NAME

grl − specification language for globally asynchronous locally synchronous system

DESCRIPTION

GRL (GALS Representation Language) is a formal imperatively styled specification language for GALS systems.

LEXICAL ELEMENTS

An identifier is defined by a letter followed by a possibly empty series of letters, digits and underscores. GRL prohibits that an identifier starts or ends with an underscore.

Identifiers are case sensitive, so that all occurrences of the same identifier must use exactly the same case, i.e., lower-case and upper-case characters have to be respected. For instance, if a variable has identifier **`XyZ`**, then all its occurrences must have the same identifier **`XyZ`**, but neither **`xyz`** nor **`Xyz`**. However, to avoid confusion, GRL forbids declaring in the same scope identifiers of the same nature (e.g., variables, constructors, functions, etc.) differing only by their case. Therefore identifiers are not considered as distinct if they differ only by their case.

Comments can be both:

- single line '**`--`** <text>'
- multi-line '**`(`** ∗ <text> ∗ **`)`**'

Keywords must be written in lowercase. Reserved words can not be used as identifiers in GRL programs. They are listed below.

`abs`	**`alias`**	**`and`**	**`any`**
`as`	**`array`**	**`block`**	**`bool`**
`by`	**`case`**	**`char`**	**`const`**
`else`	**`elsif`**	**`enable`**	**`end`**
`environment`	**`enum`**	**`equ`**	**`false`**
`for`	**`from`**	**`if`**	**`implies`**
`in`	**`int`**	**`int8`**	**`int16`**
`int32`	**`is`**	**`list`**	**`loop`**
`medium`	**`module`**	**`nat`**	**`nat8`**
`nat16`	**`nat32`**	**`not`**	**`null`**
`of`	**`or`**	**`out`**	**`range`**
`receive`	**`record`**	**`select`**	**`send`**
`static`	**`string`**	**`system`**	**`then`**
`true`	**`to`**	**`type`**	**`var`**
`when`	**`where`**	**`while`**	**`xor`**

SYNTAX DESCRIPTION

The syntax of GRL is described using Bachus-Naur Form (BNF) notations.

- Non-terminal symbols are written in italic (e.g., *type_def*).
- Keywords are written using lowercase characters and between double quotes (e.g., "module", "static", "if", etc.).
  Key symbols are written between double quotes (e.g., "|", "(", "[", "∗", etc.).
- Optional sequences of symbols are written between square brackets.
- Star (∗) denotes repetition zero or more times of the preceding expression.
  Plus (+) denotes repetition one or more times of the preceding expression.

The following table summarizes the generic terminal symbols and the most frequently used non-terminal symbols and their meaning.

	Symbol	Description
Terminal	*P*	module
	S	system
	B	block
	Bi	block instance
	N	environment
	Ni	environment instance
	M	medium
	Mi	medium instance
	F	function
	X, *Y*	variables
	T	user defined type identifier
	type	type identifier
	C	type constructor
	f	record field
	L	library
Non-termainal	*I*	statement
	E	expression
	V	pattern
	K	literal constant

SYNTAX OF MODULES

A module is the highest level syntactic construct. It contains the definition of lower level constructs. A GRL file contains exactly one module definition. This module has the same name as the file containing it.

A module **P** can import libraries. Each library can be:

- another module **Pi**. Therefore, the definitions of the imported module are visible in the main module **P**. Note that circular definitions are not allowed.
- a LNT module **module**. The file has extension lnt. See section external code for more information.
- a C filemane **filename**. The file has extension c. See section external code for more information.

prgm_def ::= "module" *P* ["(" *L* ("," *L*)* ")"] "is"
(*type_def*
| *constant_def*
| *block_def*
| *environment_def*
| *medium_def*
| *system_def*)*
"end" "module"

SYNTAX OF TYPES

type ::= "bool"
| "nat" | "nat16" | "nat32"
| "int" | "int16" | "int32"
| "char" | "string"
| *T*

type_def ::= "type" *T* "is"
type_expr
"end" "type"

type_expr ::= "array" "[" *m* "..." *n* "]" "of" *type*
| "range" *m* "..." *n*
| "record" *f* ":" *type* ("," *f* ":" *type*)*
| "enum" *C* ("," *C*)*

User-defined types are the following:

- The array type, defined using the keyword **array**, denotes a fixed-size set of elements indexed by natural numbers ranging from **m** to **n**. Where **m** and **n** are literal naturals.
- The range type, defined using the keyword **range**, denotes a finite interval of numbers ranging from **m** to **n**. Where **m** and **n** are literal naturals.
- The record type, defined using the keyword **record**, denotes a fixed-size tuple of elements indexed by field names **f0, ..., fn**.
- The enumerated type, defined using the keyword **enum**, denotes a finite and ordered set of symbolic values (identifiers) **C0, ..., Cn**.
- The external type, defined in an imported LNT library, denotes any LNT type. An external type can be used natively in GRL modules.

Example

```
type T_Array is
   array [0 ... 3] of bool
end type

type T_Range is
   range -1 ... 1
end type

type T_Record is
   record f1 : nat, f2 : bool
end type

type T_Enum is
   enum a, b, c, d, e, f
end type
```

SYNTAX OF LITERAL CONSTANTS

Literal constants may be either integer numbers, boolean constants, string constants or values of enumerated types.

K ::= *int*
| *bool*
| *string*
| *C*

SYNTAX OF OPERATORS

unary_op ::= "+" | "-" | "not" | "abs"
| "nat" | "nat16" | "nat32"
| "int" | "int16" | "int32"

binary_op ::= "and" | "or" | "xor" | "implies" | "equ"
| "+" | "-" | "%" | "^" | "*" | "/"
| "!=" | "==" | "<" | ">" | "<=" | ">="

SYNTAX OF EXPRESSIONS

E ::= *X*

| "(" *E* ")"

| *E* "." *f*

| *E* "[" *E* "]"

| *unary_op E*

| *E binary_op E*

| *K* ["of" *type*]

| *F* "(" *E* ("," *E*)* ")"

SYNTAX OF PATTERNS

Patterns may be either a literal constant or a wildcard.

V ::= *K* | "any"

SYNTAX OF PREDEFINED FUNCTIONS

Predefined functions are unary operations, binary operations, type conversion functions, functions on arrays, functions on records, and constructor of external types.

- Type conversion functions, denoted **`T (E)`**, convert an expression **`E`** from one numerical data type to another numerical data type **`T`**. Numerical data types are: **`nat`**, **`nat16`**, **`nat32`**, **`int`**, **`int16`**, **`int32`**, and all the range types. An exception is raised if the value of **`E`** does not belong to the type **`T`**.
- Given an array type defined by **`type T is array [m ... n] of Tarr end type`**, two predefined functions T : Tarr^(n-m+1) -> T and T : Tarr -> T are automatically generated (those two functions coincide into one single function if m = n). The call **`T (E`**$_m$**`, ..., E`**$_n$**`)`** builds an array **`X`** in which each element **`X[i]`** is set to the value of expression **`E`**$_i$. The call **`T (E`**$_0$**`)`** builds an array **`X`** in which all elements **`X[i]`** are set to the value of expression **`E`**$_0$.
- Given a record type defined by **`type T is record f`**$_0$ **`: Trec`**$_0$**`, ..., f`**$_n$ **`: Trec`**$_n$ **`end type`**, a predefined function T : Trec$_0$ x ... x Trec$_n$ -> T is automatically generated. The call **`T (E`**$_0$**`, ..., E`**$_n$**`)`** returns a record in which each field **`f`**$_i$ is set to the value of expression **`E`**$_i$.

SYNTAX OF STATEMENTS

I ::= "null"

| *X* ":=" *E*

| *X* "[" *E* "]" ":=" *E*

| *X*"." *f* ":=" *E*

| *I* ";" *I*

| "if" *E* "then" *I*
("elsif" *E* "then" *I*)*
["else" *I*]
"end" "if"

| "while" *E* "loop"
I
"end" "loop"

| "for" *I* "while" *E* "by" *I* "loop"
I
"end" "loop"

| "case" *E* ("," *E*)* "is"
V ("," *V*)* "->" *I*
("|" *V* ("," *V*)* "->" *I*)*
"end" "case"

| "select"
I ("[]" *I*)*
"end" "select"

| *X* ":=" "any" *type* ["where" *E*]

| "when" ["?"]*X* "->" *I*

| "when" ["?"] "<" *X* ("," *X*)* ">" "->" *I*

| "enable" *X*

| *Bi* ["{" *arg_list* "}"] "(" *arg_list* ")"

SYNTAX OF CONSTANTS

The keyword **`constant`** is used to define a variable whose value, once initialized, can not be changed. A constant is visible and can be called by all other entities defined in the module.

constant_def ::= "const" *var_list* ":" *type* ":=" *E*

Example

```
const C_Max : nat16 := 4
```

SYNTAX OF VARIABLE DECLARATIONS

decl_list ::= *var_decl* ("," *var_decl*)*
var_decl ::= *var_list* ":" *type* [":=" *E*]
decl_list_non_init ::= *var_decl_non_init*
("," *var_decl_non_init*)*
var_decl_non_init ::= *var_list* ":" *type*
var_list ::= *X* ("," *X*)*

Permanent variables, defined after the keywords **`static var`**, have a lifetime extending across the entire execution of the program, whereas variables defined after the keyword **`var`** are temporary.

local_vars ::= "static" "var" *decl_list*
| "var" *decl_list*

SYNTAX OF FORMAL PARAMETERS

Formal parameters are classified as follows.

- Constant parameters are user-fixed parameters defined as a set of variable declarations inside **`{`** and **`}`**. No value should be assigned to such parameters after their definition.

  const_param ::= *decl_list*

- Input/Output parameters are defined as a set of variable declarations preceded by the keywords **`in`** or **`out`**.

  inout_param ::= "in" *decl_list*
  | "out" *decl_list_non_init*

  inout_param_non_init ::= "in" *decl_list_non_init*
  | "out" *decl_list_non_init*

- Communication parameters are defined as a set of variable declarations preceded by the keywords **`receive`** or **`send`**.

  com_param ::= "receive" *decl_list_non_init*
  | "send" *decl_list_non_init*

- Activation parameters are defined as a variable preceded by the keyword **`block`**.

  active_param ::= "block" *X* ("," *X*)*

SYNTAX OF BLOCKS

Blocks represent the synchronous part, which is inspired by synchronous dataflow languages based on the block-diagram model. Following the definition of synchronous programs, all subblocks called in a block are governed by the clock this block.
Subblocks can be aliased to increase readability using keyword **`alias`**. Constant parameters, if any, are mandatory when aliasing a block.

block_def ::=
"block" *B* ["{" *const_param* "}"]
["(" *inout_param* ("," *inout_param*)* ")"]
["[" *com_param* ("," *com_param*)* "]"] "is"
["alias" *block_alloc* ("," *block_alloc*)*]
[*local_vars* ("," *local_vars*)*]
I
"end" "block"

| "block" *B* ["{" *const_param* "}"]
"(" *inout_param* ("," *inout_param*)* ")" "is"
"!c" *string* | "!lnt" *string*
"end" "block"

A block definition can be either user-defined or included from an external code written in C or LNT language. An external block should not have receive nor send parameters and its body consists of a "pragma" denoting the language from which the external block is imported, followed by the name of the function implementing the block. See section external code for more information.

Example

The first example is a basic block allocating no other block, the second one connects several block instances together.

```
block B {c : nat}
        (in i : bool, j : nat, in k : bool, out o : nat) is
   if (i or k)
   then
      o := j
   else
      o := c
   end if
end block

block B (in i : bool, j : nat, in k : bool, out o : nat) is
   alias B1 as Bi1; Bi2,
         B2 {true} as Bi3,
         B3 {_, _} as Bi4
   var c1, c3 : bool,
       c2 : nat
   Bi1 (i, j, ?c1);
   Bi3 (k, ?c2);
   Bi2 (c1, c2, ?c3);
   Bi4 (c3, ?o)
end block
```

SYNTAX OF ENVIRONMENTS

Environments represents the behaviour of the environment surrounding a network of blocks. They allow to constrain either inputs of separate blocks or the relative order and frequency of block executions within a network. They make possible the description of the common environment in the case of parallel systems distributed on a single platform as well as a set of separate environments in the case of geographically distributed systems.

env_param ::= *active_param* | *inout_param_non_init*
environment_def ::=
"environment" *N* ["{" *const_param* "}"]
"(" *env_param* ("," *env_param*)* ")" "is"
["alias" *block_alloc* ("," *block_alloc*)*]
[*local_vars* ("," *local_vars*)*]
I
"end" "environment"

An environment **N** interacts with blocks by connecting its input (resp., output) parameters to output (resp., input) parameters of blocks. Its body should define (or not) the behaviour of the environment when an interaction via a formal parameter list occurs. A signal of the form **when ?<X0, ..., Xn> -> I** defines

the behaviour of **N** that corresponds to the parameter list **in X0 : T0, ..., Xn : Tn** whereas a signal of the form **when <Y0, ..., Yn> -> I** defines the behaviour of **N** that corresponds to the parameter list **out Y0 : T0, ..., Ym : Tm**.

Example

The first environment computes one output using global constant **C_Max** and **C_Init**. The second one computes an output using local constants and the value given by the input. The third one is an activation environment constaining block instances **A** and **B**.

```
environment E {Step : nat16 := 1} (out No : nat16) is
   static var Last_No : nat16 := C_Init
   when No -> if ((Last_No + Step) <= C_Max)
              then
                 No := Last_No + Step
              else
                 No := C_Init
              end if;
              Last_No := No
end environment

environment E {Min    : nat16 := 0,
               Max    : nat16 := 30,
               Offset : nat16 := 3}
              (in X : nat16, out Y : nat16) is
   static var Last_X : nat16 := 0
   select
      when ?X -> Last_X := X
   []
      when Y -> select
                   Y := Last_X
                []
                   if (Last_X <= (Max - Step))
                   then
                      No := Last_X + Step
                   else
                      No := Last_X
                   end if
                []
                   if (Last_X >= (Min + Step))
                   then
                      No := Last_X - Step
                   else
                      No := Last_X
                   end if
                end select;
   end select
end environment

environment E (block A, block B) is
   static var Last_Act_A : bool := false
   if (Last_Act_A)
   then
      enable B
   else
```

```
      enable A
   end if;
   Last_Act_A := not (Last_Act_A)
end environment
```

SYNTAX OF MEDIUMS

Mediums represents the behaviour of communication mediums and enables a clean description of asynchronous interactions within a network of blocks. They enable the explicit description of the communication protocol and the rigorous design of networks whatever their topologies (star, bus, ring, etc.) and their means of communication (point-to-point, multipoint, etc.) are.

medium_def ::=
  "medium" *M* ["{" *const_param* "}"]
        "[" *com_param* ("," *com_param*)* "]" "is"
    ["alias" *block_alloc* ("," *block_alloc*)*]
    [*local_vars* ("," *local_vars*)*]
    I
  "end" "medium"

 | "medium" *M* ["{" *const_param* "}"]
        ["[" *com_param* ("," *com_param*)* "]"] "is"
    ("from" "?" "<" *var_list* ">" "to" "<" *var_list* ">")+
  "end" "medium"

A medium interacts with blocks by connecting its receive (resp., send) parameters to send (resp., receive) parameters of blocks. Its behaviour is the same as environments.

For short mediums, **from ?<X0, ..., Xn>** should correspond to the parameter list **in X0 : T0, ..., Xn : Tn** whereas **to <Y0, ..., Yn>** should corresponds to the parameter list **out Y0 : T0, ..., Yn : Tn**.

Example

The first medium is a single buffer. The second medium is the short syntax for this buffer.

```
medium M [receive a : bool, c : bool,
          send b : bool, d : bool] is
   static var p : bool := false
              q : bool := false
   select
      when ?<a, c> -> p := a;
                      q := c
   []
      when <b, d> -> b := p;
                     d := q
   end select
end medium

medium M [receive a : bool, c : bool,
          send b : bool, d : bool] is
   from ?<a, c> to <b, d>
end medium
```

SYNTAX OF INVOCATION

Actual parameters, formally defined below, denote parameters passed to an instance at invocation time.

- The corresponding formal parameter of an actual parameter denotes the formal parameter that has the same position (as the actual parameter) in the actor definition.
- An actual parameter is at constant position (respectively at input position, at output position, at receive position, and at send position) if its corresponding formal parameter is defined inside **{** and **}** (respectively after keywords **in**, **out**, **receive**, and **send**).

There are two types of actual parameters, the first one are used in systems and the second ones in other entities.

- In system, an actual parameter can have different forms according to its position. A question mark precedes both actual parameters at output and send positions. An underscore is used for unconnected parameters. An actual parameter of the form **any type** assigns the corresponding formal parameters an arbitrary value of type **type**.

chan_arg_list ::= *chan_arg* ("," *chan_args*)*

chan_arg_list ::= *chan_args* ("," *chan_args*)*

chan_args ::= "<" *chan_arg* ("," *chan_arg*)* ">"

chan_arg ::= ["?"] *X* | ["?"]_ | "any" *type*

- In other entities, an actual parameter can have different forms according to its position. A question mark precedes actual parameters at output positions. An underscore is used for unconnected parameters. An actual parameter of the form **any type** assigns the corresponding formal parameters an arbitrary value of type **type**.

arg_list ::= *arg* ("," *arg*)*

arg ::= "?" *X* | ["?"]_ | *E* | "any" *type*

Instances can be allocated before being invoked. Actual parameters at constant position should be fixed at allocation time as follows.

block_alloc ::= *B* ["{" *arg_list* "}"] "as" *Bi* (";" *Bi*)*

alloc ::= *block_alloc*
| *N* ["{" *arg_list* "}"] "as" *Ni* (";" *Ni*)*
| *M* ["{" *arg_list* "}"] "as" *Mi* (";" *Mi*)*

Instances can be invoked as follows. If instance has been allocated constant arguments are not allowed, else actual parameters at constant position should be fixed at invocation time.

block_inst ::= *Bi* ["{" *arg_list* "}"] "(" *chan_arg_list* ")"
| *Bi* ["{" *arg_list* "}"]
["(" *chan_arg_list* ")"]
["[" *chan_arg_list* "]"]

environment_inst ::= *Ni* ["{" *arg_list* "}"]
"(" *chan_arg_list* ")"

medium_inst ::= *Mi* ["{" *arg_list* "}"]
"[" *chan_arg_list* "]"

SYNTAX OF SYSTEMS

A system specifies a network of synchronous blocks. Those blocks are constrained by a set of environments and interact asynchronously via a set of mediums.

system_def ::=
"system" *S* ["{" *const_param* "}"]
["(" *decl_list_non_init* ")"] "is"
["alias" *alloc* ("," *alloc*)*]
["var" *decl_list_non_init*]
"block" "list"
block_inst ("," *block_inst*)*
["environment" "list"
environment_inst ("," *environment_inst*)*]
["medium" "list"
medium_inst ("," *medium_inst*)*]
"end" "system"

The first list *decl_list_non_init* defines the parameters that are visible from the external world whereas the second list *decl_list_non_init*, after keyword **var**, defines the invisible parameters.

Note that activated block must be aliased before activated.

Example

```
system Main {cst : nat16}
            (a, a2, b : int32, c : bool, d : nat) is
   alias B1 {cst} as Bi1,
         B2 {cst} as Bi2,
         E1 {_, 4} as Ei1,
         E2 {true, false} as Ei2,
         M1 as Mi1
   var w : nat16, x : nat16, y : nat16, z : nat16
   block list
      Bi1 (<a, a2>, ?<b>) [<w>, ?<x>],
      Bi2 (c, ?d) [y, ?z]
   environment list
      Ei1 (?<a, a2>),
      Ei2 (?c)
   medium list
      Mi1 [<x>, z, ?<w>, ?y]
end system

system Main {cst : nat16}
            (a, a2, b : int32, c : bool, d : nat) is
   var w : nat16, x : nat16, y : nat16, z : nat16
   block list
      B1 {cst} (<a, a2>, ?<b>) [<w>, ?<x>],
      B2 {cst} (c, ?d) [y, ?z]
   environment list
      E1 {_, 4} (?<a, a2>),
      E2 {true, false} (?c)
   medium list
      M1 [<x>, z, ?<w>, ?y]
end system

system Main (a, b, c, d : bool) is
```

```
   alias B1 as Bi1, B2 as Bi2
   block list
      Bi1 (a, ?b),
      Bi2 (c, ?d)
   environment list
      E (Bi1, Bi2)
end system
```

EXTERNAL CODE

External code can be either LNT code or C code.

LNT CODE

External LNT code should be written into .lnt files. A external block should be represented by a function in LNT code, following some rules.

- The number of parameters of the LNT functions and the block are the same.
- Input and constant parameters are represented by the **in** LNT keyword.
  Output parameters are represented by the **out** LNT keyword.
- Each LNT parameter should be typed using the equivalent LNT type. The following LNT types **Bool, Nat8, Nat16, Nat, Int8, Int16, Int, char, string** are respectively equivalent to the following types **bool, nat, nat16, nat32, int, int16, int32, char, string**.
  Note that user types keep the same name in the LNT code. And user LNT types can used natively in GRL modules.

Example

The following code gives the block and its LNT implementation.

```
block EXT (in a : int16, in a2 : int16,
           out b : int16, out b2 : int16) is
   !lnt "EXT_LNT"
end block

function EXT_LNT (in a : Int16, in a2 : Int16,
                  out b : Int16, out b2 : Int16) is
   b := a + a2;
   b2 := a - a2
end function
```

C CODE

External C code should be written into .c files. A external block should be represented by a function in C code, following some rules.

- The C function should have no side effect.
- The number of parameters of the C functions and the block are the same.
- Input and constant parameters are represented by C value parameters.
  Output parameters are represented by C pointer parameters.
- Each C parameter should be typed using the equivalent C type. The following C types **GRL_BOOL, GRL_NAT, GRL_NAT16, GRL_NAT32, GRL_INT, GRL_INT16, GRL_INT, GRL_CHAR, GRL_STRING** are respectively equivalent to the following types

bool, nat, nat16, nat32, int, int16, int32, char, string.
Note that user types are prefixed with the string "GRL_" and uppercase. For example the C type **GRL_TYPE_A** is equivalent to the type **type_a**.

- It is recommended to use conversion functions (e.g., **GRL_TYPE1_TO_TYPE2**) at the beginning and at the end of each C function to interface safely with the GRL code. Each function converts the given value to the required type and checks that there is no overflow.

Example

The following code gives the block and its C implementation.

```
block EXT (in a : int16, in a2 : int16,
           out b : int16, out b2 : int16) is
   !c "EXT_C"
end block

void EXT_C (GRL_INT16 a, GRL_INT16 a2,
            GRL_INT16* b, GRL_INT16* b2)
{
   signed short arg_a = GRL_INT16_TO_SIGNED_SHORT (a);
   signed short arg_a2 = GRL_INT16_TO_SIGNED_SHORT (a2);

   signed int res1 = arg_a + arg_a2;
   signed int res2 = arg_a - arg_a2;

   *b = GRL_SIGNED_INT_TO_INT16 (res1);
   *b2 = GRL_SIGNED_INT_TO_INT16 (res2);
}
```

AUTHORS

Eric Leo (version 2).
Eric Leo (version 1).
Eric Leo (initial draft, based on Fatma Jebali's definition of GRL).

SEE ALSO

For a complete description of GRL, see the **grl2lnt** reference manual. Otherwise see also the following man pages:
grl.open(LOCAL), **grl2lnt**(LOCAL).

Additional information is available from the CADP Web page located at http://cadp.inria.fr

BUGS

Please report any bug to cadp@inria.fr