

Safety Designer: AltaRica extended language

User's Guide Appendix



V5R20 – BPA SD9 Delivery 8

Table of Contents

Introduction	3
Objectives of this document	3
References	3
Syntax of AltaRica Extended language	4
Lexical conventions	4
Lexemes	4
Comments	4
Identifiers	4
Keywords	4
Constants	4
Strings	5
Structure of an AltaRica file	5
Path in hierarchy	5
Expressions	5
"if ... then ... else ..." and "(if ... then ...)" expressions	5
Atomic expressions	6
Unary operators	6
Multiplying Operators	7
Additiv operators	7
Relational operators	7
Equality operators	7
Logical AND operator	7
Logical OR operator	8
Domains statement	8
General points	8
Integer interval	8
Enumeration of symbolic constants	8
Structured domain	9
Declaration of functions or operators	10
Declaration of models	10
Declaration of flow variables	11
Declaration of state variables	11
Declarations of events	12
Declaration of sub-components	12
Definition des transitions	12
Definition of assertions	13
Definition of synchronizations	13
Definition of initialization	13
Definition of external directives	14
AltaRica Extended Grammar	18

Introduction

Objectives of this document

AltaRica Extended language is reference language of BPA DAS.

It comes from confluence works on different AltaRica dialects from 2000 to 2003 :

- AltaRica Dassault (reference language of Cécilia OCAS 2.7 Béta) [4]
- AltaRica DataFlow (reference language of Alta-X tools from ARBoost Technologie) [3]
- Original AltaRica and its extensions from LaBRI (reference language or LABRI tools) [1,2]

The objective of this language is to enables complex grammatical contructions, like common cause failures or structured flows in order to be compilable in elementary construction. Then, it's possible to add new functions without modifying computation core which can still use a language that doesn't handle this structure.

Two languages have been defined:

- A minimalist language that doesn't handle structured flows, operators, common cause failures synchronizations, and some specificities of AltaRica Dassault language:

This language relies essentially on current version of Altarica DataFlow language. The main difference with this language is syntax of assertions which is closer than the one of original AltiRaca language.

- AltaRica Extended mainly relies on functionalities expected of BPA DAS:

So, it has been built from AltaRica Dassault language and has been improved with AltaRica DataFlow notions (init and extern clause, parting component and equipment at language level).

In addition of original language functionalities, these two languages handle arithmetic operators (+, -, *, /) and "Integer" and "Real" domains (even if tools using minimal language don't take it into account).

A translator enables to translate from AltaRica Extended language to minimal language.

This document deals with grammar rules of AltaRica Extended language.

References

[1] : Projet Altarica Phase III - Le Langage Altarica - Manuel de Référence, par G. Point, 01/06/2000, réf. IXI-AltaRica/03/T01/DT00-V0

[2] : AltaRica – Manuel méthodologique, par A. Arnold, G. Point, A. Griffault, A. Rauzy, 05/10/2001

[3] : The AltaRica Data-Flow Language – Syntax, par A. Rauzy, 09/10/2002, réf. ARBoost Technologies/AltaRica/NT02-1, version 7

[4] : Atelier Cécilia OCAS version 2.7 Béta, couplé au moteur de calcul JMoteur version 4.59

Syntax of AltaRica Extended language

Lexical conventions

Lexemes

Lexemes of AltaRica are identifiers, keywords, integer or real constants, operators, and separators.

White-spaces, tabulations, end-line characters and comments are ignored unless they separate lexemes.

Comments

Two type of comments are allowed:

1. Characters `/*` mark the beginning of a multi-line comment which must ends with `*/`. Comments can't be imbricated.

```
/*
 * This is comment over many lines
 */
```

2. Characters `//` mark the beginning of a one-line comment. Comment ends with End-Of-Line or End-Of-File character.

```
// this is one-line comment
// This is another comment
```

Identifiers

An identifier is a sequence of letters, digits or `'_'`. A simple identifier always begin with a letter and can have an unlimited lenght. Uppercase and lowercase are differentiated.

Example : Engine, Engine_9, engine_9 (The last two are different identifiers).

```
<identifier> ::= '[a-z][a-zA-Z0-9_-]*'
```

Keywords

Following Identifiers are reserved keyword:

and	assert	bool	case	cnuf	const
domain	edon	else	event	extern	false
float	flow	func	if	imply	in
init	int	inverse	knil	link	local
max	min	node	not	or	out
private	state	sub	sync	term	then
trans	true				

Constants

There are different types of constants, Integer constants, Real constants, enumerated constants and boolean constants:

1. An integer constant is :
 - either 0 ;
 - either a succession of digits which doesn't begin with 0.

```
<integer> ::= '0' | '[1-9][0-9]*'
```

2. A real constant

```
<float> ::= '[0-9]*.[0-9]+([eE][+-]?[0-9]+)?'
```

3. A boolean constant in one of the two key-word: `true` or `false`.
4. An enumerated constant is an identifier declared beforehand in an enumeration domain (cf. the section called "Enumeration of symbolic constants").

Strings

A string is a sequence beginning and ending with double-quote `'"`.

```
<string> ::= "[^"]+"'
```

Structure of an AltaRica file

An altarica description is a set of declarations of constants, domains or component models. These declarations can't be imbricated, the range of objects so declared is global from the declaration point.

```
<altarica-description>
  ::= <global-declaration-list>
<global-declaration-list>
  ::= <global-declaration> ((';'?)? <global-declaration>)*
  ::=
<global-declaration>
  ::= <constant-declaration>
  ::= <domain-declaration>
  ::= <function-declaration>
  ::= <model-declaration>
```

Path in hierarchy

State variables, flow variables or events of a node can be specified by a path in hierarchy of AltaRica models. Such a path is a succession of identifier separate with points `(.)`.

```
<hierarchy-path> ::= <identifier> ('.' <hierarchy-path>)*
```

Depending on context, a path can be construed as either a variable identifier, or an event identifier, or an enumerated constant identifier, or a declared constant identifier (cf. 0). In this condition, such an identifier can not be used for a declared constant, a variable or an enumerated constant.

Expressions

This section deals with syntax of AltaRica expressions that can be found in assertions, guards or transitions of a node.

Expressions can be regrouped into different categories depending on the returned type: boolean (`<bool-expr>`), numerical (`<num-expr>`), symbolic (`<symb-expr>`) or structured (`<struct-expr>`). Expression type is not syntactically verified as made in AltaRica DataFlow language, but it's made after, during lexical verification of data.

AltaRica Extended syntax takes operators priority into account (like former version of AltaRica Dassault or AltaRica LaBRI).

```
<bool-expr>  ::= <expression>
<num-expr>   ::= <expression>
<symb-expr>  ::= <expression>
<struct-expr> ::= <expression>
```

"if ... then ... else ..." and "(if ... then ...)" expressions

Expressions *If-then*, without *Else* has been introduced in AltaRica Dassault. These expressions are equivalent to implication. Their syntaxes have been kept for compatibility reasons and user habitude. But they are translated in imply operator `=>` in low level language. Brackets must surround expressions.

Moreover, in former version of Altarica language, *If-Then-Else* expression was surrounded with brackets in order to facilitate understanding. Now, brackets are no more mandatory.

```
<expression>
  ::= <match-expr>
  ::= <unmatch-expr>
<match-expr>
  ::= <other-expr>
  ::= if <bool-expr> then <match-expr> else <match-expr>
<unmatch-expr>
  ::= '(' if <bool-expr> then <bool-expr> ')'
  ::= if <bool-expr> then <match-expr> else <unmatch-expr>
<other-expr>
  ::= <or-expression>
```

For an *If-Then-Else* expression, the first sub-expression must be boolean (predicate); it is called condition of operation. The two following sub-expressions are *Then* and *Else* of *If-Then-Else*.

Atomic expressions

Atomic expressions are either access path of variables or constants or expressions between brackets, or functions of the language (min, max, cardinality '#') or defined by user, or case operator of AltaRica DataFlow.

```
<atomic-expression>
  ::= <integer>
  ::= <float>
  ::= true
  ::= false
  ::= <id-variable>
  ::= '(' <expression> ')'
  ::= min '(' <num-expr> ',' <num-expr> '+' ')'
  ::= max '(' <num-expr> ',' <num-expr> '+' ')'
  ::= '#' '(' <bool-expr> ',' <bool-expr> '+' ')'
  ::= <identifier> '(' <expression> ',' <expression> '*' ')'
  ::= <identifier> '(' ')'
  ::= case '{'
      (<bool-expr> ':' <expression> ',' )*
      else <expression>
    '}'
<id-variable>
  ::= <hierarchy-path> '^' <identifier>
  ::= <hierarchy-path>
```

In order to access to a variable of a sub-node, point (.) must be used as separator of the path. In order to access to a field of a structured flow, circumflex accent (^) is used and followed by field name.

Unary operators

Unary operations are valuated from right to left.

```
<unary-expression>
  ::= - <unary-expression>
  ::= <operator-not> <unary-expression>
  ::= <atomic-expression>
<operator-not>
  ::= not
  ::= '~'
```

Operand of arithmetic negation - must be from arithmetic type (integer or real) and the result of this negation is the opposite of its operand.

Boolean negation is described thanks to the two following identifiers: not word or tilde (~). Operand of a boolean expression must be boolean.

Multiplying Operators

Multiplying operators are *, / and % are valued from left to right. Their operands must be arithmetic. * is multiplication, / is division and % is remainder of integer division (in this case, operands must be integer).

```
<multiplicative-expression>
::= <multiplicative-expression> '*' <unary-expression>
::= <multiplicative-expression> '/' <unary-expression>
::= <multiplicative-expression> '%' <unary-expression>
::= <unary-expression>
```

Additive operators

Additive operators + and - are valued from left to right. Operands of these operators must be arithmetic. + operator gives the sum of values of operands. + operator gives the values of the first operand minus the value of the second one.

```
<additive-expression>
::= <additive-expression> '+' <multiplicative-expression>
::= <additive-expression> '-' <multiplicative-expression>
::= <multiplicative-expression>
```

Relational operators

Relational operators are valued from left to right. Operands must be arithmetic because - in opposition to some programming language - neither enumerated constants nor boolean are assimilated with integer.

< (less than), > (greater than), <= (less than or equal to) et >= (greater than or equal to) give the value `false` if the specified comparison is not verified with the operands; otherwise the value of expression is `true`.

```
<relational-expression>
::= <relational-expression> '<' <additive-expression>
::= <relational-expression> '>' <additive-expression>
::= <relational-expression> '<=' <additive-expression>
::= <relational-expression> '>=' <additive-expression>
::= <additive-expression>
```

Equality operators

These operators are = (equal to), != or # (different from), `imply` or `=>` (imply).

Operands of = and != are boolean, arithmetic or enumerates. Operands of = operator can also be structured. The value of these expressions is `true` if the equality is verified by operands, and `false` otherwise. In boolean case, equality matches equivalence of operands, difference matches *Exclusiv or* (XOR)

Operands of `imply` and `=>` must be boolean. These operators match logical implication.

```
<equality-expression>
::= <relational-expression> '=' <relational-expression>
::= <relational-expression> <operator-neq> <relational-expression>
::= <relational-expression> <operator-imply> <relational-expression>
::= <relational-expression>
<operator-neq>
::= '!='
::= '#'
<operator-imply>
::= imply
::= '>='
```

Logical AND operator

Logical AND operator has got two identifiers : & and `and`. Its operands must be boolean. It is valued from left to right. The value is `true` if the two operands are `true`, `false` otherwise.

```
<and-expression>
    ::= <and-expression> <operator-and> <equality-expression>
    ::= <equality-expression>
<operator-and>
    ::= and
    ::= '&'
```

Logical OR operator

Logical OR operator has got two identifiers : | and or. Its operands must be boolean. It is valuated from left to right. The value is false if the two operands are false, true otherwise.

```
<or-expression>
    ::= <or-expression> <operator-or> <and-expression>
    ::= <and-expression>
<operator-or>
    ::= or
    ::= '|'
```

Domains statement

General points

Language has predefined domains: bool (for boolean variables), int (for integer variables), float (for real variables).

New domain can be created using following syntax:

```
<domain-declaration>
    ::= domain <identifier> = <domain-definition>
<domain-definition>
    ::= <link-domain>
    ::= <domain>
<domain>
    ::= <range-domain>
    ::= <enumeration-domain>
    ::= <predefined-domain>
    ::= <identifier>
<predefined-domain>
    ::= bool
    ::= int
    ::= float
```

An existing domain name can not be redefined. The declared domain can be used in the rest of description if a domain of value is required. Domain alias can be created associating a domain name to another domain name (predefined or declared).

Three domains construction are allowed: relativ integer interval, symbolic constant enumeration, and structured domain.

Integer interval

Integer interval are specified between square bracket ([,]), bounds are separate with comma. Bounds are integer. The value of left bound must be less than the value of right bound.

```
<range-domain>
    ::= '[' <integer> ',' <integer> ']'
```

Enumeration of symbolic constants

An enumeration is a list of identifiers which are between brace and separate with commas ({...}). These identifiers must not be associated with a global constant name.


```
<enumeration-domain>
::= '{' <identifier-list> '}'
```

Structured domain

Structured domain replace typedef of AltaRica Dassault language. They are essentially used to simplify entry of connection beam between components (bus, serial cable, informations transmission, ...). Only flow variables (in or out) can be declared with structured domain (private or state variables can't).

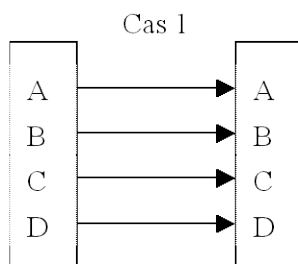
The flow clause allows to defined a set of structured domain fields. Each field is defined with an identifier and a domain (which can't be structured).

A structured domain defines two plugs: the in plug and the out plug. Each connection plug is composed of n fiels defined with flow clause. Usually, a field has the same orientation as its reference plug. The inverse clause defines plug fields having reversed orientation.

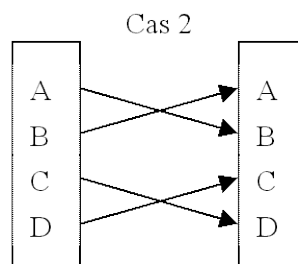
The assert clause defines equality relations between in plugin and out plug. In order to simplify understanding, equality relations are oriented and defined with assignment.

```
<link-domain>
::= link
    flow <link-flow-list> (';')?
    ( inverse <inverse-list> (';')? )?
    assert <connect-list> (';')?
    knil
<link-flow-list>
::= <link-flow-decl> (';' <link-flow-decl>)*
<link-flow-decl>
::= <identifier-list> ':' <domain>
<inverse-list>
::= <id-flow-link> (';' <id-flow-link>)*
<connect-list>
::= <connect-decl> (';' <connect-decl>)*
<connect-decl>
::= <id-flow-link> '==' <id-flow-link>
<id-flow-link>
::= in '^' <identifier>
::= out '^' <identifier>
```

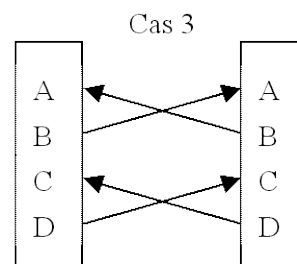
Examples :



```
domain cas1 = link
    flow A,B,C,D : int ;
    assert
        in^A := out^A;
        in^B := out^B;
        in^C := out^C;
        in^D := out^D;
    knil;
```



```
domain cas2 = link
    flow A,B,C,D : int ;
    assert
        in^A := out^B;
        in^B := out^A;
        in^C := out^D;
        in^D := out^C;
    knil;
```



```
domain cas3 = link
    flow A,B,C,D : int ;
    inverse out^A; out^C;
    in^B; in^D;
    assert
        in^A := out^B;
        out^A := in^B;
        in^C := out^D;
        out^C := in^D;
    knil;
```

Declaration of functions or operators

Operators (or functions : `func`) enable definition of output variable functions of variables in parameters.

Description of a function is close to description of a component (node). the difference is function doesn't describe a behavior. A function is a component model having an output flow, some input flows and assertions enabling definition of output flow functions of input flows.

```
<function-decl>
  ::= func <identifier>
    <function-elem>
    <function-body>
  cnuf
<function-elem>
  ::= <flow-cls> <function-elem>
  ::=
<function-body>
  ::= <assert-cls> <function-body>
  ::=
```

Domains of flow variables can be any domains, even structured domain. In this last case, structured flows having inverse clause are forbidden. Moreover, equality between structured flows takes into account crossing that are defined in `assert` clause of structured domain.

A function can be used only in expression of model assertions. It can not be used in guards of assignments. Moreover, recursive function can not be written.

Declaration of models

A model is declared using the above syntax. Declaration begins with `node` keyword, followed by the model name, a list of fields and ends with `edon`.

```
<model-declaration>
  ::= node <identifier>
    <node-elem>
    <node-body>
    <extern-1st>
  edon
```

The first part enables declaration of data manipulated in this component model, that is to say flow variables (`flow`), state variables (`state`), events (`event`) and nodes (`sub`).

```
<node-elem>
  ::= <node-elem-field> <node-elem>
  ::=
<node-elem-field>
  ::= <flow-cls>
  ::= <state-cls>
  ::= <event-cls>
  ::= <sub-cls>
```

The second part describes behavior and connections between data thanks to assertions (`assert`), transitions (`trans`), synchronizations (`sync`) and initialization (`init`).

```
<node-body>
  ::= <node-body-field> <node-body>
  ::=
<node-body-field>
  ::= <trans-cls>
  ::= <assert-cls>
  ::= <sync-cls>
  ::= <init-cls>
```

The last part (one or many `extern` clauses) enables addition of informations that are external to language. It's mainly used for tools.

```
<extern-1st>
    ::= <extern-cls> <extern-1st>
    ::=
```

The `sync` field is allowed only for hierarchical component model, that is to say if at least one sub-node has been declared in `sub` field.

`state` fields and `trans` fields are allowed only if model is not hierarchical.

Declaration of flow variables

Flow variables serve as interface with other components.

Declaration of flow variables must respect following syntax.

```
<flow-cls>
    ::= flow <flow-decl-list> (';')?
```

The `flow` keyword is followed by an unempty list of variables declaration separate with semicolon. Declarations list may end with semicolon.

```
<flow-decl-list>
    ::= <flow-decl> (';' <flow-decl>)*
```

Every declaration conforms to following syntax : A list of names of variables, a separator `:`, a value domain, a second separator `:` and the flow orientation (input flow : `in`, output flow : `out`, local flow or private component: `local` or `private`). Variables identifiers must not be followed by a declared constant, an enumerated constante or an already declared variable of the model (in the same or in another variable class).

```
<flow-decl>
    ::= <identifier-list> ':' <domain> ':' <orientation>
<orientation>
    ::= in
    ::= out
    ::= private
    ::= local
```

Declaration of state variables

State variables are used to imitate component state. Component changes its state by changing the value of one or many state variables thanks to transitions.

Declaration of state variables must respect following syntax.

```
<state-cls>
    ::= state <state-decl-list> (';')?
```

The `state` keyword is followed by an unempty list of variables declaration separate with semicolon. Declarations list may end with semicolon.

```
<state-decl-list>
    ::= <state-decl> (';' <state-decl>)*
```

Every declaration conforms to following syntax : A list of names of variables, a separator `:` and a value domain. Variables identifiers must not be followed by a declared constant, an enumerated constante or an already declared variable of the model (in the same or in another variable class).

```
<state-decl>
    ::= <identifier-list> ':' <domain>
```

Declarations of events

Declarations of events must respect following syntax.

```
<event-cls>
    ::= event <event-decl-list> (';')?
<event-decl-list>
    ::= <event-decl> (';' <event-decl>)*
<event-decl>
    ::= <identifier>
```

Declaration of event begins with `event` keyword, followed by an unempty list of event specifications separate with semicolon. Declarations list may end with semicolon.

An event is defined with its name, that is to say with an identifier. If an event is declared, a model must contain at least one transition labeled with this event.

Declaration of sub-components

Declaration of sub-nodes of an Altarica model begins with the `sub` keyword. For the rest, syntax is the same as the one used for state variables, except the fact that domain is replaced by a name of a previously declared model.

```
<sub-cls>
    ::= sub <sub-list> (';')?
<sub-decl-list>
    ::= <sub-decl> (';' <sub-decl>)*
<sub-decl>
    ::= <identifier-list> ':' <identifier>
```

Definition des transitions

Definition of model transitions begins with `trans` keyword. This keyword is followed by an unempty list of transitions specifications separate with semicolon. Declarations list may end with semicolon.

```
<trans-cls>
    ::= trans <trans-list> (';')?
<trans-list>
    ::= <transition> (';' <transition>)*
```

Transition definition always begins with a boolean expression which is the transition guard.

```
<transition>
    ::= <bool-expr> (<transition-target>)+
```

Guard of a transition is followed by an unempty list of actions linked to events.

An action begins with a separator `|` - followed by an unempty list of events previously declared in the model.(cf. 2.8.1) The list ends with `->` separator, followed by a possibly empty list of assignments of variables.

```
<transition-target>
    ::= '|' <event-name-list> '->' <assignment-list>
    ::= '|' <event-name-list> '->'
```

Assignments of variables are separate with commas.

```
<assignment-list>
    ::= <assignment> (',' <assignment>)*
```

An assignment is composed of a state variable identifier, followed by assignment separator `:=` and ends with an expression from the same type as state variable.

```
<assignment>
    ::= <identifier> ':=' <expression>
```

Definition of assertions

Definition of component assertions begins with `assert` keyword. This keyword is followed by an unempty list of boolean expressions (Cf. 2.3) separate with semicolon. Declarations list may end with semicolon.

List of assertions is construed as conjunction (logical AND) of assertions of the list.

```
<assert-cls>
    ::= assert <assert-list> (';')?
    ::= assert
<assert-list>
    ::= <bool-expr> (';' <bool-expr>)*
```

Definition of synchronizations

Declaration of synchronizations vectors begins with `sync` keyword. Declaration of synchronizations vectors is allowed if model contains at least one sub-component. (Cf. 2.8.4).

```
<sync-cls>
    ::= sync <vector-list> (';')?
```

Declaration of synchronizations vectors consists in an unempty list of vectors separate with semicolon. Declarations list may end with semicolon.

```
<vector-list>
    ::= <vector> (';' <vector>)*
```

Synchronization vector consist in an event vector beginning with `<` and ending with `>`. Events can be events of the model, or events of sub-component for hierarchical models, in this case events are prefixed with the name of the sub-component and followed by a dot.

Discussions about synchronizations have shown that there are 3 types of synchronizations:

1. Synchronization like `mec` : events can only appears at the same time

```
<vector-sync> ::= '<' <hierarchie-path> (';' <hierarchie-path>)+ '>'
```

2. Broadcast : all events that can appear at the same time appear at the same

```
<vector-diff> ::= '<' <hierarchie-path> ('|' <hierarchie-path>)+ '>'
```

3. Common cause : either events appears individually (failure without common cause), or events that can appear at the same time appear at the same time (common cause strictly speaking \approx broadcasting)

```
<vector-ccf> ::= '<' <hierarchie-path> ('?' <hierarchie-path>)+ '>'
```

A vector can be written these ways.

```
<vector>
    ::= <vector-sync>
    ::= <vector-diff>
    ::= <vector-ccf>
```

Definition of initialization

The `init` keyword enables specifications of initialization of model state variables, and initialization of sub-component state variables for hierachical models.

Declaration of component initialization begins with `init` keyword. This keyword is followed by an unempty list of state variable assignments separate with semicolon. Declarations list may end with semicolon.

List of assertions is construed as conjunction (logical AND) of assertions of the list.

```
<init-cls>
    ::= init <init-list> (';')?
<init-list>
    ::= <init-def> (';' <init-def>)*
```

An initialization consist of a path to a state variable. If the lenght of path is 1, then the referenced variable is a variable of the currently defined model, else it is a variable of a son-component of the model. Each variable is associated with a constant expression which must have the same type as that of variable.

```
<init-def>
    ::= <hierarchy-path> '==' <expression>
```

Initialization specified in a model override the ones specified in sub-components (these ones can be considered as value for default initialization).

Definition of external directives

The `extern` clause of a model has been introduced in order to give informations to tools using AltaRica language.

Declaration of directives begins with `extern` keyword. This keyword is followed by an unempty list of directives separate with semicolon. Declarations list may end with semicolon.

```
<extern-cls>
    ::= extern <extern-list> (';')?
<extern-list>
    ::= <extern-decl> (';' <extern-decl>)*
```

Contrary to previous versions of AltaRica Dassault (and AltaRica LaBRI), the syntax of external clause is specified. It takes and extends the one defined in AltaRica DataFlow language.

```
<extern-decl>
    ::= <identifier> <extern-term> '=' <extern-term>
    ::= <identifier> <extern-term>
<extern-term>
    ::= true
    ::= false
    ::= <integer>
    ::= <float>
    ::= <string>
    ::= <identifier>
    ::= <identifier> '(' <extern-term> (',' <extern-term>)* ')'
    ::= '{' <extern-term> (',' <extern-term>)* '}'
    ::= '<' flow <hierarchy-path> '>'
    ::= '<' state <hierarchy-path> '>'
    ::= '<' event <hierarchy-path> '>'
    ::= '<' sub <hierarchy-path> '>'
    ::= '<' local <hierarchy-path> '>'
    ::= '<' term '(' <expression> ')' '>'
```

The second way to declare `extern` clause enables description of a set of term having specific priority (defined by identifier before this set).

Thus, a term (of `extern` clause) is a number, a string (surrounded with double-quotes), an identifier, a function using some parameters (external type), or a set of term (of `extern` clause).

A term (of `extern` clause) can also refer to a part of the described component (flow variables, state variables, events or sub-components).

The `< local ... >` term enables definition of a parameter linked to current component, but only usable in external clauses. On the contrary, a term composed by only one identifier is defined as a global parameter.

The `< term ... >` term enable the use of expression inside external clauses.

Following chapter deals with wanted syntax of usual extern directives.

Comments

It's possible to add a comment, that is to say a string, to a part of a component (flow variable, state variable, event or sub-component).

This comment can be used by tools in order to display extra informations. For example, let be a comment linked to events representing failure of the component. This comment can be used by fault-tree generator for commenting basic event of tree for more traceability.

```
<remark-decl>
    ::= remark <objects> '=' <string>
<objects>
    ::= <object>
    ::= <objects-set>
<objects-set>
    ::= '{' <object> (',' <object>)* '}'
<object>
    ::= '<' flow <hierarchy-path> '>'
    ::= '<' state <hierarchy-path> '>'
    ::= '<' event <hierarchy-path> '>'
    ::= '<' sub <hierarchy-path> '>'
    ::= '<' local <hierarchy-path> '>'
```

Named parameters

Named parameters can be used for parameters of probability laws. It links identifier (local or global) to a law parameter, that is to say a number or a density function.

```
<param-decl>
    ::= parameter <identifier> '=' <param>
    ::= parameter '<' local <hierarchy-path> '>' '=' <param>
<param>
    ::= <float>
    ::= <identifier>
    ::= <identifier> '(' <extern-term> (',' <extern-term>)* ')'
    ::= '<' local <hierarchy-path> '>'
```

The identified density functions are :

- uniform(min-value, max-value)
- normal(mean, standard-deviation)
- lognormal(mean, error-factor)

Probability laws

Model events can be linked to probability law of occurrence. This is mainly used by tools generating fault-trees. These probability laws are described with functions using one or many parameters.

```
<law-decl>
    ::= law <events> '=' <law>
<events>
    ::= '<' event <hierarchy-path> '>'
    ::= <events-set>
<events-set>
    ::= '{' <events> (',' <events>)* '}'
<law>
    ::= <identifier> '(' <param> (',' <param>)* ')'
```

Identified probability laws are :

- exponential(lambda)

- Weibull(alpha, beta)
- Dirac(delay)
- constant(probability)
- asymptotic_exponential(lambda, mu)
- GLM(gamma, lambda, mu)
- periodic_test(lambda, period, t0)
- CMT(lambda, mission-time, Q)
- uniform(min, max)
- periodic(T, t0)

Attributs

Attributs can be linked to model events. These attributs are couples (name, value) mainly used by tools generating fault-trees.

Syntax used to define attributs on base events is the following :

```
<attribute-decl>
    ::= attribute <identifier> '(' <events> ')' '=' <attribute-value>
<attribute-value>
    ::= true
    ::= false
    ::= <integer>
    ::= <float>
    ::= <string>
    ::= <identifier>
```

<events> is defined in chapter dealing with laws.

For example, `attribute type(<event def>)= "CircuitBreaker"` means value of "type" attribut for "def" event is equal to "CircuitBreaker".

Priority

In case of deterministic events, two events (instantaneous or time delay) may appear at the same time. In order to choose the firing order of transition, a priority can be defined as shown below:

```
<priority-decl>
    ::= priority <events> '=' <integer>
```

<events> is defined in chapter dealing with laws and <integer> is an integer greater or equal to zero.

Events with high priority will be the more priority events.

Conditional events

Conditional transition are a specific case of immediate transitions. They are merged and defined with a bucket external clause. Every transition of a bucket must have the same guard. When transitions of a bucket becomes fireable, only one transition is randomly choosen (fired) depending on their probability law.

In order to say that an event is conditionnal, we have to link it to a constant probability, and we have to declare it in a bucket with the following syntax:

```
<bucket-decl>
    ::= bucket <events>
```

<events> is defined in chapter dealing with laws.

Sum of event probabilities of a bucket must me equal to 1.

Predicates and properties

Predicates and properties are quantity computed from model and can be observed with different tools. These quantity can be defined from state or flow variable, and generally from expression define with `<term (expr)>` clause. Predicates are boolean quantity, properties are numerical quantity (that is to say integer or real).

The role of this quantity is to give a way to observe model to external tools.

They are defined with the following syntax:

```
<predicate-decl>
    ::= predicate <identifier> '=' '<' term '(' <expression> ')' '>'
    ::= predicate '<' local <hierarchy-path> '>' '=' '<' term '(' <expression> ')' '>'
<property-decl>
    ::= property <identifier> '=' '<' term '(' <expression> ')' '>'
    ::= property '<' local <hierarchy-path> '>' '=' '<' term '(' <expression> ')' '>'
```

Events with memory (preemptible)

When transition is valid, a delay is defined functions of probability law assigned to event. Delay is randomly fired for stochastic transitions, or fire with a determinate way for deterministic transitions. Transition will be fired at the current time of simulation plus this delay if transition stay valid until firing time. If transition doesn't stay valid (because of other transition firing) until this time, there are two way to handle transitions when they become valid again:

- The timed elapsed when transition was valid is forgotten. When transition becomes valid again, a new delay is computed. It's AltaRica normal behavior.
- The timed elapsed when transition was valid is memorised. When transition becomes valid again, remaining time is used instead of a new one.

This property of events is declared as shown below:

```
<preemptible-decl>
    ::= preemptible <events>
```

`<events>` is defined in chapter dealing with laws.

Properties of component (nodeproperty)

It can be useful to memorise informations. It useful to keep informations about components in results files, for example their name, their creation date, their version or owner.

It can be made thanks to nodeproperty external clause as shown below:

```
<nodeproperty-decl>
    ::= nodeproperty <identifier> '=' <attribute-value>
    ::= nodeproperty '<' local <hierarchy-path> '>' '=' <attribute-value>
```

`<attribute-value>` is defined in chapter dealing with attributs.

It's also possible to memorise informations about generation of AltaRica file, and other informations (user who generated it ...), adding external clause nodeproperty to main node of AltaRica file.

AltaRica Extended Grammar

```

<identifier> ::= '[a-zA-Z][a-zA-Z0-9_]*'

<integer> ::= '0 | ([1-9][0-9]*)'

<float> ::= '[0-9]*.[0-9]+([eE][+-]?[0-9]+)?'

<string> ::= '"[^"]+"'

<altarica-description>
    ::= <global-declaration-list>
<global-declaration-list>
    ::= <global-declaration> ((';'? <global-declaration>)*
    ::=
<global-declaration>
    ::= <constant-declaration>
    ::= <domain-declaration>
    ::= <function-declaration>
    ::= <model-declaration>

<hierarchy-path> ::= <identifier> ('.' <hierarchy-path>)*

<bool-expr> ::= <expression>
<num-expr> ::= <expression>
<symb-expr> ::= <expression>
<struct-expr> ::= <expression>

<expression>
    ::= <match-expr>
    ::= <unmatch-expr>
<match-expr>
    ::= <other-expr>
    ::= if <bool-expr> then <match-expr> else <match-expr>
<unmatch-expr>
    ::= '(' if <bool-expr> then <bool-expr> ')'
    ::= if <bool-expr> then <match-expr> else <unmatch-expr>
<other-expr>
    ::= <or-expression>

<atomic-expression>
    ::= <integer>
    ::= <float>
    ::= true
    ::= false
    ::= <id-variable>
    ::= '(' <expression> ')'
    ::= min '(' <num-expr> ',' <num-expr> ')'
    ::= max '(' <num-expr> ',' <num-expr> ')'
    ::= '#' '(' <bool-expr> ',' <bool-expr> ')'
    ::= <identifier> '(' <expression> ',' <expression> '*' ')'
    ::= <identifier> '(' ')'
    ::= case '{'
        (<bool-expr> ':' <expression> ',')*
        else <expression>
    '}'

<id-variable>
    ::= <hierarchy-path> '^' <identifier>
    ::= <hierarchy-path>

<unary-expression>
    ::= - <unary-expression>
    ::= <operator-not> <unary-expression>
    ::= <atomic-expression>

```

```

<operator-not>
    ::= not
    ::= '~'

<multiplicative-expression>
    ::= <multiplicative-expression> '*' <unary-expression>
    ::= <multiplicative-expression> '/' <unary-expression>
    ::= <multiplicative-expression> '%' <unary-expression>
    ::= <unary-expression>

<additive-expression>
    ::= <additive-expression> '+' <multiplicative-expression>
    ::= <additive-expression> '-' <multiplicative-expression>
    ::= <multiplicative-expression>

<relational-expression>
    ::= <relational-expression> '<' <additive-expression>
    ::= <relational-expression> '>' <additive-expression>
    ::= <relational-expression> '<=' <additive-expression>
    ::= <relational-expression> '>=' <additive-expression>
    ::= <additive-expression>

<equality-expression>
    ::= <relational-expression> '=' <relational-expression>
    ::= <relational-expression> <operator-neq> <relational-expression>
    ::= <relational-expression> <operator-impl> <relational-expression>
    ::= <relational-expression>

<operator-neq>
    ::= '!='
    ::= '#'

<operator-impl>
    ::= imply
    ::= '=>'

<and-expression>
    ::= <and-expression> <operator-and> <equality-expression>
    ::= <equality-expression>

<operator-and>
    ::= and
    ::= '&'

<or-expression>
    ::= <or-expression> <operator-or> <and-expression>
    ::= <and-expression>

<operator-or>
    ::= or
    ::= '|'

<domain-declaration>
    ::= domain <identifier> = <domain-definition>

<domain-definition>
    ::= <link-domain>
    ::= <domain>

<domain>
    ::= <range-domain>
    ::= <enumeration-domain>
    ::= <predefined-domain>
    ::= <identifier>

<predefined-domain>
    ::= bool
    ::= int
    ::= float

<range-domain>
    ::= '[' <integer> ',' <integer> ']'

<enumeration-domain>

```

```

::= '{' <identifier-list> '}'

<link-domain>
::= link
    flow <link-flow-list> (';')?
    ( inverse <inverse-list> (';')? )?
    assert <connect-list> (';')?
    knil

<link-flow-list>
::= <link-flow-decl> (';' <link-flow-decl>)*

<link-flow-decl>
::= <identifier-list> ':' <domain>

<inverse-list>
::= <id-flow-link> (';' <id-flow-link>)*

<connect-list>
::= <connect-decl> (';' <connect-decl>)*

<connect-decl>
::= <id-flow-link> ':' <id-flow-link>

<id-flow-link>
::= in '^' <identifier>
::= out '^' <identifier>

<function-decl>
::= func <identifier>
    <function-elem>
    <function-body>
    cnuf

<function-elem>
::= <flow-cls> <function-elem>
::=

<function-body>
::= <assert-cls> <function-body>
::=

<model-declaration>
::= node <identifier>
    <node-elem>
    <node-body>
    <extern-lst>
    edon

<node-elem>
::= <node-elem-field> <node-elem>
::=

<node-elem-field>
::= <flow-cls>
::= <state-cls>
::= <event-cls>
::= <sub-cls>

<node-body>
::= <node-body-field> <node-body>
::=

<node-body-field>
::= <trans-cls>
::= <assert-cls>
::= <sync-cls>
::= <init-cls>

<extern-lst>
::= <extern-cls> <extern-lst>
::=

<flow-cls>
::= flow <flow-decl-list> (';')?

<flow-decl-list>

```

```

    ::= <flow-decl> (';' <flow-decl>)*

<flow-decl>
    ::= <identifier-list> ':' <domain> ':' <orientation>
<orientation>
    ::= in
    ::= out
    ::= private
    ::= local

<state-cls>
    ::= state <state-decl-list> (';')?

<state-decl-list>
    ::= <state-decl> (';' <state-decl>)*

<state-decl>
    ::= <identifier-list> ':' <domain>

<event-cls>
    ::= event <event-decl-list> (';')?
<event-decl-list>
    ::= <event-decl> (';' <event-decl>)*
<event-decl>
    ::= <identifier>

<sub-cls>
    ::= sub <sub-list> (';')?
<sub-decl-list>
    ::= <sub-decl> (';' <sub-decl>)*
<sub-decl>
    ::= <identifier-list> ':' <identifier>

<trans-cls>
    ::= trans <trans-list> (';')?
<trans-list>
    ::= <transition> (';' <transition>)*

<transition>
    ::= <bool-expr> (<transition-target>)+

<transition-target>
    ::= '|' <event-name-list> '->' <assignment-list>
    ::= '|' <event-name-list> '->'

<assignment-list>
    ::= <assignment> (';' <assignment>)*

<assignment>
    ::= <identifier> ':' <expression>

<assert-cls>
    ::= assert <assert-list> (';')?
    ::= assert
<assert-list>
    ::= <bool-expr> (';' <bool-expr>)*

<sync-cls>
    ::= sync <vector-list> (';')?

<vector-list>
    ::= <vector> (';' <vector>)*

<vector-sync> ::= '<' <hierarchie-path> (';' <hierarchie-path>)+ '>'
<vector-diff> ::= '<' <hierarchie-path> ('|' <hierarchie-path>)+ '>'

```

```

<vector-ccf> ::= '<' <hierarchie-path> ('?' <hierarchie-path>)+ '>'

<vector>
  ::= <vector-sync>
  ::= <vector-diff>
  ::= <vector-ccf>

<init-cls>
  ::= init <init-list> (';')?
<init-list>
  ::= <init-def> (';' <init-def>)*

<init-def>
  ::= <hierarchy-path> '=' <expression>

<extern-cls>
  ::= extern <extern-list> (';')?
<extern-list>
  ::= <extern-decl> (';' <extern-decl>)*

<extern-decl>
  ::= <identifier> <extern-term> '=' <extern-term>
  ::= <identifier> <extern-term>
<extern-term>
  ::= true
  ::= false
  ::= <integer>
  ::= <float>
  ::= <string>
  ::= <identifier>
  ::= <identifier> '(' <extern-term> (';' <extern-term>)* ')'
  ::= '{' <extern-term> (';' <extern-term>)* '}'
  ::= '<' flow <hierarchy-path> '>'
  ::= '<' state <hierarchy-path> '>'
  ::= '<' event <hierarchy-path> '>'
  ::= '<' sub <hierarchy-path> '>'
  ::= '<' local <hierarchy-path> '>'
  ::= '<' term '(' <expression> ')' '>'

<remark-decl>
  ::= remark <objects> '=' <string>
<objects>
  ::= <object>
  ::= <objects-set>
<objects-set>
  ::= '{' <object> (';' <object>)* '}'
<object>
  ::= '<' flow <hierarchy-path> '>'
  ::= '<' state <hierarchy-path> '>'
  ::= '<' event <hierarchy-path> '>'
  ::= '<' sub <hierarchy-path> '>'
  ::= '<' local <hierarchy-path> '>'

<param-decl>
  ::= parameter <identifier> '=' <param>
  ::= parameter '<' local <hierarchy-path> '>' '=' <param>
<param>
  ::= <float>
  ::= <identifier>
  ::= <identifier> '(' <extern-term> (';' <extern-term>)* ')'
  ::= '<' local <hierarchy-path> '>'

<law-decl>
  ::= law <events> '=' <law>
<events>
  ::= '<' event <hierarchy-path> '>'
  ::= <events-set>

```

```

<events-set>
    ::= '{' <events> (',' <events>)* '}'

<law>
    ::= <identifier> '(' <param> (',' <param>)* ')'

<attribute-decl>
    ::= attribute <identifier> '(' <events> ')' '=' <attribute-value>
<attribute-value>
    ::= true
    ::= false
    ::= <integer>
    ::= <float>
    ::= <string>
    ::= <identifier>

<priority-decl>
    ::= priority <events> '=' <integer>

<bucket-decl>
    ::= bucket <events>

<predicate-decl>
    ::= predicate <identifier> '=' '<' term '(' <expression> ')' '>'
    ::= predicate '<' local <hierarchy-path> '>' '=' '<' term '(' <expression> ')' '>'
<property-decl>
    ::= property <identifier> '=' '<' term '(' <expression> ')' '>'
    ::= property '<' local <hierarchy-path> '>' '=' '<' term '(' <expression> ')' '>'

<preemptible-decl>
    ::= preemptible <events>

<nodeproperty-decl>
    ::= nodeproperty <identifier> '=' <attribute-value>
    ::= nodeproperty '<' local <hierarchy-path> '>' '=' <attribute-value>

```