \rangle \rrbracket_C).begin(), (\llbracket \langle expr2 \rangle \rrbracket_C).end()) \ \&\& \ ((\llbracket \langle expr2 \rangle \rrbracket_C).size() < (\llbracket \langle expr2 \rangle \rrbracket_C).size()))$$

### 3.10.2 Equality comparison

---

#### EQUALS Basic

$\llbracket \langle expr1 \rangle = \langle expr2 \rangle \rrbracket^E$

---

#### CONTEXT:

$Type(\langle expr1 \rangle) = Type(\langle expr2 \rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time} \mid \text{Text} \mid \text{Bool} \mid \langle enum \rangle$

---

#### CODE:

- DEVS-Suite:  
 $(\llbracket \langle expr1 \rangle \rrbracket_J).equals(\llbracket \langle expr2 \rangle \rrbracket_J)$
  - PowerDEVS:  
 $(\llbracket \langle expr1 \rangle \rrbracket_C) == (\llbracket \langle expr2 \rangle \rrbracket_C)$
- 

---

#### EQUALS Union1

$\llbracket \langle expr1 \rangle = \langle expr2 \rangle \rrbracket^E$

---

#### CONTEXT:

$Type(\langle expr1 \rangle) = Type_1 \cup \dots \cup Type_n$   
 $Type(\langle expr2 \rangle) = Type_i \in \{Type_1, \dots, Type_n\}$

---

#### CODE:

- DEVS-Suite:  
 $((\llbracket \langle expr1 \rangle \rrbracket_J).type == \llbracket Type_i \rrbracket^N) \ \&\& \ ((\llbracket \langle expr1 \rangle \rrbracket_J).v_{\llbracket Type_i \rrbracket^N} = \langle expr2 \rangle \rrbracket^E)$
  - PowerDEVS:  
 $((\llbracket \langle expr1 \rangle \rrbracket_C).type == \llbracket Type_i \rrbracket^N) \ \&\& \ ((\llbracket \langle expr1 \rangle \rrbracket_C).v_{\llbracket Type_i \rrbracket^N} = \langle expr2 \rangle \rrbracket^E)$
- 

#### COMMENT:

The Union type is already normalized

---

---

#### EQUALS Union2

$\llbracket \langle expr1 \rangle = \langle expr2 \rangle \rrbracket^E$

---

#### CONTEXT:

$Type(\langle expr1 \rangle) = Type_1 \cup \dots \cup Type_n$   
 $Type(\langle expr2 \rangle) = Type_{n+1} \cup \dots \cup Type_m$

---

#### CODE:

- DEVS-Suite:  
 $((\llbracket \langle expr1 \rangle \rrbracket_J).type == (\llbracket \langle expr2 \rangle \rrbracket_J).type) \ \&\& \ ((\llbracket \langle expr1 \rangle \rrbracket_J).v_{\llbracket Type_i \rrbracket^N} = (\llbracket \langle expr2 \rangle \rrbracket_J).v_{\llbracket Type_j \rrbracket^N} \rrbracket^E)$
  - PowerDEVS:  
 $((\llbracket \langle expr1 \rangle \rrbracket_C).type == (\llbracket \langle expr2 \rangle \rrbracket_C).type) \ \&\& \ ((\llbracket \langle expr1 \rangle \rrbracket_C).v_{\llbracket Type_i \rrbracket^N} = (\llbracket \langle expr2 \rangle \rrbracket_C).v_{\llbracket Type_j \rrbracket^N} \rrbracket^E)$
- 

#### COMMENT:

The Union types are already normalized

---

---

#### EQUALS Tuple

$\llbracket \langle exprA \rangle = \langle exprB \rangle \rrbracket^E$

---

#### CONTEXT:

$Type(\langle exprA \rangle) = Type(\langle exprB \rangle) = Type_1 \times \dots \times Type_n$

---

#### CODE:

Case  $\langle exprA \rangle, \langle exprB \rangle$  of:

- $\langle exprA \rangle = \text{"varName1"}, \langle exprB \rangle = \text{"varName2"}$ 
  - DEVS-Suite:  
 $varName1.equals(varName2)$
  - PowerDEVS:

varName1 == varName2

- $\langle \langle exprA \rangle = \text{"varName"}, \langle exprB \rangle = (expr_1, \dots, expr_n) \rangle$   
 $\vee \langle \langle exprA \rangle = (expr_1, \dots, expr_n), \langle exprB \rangle = \text{"varName"} \rangle$ 
  - DEVS-Suite:  
 $\llbracket \text{varName}.1 = expr_1 \rrbracket^E$   
 $\vdots$   
 $\llbracket \text{varName}.n = expr_n \rrbracket^E$
  - PowerDEVS:  
 $\llbracket \text{varName}.1 = expr_1 \rrbracket^E$   
 $\vdots$   
 $\llbracket \text{varName}.n = expr_n \rrbracket^E$
- $\langle \langle exprA \rangle = (expr_1^A, \dots, expr_n^A), \langle exprB \rangle = (expr_1^B, \dots, expr_n^B) \rangle$ 
  - DEVS-Suite:  
 $\llbracket expr_1^A = expr_1^B \rrbracket^E$   
 $\vdots$   
 $\llbracket expr_n^A = expr_n^B \rrbracket^E$
  - PowerDEVS:  
 $\llbracket expr_1^A = expr_1^B \rrbracket^E$   
 $\vdots$   
 $\llbracket expr_n^A = expr_n^B \rrbracket^E$

## EQUALS Set

$\llbracket \langle exprA \rangle = \langle exprB \rangle \rrbracket^E$

## CONTEXT:

$\text{Type}(\langle exprA \rangle) = \text{Type}(\langle exprB \rangle) = \mathbb{P} \text{ Type}$

## CODE:

Case  $\langle exprA \rangle, \langle exprB \rangle$  of:

- $\langle exprA \rangle = \text{"varName1"}, \langle exprB \rangle = \text{"varName2"}$ 
  - DEVS-Suite:  
`varName1.equals(varName2)`
  - PowerDEVS:  
`varName1 == varName2`
- $\langle \langle exprA \rangle = \text{"varName"}, \langle exprB \rangle = \{expr_1, \dots, expr_n\} \rangle$   
 $\vee \langle \langle exprA \rangle = \{expr_1, \dots, expr_n\}, \langle exprB \rangle = \text{"varName"} \rangle$ 
  - DEVS-Suite:  
`varName.equals(buildSet( $\llbracket expr_1 \rrbracket_J, \dots, \llbracket expr_n \rrbracket_J$ ))`
  - PowerDEVS:  
`varName==(buildSet< $\llbracket \text{Type} \rrbracket_c$ >(n,  $\llbracket expr_1 \rrbracket_c, \dots, \llbracket expr_n \rrbracket_c$ ))`
- $\langle \langle exprA \rangle = \{expr_1^A, \dots, expr_n^A\}, \langle exprB \rangle = \{expr_1^B, \dots, expr_n^B\} \rangle$ 
  - DEVS-Suite:  
`(buildSet( $\llbracket expr_1^A \rrbracket_J, \dots, \llbracket expr_n^A \rrbracket_J$ )).equals(  
buildSet( $\llbracket expr_1^B \rrbracket_J, \dots, \llbracket expr_n^B \rrbracket_J$ ))`
  - PowerDEVS:  
`(buildSet< $\llbracket \text{Type} \rrbracket_c$ >(n,  $\llbracket expr_1^A \rrbracket_c, \dots, \llbracket expr_n^A \rrbracket_c$ ))  
==(buildSet< $\llbracket \text{Type} \rrbracket_c$ >(n,  $\llbracket expr_1^B \rrbracket_c, \dots, \llbracket expr_n^B \rrbracket_c$ ))`

---

---

**EQUALS List**
$$\llbracket \langle \text{exprA} \rangle = \langle \text{exprB} \rangle \rrbracket^E$$

---

**CONTEXT:**
$$\text{Type}(\langle \text{exprA} \rangle) = \text{Type}(\langle \text{exprB} \rangle) = \text{List Type}$$

---

**CODE:**

Case  $\langle \text{exprA} \rangle, \langle \text{exprB} \rangle$  of:

- $\langle \text{exprA} \rangle = \text{"varName1"} \wedge \langle \text{exprB} \rangle = \text{"varName2"}$ 
    - DEVS-Suite:  
varName1.equals(varName2)
    - PowerDEVS:  
varName1 == varName2
  - $(\langle \text{exprA} \rangle = \text{"varName"}, \langle \text{exprB} \rangle = \langle \text{expr}_1, \dots, \text{expr}_n \rangle)$   
 $\vee (\langle \text{exprA} \rangle = \langle \text{expr}_1, \dots, \text{expr}_n \rangle, \langle \text{exprB} \rangle = \text{"varName"})$ 
    - DEVS-Suite:  
varName.equals(buildList( $\llbracket \text{expr}_1 \rrbracket_J, \dots, \llbracket \text{expr}_n \rrbracket_J$ ))
    - PowerDEVS:  
varName==(buildList<Type>(n, $\llbracket \text{expr}_1 \rrbracket_C, \dots, \llbracket \text{expr}_n \rrbracket_C$ ))
  - $\langle \text{exprA} \rangle = \langle \text{expr}_1^A, \dots, \text{expr}_n^A \rangle, \langle \text{exprB} \rangle = \langle \text{expr}_1^B, \dots, \text{expr}_n^B \rangle$ 
    - DEVS-Suite:  
(buildList( $\llbracket \text{expr}_1^A \rrbracket_J, \dots, \llbracket \text{expr}_n^A \rrbracket_J$ )).equals(buildList( $\llbracket \text{expr}_1^B \rrbracket_J, \dots, \llbracket \text{expr}_n^B \rrbracket_J$ ))
    - PowerDEVS:  
(buildList<Type>(n, $\llbracket \text{expr}_1^A \rrbracket_C, \dots, \llbracket \text{expr}_n^A \rrbracket_C$ ))==  
(buildList<Type>(n, $\llbracket \text{expr}_1^B \rrbracket_C, \dots, \llbracket \text{expr}_n^B \rrbracket_C$ ))
- 
- 

---

---

**EQUALS Partial Function**
$$\llbracket \langle \text{exprA} \rangle = \langle \text{exprB} \rangle \rrbracket^E$$

---

**CONTEXT:**
$$\text{Type}(\langle \text{exprA} \rangle) = \text{Type}(\langle \text{exprB} \rangle) = \text{TypeL} \rightarrow \text{TypeR}$$

---

**CODE:**

Case  $\langle \text{exprA} \rangle, \langle \text{exprB} \rangle$  of:

- "varName1", "varName2":
  - DEVS-Suite:  
varName1.equals(varName2)
  - PowerDEVS:  
varName1 == varName2
- "varName",  $\{(\langle \text{expr} \rangle_l^1, \langle \text{expr} \rangle_r^1), \dots, (\langle \text{expr} \rangle_l^n, \langle \text{expr} \rangle_r^n)\}$ :
  - DEVS-Suite:  
varName.equals(buildMap(new Pair< $\llbracket \text{TypeL} \rrbracket_J, \llbracket \text{TypeR} \rrbracket_J$ >( $\llbracket \langle \text{expr} \rangle_l^1 \rrbracket_J, \llbracket \langle \text{expr} \rangle_r^1 \rrbracket_J$ ) ...,  
new Pair< $\llbracket \text{TypeL} \rrbracket_J, \llbracket \text{TypeR} \rrbracket_J$ >( $\llbracket \langle \text{expr} \rangle_l^n \rrbracket_J, \llbracket \langle \text{expr} \rangle_r^n \rrbracket_J$ )))
  - PowerDEVS:  
varName==(buildMap< $\llbracket \text{TypeL} \rrbracket_C, \llbracket \text{TypeR} \rrbracket_C$ >(n, Pair< $\llbracket \text{TypeL} \rrbracket_C, \llbracket \text{TypeR} \rrbracket_C$ >( $\llbracket \langle \text{expr} \rangle_l^1 \rrbracket_C, \llbracket \langle \text{expr} \rangle_r^1 \rrbracket_C$ ) ...,  
Pair< $\llbracket \text{TypeL} \rrbracket_C, \llbracket \text{TypeR} \rrbracket_C$ >( $\llbracket \langle \text{expr} \rangle_l^n \rrbracket_C, \llbracket \langle \text{expr} \rangle_r^n \rrbracket_C$ )))
- $\{(\langle \text{exprA} \rangle_l^1, \langle \text{exprA} \rangle_r^1), \dots, (\langle \text{exprA} \rangle_l^n, \langle \text{exprA} \rangle_r^n)\}, \{(\langle \text{exprB} \rangle_l^1, \langle \text{exprB} \rangle_r^1), \dots, (\langle \text{exprB} \rangle_l^n, \langle \text{exprB} \rangle_r^n)\}$ :
  - DEVS-Suite:

```

    (buildMap(new Pair<[[TypeL]]J, [[TypeR]]J>([[<exprA>i1]]J, [[<exprA>r1]]J) ...,
      new Pair<[[TypeL]]J, [[TypeR]]J>([[<exprA>in]]J, [[<exprA>rn]]J)).equals(buildMap(
        new Pair<[[TypeL]]J, [[TypeR]]J>([[<exprB>i1]]J, [[<exprB>r1]]J) ...,
        new Pair<[[TypeL]]J, [[TypeR]]J>([[<exprB>im]]J, [[<exprB>rm]]J)))

– PowerDEVS:
    (buildMap<[[TypeL]]c, [[TypeR]]c>(n, Pair<[[TypeL]]c, [[TypeR]]c>([[<exprA>i1]]c, [[<exprA>r1]]c) ...,
      Pair<[[TypeL]]c, [[TypeR]]c>([[<exprA>i1]]c, [[<exprA>r1]]c)) ==
    (buildMap<[[TypeL]]c, [[TypeR]]c>(n, Pair<[[TypeL]]c, [[TypeR]]c>([[<exprB>i1]]c, [[<exprB>r1]]c) ...,
      Pair<[[TypeL]]c, [[TypeR]]c>([[<exprB>im]]c, [[<exprB>rm]]c)))

```

---

### 3.10.3 Comparison “less” (<)

---

#### LESS Numbers

$[[\langle expr1 \rangle < \langle expr2 \rangle]]^L$

---

#### CONTEXT:

$Type(\langle expr1 \rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid Time$

$Type(\langle expr2 \rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid Time$

---

#### CODE:

- DEVS-Suite:
 
$$[[\langle expr1 \rangle]]_J < [[\langle expr2 \rangle]]_J$$
  - PowerDEVS:
 
$$[[\langle expr1 \rangle]]_c < [[\langle expr2 \rangle]]_c$$
- 

#### LESS Union and number

$[[\langle expr1 \rangle < \langle expr2 \rangle]]^L$

---

#### CONTEXT:

$Type(\langle expr1 \rangle) = Type_1 \cup \dots \cup Type_n$

$Type(\langle expr2 \rangle) = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid Time$

$Type_i = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid Time$

$\langle expr1 \rangle.type = "[Type_i]^N"$

---

#### CODE:

- DEVS-Suite:
 
$$[[\langle expr1 \rangle]]_J.v\_ [Type_i]^N < [[\langle expr2 \rangle]]_J$$
  - PowerDEVS:
 
$$[[\langle expr1 \rangle]]_c.v\_ [Type_i]^N < [[\langle expr2 \rangle]]_c$$
- 

#### COMMENT:

The Union type is already normalized

It must be checked before if  $\langle var \rangle$  is currently a number, e.g.  $\langle var \rangle \in \mathbb{R}$ .

---

#### LESS Numbers in Unions

$[[\langle expr1 \rangle < \langle expr2 \rangle]]^L$

---

#### CONTEXT:

$Type(\langle expr1 \rangle) = Type_1 \cup \dots \cup Type_n$

$Type(\langle expr2 \rangle) = Type_{n+1} \cup \dots \cup Type_m$

$Type_i = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid Time$

$\langle expr1 \rangle.type = "[Type_i]^N"$

$Type_j = \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid Time$

$\langle expr2 \rangle.type = "[Type_j]^N"$

---

#### CODE:

- DEVS-Suite:

```

((⟦⟨expr1⟩⟧J).type=="⟦Typei⟧N")
  && ((⟦⟨expr2⟩⟧J).type=="⟦Typej⟧N")
  && (⟦⟨⟨expr1⟩⟩.v_⟦Typei⟧N < ⟨⟨expr2⟩⟩.v_⟦Typej⟧N⟧E)

```

• PowerDEVS:

```

((⟦⟨expr1⟩⟧C).type=="⟦Typei⟧N")
  && ((⟦⟨expr2⟩⟧C).type=="⟦Typej⟧N")
  && (⟦⟨⟨expr1⟩⟩.v_⟦Typei⟧N < ⟨⟨expr2⟩⟩.v_⟦Typej⟧N⟧E)

```

**COMMENT:**

The Union type is already normalized

It must be checked before if  $\langle expr1 \rangle$  and  $\langle expr2 \rangle$  are currently numbers, e.g.  $\langle expr1 \rangle \in \mathbb{R}$ .

### 3.10.4 Comparison “less or equal” ( $\leq$ )

$\llbracket \langle expr1 \rangle \leq \langle expr2 \rangle \rrbracket^L$

These are the same rules than Section 3.10.3 changing  $<$  for  $\leq$  in the translated code.

### 3.10.5 Comparison “greater” ( $>$ )

$\llbracket \langle expr1 \rangle > \langle expr2 \rangle \rrbracket^L$

These are the same rules than Section 3.10.3 changing  $<$  for  $>$  in the translated code.

### 3.10.6 Comparison “greater or equal” ( $\geq$ )

$\llbracket \langle expr1 \rangle \geq \langle expr2 \rangle \rrbracket^L$

These are the same rules than Section 3.10.3 changing  $<$  for  $\geq$  in the translated code.

### 3.10.7 Belongs

**BELONGS 1**

$\llbracket \langle exprA \rangle \in \langle exprB \rangle \rrbracket^B$

**CONTEXT:**

$\text{Type}(\langle exprB \rangle) = \mathbb{P} \text{Type}(\langle exprA \rangle) = \mathbb{P} \text{Type}$

**CODE:**

Case  $\langle exprB \rangle$  of:

- $\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle$

– DEVS-Suite:

$(\llbracket \langle exprB \rangle \rrbracket_J).contains(\llbracket \langle exprA \rangle \rrbracket_J)$

– PowerDEVS:

$findInSet<\text{Type}>(\llbracket \langle exprB \rangle \rrbracket_C, \llbracket \langle exprA \rangle \rrbracket_C)$

- $\langle set \rangle = \{\langle expr \rangle_1, \dots, \langle expr \rangle_n\}$

– DEVS-Suite:

$(buildSet(\llbracket \langle expr \rangle_1 \rrbracket_J, \dots, \llbracket \langle expr \rangle_n \rrbracket_J)).contains(\llbracket \langle exprA \rangle \rrbracket_J)$

– PowerDEVS:

$findInSet<\text{Type}>(buildSet<\text{Type}>(n, \llbracket \langle expr \rangle_1 \rrbracket_C, \dots, \llbracket \langle expr \rangle_n \rrbracket_C), \llbracket \langle exprA \rangle \rrbracket_C)$

- $\mathbb{N}$

– DEVS-Suite:

$isNat(\llbracket \langle exprA \rangle \rrbracket_J)$

– PowerDEVS:

$isNat(\llbracket \langle exprA \rangle \rrbracket_C)$

- $\mathbb{Z}$

– DEVS-Suite:

$isInt(\llbracket \langle exprA \rangle \rrbracket_J)$

- PowerDEVS:  
isInt( $\llbracket \langle exprA \rangle \rrbracket_c$ )
- $\mathbb{R}$ 
  - DEVS-Suite:  
isReal( $\llbracket \langle exprA \rangle \rrbracket_j$ )
  - PowerDEVS:  
isReal( $\llbracket \langle exprA \rangle \rrbracket_c$ )

## BELONGS 2

$\llbracket \langle exprA \rangle \in \langle exprB \rangle \rrbracket^B$

## CONTEXT:

$Type(\langle exprA \rangle) = Type_1 \cup \dots \cup Type_n$

$Type(\langle exprB \rangle) = \mathbb{P} Type_i, Type_i \in \{Type_1, \dots, Type_n\}$

## CODE:

- DEVS-Suite:  
( $\llbracket \langle exprA \rangle \rrbracket_j.type == \llbracket Type_i \rrbracket^N$ )  
&&  $\llbracket \langle exprA \rangle \rrbracket_j.v_{\llbracket Type_i \rrbracket^N} \in \langle exprB \rangle \rrbracket^B$
- PowerDEVS:  
( $\llbracket \langle exprA \rangle \rrbracket_c.type == \llbracket Type_i \rrbracket^N$ )  
&&  $\llbracket \langle exprA \rangle \rrbracket_c.v_{\llbracket Type_i \rrbracket^N} \in \langle exprB \rangle \rrbracket^B$

## 3.10.8 Proper subset

## PROPER SUBSET

$\llbracket \langle exprA \rangle \subset \langle exprB \rangle \rrbracket^{PS}$

## CONTEXT:

$Type(\langle exprB \rangle) = Type(\langle exprA \rangle) = \mathbb{P} Type$

## CODE:

Case  $\langle exprA \rangle, \langle exprB \rangle$  of:

- ( $\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle$ ), ( $\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle$ )
  - DEVS-Suite:  
isProperSubset( $\llbracket \langle exprA \rangle \rrbracket_j, \llbracket \langle exprB \rangle \rrbracket_j$ )
  - PowerDEVS:  
isProperSubset( $\llbracket \langle exprA \rangle \rrbracket_c, \llbracket \langle exprB \rangle \rrbracket_c$ )
- ( $\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle$ ),  $\langle set \rangle = \{expr_1, \dots, expr_n\}$ 
  - DEVS-Suite:  
isProperSubset( $\llbracket \langle exprA \rangle \rrbracket_j, (buildSet(\llbracket expr_1 \rrbracket_j, \dots \llbracket expr_n \rrbracket_j))$ )
  - PowerDEVS:  
isProperSubset( $\llbracket \langle exprA \rangle \rrbracket_c, (buildSet<\llbracket Type \rrbracket_c>(n, \llbracket expr_1 \rrbracket_c, \dots \llbracket expr_n \rrbracket_c))$ )
- $\langle set \rangle = \{expr_1, \dots, expr_n\}$ , ( $\langle var \rangle \mid \langle defFun \rangle \mid \langle operation \rangle$ )
  - DEVS-Suite:  
isProperSubset( $buildSet(\llbracket expr_1 \rrbracket_j, \dots \llbracket expr_n \rrbracket_j), \llbracket \langle exprA \rangle \rrbracket_j$ )
  - PowerDEVS:  
isProperSubset( $buildSet<\llbracket Type \rrbracket_c>(n, \llbracket expr_1 \rrbracket_c, \dots \llbracket expr_n \rrbracket_c), \llbracket \langle exprA \rangle \rrbracket_c$ )
- $\langle set \rangle = \{expr_1^A, \dots, expr_n^A\}$ ,  $\langle set \rangle = \{expr_1^B, \dots, expr_m^B\}$ 
  - DEVS-Suite:



```

isProperSubset(buildSet([ $\llbracket \text{expr}_1^A \rrbracket_J, \dots \llbracket \text{expr}_n^A \rrbracket_J$ ],
                        buildSet([ $\llbracket \text{expr}_1^B \rrbracket_J, \dots \llbracket \text{expr}_m^B \rrbracket_J$ ]))
– PowerDEVS:
isProperSubset(buildSet<[Type]c>(n, [ $\llbracket \text{expr}_1^A \rrbracket_c, \dots \llbracket \text{expr}_n^A \rrbracket_c$ ],
                        buildSet<[Type]c>(n, [ $\llbracket \text{expr}_1^B \rrbracket_c, \dots \llbracket \text{expr}_m^B \rrbracket_c$ ]))

```

---

### 3.10.9 Subset

---

#### SUBSET

$\llbracket \langle \text{expr}A \rangle \subseteq \langle \text{expr}B \rangle \rrbracket^{\text{PS}}$

---

#### CONTEXT:

$\text{Type}(\langle \text{expr}B \rangle) = \text{Type}(\langle \text{expr}A \rangle) = \mathbb{P} \text{ Type}$

---

#### CODE:

Case  $\langle \text{expr}A \rangle, \langle \text{expr}B \rangle$  of:

- $(\langle \text{var} \rangle \mid \langle \text{defFun} \rangle \mid \langle \text{operation} \rangle), (\langle \text{var} \rangle \mid \langle \text{defFun} \rangle \mid \langle \text{operation} \rangle)$ 
    - DEVS-Suite:
 

```
isSubset([ $\llbracket \langle \text{expr}A \rangle \rrbracket_J, \llbracket \langle \text{expr}B \rangle \rrbracket_J$ ])
```
    - PowerDEVS:
 

```
isSubset([ $\llbracket \langle \text{expr}A \rangle \rrbracket_c, \llbracket \langle \text{expr}B \rangle \rrbracket_c$ ])
```
  - $(\langle \text{var} \rangle \mid \langle \text{defFun} \rangle \mid \langle \text{operation} \rangle), \langle \text{set} \rangle = \{\text{expr}_1, \dots, \text{expr}_n\}$ 
    - DEVS-Suite:
 

```
isSubset([ $\llbracket \langle \text{expr}A \rangle \rrbracket_J, \text{buildSet}([ $\llbracket \text{expr}_1 \rrbracket_J, \dots \llbracket \text{expr}_n \rrbracket_J$ ])$ 
```
    - PowerDEVS:
 

```
isSubset([ $\llbracket \langle \text{expr}A \rangle \rrbracket_c, \text{buildSet}<[Type]_c>(n, [ $\llbracket \text{expr}_1 \rrbracket_c, \dots \llbracket \text{expr}_n \rrbracket_c$ ])$ 
```
  - $\langle \text{set} \rangle = \{\text{expr}_1, \dots, \text{expr}_n\}, (\langle \text{var} \rangle \mid \langle \text{defFun} \rangle \mid \langle \text{operation} \rangle)$ 
    - DEVS-Suite:
 

```
isSubset(buildSet([ $\llbracket \text{expr}_1 \rrbracket_J, \dots \llbracket \text{expr}_n \rrbracket_J$ ]), [ $\llbracket \langle \text{expr}A \rangle \rrbracket_J$ ])
```
    - PowerDEVS:
 

```
isSubset(buildSet<[Type]c>(n, [ $\llbracket \text{expr}_1 \rrbracket_c, \dots \llbracket \text{expr}_n \rrbracket_c$ ]), [ $\llbracket \langle \text{expr}A \rangle \rrbracket_c$ ])
```
  - $\langle \text{set} \rangle = \{\text{expr}_1^A, \dots, \text{expr}_n^A\}, \langle \text{set} \rangle = \{\text{expr}_1^B, \dots, \text{expr}_m^B\}$ 
    - DEVS-Suite:
 

```
isSubset(buildSet([ $\llbracket \text{expr}_1^A \rrbracket_J, \dots \llbracket \text{expr}_n^A \rrbracket_J$ ],
                    buildSet([ $\llbracket \text{expr}_1^B \rrbracket_J, \dots \llbracket \text{expr}_m^B \rrbracket_J$ ]))
```
    - PowerDEVS:
 

```
isSubset(buildSet<[Type]c>(n, [ $\llbracket \text{expr}_1^A \rrbracket_c, \dots \llbracket \text{expr}_n^A \rrbracket_c$ ],
                    buildSet<[Type]c>(n, [ $\llbracket \text{expr}_1^B \rrbracket_c, \dots \llbracket \text{expr}_m^B \rrbracket_c$ ]))
```
- 

## 4 Sentences “Where”

---

```

 $\langle \text{whereblock} \rangle ::= \text{defwhere}$ 
     $\langle \text{sentence} \rangle$ 
    { $\langle \text{sentence} \rangle$ }
    where
    [ $\langle \text{definition} \rangle$ ]
    [ $\langle \text{synonyms} \rangle$ ]
    { $\langle \text{sentence} \rangle$ }
    end defwhere

```

---

---

**Where**

```
[[defwhere
  <sentencesA>
  where
    <definitions>
    <synonyms>
    <sentencesB>
  end defwhere ]]
```

---

**CODE:**

- DEVS-Suite:  
[[<definitions>]]<sub>J</sub>  
[[<synonyms>]]<sub>J</sub>  
[[<sentencesB>]]<sub>J</sub>  
[[<sentencesA>]]<sub>J</sub>
  - PowerDEVS:  
[[<definitions>]]<sub>C</sub>  
[[<synonyms>]]<sub>C</sub>  
[[<sentencesB>]]<sub>C</sub>  
[[<sentencesA>]]<sub>C</sub>
- 

## 5 Used-defined functions

---

```
<functions> ::= functions <id> is
  <function>
  { <function> }
end functions

<function> ::= function <id> is
  <id>: <type> {, <id>: <type> } → <id>: <Type>;
  [ <definitions> ]
  [ <synonyms> ]
  <sentence>;
  { <sentence>; }
end function
```

---

---

**Function**

[[<function>]]<sup>F</sup>

---

**CONTEXT:**

```
[<id>] = [funId]
<id>: <type> {, <id>: <type> } → <id>: <Type> = var1: Type1, ..., varn: Typen → retVal: funType
```

---

**CODE:**

- DEVS-Suite:  
public funType funId(Type1 var1, ..., Typen varn){  
 [[retVal: funType]]<sup>P</sup>  
 [[<definitions>]]<sup>D</sup> /\*If any\*/  
 [[<sentence>]]<sup>S</sup>  
 :  
 [[<sentence>]]<sup>S</sup>  
 return retVal;  
}
- PowerDEVS:
  - In the header file (.h file):  
funType modeName::funId(Type1, ..., Typen);

- In the source code file (.cpp file):

```
funType modeName::funId(Type1 var1, ..., Typen varn){
    [[retVal: funType]]D
    [[⟨definitions⟩]]D /*If any*/
    [[⟨sentence⟩]]S
    :
    [[⟨sentence⟩]]S
    return retVal;
}
```

---

**COMMENT:**

If the type of the function has not been defined, define it.

---

## 6 Java code and C++ code

### 6.1 Java Code (Expr)

---

**JAVA CODE Expr**

$\llbracket \langle expr \rangle \rrbracket_J$

---

**CONTEXT:**

$TypeName(\langle expr \rangle) = TypeName$

---

**CODE:**

*Case*  $\langle expr \rangle$  *of:*

- $\langle id \rangle = \text{"Name"}:$

Name

- $\langle idComp \rangle = \langle id \rangle "." \langle digit \rangle \{ "." \langle digit \rangle \}:$

$\llbracket \langle id \rangle \rrbracket_J . v \llbracket \langle digit \rangle \rrbracket_J . \dots . v \llbracket \langle digit \rangle \rrbracket_J$

- $\langle digit \rangle = \text{"7"}:$

7

- $\langle val \rangle = \text{"value"}:$

*Case*  $Type(\langle val \rangle)$  *of:*

- $\mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Bool}:$

value

- Text:

"value"

- Time:

\* value =  $\infty$   
INFINITY

\* otherwise:  
value

- $\langle enum \rangle = \text{value}:$

"value"

- $\langle Type \rangle \cup \dots \cup \langle Type \rangle:$

new TypeName( $\llbracket \text{value} \rrbracket_C$ )

- $\sigma:$

sigma

- port:

port

- value:

value

- $\langle vals \rangle = (\text{expr}_1, \dots, \text{expr}_n):$

TypeName( $\llbracket \text{expr}_1 \rrbracket_J, \dots, \llbracket \text{expr}_n \rrbracket_J$ )

- $\langle set \rangle = \{expr_1, \dots, expr_n\} \wedge \text{Type}(\langle set \rangle) = \mathbb{P} \text{ Type}$ :  
`buildSet( $\llbracket expr_1 \rrbracket_J, \dots, \llbracket expr_n \rrbracket_J$ )`
- $\langle list \rangle = \langle expr_1, \dots, expr_n \rangle \wedge \text{Type}(\langle list \rangle) = \text{List Type}$ :  
`buildList( $\llbracket expr_1 \rrbracket_J, \dots, \llbracket expr_n \rrbracket_J$ )`
- $\langle operation \rangle = \langle unOp \rangle \langle operand \rangle$   
*Case  $\langle unOp \rangle$  of:*
  - “-”:  
`-( $\llbracket \langle operand \rangle \rrbracket_J$ )`
  - “rev”:  
`listRev( $\llbracket \langle operand \rangle \rrbracket_J$ )`
  - “head”:  
`( $\llbracket \langle operand \rangle \rrbracket_J$ ).get(0)`
  - “last”:  
`( $\llbracket \langle operand \rangle \rrbracket_J$ ).get(( $\llbracket \langle operand \rangle \rrbracket_J$ ).size()-1)`
  - “front”:  
`( $\llbracket \langle operand \rangle \rrbracket_J$ ).subList(0, ( $\llbracket \langle operand \rangle \rrbracket_J$ ).size()-2)`
  - “tail”:  
`( $\llbracket \langle operand \rangle \rrbracket_J$ ).subList(1, ( $\llbracket \langle operand \rangle \rrbracket_J$ ).size()-1)`
  - “#”:  
`( $\llbracket \langle operand \rangle \rrbracket_J$ ).size()`
  - “max”:  
`Collections.max( $\llbracket \langle operand \rangle \rrbracket_J$ )`
  - “min”:  
`Collections.min( $\llbracket \langle operand \rangle \rrbracket_J$ )`
  - “dom”:  
`( $\llbracket \langle operand \rangle \rrbracket_J$ ).keySet()`
  - “ran”:  
`( $\llbracket \langle operand \rangle \rrbracket_J$ ).values()`
- $\langle operation \rangle = \langle operand1 \rangle \langle binOp \rangle \langle operand2 \rangle$   
*Case  $\langle binOp \rangle$  of:*
  - “+”:  
`( $\llbracket \langle operand1 \rangle \rrbracket_J$ ) + ( $\llbracket \langle operand2 \rangle \rrbracket_J$ )`
  - “-”:  
`( $\llbracket \langle operand1 \rangle \rrbracket_J$ ) - ( $\llbracket \langle operand2 \rangle \rrbracket_J$ )`
  - “\*”:  
`( $\llbracket \langle operand1 \rangle \rrbracket_J$ ) * ( $\llbracket \langle operand2 \rangle \rrbracket_J$ )`
  - “\”:  
*Case  $\text{Type}(\langle operand1 \rangle)$  of:*
    - \*  $\langle \mathbb{R} \rangle \mid \langle \mathbb{Z} \rangle \mid \langle \mathbb{N} \rangle$ :  
`( $\llbracket \langle operand1 \rangle \rrbracket_J$ ) \ ( $\llbracket \langle operand2 \rangle \rrbracket_J$ )`
    - \*  $\mathbb{P} \text{ Type}$ :  
`setDiff( $\llbracket \langle operand1 \rangle \rrbracket_J, \llbracket \langle operand2 \rangle \rrbracket_J$ )`
  - “ $\cap$ ”:  
`listCat( $\llbracket \langle operand1 \rangle \rrbracket_J, \llbracket \langle operand2 \rangle \rrbracket_J$ )`
  - “ $\cap$ ”:  
`setInter( $\llbracket \langle operand1 \rangle \rrbracket_J, \llbracket \langle operand2 \rangle \rrbracket_J$ )`
  - “ $\cup$ ”:  
`setUnion( $\llbracket \langle operand1 \rangle \rrbracket_J, \llbracket \langle operand2 \rangle \rrbracket_J$ )`



- *sing*:  
    `singnum( $\llbracket \langle parameter \rangle \rrbracket_J$ )`
  - $\langle defFun \rangle = funName(param1, \dots, paramn)$ :  
    `funName( $\llbracket param1 \rrbracket_J, \dots, \llbracket paramn \rrbracket_J$ )`
  - $\langle Type \rangle$ :  
    *Case*  $\langle Type \rangle$  *of*:
    - $\mathbb{N}$ :  
    `Integer`
    - $\mathbb{Z}$ :  
    `Integer`
    - $\mathbb{R} \mid \text{Time}$ :  
    `Double`
    - `Text` | `Enum`:  
    `String`
    - `Boolean`:  
    `Boolean`
    - $\langle synonym \rangle = synName$ :  
    `T_synName`
- 

## 6.2 C++ code (Expr)

---

### C++ CODE Expr

$\llbracket \langle expr \rangle \rrbracket_c$

---

#### CONTEXT:

`TypeName( $\langle expr \rangle$ )=TypeName`

---

#### CODE:

- Case*  $\langle expr \rangle$  *of*:
- $\langle id \rangle = \text{"Name"}$ :  
    `Name`
  - $\langle idComp \rangle = \langle id \rangle \text{"."} \langle digit \rangle \{ \text{"."} \langle digit \rangle \}$ :  
     `$\llbracket \langle id \rangle \rrbracket_c.v \llbracket \langle digit \rangle \rrbracket_c \dots v \llbracket \langle digit \rangle \rrbracket_c$`
  - $\langle digit \rangle = \text{"7"}$ :  
    `7`
  - $\langle val \rangle = \text{"value"}$ :  
    *Case* *Type*( $\langle val \rangle$ ) *of*:
    - $\mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Bool}$ :  
    `value`
    - `Text`:  
    `"value"`
    - `Time`:
      - \* `value =  $\infty$`   
    `INFINITY`
      - \* *otherwise*:  
    `value`
    - $\langle enum \rangle$ :  
    `"value"`
    - $\langle Type \rangle \cup \dots \cup \langle Type \rangle$ :  
    `TypeName( $\llbracket value \rrbracket_c$ )`
  - $\sigma$ :  
    `sigma`
  - `port`:  
    `port`

- value:  
value
- $\langle vals \rangle = (\text{expr}_1, \dots, \text{expr}_n)$ :  
TypeName( $\llbracket \text{expr}_1 \rrbracket_c, \dots, \llbracket \text{expr}_n \rrbracket_c$ )
- $\langle set \rangle = \{\text{expr}_1, \dots, \text{expr}_n\} \wedge \text{Type}(\langle set \rangle) = \mathbb{P} \text{ Type}$ :  
buildSet<Type>(n,  $\llbracket \text{expr}_1 \rrbracket_c, \dots, \llbracket \text{expr}_n \rrbracket_c$ )
- $\langle list \rangle = (\text{expr}_1, \dots, \text{expr}_n) \wedge \text{Type}(\langle list \rangle) = \text{List Type}$ :  
buildList<Type>(n,  $\llbracket \text{expr}_1 \rrbracket_l, \dots, \llbracket \text{expr}_n \rrbracket_l$ )
- $\langle operation \rangle = \langle unOp \rangle \langle operand \rangle$   
Case  $\langle unOp \rangle$  of:
  - “-”:  
- ( $\llbracket \langle operand \rangle \rrbracket_c$ )
  - “rev”:  
listRev( $\llbracket \langle operand \rangle \rrbracket_c$ )
  - “head”:  
( $\llbracket \langle operand \rangle \rrbracket_c$ ).front()
  - “last”:  
( $\llbracket \langle operand \rangle \rrbracket_c$ ).back()
  - “front”:  
listFront( $\llbracket \langle operand \rangle \rrbracket_c$ )
  - “tail”:  
listTail( $\llbracket \langle operand \rangle \rrbracket_c$ )
  - “#”:  
( $\llbracket \langle operand \rangle \rrbracket_c$ ).size()
  - “max”:  
Case Type( $\langle operand \rangle$ ) of:
    - \*  $\mathbb{P} \text{ Type}$ :  
setMax( $\llbracket \langle operand \rangle \rrbracket_c$ );
    - \* List Type:  
listMax( $\llbracket \langle operand \rangle \rrbracket_c$ );
  - “min”:  
Case Type( $\langle operand \rangle$ ) of:
    - \*  $\mathbb{P} \text{ Type}$ :  
setMin( $\llbracket \langle operand \rangle \rrbracket_c$ );
    - \* List Type:  
listMin( $\llbracket \langle operand \rangle \rrbracket_c$ );
  - “dom”:  
getKeys( $\llbracket \langle operand \rangle \rrbracket_c$ );
  - “ran”:  
getValues( $\llbracket \langle operand \rangle \rrbracket_c$ );
- $\langle operation \rangle = \langle operand1 \rangle \langle binOp \rangle \langle operand2 \rangle$   
Case  $\langle binOp \rangle$  of:
  - “+”:  
( $\llbracket \langle operand1 \rangle \rrbracket_c$ ) + ( $\llbracket \langle operand2 \rangle \rrbracket_c$ )
  - “-”:  
( $\llbracket \langle operand1 \rangle \rrbracket_c$ ) - ( $\llbracket \langle operand2 \rangle \rrbracket_c$ )
  - “\*”:  
( $\llbracket \langle operand1 \rangle \rrbracket_c$ ) \* ( $\llbracket \langle operand2 \rangle \rrbracket_c$ )
  - “\”:  
Case Type( $\langle operand1 \rangle$ ) of:





```

    cos( $\llbracket \langle parameter \rangle \rrbracket_c$ )
  - tan:
    tan( $\llbracket \langle parameter \rangle \rrbracket_c$ )
  - arcsin:
    asin( $\llbracket \langle parameter \rangle \rrbracket_c$ )
  - arccos:
    acos( $\llbracket \langle parameter \rangle \rrbracket_c$ )
  - arctan:
    atan( $\llbracket \langle parameter \rangle \rrbracket_c$ )
  - log:
    log( $\llbracket \langle parameter \rangle \rrbracket_c$ )
  - sing:
    ((( $\llbracket \langle parameter \rangle \rrbracket_c$ ) == 0.0) ? 0.0 : ((( $\llbracket \langle parameter \rangle \rrbracket_c$ ) < 0.0) ? -1.0 : 1.0))

•  $\langle defFun \rangle = funName(param1, \dots, paramn)$ :
  funName( $\llbracket param1 \rrbracket_c, \dots, \llbracket paramn \rrbracket_c$ )

•  $\langle Type \rangle$ :
  Case  $\langle Type \rangle$  of:
    -  $\mathbb{N}$ :
      unsigned int
    -  $\mathbb{Z}$ :
      int
    -  $\mathbb{R} \mid \text{Time}$ :
      double
    - Text | Enum:
      string
    - Boolean:
      bool
    -  $\langle synonym \rangle = synName$ :
      T_synName

```

---

## 6.3 Names

---

### Names

$\llbracket \langle Type \rangle \rrbracket^N$

---

#### CODE:

```

Case  $\langle Type \rangle$  of:
•  $\mathbb{N} \mid \mathbb{Z} \mid \mathbb{R} \mid \text{Time}$ :
  Number
• Text:
  Text
• Boolean:
  Boolean
• Enum:
  Enum
•  $\langle synonym \rangle = synName$ :
  synName

```

---

## 7 Auxiliary functions

---

### AUXILIARY FUNCTIONS

---

#### CODE:

- **setUnion:**

- DEVS-Suite:

```
public static <T> Set<T> setUnion(Set<T> setA, Set<T> setB){
    Set<T> union = new HashSet<T>(setA);
    union.addAll(setB);
    return union;
}
```

- PowerDEVS:

```
template <class T>
std::set<T> setUnion(std::set<T> setA,
                    std::set<T> setB){
    std::set<T> res;
    std::set_union(setA.begin(), setA.end(), setB.begin(),
                  setB.end(), inserter(res, res.begin()));
    return res;
}
```

- **setInter:**

- DEVS-Suite:

```
public static <T> Set<T> setInter(Set<T> setA, Set<T> setB){
    Set<T> inter = new HashSet<T>(setA);
    inter.retainAll(setB);
    return inter;
}
```

- PowerDEVS:

```
template <class T>
std::set<T> setInter(std::set<T> setA,
                    std::set<T> setB){
    std::set<T> res;
    std::set_intersection(setA.begin(),
                          setA.end(),
                          setB.begin(),
                          setB.end(),
                          inserter(res, res.begin()));
    return res;
}
```

- **setDiff:**

- DEVS-Suite:

```
public static <T> Set<T> setDiff(Set<T> setA, Set<T> setB){
    Set<T> diff = new HashSet<T>(setA);
    diff.removeAll(setB);
    return diff;
}
```

- PowerDEVS:

```
template <class T>
std::set<T> setDiff(std::set<T> setA,
                   std::set<T> setB){
    std::set<T> res;
    std::set_difference(setA.begin(), setA.end(),
                       setB.begin(), setB.end(),
                       inserter(res, res.begin()));
    return res;
}
```

- **listCat:**

- DEVS-Suite:

```
public static <T> List<T> listCat(List<T> listA, List<T> listB){
    List<T> cat = new ArrayList<T>(listA);
    cat.addAll(listB);
    return cat;
}
```

- PowerDEVS:

```
template <class T>
std::list<T> listCat(std::list<T> listA,
                   std::list<T> listB){
    std::list<T> res = listA;
    res.insert(res.end(), listB.begin(), listB.end());
    return res;
}
```

- **listFront:**

- PowerDEVS:

```

template <class T>
std::list<T> listFront(std::list<T> list){
    std::list<T> res=list;
    res.erase(--res.end());
    return res;
}

```

- **listTail:**

- PowerDEVS:

```

template <class T>
std::list<T> listTail(std::list<T> list){
    std::list<T> res=list;
    res.erase(res.begin());
    return res;
}

```

- **listRev:**

- DEVS-Suite:

```

public static <T> List<T> listRev(List<T> list){
    List<T> rev= new ArrayList<T>(list);
    Collections.reverse(rev);
    return rev;
}

```

- PowerDEVS:

```

template <class T>
std::list<T> listRev(std::list<T> list){
    std::list<T> res=list;
    res.reverse();
    return res;
}

```

- **buildSet:**

- DEVS-Suite:

```

public static <T> Set<T> buildSet(T ...elements){
    Set<T> set = new HashSet<T>(Arrays.asList(elements));
    return set;
}

```

- PowerDEVS:

```

template <class T>
std::set<T> buildSet(int n, ...){
    std::set<T> res;
    va_list vl;
    va_start(vl,n);
    for (int i=0; i<n; i++){
        T val=(T)(va_arg(vl,T));
        res.insert(val);
    }
    va_end(vl);
    return res;
}

```

- **buildList:**

- DEVS-Suite:

```

public static <T> List<T> buildList(T ...elements){
    List<T> list=new ArrayList<T>(Arrays.asList(elements));
    return list;
}

```

- PowerDEVS:

```

template <class T>
std::list<T> buildList(int n, ...){
    std::list<T> res;
    T val;
    va_list vl;
    va_start(vl,n);
    for (int i=0; i<n; i++){
        val=va_arg(vl,T);
        res.push_back(val);
    }
    va_end(vl);
    return res;
}

```

- **buildMap:**

- DEVS-Suite:

```

public static <T1,T2> Map<T1,T2> buildMap(Pair<T1,T2> ...pairs){
    Map<T1,T2> map = new HashMap<T1,T2>();
    for (int i=0; i < pairs.length; i++){
        map.put(pairs[i].f(), pairs[i].s());
    }
    return map;
}

```

- PowerDEVS:

```

template <class F, class S>
std::map<F,S> buildMap(int n, ...){
    std::map<F,S> res;
    typedef Pair<F,S> P;
    P p;
    va_list vl;
    va_start(vl,n);
    for (int i=0; i<n; i++){
        p=va_arg(vl,P);
        std::pair<F,S> pair(p.f,p.s);
        res.insert(pair);
    }
    va_end(vl);
    return res;
}

```

- **oPlus:**

- PowerDEVS:

```

template <typename F, typename S>
std::map<F,S> oplus(std::map<F,S> m1, std::map<F,S> m2){
    std::map<F,S> res = m1;
    typename std::map<F,S>::iterator it;
    for (it=m2.begin(); it!=m2.end(); it++){
        res[it->first]=it->second;
    }
    return res;
}

```

- **dRes:**

- DEVS-Suite:

```

public static <T1,T2> Map<T1,T2> dRes(Set<T1> s, Map<T1,T2> m){
    Map<T1,T2> map = new HashMap<T1,T2>(m);
    map.keySet().retainAll(s);
    return map;
}

```

- PowerDEVS:

```

template <typename T1, typename T2>
std::map<T1,T2> dRes(std::set<T1> s, std::map<T1,T2> m){
    std::map<T1,T2> res;
    typename std::map<T1,T2>::iterator it;
    for (it=m.begin(); it!=m.end(); it++){
        if (s.find((it->first)) != s.end()){
            res[it->first]=it->second;
        }
    }
    return res;
}

```

- **rRes:**

- DEVS-Suite:

```

public static <T1,T2> Map<T1,T2> rRes(Map<T1,T2> m, Set<T2> s){
    Map<T1,T2> map = new HashMap<T1,T2>(m);
    map.values().retainAll(s);
    return map;
}

```

- PowerDEVS:

```

template <typename T1, typename T2>
std::map<T1,T2> rRes(std::map<T1,T2> m, std::set<T2> s){
    std::map<T1,T2> res;
    typename std::map<T1,T2>::iterator it;
    for (it=m.begin(); it!=m.end(); it++){
        if (s.find((it->second)) != s.end()){
            res[it->first]=it->second;
        }
    }
    return res;
}

```

- **isProperSubset:**

- DEVS-Suite:

```

public static <T> Boolean isProperSubset(Set<T> setA,
                                       Set<T> setB){
    return setB.containsAll(setA) &&
           (setA).size()<(setB).size();
}

```

- PowerDEVS:

```

template <class T>
bool isProperSubset(std::set<T> setA,
                   std::set<T> setB){
    return std::includes(setB.begin(), setB.end(),
                        setA.begin(), setA.end())
           && setA.size()<setB.size();
}

```

- **isSubset:**

- DEVS-Suite:

```

public static <T> Boolean isSubset(Set<T> setA, Set<T> setB){
    return setB.containsAll(setA);
}

```

– PowerDEVS:

```
template <class T>
bool isSubset(std::set<T> setA,
             std::set<T> setB){
    return std::includes(setB.begin(), setB.end(),
                        setA.begin(), setA.end());
}
```

• isNat:

– DEVS-Suite:

```
public static Boolean isNat(Object var){
    if (var instanceof Integer)
        if ((Integer)var >=0) return true;
        else return false;
    else return false;
}
```

– PowerDEVS:

```
bool isNat(int var){
    return (var>0);
}
template <typename T> bool isNat(T var){
    return false;
}
```

• isInt:

– DEVS-Suite:

```
public static Boolean isInt(Object var){
    if (var instanceof Integer) return true;
    else return false;
}
```

– PowerDEVS:

```
template <typename T> bool isInt(T var){
    return (typeid(var)==typeid(int));
}
```

• isReal:

– DEVS-Suite:

```
public static Boolean isReal(Object var){
    if ((var instanceof Double)
        || (var instanceof Integer)) return true;
    else return false;
}
```

– PowerDEVS:

```
template <typename T> bool isReal(T var){
    return (typeid(var)==typeid(double)
        || typeid(var)==typeid(int));
}
```

• findInSet:

– PowerDEVS:

```
bool findInSet(std::set<const char*> s, const char* x){
    for (std::set<const char*>::iterator it = s.begin(); it != s.end(); it++){
        const char* elem = *it;
        if (strcmp(elem,x)==0) return true;
    }
    return false;
}
```

• toInteger:

– DEVS-Suite:

```
public static Integer toInteger(Object var){
    Integer res=null;
    if (var instanceof Integer){
        res=((Integer) var).intValue();
    }
    else if (var instanceof Double){
        res=((Double) var).intValue();
    }
    return res;
}
```

• toDouble:

– DEVS-Suite:

```
public static Double toDouble(Object var){
    Double res=null;
    if (var instanceof Integer){
        res=((Integer) var).doubleValue();
    }
    else if (var instanceof Double){
        res=((Double) var).doubleValue();
    }
    return res;
}
```

- **setMin:**

- PowerDEVS:

```
template <class T> T setMin(std::set<T> s){
    return *std::min_element(s.begin(), s.end());
}
```

- **setMax:**

- PowerDEVS:

```
template <class T> T setMax(std::set<T> s){
    return *std::min_element(s.begin(), s.end());
}
```

- **listMin:**

- PowerDEVS:

```
template <class T> T listMin(std::set<T> l){
    return *std::min_element(l.begin(), l.end());
}
```

- **listMax:**

- PowerDEVS:

```
template <class T> T listMax(std::list<T> l){
    return *std::min_element(l.begin(), l.end());
}
```

- **class Pair:**

- PowerDEVS:

```
template <typename F, typename S> class Pair{
public:
    F f;
    S s;
    Pair<F,S>(){};
    Pair<F,S>(F f, S s){
        this->f=f;
        this->s=s;
    }
    Pair<F,S>& operator=(const Pair<F,S>& other){
        this->f=other.f;
        this->s=other.s;
        return *this;
    }
    bool operator<(const Pair<F,S>& other) const{
        return !(*this==other);
    }
    bool operator==(const Pair<F,S>& other) const{
        return ((this->f) == (other.f)) && ((this->s) == (other.s));
    }
};
```

---

## 8 Post-processing

Once the model has been translated, both in DEVS-Java and in PowerDEVS, a post-processing must be performed where each variable involved in the state of the model is replaced, inside the transition function definitions ( $\delta_{int}$  and  $\delta_{ext}$ ), in each occurrence of it on the right side of an assignment or in a comparison. For this purpose it is used the copy of this variables made before. (`modelName prev=modelName(this);` -in the case of Java-).

For instance, suppose that `varX` is part of the model state, and it appear on the right side of an assignment:

```
varY=varX+7;
```

it is replaced by:

```
varY=prev.varX+7;
```

and, instead, it appear within a comparison (on either side):

```
(varX >= varY)
```

it is also replaced:

```
(prev.varX >= varY)
```

This replaced is due to, in the transition functions, these variables may modify its values but, when referring to these variables in the abstract model, the reference is about the value of them in the previous state of the transition.

Suppose that in a model the state has the following representation:

$$S = \mathbb{N} \times \mathbb{R} \times \text{Time}$$

and the internal transition function,  $\delta_{int}$ , is as follows:

$$\delta_{int}((n, r, \sigma)) = (n + 1, r * n, \sigma + n)$$

In this case, both in  $r * n$  and in  $\sigma + n$  must be consider the value of  $n$  before performing such transition, i.e. before  $n$  assumes the value  $n + 1$ . Let us see how it would be the Java code of the model after the translations. The state definition could be:

```
Integer n;  
Double r;  
Double t;
```

and the internal transition function:

```
public void deltint(){  
    n = n+1;  
    r = r*n;  
    sigma = sigma+n;  
}
```

However, this clearly will not behave as expresses the model (once the function ends, the value of  $\sigma$  will be  $\sigma + (n + 1)$  instead of  $\sigma + n$ , and a similar case occurs with  $r$ ). Therefore, the occurrences of the state variables on right side of the assignments are replaced by the copy made before:

```
public void deltint(){  
    modelName prev=new modelName(this);  
    n = prev.n+1;  
    r = prev.r*prev.n;  
    sigma = prev.sigma+prev.n;  
}
```

Note that not all replacement are necessarily to preserved the behavior of the model. However, to avoid having to determinate which variables change its values before being used on an assignment or in a comparison, all of them are replaced without affecting the complexity of the model.