ERTMSFormalSpec Workbench
Technical design

Version 0.9.0

1 Table of contents

2 Revision history

Version	Date	Name	Description	Paragraphs
0.2.1	13/12/2010	Laurent Ferier	First version	All
0.3.1	31/01/2011	Laurent Ferier	Updates according to WP2 release 1	All
0.4.1	04/02/2011	Laurent Ferier	Updates according to WP2 release 2	All
0.5.1	05/04/2011	Laurent Ferier	Updates according to WP2 release 3	All
0.6.1	13/05/2011	Laurent Ferier	Updates according to WP2	All
0.7.1	19/10/2012	Svitlana Lukicheva, Laurent Ferier	Updates according to the release of the version 0.7	All
0.9.0	12/02/2013	Svitlana Lukicheva	Added missing predefined function and keywords	4.3.3.1, 5.1.1.9, 5.1.4

3 Introduction

This document holds technical data about the ERTMSFormalSpecs Workbench (EFSW). This document is decomposed into two parts

- Section 4 presents the underlying XML structure used to model the EFSW logic
- Section 5 presents how this model should be interpreted by a computer software, be it a test engine (as in the ESFW) or a real EVC target.

3.1 References

Document	Title	Date

3.2 Terms and definitions

Term	Definition
ESFW	ERTMSFormalSpec Workbench
EFS	ERTMSFormalSpecs

4 Exported model

The exported model is an XML file which respects the XML schema named `DataDictionary.xsd` provided with the distributed environment. The following sections present the elements defined in that schema.[1]

4.1 Element Dictionary

The dictionary is the main element of the exported model. This dictionary is composed of five elements

- The *specification* used to model the system (see Section 4.2)
- The ERTMS *namespaces* defined (e.g. Kernel, DMI, TIU, …) presented in Section 4.3
- The *tests* presented in Section 4.4
- The *test translations* presented in Section 4.5
- The *shortcuts* presented in Section 4.6

4.2 Specifications

The specification related to the model holds the following information

- **Name**: the name which identifies the specification
- **Chapters**: a list of chapters
- **version**: the version of the baseline

Each chapter holds a sequence of paragraphs, which hold the following information

- **Name**: the name which identifies the paragraph in the specification
- **Comment**: a comment related to the paragraph
- **ReqRefs**: references to requirements related to that paragraphs
- **Messages**: a message describing the content of variables described in the paragraph, if any
- **Subs**: list of sub-paragraphs
- **TypeSpecs**: specification of special or reserved values described by the paragraph, if any
- **reviewed**: indicates if the paragraph has been reviewed by the requirement analyst
- **infoRequired**: indicates if the implementation of this paragraph requires some additional information to be completed
- **specIssue**: indicates if there is an issue in the specification (incoherence, paragraph incomplete, …)
- **status**: the implementation status of the paragraph (N/A, NotImplementable, Implemented, NewRevisionAvailable)
- **bl**: the baseline of the paragraph
- **id**: the id of the paragraph, corresponding to the id from the Subset-026
- **name**: the name of the paragraph, needed for the backward compatibility
- **optional**: needed for the backward compatibility

[1] Note that the EFSWorkbench is built using classes generated by XMLBooster (http://www.xmlbooster.com) using an equivalent meta definition, available in the EFS distribution.

- **scope**: scope of the paragraph (OBU, TRACK or OBU_AND_TRACK)
- **type**: type of the paragraph (TITLE, DEFINITION, REQUIREMENT, NOTE, DELETED, PROBLEM or TABLE_HEADER)
- **version**: version of the baseline

4.3 Namespaces

The model is split into several namespaces which can hold sub namespaces, ranges, enumerations, structures, collections, functions, procedures, variables and rules, which shall be further described below. Namespaces are identified by a name.

4.3.1 Information common to all model elements

The elements in the model reference requirements for which they are created. They hold the following information

- **ReqRefs**: the requirement paragraph ids
- **Comment**: comments associated to this relation
- **Implemented**: the implementation status
- **Verified**: the verification status
- **NeedsRequirement**: indicates if this elements needs to be attached to a requirement

4.3.2 Types

The following section presents types that can be specified inside a namespace

4.3.2.1 Range type

Ranges allow to specify that a value is integral and has a minimum and a maximum value. They hold the following information

- **Name**: the name which identifies the range in the document
- **Default**: the default value to use when variables of this type are instantiated
- **MinValue**: the minimum value
- **MaxValue**: the maximum value
- **Precision**: the precision (integer or floating point)
- **SpecialValues**: the special values associated to a range; each special value provides a meaningful name to a specific value of the range

4.3.2.2 Enumeration type

An enumeration allows to define the possible (literal) values a variable can take. They hold the following information

- **Name**: the name which identifies this enumeration in the document
- **Default**: the default value to use when variables of this type are instantiated
- **EnumValues**: the values associated to this enumeration. Each one of these values is identified by a name and a value.
- **SubEnums**: enumerations that share the enclosing enumeration type, but whose range is shorter than the enclosing's

4.3.2.3 Structures

Structures allow to structure variables using a C-like struct mechanism. They hold the following information

- **Name**: the name which identifies this structure in the document
- **StructureElements**: elements composing this structure
- **Rules**: lists the rules attached to this structure, which will be applied to all instances of the latter; the rules are described in the Section □
- **StructureProcedures**: lists the procedures related to this structure, which will be executed on each instance of the latter; the procedures are described in the Section 4.3.3.1

The structure elements hold the following information

- **Name**: the name of the field identified by this structure element
- **Default**: the default value of the element
- **TypeName**: the name of the type of this structure element; it can be either a range, an enumeration, a structure, …
- **Mode**: the mode of the structure element, the same as the mode of variables described in the Section 0

4.3.2.4 Collections

A collection allows to define a variable which can hold a set of values, all these values must be of the collection's type. They hold the following information

- **Name**: the name which identifies this collection in the document
- **Default**: the default value to be used when instances of that collection are created
- **TypeName**: the name of the type of the elements stored in the collection
- **MaxSize**: the maximum size of the collection

4.3.3 Procedures and Variables

The system state is represented in the model using two artifacts

- **Variables**: which can be of any type defined in Section 0
- **Procedures**: whose value is the current state in the state machine they define

4.3.3.1 Procedures

Procedures allow to define a specific process, which can be split in several Kernel activations. These procedures are modeled using state machines. A procedure holds the following information

- **Name**: the name which identifies the procedure in the document
- **StateMachine**: the state machine associated to this procedure; it identifies the possible states in which this procedure may be
- **Rules**: lists the rules to be applied when invoking the procedure
- **Parameters**: lists the formal parameters of this procedure; during invocation, the actual values of the parameters are associated to each formal parameter

A procedure parameter is a simple association between a name and a type. It holds the following information

- **Name**: the name which identifies this parameter in the procedure
- **Type**: the type of the parameter

State machines allow to define the possible states in which a procedure can be. A state machine holds the following information

- **Name** : the name of the state machine
- **InitialState**: the initial state of the state machine
- **States**: the set of states defined for this state machine

Each state can be further decomposed into another state machine. A state holds the following information

- **Name**: the name which identifies this state in the document
- **StateMachines**: the set of sub state machines
- **X, Y, Width, Height**: rendering information

The special variable **CurrentState**, associated to each state machine, indicates which is the current state of the state machine. For example, to test if the state machine SM is in the state A, the following expression should be used: `SM.CurrentState in A`.

4.3.3.2 Variables

Variables are used to keep track of the state of a sub system. A variable holds the following information

- **Name**: the name which identifies this variable in the document
- **DefaultValue**: the default value to use when instantiating this variable; it overrides the type default value
- **Type**: the type of the variable
- **Mode**: the mode of the variable which can be either
  - **Incoming**: variable which is used to provide information to the enclosing system
  - **Outgoing**: output variable, provides information to the external world
  - **In Out**: the variable can be modified by the external world and provides information to the enclosing system
  - **Internal**: internal variable
  - **Constant**: variable that can be assigned once, then only read
- **SubVariables**: the set of sub variables (in case the variable's type is struct)

4.3.4 Functions

Functions are used to compute a value, according to the function's parameters. Each function holds the following information

- **Name**: the name which identifies this function in the document
- **Type**: the type of the function
- **Parameters**: define the formal function parameters
- **Cases**: define the values the function can take, depending on conditions over the parameters

A function parameter is a simple association between a name and a type. It holds the following information

- **Name**: the name which identifies this parameter in the function
- **Type**: the type of the parameter

The value of a function is expressed in term of the case. A case of a function associates a set of pre-conditions, based on the parameter's value, and an expression which provides the function value in case of the corresponding pre-conditions are satisfied, as defined in Section 4.3.5.1. A case holds the following information

- **Name**: the name of the case
- **PreConditions** : the set of pre-conditions
- **Expression**: the expression of the case

4.3.5 Rules

Rules allow to define the behavior of the system. A rule can hold the following information

- **Name**: the name of the rule, which describes the rule behavior.
- **Priority:** which defines when the rule should be activated
  - **Verification:** the rule is applied during the verification phase
  - **UpdateINTERNAL:** the rule is applied to update internal values, to prepare the processing phase
  - **Processing:** this phase is the main processing phase
  - **UpdateOUT:** during this phase, the OUT variables are updated
- **Conditions:** lists the mutually exclusive cases for this rule: when executing a rule, at most one of these conditions is executed; when two conditions could be executed, the first one takes precedence

Each RuleCondition is composed of the following items

- **PreConditions**: allow to determine when the rule must be activated (see 4.3.5.1)
- **Actions**: define the actions to take when the rule condition is activated (see 4.3.5.2)
- **SubRules**: lists the sub-rules of the rule condition

4.3.5.1 PreConditions

Pre-conditions are evaluated to check if a rule must be activated. The pre-condition body holds the conditional expression, which should evaluate to a Boolean, see 5.1.1 for more information about available expressions.

4.3.5.2 Actions

Actions are system state modifications which are applied when the rule's pre-conditions are satisfied. Actions body holds the statement to be performed, see 5.1.2 for more information about available statements.

4.4 Tests

4.4.1 Sub sequence

Tests are decomposed into sub sequences. A sub sequence holds the following information

- **Name**: the name of the sub sequence
- **TestCases**: lists the test cases that compose this sub sequence
- **Level**: needed for the backward compatibility with the sub sequences of the Subset-076
- **Mode**: needed for the backward compatibility with the sub sequences of the Subset-076
- **D_LRBG**: needed for the backward compatibility with the sub sequences of the Subset-076
- **NID_LRBG**: needed for the backward compatibility with the sub sequences of the Subset-076
- **Q_DIRLRBG**: needed for the backward compatibility with the sub sequences of the Subset-076
- **Q_DIRTRAIN**: needed for the backward compatibility with the sub sequences of the Subset-076
- **Q_DLRBG**: needed for the backward compatibility with the sub sequences of the Subset-076
- **RBCPhone**: needed for the backward compatibility with the sub sequences of the Subset-076
- **RBC_Id**: needed for the backward compatibility with the sub sequences of the Subset-076

4.4.2 Test case

Test cases hold the following information

- **Name**: the name of the test case
- **Steps**: lists the steps to be executed for this test case
- **Case**: needed for the backward compatibility with the sub sequences of the Subset-076
- **Feature**: needed for the backward compatibility with the sub sequences of the Subset-076
- **ReqRefs**: the requirement paragraph ids
- **Implemented**: the implementation status
- **Verified**: the verification status

- **NeedsRequirement**: indicates if this elements needs to be attached to a requirement

4.4.3 Test step

A test step consists of a specific step to be taken during a test. It contains

- **Name**: the name of the test step
- **SubSteps**: lists the sub steps to be executed for this step
- **IO**: needed for the backward compatibility with the sub sequences of the Subset-076
- **LevelIN**: needed for the backward compatibility with the sub sequences of the Subset-076
- **LevelOUT**: needed for the backward compatibility with the sub sequences of the Subset-076
- **ModeIN**: needed for the backward compatibility with the sub sequences of the Subset-076
- **ModeOUT**: needed for the backward compatibility with the sub sequences of the Subset-076
- **TranslationRequired**: indicates if this step needs to be translated

4.4.4 Sub step

A sub step holds the following information

- **Name**: the name of the sub step
- **Actions**: the set of variable modifications to be taken at the beginning of this sub step (see 4.3.5.2)
- **Expectations**: the set of conditions that must be satisfied to consider this step as successful (see 4.4.4.1).

4.4.4.1 Expectation

An expectation describes the expected state of a variable to be verified at least once before the end of a test step. An expectation holds the following information

- **Variable**: no more used.
- **Blocking**: indicates that the next sub step cannot be executed until this expectation has not be satisfied. When this flag is set to true, the test scenario is stopped until the expectation is reached. Otherwise, the test scenario continues with this expectation pending.
- **DeadLine**: the time before which this expectation should be satisfied
- The expectation PCData holds the expression which describes should be evaluated to true.

4.5 Translations

Translations define pattern matching rules and are used to automatically create actions and expectations for sub steps imported from Subset-076.

4.6 Shortcuts dictionary

The shortcuts allow to access directly to model elements. The shortcut dictionary contains the following information

- **Name**: the name of the dictionary
- **Folders**: the set of folders containing each one a set of shortcuts; the folders are identified by their name
- **Shortcuts**: the set of shortcuts; a shortcut is identified by its name and contains the path to the corresponding model element

5 Interpretation model

This section presents the interpretation model of the data dictionary.

5.1.1 Expression evaluation

Expressions follow the following grammar[2][3]

```
Expression_i ::=  Expression_i+1 { Op_i Expression_i }?      for i=0..5
Expression_7 ::=  DerefExpression
                | FunctionCall
                | ListExpression
                | UnaryExpression
```

Where the following table presents the operators by level

op_0	*OR*
op_1	*AND*
op_2	*== | != | <= | >= | not in | in*
op_3	*+ | -*
op_4	** | /*
op_5	*^*

We define the value (the expression's semantics) of the expression $\sigma($`Expression`$)$ by

$$\sigma(\texttt{Expression}_{i+1}\ op_i\ \texttt{Expression}_i) = \sigma_{op_i}(\sigma(\texttt{Expression}_{i+1}), \sigma(\texttt{Expression}_i))$$

This simply states that, to compute the value of an expression, one must first evaluate the left part of the expression, then the right part of the expression and combine those two values using the operator.

The semantics of each operator depends on the type it is applied to.

---

[2] Note that this grammar is subject to further evolutions (for instance, to allow more arithmetic operators).

[3] The notation used is the an extended BNF grammar (see for instance http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form) where :

- **::=** defines a new grammar rule
- ***** stands for 0, 1 or more
- **+** stands for 1 or more
- braces **{ }** are used to group elements together
- **[]** is used to indicate that one of the enclosed element should be chosen (for instance, [0..9] means either 0, 1, 2, … or 9
- ^ is a negation. For instance, [^0..9] means everything but a number.
- **…** is used for lists, and represents 0, 1 or more elements, separated by the list separator
- As a shortcut, literal are presented in green, non terminals are in blue, whereas grammar operators are displayed in **bold**

5.1.1.1 Operators for Integers, Ranges, EnumValues, States and Strings

For variables of type Integer, Range, EnumValue, State and String, the semantics of the operators is provided by the following table

Op	Semantics $\sigma_{op}(x, y)$ =	Constraints
OR	*x = true or y = true*	*x.type* = Boolean and *y.type = Boolean*
AND	*x = true and y = true*	*x.type* = Boolean and *y.type = Boolean*
==	*x = y*	*x.type = y.type and x.type is an integer type*
	$\lvert x - y \rvert < \varepsilon$	*x.type = y.type and x.type is a float type. In this case,* $\varepsilon = 0.000000001$
!=	**not** $\sigma_{==}(x,y)$	*x.type = y.type*
IN	N/A	N/A
NOT IN	N/A	N/A
+	*x + y*	*x.type = y.type* and (*x.type* is Integer or *x.type = Range)*
-	*x – y*	*x.type = y.type* and (*x.type* is Integer or *x.type = Range)*
*	*x * y*	*x.type = y.type* and (*x.type* is Integer or *x.type = Range)*
/	*x / y*	*x.type = y.type* and (*x.type* is Integer or *x.type = Range)* and $y \neq 0$
^	x^y	*x.type* is Integer *and y.type* is Integer

This table states that the common semantics for equality and inequality are used to interpret the `==` and `!=` operators. The operators `in` and `not in` are not available for Ranges, EnumValues, Procedures and Stings.

5.1.1.2 Operators for collections

For variables of type Collection, the semantics of the operators is provided by the following table

Op	Semantics $\sigma_{op}(x, y)$ =	Constraints
OR	N/A	N/A
AND	N/A	N/A
==	*x*.Size = *y*.Size and $\forall i : \sigma_{==}(x[i], y[i])$	*x.type* is Collection and *y.type* is Collection
!=	**not** $\sigma_{==}(x,y)$	*x.type* is Collection and *y.type* is Collection
IN	$\exists\ i : \sigma_{==}(x , y[i])$	*y.type* is Collection
NOT IN	**not** $\sigma_{in}(x, y)$	*y.type* is Collection
+	N/A	N/A
-	N/A	N/A
*	N/A	N/A
/	N/A	N/A
^	N/A	N/A

This table states two lists are equal when they hold the same number of elements and each element of one of them can be found at the same location in the second list. The `in` operator is a simple "belongs to" operation between an element and a list.

5.1.1.3 Unary expression and Term

A term can either be a designator, function call, a string literal, an integer or a list[4].

```
UnaryExpression ::= NOT ( Expression )
                  | ( Expression )
                  | StructureExpression
                  | Term


Term ::= Designator
       | String
       | Integer
       | Double
       | List
```

The semantics of parenthesed expressions is straightforward, the next sections will focus on structure expressions and terms.

[4] Note that while procedure calls are available in EFS, these are not considered as unary expression (or term), since a procedure call does not return a value.

5.1.1.4 Structure expression

Structure expression defines an instance of a structure and allows to set the value or that instance's members. It follows the grammar

```
StructureExpression ::= Expression { Designator  => Expression,…}
```

The first expression of the structure expression must reference the structure (defined in the model) to be instantiated. The expressions enclosed within the braces define the value to be set to specific sub elements of the instance.

Op	Semantics $\sigma_{op}(x, y)$ =	Constraints
expr{…}	Allocates a new instance of the structure identified by σ(expr). All field values are set to their default value.	σ(expr) references a structure defined in the model
Id => expr	Assigns σ(expr) to the field value referenced by *inst*.Id, where *inst* is the instance currently being built	

5.1.1.5 Designator

A designator is used to identify elements of the model. It is a sequence of letters and digits as presented in the following rule

```
Designator ::=  Letter { Letter | Digit | '_' }*
```

Finding the model element referenced by a designator depends on its position in the expression:

- The first element of a sequence of designators separated by the "." operator is searched for at the following locations
  - The parameters of the enclosing function or procedure (the evaluation context stack)
  - The variable bound by an enclosing list operator expression
  - The predefined elements defined in the system
  - One of the enclosing scope in which the expression is declared
  - The default namespace of the EFS system
- The following elements of the sequence reference sub elements of the previous dereference expression. For instance, in the expression a.b.c, the designator c references the sub element named c in the model element designated by a.b

5.1.1.6 DerefExpression

A DerefExpression is used as a scoping mechanism to identify types, variables, functions, procedures, state machines or literals. It follows the rule

```
DerefExpression ::=  Expression { . Expression }*
```

This is a sequence of expressions, separated by dots. Consider

$$DerefExpression = Expression_0 \ . \ Expression_1 \ . \ \dots \ . \ Expression_n$$

The semantics of such deref expression is computed piecewise.

5.1.1.7 Function call

The function call follows the grammar

```
FunctionCall ::= Designator ( Expression, … )
```

Evaluating the value of a function call is performed the following way

1. find the function which corresponds to the designator (as presented in Section 5.1.1.5) and create a new evaluation context stack.
2. for each parameter expression, evaluate the expressions and bind a new variable with the value to the corresponding function parameter in the last entry of the evaluation context stack.
3. find the first case preconditions (see Section 4.3.5.1) which evaluates to true to determine which expression should be used to evaluate the function call
4. the function call is evaluated to the value of the corresponding case expression.
5. Remove the last entry from the evaluation context stack.

5.1.1.8 Literal values

Literal values are either string or integer values. They follow the following grammar.

```
String ::=  ' [^']* '
Integer ::= 0 [0..9]*
```

The semantics of literal values is the corresponding value.

5.1.1.9 Lists

List values are expressed by enclosing a list of terms (separated by comma) with brackets.

```
List ::= [ Term, … ]
```

The semantics of a list is the list value which holds the semantics of the terms.

5.1.1.10 List Expression

A list expression follows the grammar

```
ListExpression ::=
      | THERE_IS_IN ListValue
      | FORALL_IN ListValue
      | FIRST_IN ListValue
      | LAST_IN ListValue
      | COUNT ListValue
      | REDUCE ListValue USING Expression INITIAL_VALUE Expression
      | SUM ListValue USING Expression
      | MAP ListValue USING Expression

ListValue ::=  Expression [ | Condition ]?
Condition ::=  Expression
```

Where the list value provides a value of type collection, where elements from the Expression (which also evaluates to a list expression) have been filtered out by the condition (if present).

The CONDITION and USING expressions can use the bound variable named X which references the list element currently being processed. For instance, the expression

```
THERE_IS_IN List | X > 3
```

will be true if there is at least an element in the list which value is above 3 (see below for the operator semantics)

Moreover, the USING expression can use the bound variable named RESULT which is the partial result actually build when considering the current list element. At the start of the evaluation, RESULT holds

- the INITIAL_VALUE for REDUCE statements,
- an empty list for MAP statements

For instance, the expression

```
REDUCE List
        USING X.Size + RESULT
        INITIAL_VALUE 0
```

would sum the size attribute stored in elements in the list. This expression can be written more concisely using the following expression

```
SUM List USING X.Size
```

The following table summarizes the list operators and their associated semantics. There operators can only be applied under the following conditions

- ListValue.Type is Collection.
- Condition expression, when defined, evaluate to a Boolean

Moreover, INITIAL_VALUE and USING expressions must evaluate to the same type for the REDUCE statement.

ListOp	σ_{ListOp} (*List, cond*) =
THERE_IS_IN	$\exists$ i : σ_{cond}(*List*[i])
FORALL_IN	$\forall$ i : σ_{cond}(*List*[i])
FIRST_IN	*List*[i] where $\forall$ j<i : σ_{cond}(*List*[j]) = *False* and σ_{cond}(*List*[i])
LAST_IN	*List*[i] where $\forall$ j>i : σ_{cond}(*List*[j]) = *False* and σ_{cond}(*List*[i])
COUNT	Counts the number of elements in the list

REDUCE	Reduces the collection to a single value, by applying the expression defined in the USING clause on each element of the collection, where X is the value of the current collection element and RESULT the current reduced result (when only part of the list has been processed). The initial value clause is used to initiate the value of RESULT.
	For instance, the sum of all elements of a collection is written
	`REDUCE List`
	`  USING X + RESULT`
	`  INITIAL_VALUE 0`
	If a condition is expressed, the source list is first limited to the element matching the condition before applying the USING clause. For instance, the following expression
	`REDUCE List | X.Enabled`
	`  USING X.Size + RESULT`
	`  INITIAL_VALUE 0`
	sums the size of all elements where the enabled flag is set to true.

SUM	Perform a sum over the elements of a list, by applying the USING expression to compute the value of a single element. For instance, the sum of all elements of a list is written `SUM List USING X`

MAP	Creates a new list from the list referenced by the ListValue, by applying the expression in the USING clause on each element on this list. For instance, the expression
	`MAP List USING X.Size`
	creates a new collection holding the size attribute of all elements of the original collection.
	The MAP expression can also be used to limit a collection to the elements which match a given criteria. For instance, the following expression
	`MAP List | X.Enabled USING X`
	provides a new collection containing only the elements for which the Enabled attribute is true.

5.1.2 Statements

Statements change the state of the system. They follow the grammar

```
Statement ::=
      Expression0 <- Expression1
      Expression0 ( Expression , … )
      ApplyStatement

StatementList ::=
      { Statement ; }+
```

The first form of statement is a simple variable assignment, whereas the second one is a procedure call.[5] A statement list consists of a list of statement to be executed in sequence.

5.1.2.1 Variable assignation

Assigning a new value for a variable changes the state of the system. After this statement has been executed, the new value of the variable identified by $\sigma(expression_0)$ is $\sigma(expression_1)$.

5.1.2.2 Procedure calls

Executing a procedure call is performed the following way

1. find the procedure which corresponds to the $expression_0$ (as presented in Section 5.1.1.5) and create a new evaluation context stack.
2. for each parameter expression, evaluate the expressions and bind a new variable with the value to the corresponding procedure parameter in the last entry of the evaluation context stack.
3. find the first case preconditions (see Section 4.3.5.1) which evaluates to true to determine which statement should be used to execute the procedure call
4. execute the corresponding statement list.

[5] Note that function calls cannot appear as a statement since in that case, their return value would be lost.

5. Remove the last entry of the evaluation context stack.

5.1.2.3 APPLY Statement

The apply statement is defined as following

```
Statement ::= APPLY Expression ON ListValue
```

It applies a procedure on each element of the list referenced by the ListValue. For instance, the expression

```
APPLY StoreInfo(X) ON Message.Packets | X.NID = 4
```

will call the procedure StoreInfo for each Packet of the variable Message whose NID is 4.

5.1.3 Instantiate a variable

When a variable is instantiated, a default value is provided to that variable. If the default value is defined at the variable declaration, the corresponding expression is evaluated and assigned as the variable's value. If no default expression is available, the default expression provided for the variable's type is used instead.

5.1.4 Predefined functions

This section presents the predefined functions available in the model.

5.1.4.1 Allocate function

This function responds to the following prototype

```
Allocate (Collection<T>): T
```

It finds an empty entry in the collection and returns the corresponding entry.

5.1.4.2 Available function

This function responds to the following prototype

```
Available (Collection<T>): Boolean
```

It returns true when there is an empty entry in the collection provided as parameter, and false otherwise.

5.1.4.3 Min

This function responds to the following prototype

```
Min (T, T): T
```

It provides the minimum value between two of them.

5.1.4.4 MinSurface

This function responds to the following prototype

```
MinSurface (f1, f2): f
```

It is an higher order function which takes two functions as parameters and provides the function which minimize both of them.

5.1.4.5 Max

This function responds to the following prototype

```
Max (T, T): T
```

It provides the maximum value between two of them.

5.1.4.6 Targets

This function responds to the following prototype

```
Targets (f1, f2, f3): Collection<Target>
```

This function computes the list of targets of step functions provided as parameter. A target is a structure with the following fields

- Speed: the associated speed (that is, f(x))
- Location: the x value where the function is discontinuous
- Length: the length of the current step

This structure is described as *Kernel.SpeedAndDistanceMonitoring.TargetSupervision.Target*.

The resulting type is a collection of such type. The resulting value combines all the targets for the functions provided as parameter.

5.1.4.7 RoundToMultiple

This function responds to the following prototype

```
RoundToMultiple (double, double): double
```

It rounds the first parameter to a multiple of the second parameters. This function is needed to implement the requirement 3.11.11.6 of the Subset-026.

For instance,

RoundToMultiple (21.4, 5.0) = 20.0

RoundToMultiple (0.28, 5.0) = 25.0

5.1.4.8 DecelerationProfile

This function responds to the following prototype

```
DecelerationProfile (SpeedRestriction, DecelerationFactor): f
```

It creates a new quadratic function which computes the deceleration profile according to a given speed restriction and a deceleration factor function, as presented on the Figure 38 of the Subset-026.

5.1.4.9 Before

This function responds to the following prototype

```
Before (ExpectedFirst, ExpectedSecond, Collection): Boolean
```

It indicates if the expected first entry is before the expected second entry in a collection.

5.1.4.10 AddIncrement

This function responds to the following prototype

```
AddIncrement (f, Increment): f
```

It is a higher order function which adds an increment to an existing function, as presented in Subset-026 Paragraph 3.13.9.2.2.

5.1.4.11 Override

This function responds to the following prototype

```
Override (Default, Override): f
```

It is a higher order function which overrides the values of a defaults function by the values of the override (partial) function and returns the corresponding overridden function.

For instance, consider the constant function

f1(x) = 1 for all x

and the partial function

f2(x) = 3 when x > 2 and x < 3

then, the function

Override(f1, f2)	= 3	when x > 2 and x < 3
	= 1	otherwise

5.1.4.12 DistanceForSpeed

This function responds to the following prototype

```
DistanceForSpeed (f, speed): distance
```

This function computes the distance where the function f will reach the speed provided as parameter.

5.1.4.13 IntersectAt

This function responds to the following prototype

```
IntersectAt (f1, f2): double
```

This function computes the intersection of a step function (f1) with a curve (f2). The functions f1 and f2 must have the following prototypes:

```
f1 (distance): speed
f2 (speed):    distance
```

This function is used for example to compute the braking to target Indication supervision limit, which is not used for estimated speeds higher than V_MRSP (see Paragraph 3.13.10.4.11 in Subset-026). It allows to determine the location of the Indication supervision limit valid for V_MSRP.

5.2 Rule evaluation

Rules are used in EFS to change the state of the system, when specific conditions have been encountered. Processing the system rules is performed in two phases:

1. The system evaluates the set of rules to be activated by considering each rule preconditions (see Section 5.2.1)
2. The state of the system is altered by applying all activated rule's statements. There is no guaranteed rule activation order, hence, two distinct rules are inconsistent when they can be activated at the same time and alter the same model element with different values.

Rule activation has been split into several phases to handle a complete execution cycle, as depicted by the Figure 1.

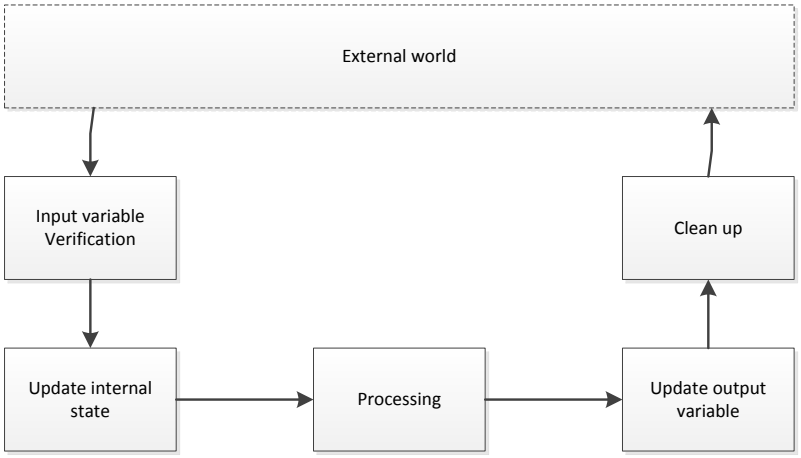

Figure 1 - Processing cycle

The phase "Input variable Verification" is used to ensure the consistency of the input variables and set the system in the right state according to these verifications.

The next phase updates the system state according to the input variables. When this phase is completed, processing can then be performed to handle the core business logic. The system response is performed in the "Update output variable" part of the process, before cleaning up all intermediate results (in the "Clean up" phase). Control is then given back to the enclosing system and the EFS interpreter is ready to processa new cycle.

5.2.1 Rules to be evaluated

Rules are defined at several places in the model. When activating the interpretation machine, the following rules are evaluated to determine the actions to apply.

1. All rules declared in a namespace. The instance on which the rule is evaluated is the namespace in which the rule is declared
2. For all variables declared in a namespace,
    a. If the variable is a structure,

i. All rules declared in that structure. The instance on which the rules are evaluated is the corresponding variable.

ii. Apply the same algorithm to all sub variables of the structure

b. If the variable is a procedure

i. All rules declared in the current state and its enclosing states. The instance on which the rules are evaluated is the procedure's state variable.

5.2.2 Evaluating the actions to apply

Evaluating a rule consists of determining the first rule condition which is activated.

A rule condition is activated when all its preconditions evaluate to true.

1. When a rule condition is activated, all its actions are selected to be applied, in a second step. The model element on which the expressions have been computed is kept, to be used in the action application phase.
2. All the sub rules of the rule condition are evaluated to be activated.

5.2.3 Applying the actions

When the selection phase completes, the statements corresponding to the selected actions are executed to update the model state. Remember that, since there is no guaranteed rule activation order, two distinct rules are inconsistent when they can be activated at the same time and alter the same model element with different values.

5.3 Test execution

5.3.1 Frame

Executing the frame consists of executing all sub sequences located in that frame. The frame holds an expression which evaluates to the business logic cycle time.

A frame is successful when all the sub sequences located in that frame are also successful.

5.3.2 Sub sequence

Executing a sub sequence consists of executing all test cases located in that sub sequence.

A sub sequence is successful when all the test cases located in that sub sequence are also successful.

5.3.3 Test case

Executing a test case consists of executing all steps located in that test case.

A test case is successful when all the steps located in that test case are also successful.

5.3.4 Step and sub-step

A **step** is composed of several **sub-steps**, each sub-step is composed of

- a sequence of actions. Applying the sub-step on the system consists of applying the action's statement in sequence to modify the system state.
- a sequence of expectation. A sub-step is considered completed when all its blocking expectations have been satisfied or are failed.

Executing a **step** consists of executing its sub-steps in sequence: a sub-step in the sequence can be executed as soon as its previous sub-step is completed.

Interpreting a **sub-step** is performed as follows

1. Apply all actions on the model and add all sub-step expectation to the list of encountered expectations, along with their deadline.
2. Activate the EFS Rule interpretation machine until all expectations are either satisfied or failed.
    1. An expectation is satisfied when its expectation expression is evaluated to true
    2. An expectation is failed when it has not been satisfied and its deadline is elapsed