Copy Number .....

PDR Library Primitives

P6957-11-004
13-April-2000 Issue 1

Written by

.....

E.D.Willink

Reviewed and Approved by

Project Manager

.....

C.Waugh

Authorised by

Project Director

Date ....

C.P.Ash

© DERA 2000

DERA Portsdown
Portsdown Hill Road
Fareham
Hants
PO17 6AD

The investigation, which is the subject of this report, was carried out under the terms of Contract CU009-0000002745. All recipients of this report are advised that it is not to be copied in part or in whole or be given further distribution without the written approval of Intellectual Property Department, DERA Farnborough.

Configuration Management	
CM Tool/Version	Visual SourceSafe v5.0 and v6.0
Library Location	Q:\P6957\Vss\…
Library Project	$/P6957/Project Library/11 Technical Reports & Notes/ 004 Library Primitives
Baseline	
Elements	`LibraryPrimitives.doc` - words `LibraryPrimitives.vsd` - pictures `Primitives.wdl` - syntax checked examples `Library.wsg` - picture icon templates

Modification History			
Author	**Ident**	**Date**	**Description**
EDW	28 February 2000		Very preliminary draft made available to interested parties
EDW	17-March-2000		Further draft tracking language and showing how to polymorphise decoders
EDW	Issue 1	13-April-2000	End of phase 1

Table of Contents

Configuration Management ii
Modification History ii
Table of Contents iii
1 Introduction 1
1.1 Document Structure 1
1.2 Coverage 1
1.3 Further Work 1
2 Dimensions 3
2.1 Time 3
2.2 Angles 3
3 Types 4
3.1 boolean, natural, integer, real, complex 4
3.2 GaloisField 4
3.3 Polynomial 5
3.4 PrimitivePolynomial 5
4 Generic Parameters 6
5 Computations 7
5.1 Add 9
5.2 Compare 10
6 Assignments 11
6.1 Overview 11
6.1.1 Conversions 11
6.1.2 Arrangements 12
6.2 Bundler 14
6.3 Composer 15
6.4 Concatenator 16
6.5 Constructor 18
6.6 Decomposer 19
6.7 Deconstructor 20
6.8 Index 21
6.9 Parallelizer 23
6.10 Serializer 24
6.11 Switch 25
7 Scheduling 26
7.1 Clock 27
7.2 Delay 28
7.3 Event 30
7.4 Exit 31
7.5 Poll 32
7.6 Recorder 33
7.7 Sink 34
7.8 Store 35
7.9 Synch 36
7.10 When 37
8 Generators 38
8.1 F(t) 38
8.2 F1(t) 38

8.3 Impulse, Sine, Square 38
8.4 Gaussian, Uniform 38
8.5 Nco 38
8.6 Constant 39
8.7 Ramp 40
9 Higher Order Entities 41
9.1 ArrayOf 41
10 Utilities 42
10.1 CounterProcess 43
10.2 ValueProcess 45
10.3 ViewProcess 46
11 Special Purpose 47
11.1 Codecs 47
11.1.1 Abstract Codecs 47
11.1.1.1 CodecParams 47
11.1.1.2 Decoder 48
11.1.1.3 Encoder 49
11.1.2 Reed Solomon Codecs 50
11.1.2.1 RsParams 50
11.1.2.1.1 SystematicRsParams 50
11.1.2.1.2 TransformRsParams 50
11.1.2.2 RsDecoder 51
11.1.2.2.1 SystematicRsDecoder 53
11.1.2.2.2 TransformRsDecoder 53
11.1.2.3 RsEncoder 54
11.1.2.3.1 SystematicRsEncoder 55
11.1.2.3.2 TransformRsEncoder 56
11.2 Filters 57
11.3 Modems 57
11.4 Transforms 57
12 Test aids 58
12.1 GUI Sources 58
12.2 GUI Sinks 58
Index 59

1 Introduction

The Waveform Description Language provides an implementation-neutral language for unambiguous specification of the externally observable behaviour of a system . A system specification is composed hierarchically from sub-system (or entity) sub-specifications. Repeated decomposition leads to simple re-usable entities that can be provided by a library.

This document provides the specifications for a variety of entities that may be used to assist in writing specifications. The library comprises:

Primitive entities: entities that cannot sensibly be decomposed into more primitive entities and so must have associated implementations.

Semi-primitive entities: entities that can be further decomposed, but for which direct implementations are likely to be beneficial.

1.1 Document Structure

General purpose declarations are described first involving standard dimensions in Section 2, types in Section 3, and the parameters common to all entities in Section 4. The library entities are then specified with computational entities in Section 5, assignment entities in Section 6, scheduling entities in Section 7 and (continuous time) generators in Section 8. General purpose entities are sprecified in Section 9, with more special purpose entities in Section 11. Finally Section 12 specifies entities support for testing and simulation.

1.2 Coverage

The listed library entities comprise the primitive entities that are necessary to support WDL, some further semi-primitive entities that are likely to be useful, or and further entities that where found to be useful for the FM3TR decomposition.

The `Encoder` and `Decoder` entities, although only partially elaborated for Reed Solomon coding, indicate how a family of isomorphic entities can be defined so that configuration can be performed remote from the implementation.

1.3 Further Work

This document is a working document with the current version documenting the work conducted on phase 1 of the PDR programme. It is envisaged that the document will be more fully completed at the end of phase 2, when WDL will also be more full defined.

Further work involves

- Definitions of the entities defined only by implication or defined only in note form.
  - Most arithmetic entities
  - A few assignment entities
  - A few scheduling entities
  - Most continuous time generators and operators
- Clearer definitions and tutorial examples are required for most of the other entities.

- Tutorial and consideration of data parallel (array interpretation) of many operators.
- Mnemonic pictures for most entities
- Application domain entities such as Filters, Modems, Transforms
- Test support entities for GUI, file, script and hardware interaction

Areas that need further consideration have been grayed over to assist completion of phase 2.

2 Dimensions

2.1 Time

The concept of time is necessary for a variety of library entities.

The time dimension is therefore defined and a corresponding type using that dimension.

```
dimension Time;
type Time : real‘Time;
```

Time is measured from the ideal start time of the entity in which time is used. There is therefore no inherent consistency between entities with different life-times, and the consistency between entities with the same life-time is TBD.

There is no concept of absolute time in the language. If needed, an AbsoluteTime can be defined as a user type and maintained using library entities.

The scaling of Time is not specified, so a declaration such as

```
dimension s;
dimension Time = 1*s;
```

should be provided so that applications may specify `10‘s`.

2.2 Angles

Trigonometric functions such as `sin` and `acos` make use of angles. These are usually expressed in radians for computing purposes, but are often specified in degrees. Confusion between the two schemes can lead to significant errors. The WDL specification values therefore have an angular dimension and eliminate the opportunity for error.

The built-in angular dimension is the `quadrant` dimension and represents 90 degrees. Other dimensions may be user-defined.

```
dimension degree = quadrant / 90;
dimension radian = quadrant * pi / 2;
```

3 Types

The WDL type system should offer similar facilities to those of modern Object Oriented languages. However since WDL has more general concepts of multi-dimensional arrays, discriminated unions and genericity, it is not possible to adopt the type system of any particular language as is.

The following definitions are indicative of the kind of facilities that may be appropriate.

3.1 boolean, natural, integer, real, complex

Five behavioural types are built-in to WDL with a (very incomplete) hypothetical interface such as

```
record natural : boolean
{
    operation add() : natural ;
    operation multiply() : natural ;
    operation integer_divide() : natural ;
    operation pow() : natural ;
};

record integer : natural
{
    operation add() : integer ;
    operation subtract() : integer ;
    operation multiply() : integer ;
    operation integer_divide() : integer ;
    operation pow() : integer ;
};

record real : integer
{
    operation add() : real ;
    operation subtract() : real ;
    operation multiply() : real ;
    operation divide() : real ;
    operation pow() : real ;
};

record complex : real
{
    operation add() : complex ;
    operation subtract() : complex ;
    operation multiply() : complex ;
    operation divide() : complex ;
    operation pow() : complex ;
};
```

3.2 GaloisField

A `GaloisField` defines a data element whose mathematical operations are performed using Galois Fields.

```
record GaloisField
{
    generic degree : value = natural;
    generic polynomial : value = PrimitivePolynomial;
    constraint: polynomial.ElementType = natural;
    constraint: polynomial.terms.shape = [degree+1];
    type Value : natural { maximum (1 << degree) - 1; };
```

```
    attribute value = 0 : Value;
    operation add() : GaloisField;
    operation subtract() : GaloisField;
    operation multiply() : GaloisField;
    operation divide() : GaloisField;
    operation pow() : GaloisField;
};
```

The specifications of the mathematical operation have not yet been defined.

3.3 Polynomial

A polynomial type maintains the coefficients of a polynomial as a vector (one-dimensional array) such that index zero comprises the constant (x^0) term, index one the linear (x^1) term, etc.

Note that this ordering of terms is consistent, but conflicts with the lexical order of array initializers. The polynomial `2x+1` is therefore `Polynomial([1,2])` or `Polynomial(Reverser([2,1]))`.

Mathematical operations such as additions and subtraction operate in conventional element-wise fashion. Multiplication, division and modulus exhibit the behaviour of a polynomial rather than that of a vector.

```
record Polynomial
{
    generic ElementType : type;
    attribute terms : ElementType[*];
    constraint: terms._shape[0] > 0;
    constant Order = terms._shape[0]-1 : natural;
    constraint: (Order = 0) | (terms[Order] != 0);
    operation add() : Polynomial;
    operation subtract() : Polynomial;
    operation multiply() : Polynomial;
    operation modulus() : Polynomial;
    operation divide() : Polynomial;
};
```

The specifications of the mathematical operation have not yet been defined.

Can a single definition of Polynomial handle fixed and variable length polynomials?

How is overflow of fixed length polynomial multiply resolved?

3.4 PrimitivePolynomial

A `PrimitivePolynomial` adds the no-factors constraint to the specification of a `Polynomial`; there must be no instance of a `Polynomial` other than the unit polynomial and the `PrimitivePolynomial`, for which the remainder of the `PrimitivePolynomial` divided by the polynomial yields the zero polynomial.

```
record PrimitivePolynomial : Polynomial
{
    constraint no_factors: for no p in Polynomial
        where ((p != Polynomial([1])) & (p != this))
            { this % p = Polynomial([0]); };
};
```

4 Generic Parameters

Certain parameters are implicitly defined for every entity, and so are not repetitively defined for each.

Scheduling

The computation of each entity occurs at the rendezvous of each input to a response. This rendezvous occurs at the limit of causality and so precisely defines the behaviour for an infinite computing resource. Practical implementations cannot meet this ideal and so a latency must be specified.

`_inertia`

The `_inertia` specifies an upper-bound on the tardy production of any output. The default `_inertia`is infinite. `_inertia`will typically be specified only at system outputs leaving an (automated) design tool to establish an `_inertia`budget for each subsystem.

`_runaway`

The `_runaway` specifies an upper bound on the premature production of any input. The default `_runaway`is infinite. `_runaway`may not need to be specified at all since the lack of available inputs may constrain premature calculation. However in some applications it may be necessary to prevent some computation proceeding too soon and thereby ignoring the effects that value flows could involve.

`_after`

Precedence between 'concurrent' events is enforced by requiring that the entities (or their parents) that generate concurrent events specify an `_after` precedence.

Partitioning

Implementations may allocate entities directly to processors, using the `_processor` parameter, or indirectly using the `_component` parameter. Specification of the placement of an entity, automatically supplies a default placement for child entities.

`_component`

Specifies implementation of the entity behaviour within a specific component.

`_processor`

Specifies implementation of the entity behaviour within a specific processor.

Resource usage

power-consumption

processing time

memory

5 Computations

In so far as possible all library entities operate on all data types. In particular they work element wise on arrays of data types with a distinct local context for each element thus a PRBG may generate a 4 by 3 array output using 12 distinct shift registers that are clocked once, rather than one register 12 times. Library entities involving arithmetic do not work directly (compile-time error) on records or discriminated unions. However records and discriminated unions may define overloaded implementations for the standard arithmetic operations, enabling for instance an FFT to be performed using Galois field arithmetic.

The computational operators exhibit conventional arithmetic behaviours, so there is relatively little point in padding out this document with 50 pages of obvious specifications. Most operators are therefore just listed by name, with any special behavioural aspects highlighted. `Add` and `Compare` are documented to give an indication of the more complete specifications.

Add, Div, Idiv, Matmul, Mod, Mul, Neg, Recip, Sub, ...

`Idiv` (integer divide) is a distinct operator from `Div` to avoid anomalies when excess precision is used. It returns the integer nearest to zero.

`Recip` is a variant of `Div` using 1 as the dividend.

`Matmul` is a variant of `Mul` that performs a cross rather than dot product for matrices.

`Mod` returns the signed remainder such that `mod(x, y) + y * idiv(x,y) == x`.

And, Not, Or, Xor

ShiftLeft, ShiftRight, RotateLeft, RotateRight

Rotation is only defined for types with an explicit number of bits.

Abs, Max, Min, AbsSquared
Ceil, Floor, Limit, Quant, Round, Sign,Trunc, ...

Sqrt,
Sin, Cos, Tan,
Asin, Acos, Atan,
Sinh, Cosh, Tanh,
Asinh, Acosh, Atanh,
Exp, Log2, Loge, Log10, Pow,
Sinc, ...

`Atan` takes two arguments as a complex number and has defined zero behaviour for `Atan2(0+0i)`.

Conj, Complex, Imag, Real,
Int, Frac,

Function

The generic function block supports definition of its output as an expression involving its inputs and hierarchically visible names.

Reducer

Applies a mathematical operation repetitively to reduce an array dimension to an

element. Reduction using an `add` operator along a matrix row produces a vector of row sums. Reduction using a `min` operator along a matrix row and column returns the minimum matrix element.

5.1 Add

Graphical usage

The input is a multi-port and so the icon should be adapted to suit the required number of inputs.

Expression usage

```
out = Add(in1)
out = Add(in1, in2)
out = Add(in1, in2, in3)
etc.
```

Description

The output is the sum of all the inputs as determined by repeated invocation of `ElementType.add()`.

There are some interesting dimensional issues to resolve to ensure consistency for addition of a continuous or discrete set of values along a time axis.

Ports

```
generic ElementType : type;
```

Inputs and output must share a common type (and shape).

```
generic Shape : shape;
```

The multi-port inputs may have any shape.

```
in in[Shape] : ElementType;
```

The input is a multi-port array of the element type.

```
out out : ElementType;
```

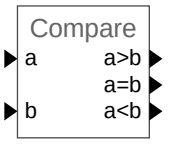
The output is of the element type.

Specification

```
response in
{
    constraint: out = reduce ElementType.add for all i in Shape
                                    { in[i]; };
};
```

5.2 Compare

Graphical usage



Expression usage

```
a < b
```

Description

The `Compare` entity generates results indicating the relative relationship of two inputs.

See Also

`When` may be used to generate events when the relationship between inputs change.

If may be used to generate tokens depending on the relationship of a pair of tokens.

Generics

```
generic AnyFlow : flow;
generic AnyType : type;
```

Ports

```
in:AnyFlow a : AnyType[];
in:AnyFlow b : AnyType[];
```

The input signals are of the same flow and arbitrary scalar type.

```
out:AnyFlow gt = 1 : natural;
out:AnyFlow eq = 0 : natural;
out:AnyFlow lt = -1 : natural;
```

The outputs indicate the required condition, and carry an identification value to facilitate a merge of the outputs.

Constraints

The use of two inputs by the same response prohibits the use of a pair of events.

Specification

```
response a b
{
    if (a < b) then { lt(-1); }
    else if (b < a) then { gt(1); }
    else { eq(0); };
};
```

6 Assignments

The many requirements to change the format of data are accommodated by the entities in this section. They are loosely referred to as assignment entities, although many define a more general bidirectional identity between inputs and outputs.

6.1 Overview

6.1.1 Conversions

Simple assignments just convert from one format to another.

Array element conversion - `Composer` (`Decomposer`)

Composes multiple elements into an array of those elements.

(Decomposes one dimension of an array into multiple elements).

In_i: 1 token of 1 element

Out: 1 token of an array of i elements (i=0 first)

Record/Field conversion - `Constructor` (`Deconstructor`)

Constructs a message from its constituent fields (populates a C struct).

(Splits a message into its constituent fields)

In: 1 token of 1 element for each message field

Out: 1 token of 1 element of the message type

Discriminated union conversion - `Discriminator` (`Harmonizer`)

Splits a message into its data dependent part.

(Combines alternative message formats into a composite message).

In: 1 token of 1 element for one of the message field

Out: 1 token of 1 element of the message type

Time conversion - `Serializer` (`Parallelizer`)

Convert an array to a time sequence of samples.

(Convert a time sequence of samples to an array).

In: N token of 1 element

Out: 1 token of 1 element to each of N outputs

A `Distributor` is a concatenation of `Parallelizer` and `Decomposer`.

A `Commutator` is a concatenation of `Composer` and `Serializer`.

A `DownSampler` is a specialised form of `Distributor` for which all but the initial phase output are fed to `sinks`.

An `UpSampler` is a specialised form of `Commutator` for which all but the initial phase input are typically fed by the 0 `const`.

A `Repeater` is a specialised form of `Commutator` for which all inputs are fed by the sole input.

Bit conversion - `Nibbler` (`Unnibbler`)

Packs multiple input values (typically one or just a few bits) into a single value least significant value first. Each input (output) has an associated bit width.

6.1.2 Arrangements

More complicated assignments rearrange or select data values.

`Concatenator` (`Partitioner`)

Concatenates multiple input arrays to a wider array.

(Partitions an array into multiple narrower array outputs).

In_i: 1 token of array of N_i elements

Out: 1 token of $\Sigma_i N_i$ elements (elements of i=0 first)

This operation is apparently very similar to `composer`/`decomposer`, the difference lies in enforcement of strong typing. The input and output arrays of `composer`/`decomposer` have different array dimension depth. The input and outputs of `Concatenator`/`Partitioner` have the same array dimension depth.

A `Stacker` is a specialised form of `Concatenator` for which all inputs are fed by the sole input.

`Index`

Selects a set of values by array index from the input.

`Switch`

Selects a values by name from the input.

`Reverser`

Element indexes are reversed along a specified array dimension. Use on a vector to interchange least significant/most significant first for a `Serializer` or `Nibbler`.

`Transposer`

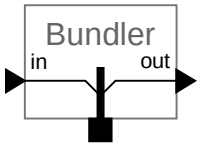
Two nominated array dimensions are interchanged.

Find

Find first occurrence etc..

6.2 Bundler

Graphical usage



The inputs and/or outputs correspond by name to the fields of the bundle type.

Expression usage

```
aBundle.field = aValue;
aValue = aBundle.field;
```

Description

The Bundler provides access to the fields in the bundle.

Generics

```
generic BundleType : type;
```

The bundle type is the composite of all the fields..

Ports

```
inout bundle : BundleType;
```

The bundle port has the bundle type..

```
for all i in BundleType.attributes
{
    inout i._name = bundle.i._name : i._type;
};
```

A port of unspecified direction is provided for each element of the `BundleType`, with the data type as the same-named field in the bundle.

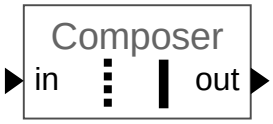
Specification

The bundled and unbundled elements are the same.

```
constraint: for all i in BundleType.attributes
{
    i._name = bundle.i._name;
};
```

6.3 Composer

Graphical usage



The input is a multi-port, and so the icon should be adapted to the required number of inputs.

Expression usage

```
out = Composer([in0, in1, in2, in3]);
```

Description

The `Composer` entity is a port combiner; a multi-port array input is converted into a conventional array output.

See Also

The `Concatenator` entity is also a port combiner; the elements of a multi-port array are concatenated along a nominated dimension to create a larger array.

Generics

```
generic ElementType : type;
generic Shape : shape;
```

The converted element may be of any type (and shape).

Ports

```
in[Shape] in : ElementType;
```

The input is a multi-port array of the element type.

```
out out : ElementType[Shape];
```

The output is a conventional array of the element type.

Specification

Each input port element (which need not be scalar) is matches its corresponding array output. A rendezvous is performed over the input ports.

```
response in
{
    constraint : out = in;
};
```

6.4 Concatenator

Graphical usage

The input is a multi-port, and so the icon should be adapted to the required number of inputs.

Expression usage

```
out = Concatenator(dimension, [in1]);
out = Concatenator(dimension, [in1, in2]);
out = Concatenator(dimension, [in1, in2, in3]);
etc.
```

Description

The `Concatenator` entity is an extent combiner; multiple inputs are concatenated along a nominated array dimension to form a wider array.

See Also

A `Combiner` entity is also an extent combiner; multiple inputs are combined to form an array of the input elements.

Generics

```
generic ElementType: type;
generic OutShape : shape;
```

The output can be any type and shape, subject to consistency with the concatenation constraints.

Constants

```
constant Dimension = 0 : natural;
```

The array dimension along which concatenation occurs defaults to the first dimension.

Ports

```
in[*] in : ElementType[**];
```

The input is a multi-port array of inputs whose shape may vary along the concatenated dimension. The type is therefore declared as of arbitrary dimensionality and subsequently constrained.

```
out out : ElementType[OutShape];
```

The output is a conventional array.

Constraints

```
//
//  Each input and the output has same dimensionality.
//
constraint : for all i in in._shape
{
    OutShape._rank = in[i]._rank;
};
//
//  Each input and the output has same extent except along Dimension.
//
constraint : for all i in in._shape
{
    for all j in OutShape where (j != Dimension)
        { OutShape[j] = in._shape[j]; };
```

```
};
//
//  Output extent along Dimension is sum of input extents.
//
constraint : OutShape[Dimension] =
    reduce add for all i in OutShape { in[i]._shape[Dimension]; };
//
//  Start indexes along Dimension are sum of preceding input extents.
//
let starts = collect i in in._shape[0]
    { reduce add for all j in in._shape[0] where (j < i)
        { in[j]._shape[Dimension]; }; };
//
//  Output element matches corresponding input element.
//
constraint : for all i in in._shape      // For each input port
{
    for all y in in[i]._type             // For each input element
    {
        for one x : natural[OutType.rank] // For the output index
            where (                      // that matches input off Dimension
            {                            // but sums along Dimension
                for all j in OutShape where (j != Dimension)
                    { x[j] = y[j]; };
                x[Dimension] = y[Dimension] + starts[i];
            })
        {
            out[x] = in[y];              // elements match
        };
    };
};
```

6.5 Constructor

Graphical usage

There should be an input for each attribute of the constructed type, and so the icon should be adapted to the required number of named inputs.

Expression usage

```
out = TypeName(in0, in1, in2, in3);
```

Description

The Constructor entity is a field combiner; a complete set of the input attributes of a type are combined to yield an output of the composite type.

Construction involves just field grouping. There is no computation involved. Perhaps constructor is the wrong name since in C++ construction is more intelligent.

Generics

```
generic RecordType : type;
```

The constructed element may be of any record type.

Ports

```
for all i in RecordType._attributes
{
   in i._name  :  RecordType.i._name;
};
```

Each input is named by and has corresponding type to an attribute of the output type..

```
out out : RecordType;
```

The output is a token, value or signal of the constructed type.

Specification

The fields of the output match the corresponding named inputs.

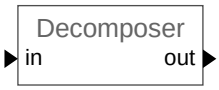
```
constraint: for all i in RecordType._attributes
{
   out.i._name = i._name;
};
```

Input flows are constrained by the requirements to perform a rendezvous of all inputs.

```
response _ins {};
```

6.6 Decomposer

Graphical usage



The output is a multi-port, and so the icon should be adapted to the required number of inputs.

Expression usage

Not applicable; multiport output.

Description

The Decomposer entity is a port splitter; a conventional array input is converted into a multi-port array output.

Generics

```
generic ElementType : type;
generic Shape : shape;
```

The converted element may be of any type (and shape).

Ports

```
in in : ElementType;
```

The input is a conventional array of the element type.

```
out[Shape] out : ElementType[Shape];
```

The output is a multi-port array of the element type.

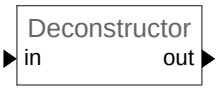
Specification

Each input port element (which need not be scalar) is copied to its corresponding array output.

```
constraint : out = in;
```

6.7 Deconstructor

Graphical usage



There should be an output for each attribute of the deconstructed type, and so the icon should be adapted to the required number of named outputs.

Expression usage

An expression cannot use a multiple output, however the dot operator may access any individual field.

```
out = in.field;
```

Description

The Deconstructor entity is a field splitter; the composite type is decomposed to yield an output for each field.

An event yields an event for each field, only one of which may be connected.

A token input yields a token for each field, all of which must be connected.

A value input yields value outputs which may be left unconnected.

A signal input yields signal outputs which may be left unconnected.

Generics

```
generic RecordType : type;
```

The constructed element may be of any record type.

Ports

```
in in : RecordType;
```

The input is of the constructed type.

```
for all i in RecordType._attributes
{
    out i._name : RecordType.i._name;
};
```

Each output is named by and has corresponding type to an attribute of the input type.

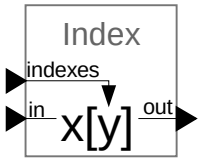
Specification

The named outputs match the corresponding fields of the input.

```
constraint: for all i in RecordType._attributes
{
    i._name = in.i._name;
};
```

6.8 Index

Graphical usage



Expression usage

```
out = Index(in, indexes)
```

Description

The Index entity performs the data parallel generalisation of the familiar indexing operation:

$q = vector[0]$ $\quad vector_{\{0\}}$

$q = matrix[0,1]$ $\quad matrix_{\{0,1\}}$

$q = matrix\left[\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}\right]$ $\quad \begin{bmatrix}matrix_{\{0,1\}}\\matrix_{\{1,0\}}\end{bmatrix}$

$q = matrix\left[\begin{bmatrix}1&5\\3&4\\2&2\end{bmatrix},\begin{bmatrix}2&3\\3&1\\0&4\end{bmatrix}\right]$ $\quad \begin{bmatrix}matrix_{\{1,2\}} & matrix_{\{5,3\}}\\matrix_{\{3,3\}} & matrix_{\{4,1\}}\\matrix_{\{2,0\}} & matrix_{\{0,4\}}\end{bmatrix}$

Generics

```
generic ElementType : type;
generic OutShape : shape;
```

The arbitrary type and shape of the output are defined by two generics, which constrain the declarations of the inputs.

```
generic InRank : natural;
generic InShape : natural[InRank];
```

The arbitrary (dimensionality and) shape of the input is expressed by two generics, so that constraints can again be expressed more directly.

Ports

```
in in : ElementType[InShape];
```

The source datum has its own shape but must be of the same element type as the output.

```
in indexes : natural[OutShape][InRank];
```

The vector of indices provides the an indexing array for each dimension of the input.

```
out out : ElementType[OutShape];
```

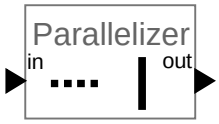
The output datum has its own shape but must be of the same element type as the input.

Specification

```
response in indexes
{
    constraint: for all x in OutShape     // For each possible output index vector
    {
        for one y in InShape           // For the corresponding input index vector
            where(for all j in OutShape.shape     // which has each of its elements
                { y[j] = indexes[j][x[j]]; })     //  matching the lookup entry
        {
            out[x] = in[y];               // Output element matches input element
        };
    };
};
```

6.9 Parallelizer

Graphical usage



Expression usage

Not applicable: input data is discontiguous.

Description

The Parallelizer entity is a time combiner; a temporal array input is converted into an array output.

See also

A Parallelizer operates on deterministic array shapes. A Recorder may be used to when the length of the temporal array is defined by control events.

Generics

```
generic ElementType : type;
generic Shape : shape;
```

The converted element may be of any type (and shape).

Ports

```
in in[Shape] : ElementType;
```

The input is a temporal array of the element type.

```
out out : ElementType[Shape];
```

The output is a conventional array of the element type.

Specification

Each input element (which need not be scalar) is copied to its corresponding array output.

```
constraint : out = in;
```

6.10 Serializer

Graphical usage

Expression usage

Not applicable: output data is discontiguous.

Description

The Serializer entity is a time splitter; a conventional array input is converted into temporal array output.

Generics

```
generic ElementType : type;
generic Shape : shape;
```

The converted element may be of any type or shape.

Ports

```
in in : ElementType[Shape];
```

The input is a conventional array of the element type.

```
out out[Shape] : ElementType;
```

The output is a temporal array of the element type.

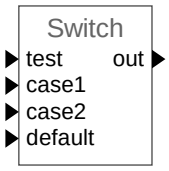
Specification

Each input array element (which need not be scalar) is copied to its corresponding sequential output.

```
constraint : out = in;
```

6.11 Switch

Graphical usage



There should be an input for each enumerator of the control type.

Expression usage

```
out = switch(test)
       {
       case case1: { case1; };
       case case2: { case2; };
       default: { default; };
       };
```

Description

The Switch entity selects one of a set of named inputs.

Generics

```
generic SwitchType : type;
```

The type of the test condition is externally defined.

```
generic CaseType : type;
```

The type passed through the switch is externally defined.

Ports

```
in test : SwitchType;
```

The output is a token, value or signal of the constructed type.

```
for all i in SwitchType._enumerators
{
   in i._name : CaseType;
};
in default : CaseType;
```

Each input is named by an enumerator of the switch type..

```
out out : CaseType;
```

The output is a token, value or signal of the constructed type.

Specification

What are the most general but safe rendezvous semantics?

7 Scheduling

After

Generates an output after a designated number of tokens have arrived. (Use `Clock` for a timed rather than counted delay).

Case

Directs an input token or event, to one of N outputs under control of another input.

Discarder

Discards the first N tokens or events and then becomes transparent.

EndCase

Merges the multiple flows resulting from a Case.

Fork

Replicates its input at N outputs. This may be realised as just a blob graphically.

If

Merge

Merges N (token or) event streams to produce a composite event stream.

PriorityScheduler

In response to a trigger event or token, passes a multi-port input token to the corresponding multi-port output. Each of the N possible inputs are considered in a priority order, index 0 first. If an input token is present at the considered input, that input is propagated. If no input token is present, the next input is considered. If no token is available at any input, a token is emitted from the nothing-to-do output.

RoundRobinScheduler

In response to a trigger event or token, passes a multi-port input token to the corresponding multi-port output. Each of the N possible inputs are considered in a cyclic order. If an input token is present at the considered input, that input is propagated. If no input token is present, the next input is considered. If no token is available at any input, a token is emitted from the nothing-to-do output.

LIFO, FIFO, Stack, ShiftRegister

CyclicDelay

for in-place FIRs etc

Sender

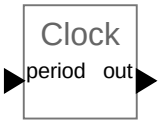
Originator and terminator of a return flow.

Receiver

Outgoing recipient and returned originator of a return flow.

7.1 Clock

Graphical usage



Expression usage

Not applicable.

Description

The Clock entity generates clock events at an interval determined by its period input, which need not be constant. A rendezvous between a clock event and a token flow at a Synch may be used to enforce real-time constraints on the token flow.

Constants

```
constant delay = 0‘s : NonNegativeTime;
```

The first clock event normally occurs instantly, but may be delayed.

```
constant latency = 0‘s : NonNegativeTime;
```

The tardiness of response to the clock event may be specified.

```
constant clock_number = 0‘s : natural;
```

The `clock_number` establishes a scheduling precedence between multiple clock sources within an event region (a region connected by event flows). Concurrent clock events are sequenced lowest `clock_number` first. It is illegal for a `clock_number` to be reused within an event region.

Ports

```
in:value period : PositiveTime;
```

The clock period.

```
out:event out : void;
```

The clock event.

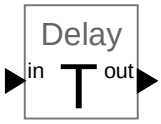
Specification

A rendezvous with a clock event forces to synchronization in time thereby enforcing a real-time constraint on all other flows, so that inputs are pre-calculated and outputs well-timed.

The first clock occurs as soon as the encompassing context is created, but may be delayed by specifying a `delay`. An infinitely tight real-time constraint cannot be met and so, some tolerance must be allowed if only to accommodate the phase noise of the system clock. The default zero `latency` avoids the need to specify a latency for untimed rendezvous, but the constraint mandates a non-zero value for real-time rendezvous.

7.2 Delay

Graphical usage



Expression usage

Not applicable; a delay has internal state.

Description

The Delay entity imposes a delay between its input and output, which must each be token flows.

Consistency between continuous time (analogue) and sampled time (DSP) usage is achieved by specifying the delay as a time, but supporting conversion from the Nyquist rate.

Generics

```
generic ElementType : type;
```

The delayed value may have any type (and shape).

Constants

```
constant delay = in.period : NonNegativeTime;
```

The delay is defaults to one clock period.

```
constant latency = 0 : NonNegativeTime;
```

The permitted late response of the output should be specified.

Ports

```
in:value initial_value = 0 : ElementType;
```

The initial value of the delay is normally zero.

```
in:token in : ElementType;
```

The input is of the generic type.

```
out:token out : ElementType;
```

The output is of the generic type.

Specification

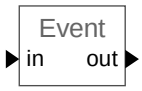
The internal state is maintained in a `stored_value` attribute which is updated whenever an `in` event occurs. The `in` input must therefore be an event or token flow.

```
response in
{
    specification discrete
    {
        attribute delay_line = initial_value : ElementType[*];
        constraint : out = delay_line[DelayLength-1];
```

```
                let shiftIndexes = Ramp(0, DelayLength-1);
                let shiftData = Index(delay_line, shiftIndexes);
                delay_line := Concatenator(in, shiftData);
            };
            specification continuous
            {
                out = in;
            };
        };
```

7.3 Event

Graphical usage



Expression usage

Not applicable.

Description

The Event entity generates one or more events from a token (or event). The generated events are sequenced in raster scan order of the multi-port output. If the input is a token rather than an event, the sequencing of the token to event conversion must be specified.

Generics

```
generic AnyInType : type;
generic AnyOutType : type;
generic AnyShape : shape;
```

The ports may be of any type.

Ports

```
in:token in : AnyInType ;
```

The input may be an `event` flow.

```
out[AnyShape]:event in : AnyOutType ;
```

The outputs may be any number (shape) of event flows in raster scan order.

7.4 Exit

Graphical usage

Expression usage

Not applicable.

Description

The Exit entity causes its parent state to exit, consequently causing any default transition to be taken.

Ports

```
in in;
```

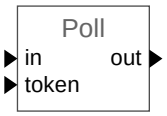
The input must be an `event` flow. The input type is unconstrained.

Specification

The specification cannot be written in WDL. Synchronization is a fundamental concept.

```
response in
{
    exit;
};
```

7.5 Poll



Description

The Poll entity conditionally propagates an event depending upon the presence of a token at an event input. This functionality may be used to avoid a compile-time error or indeterminacy that could result from an attempt to rendezvous an event and a missing token.

Generics

```
generic AnyEventType : type;
generic AnyTokenType : type;
```

The inputs may be of any type.

Ports

```
in:event in : AnyEventType;
```

The polling input must be an event.

```
in:token token : AnyTokenType;
```

The polled input must be a token flow.

```
out:event out : AnyEventType;
```

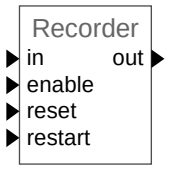
The output is an event flow.

Specification

```
response in
{
    out(token._is_present());
};
```

7.6 Recorder

Graphical usage



Expression usage

Not applicable: entity has internal state.

Description

The Recorder entity is a time combiner; a possibly discontiguous sequence of the input is buffered and propagated to the output whenever a restart occurs. The enable input may be used to suspend recording. The reset input may be used to discard recorded input.

See also

A Serializer is more suitable when the size of the source data is known.

Generics

```
generic ElementType : type;
```

The recorded element may be of any type (and shape).

Constants

```
constant initially_recording = false : boolean;
```

The initial behaviour of the recorder is determined by the initially recording configuration parameter: true to record immediately.

Constraints

The input must be a signal or token flow.

Ports

```
in in : ElementType;
```

The input is a time sequence of the element type.

```
event in restart = true : boolean;
```

The restart input transfers all recorded input to the output. Subsequent recording is enabled if the value of the event is true.

```
out:event out : ElementType[Shape];
```

The output is a conventional array of the element type gener

```
in:event enable = true : boolean;
```

A true value of an enable input event enables subsequent recording of the input flow, a false value suspends recording..

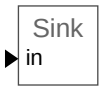
```
in:event reset = true : boolean;
```

A reset event discards all recorded input. Subsequent recording is enabled if the value if the event is true.

Specification

7.7 Sink

Graphical usage



Expression usage

Not applicable.

Description

The Sink entity discards all its input. A sink must be used to terminate unwanted token flows. A sink may be used to terminate any flow

Ports

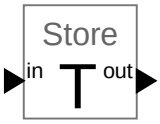
```
in in;
```

The input may be of any type (and shape) or flow.

```
response in {};
```

7.8 Store

Graphical usage



Expression usage

Not applicable; a store has internal state.

Description

The Store entity makes the most recent value of a token or event flow continuously available; a store converts a token or event flow to a value flow.

(The inverse conversion from value to token or event flow may be performed by a Synch.)

Ports

```
generic ElementType : type;
```

The stored value may have any type (and shape).

```
in : value initial_value = 0 : ElementType;
```

The initial value of a store is normally zero.

```
constant latency = 0 : NonNegativeTime;
```

The permitted late response of the output should be specified.

```
in in : ElementType;
```

The input is of the generic type.

```
out : value out : ElementType;
```

The output is of the generic type and may be accessed at any time.

Specification

The internal state is maintained in a `stored_value` attribute which is updated whenever an `in` event occurs. The `in` input must therefore be an event or token flow.

```
response in
{
    stored_value := in;
};
```

7.9 Synch

Graphical usage

The icon comprises a synchronization bar. By convention, synchronized flows (comprising an input and an identical output) should be drawn with input and output at the same level. Input only or output only flows should be unpaired. The icon should be adapted to accommodate the required inputs, outputs and clock control.

Expression usage

Not applicable.

Description

The Synch entity is the fundamental synchronization establishing a rendezvous between its inputs and outputs.

A Synch entity may be used for a wide variety of purposes:

- enforcing real-time clocking (using a Clock as an input)
- synchronizing multiple flows
- generating additional flows
- converting a value flow to a token or event flow

Ports

```
in[] in : any;
```

The input is a multi-port array of any type.

```
out[] out : any;
```

The output is a multi-port array of any type.

Specification

The specification cannot be written in WDL. Synchronization is a fundamental concept.

A rendezvous may involve

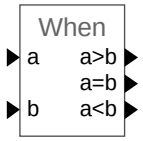
- any number of `value` or `constant` flows
- any number of `token` flows
- at most one `event` flow

Each output may be defined using an expression that may reference the values of input flows and the values of event flow returns (in addition to hierarchically visible names).

```
response _ins
{
    // Outputs defined by user constraints
};
```

7.10 When

Graphical usage



Expression usage

Not applicable.

Description

The When entity generates an event each time the associated output condition becomes true. This supports behaviour such as zero crossing detectors for continuous signals.

If an implementation chooses to use a discrete time representation of the source signals, it should be noted that the specified behaviour requires the event to be generated at the instant the continuous signals satisfied the output condition. This requires some form of interpolation to be performed, unless the sample rate is high enough to accommodate the interpolation errors in the overall implementation loss.

It is probably appropriate to incorporate some hysteresis in a practical application.

See Also

Compare may be used to return values indicating the prevailing relationship between inputs.

Generics

```
generic AnyType : type;
```

Ports

```
in:signal a : AnyType;
in:signal b : AnyType;
```

The input signals are of the same but arbitrary type.

```
out:event gt = 1: natural;
out:event eq = 0: natural;
out:event lt = -1: natural;
```

The output events indicate the required condition, and carry an identification value to facilitate a merge of the outputs.

Constraints

```
constraint AnyType._rank = 0;
```

The inputs must be scalar.

Specification

8 Generators

The family of generator entities support generation of continuous flows.

8.1 F(t)

Generates a waveform defined purely as a property of time since the existence of the encompassing state.

8.2 F1(t)

Generates a waveform defined as a property of one input and the time since the existence of the encompassing state.

8.3 Impulse, Sine, Square

signal sources

8.4 Gaussian, Uniform

noise sources

8.5 Nco

Numerically controlled oscillator

8.6 Constant

Graphical usage

Expression usage

`out = constant;`

Description

The Constant provides a defined output value. The output flow is necessarily a value flow, which may be automatically converted to a token or signal flow when required.

Generics

```
generic DataType : type;
```

The constant may be of any data type.

Constants

```
constant value : DataType;
```

The output is a token, value or signal of the constructed type.

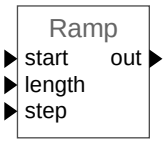
Ports

```
out : value out = value : DataType;
```

The output is a value with the specified value.

8.7 Ramp

Graphical usage



Expression usage

```
out = Ramp(1, 3, 2);          // [1, 3, 5]
```

Description

A Ramp generates a vector comprising a linear sequence, and may be used for any type of flow other than events.

For discrete flows the length is the dimension of the output vector with step determining the step size between elements/

For continuous flows the length is the duration of the output with step defining the gradient.

Generics

```
generic FlowType : flow;
```

The constant may be of any data type.

```
generic DataType : type;
```

The constant may be of any data type.

Constants

```
constant value : DataType;
```

The output is a token, value or signal of the constructed type.

Ports

```
in : FlowType start = 0 : DataType;
```

The output is a value with the specified value.

```
in length = 0 : FlowType._length;
```

The length has the appropriate type for the flow.

```
in step = 1 : DataType / FlowType._length;
```

The step has the appropriate gradient for the flow.

```
out: FlowType out = value : DataType[length];
```

The output is a vector of the specified length

```
response start length step
{
    constraint : out = Ramp(start, length, step);
};
```

9 Higher Order Entities

A higher order entity uses an entity as an input.

9.1 ArrayOf

The ArrayOf entity provides a graphical notation for the Wdl statement.

```
attribute name : type[shape];
```

10 Utilities

Integrator

Returns the sum of its input over time.

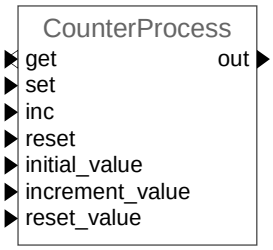
This is a particularly interesting/challenging block to specify because it should somehow demonstrate the compatibility of discrete and continuous interpretation of the signal flow type in spite of the distinct forms of integration and time skew of the discrete case.

ModuloIntegrator

As Integrator but using wrap-around arithmetic.

10.1 CounterProcess

Graphical usage

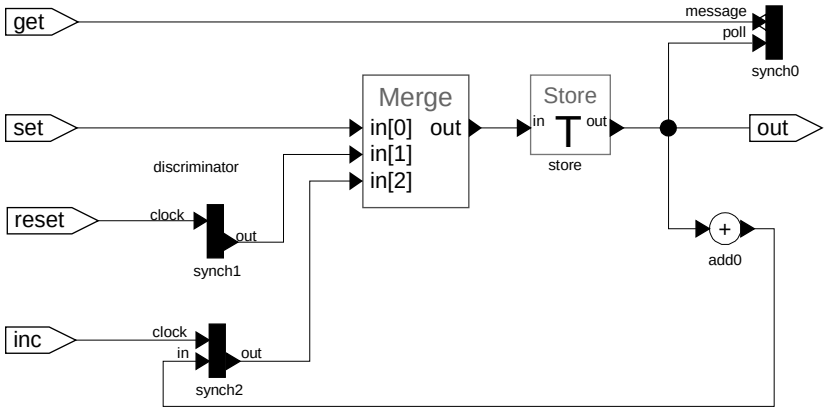


Expression usage

Not applicable; entity has internal state.

Description

The CounterProcess entity provides a continuously available output value that may be set and accessed via its message input. Additional counter control is supplied by the increment and reset inputs.



The value is maintained by the Store, with the `set` message passing straight through. The `get` message performs a rendezvous at the synchronization bar to obtain the value from the Store.

The `reset` input performs a rendezvous with the `reset_value` to generate an alternate Store update message.

The `inc` input performs a rendezvous with the incremented store value to provide a third alternative store update.

All three possible store update events are merged by the Merge.

Generics

CounterType

```
generic CounterType :  type;
```

Any type may be used for the counter.

Ports

```
in : value increment_value = 1 : CounterType;
in : value initial_value = reset_value : CounterType;
in : value reset_value = 0 : CounterType;
```

The initial, reset and increment values may be specified.

```
in : event get : void : CounterType;
in : event set : CounterType;
```

Get and Set message interaction is supported.

```
in : event inc : void;
```

The counter may be incremented (by the `increment_value`)

```
in : event reset : void;
```

The counter may be reset (to the `reset_value`)

```
out : value out : CounterType;
```

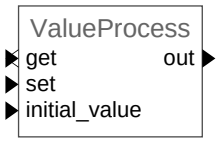
The output is of the generic type and may be accessed at any time.

Specification

```
constraint : store.initial_value = initial_value;
constraint : synch0.message = poll;
constraint : synch1.out = reset_value;
constraint : synch2.out = in;
constraint : add0.in[1] = increment_value;
```

10.2 ValueProcess

Graphical usage

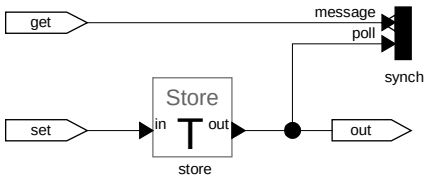


Expression usage

Not applicable; entity has internal state.

Description

The ValueProcess entity provides a continuously available output value that may be set and accessed via its message inputs.



The value is maintained by the Store. The the SET message passes straight to the Store. The GET message performs a rendezvous at the synchronization bar to obtain the value from the Store.

Generics

```
generic ElementType : type;
```

The maintained value may have any type (and shape).

Ports

```
in : value initial_value = 0 : ElementType;
```

The initial value of a store is normally zero.

```
in : event get : void : ElementType;
in : event set : ElementType;
```

Get and Set messages are supported.

```
out : value out : ElementType;
```

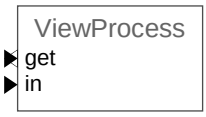
The output is of the generic type and may be accessed at any time.

Specification

```
constraint : store.initial_value = initial_value;
constraint : synch.message = synch.poll;
```

10.3 ViewProcess

Graphical usage

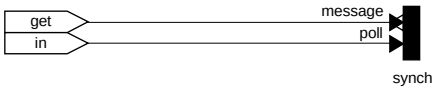


Expression usage

Not applicable.

Description

The ViewProcess entity provides polled interrogation of a value.



The GET message performs a rendezvous at the synchronization bar to obtain the value.

Generics

```
generic ElementType : type;
```

The maintained value may have any type (and shape).

Ports

```
in : event get : void : ElementType;
```

A Get message is supported.

```
in : value in : ElementType;
```

The value to be polled must be continuously available.

Specification

```
constraint : synch.message = synch.poll;
```

11 Special Purpose

11.1 Codecs

Codecs support the encoding and matching decode of data. There are a wide variety of alternative algorithms available, and these are to some extent interchaneable. The library therefore provides an abstract encoder and decodr that may be remotely paramterised to support a particular application requiremernt.

11.1.1 Abstract Codecs

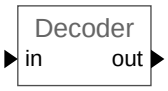
11.1.1.1 CodecParams

The abstract interface of any Codec is defined by CodecParams, and requires that entities be identified whose interfaces complies with the Encoder and Decoder interfaces.

```
record CodecParams
{
    generic EncodingEntity : entity = Encoder;
    generic DecodingEntity : entity = Decoder;
};
```

11.1.1.2 Decoder

Graphical usage



Expression usage

out = Decoder(in, params)

Description

The many different decoder behaviours are selected by the appropriate parameters. The Decoder therefore delegates its specification to the entity identified by the parameters.

Generics

```
generic Params : type = CodecParams;
```

Ports

```
in in;
in params : Params;
out out;
```

Specification

It is not clear how to express this graphically. It looks suspiciously like the curiously recurring template pattern, but there is no entity inheritance to help out.

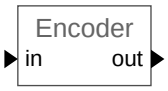
In Wdl it is:

```
entity Decoder
{
    generic Params : type = CodecParams;
    in in;
    in params : Params;
    out out;
    entity decoder : Params.DecodingEntity;
    relation r1 { link in; link decoder.in; };
    relation r2 { link out; link decoder.out; };
};
```

The absence of any flow, type or shape specification is left to be resolved by connection to the embedded decoder.

11.1.1.3 Encoder

Graphical usage



Expression usage

out = Encoder(in, params)

Description

The many different encoder behaviours are selected by the appropriate parameters. The Encoder therefore delegates its specification to the entity identified by the parameters.

Generics

```
generic Params : type = CodecParams;
```

Ports

```
in in;
in params : Params;
out out;
```

Specification

It is not clear how to express this graphically. It looks suspiciously like the curiously recurring template pattern, but there is no entity inheritance to help out.

In Wdl it is:

```
entity Encoder
{
    generic Params : type = CodecParams;
    in in;
    in params : Params;
    out out;
    entity encoder : Params.EncodingEntity;
    relation r1 { link in; link encoder.in; };
    relation r2 { link out; link encoder.out; };
};
```

The absence of any flow, type or shape specification is left to be resolved by connection to the embedded encoder.

11.1.2 Reed Solomon Codecs

11.1.2.1 RsParams

The RsParams type defines the interface of the type to define an RsEncoder or RsDecoder.

```
record RsParams : CodecParams
{
    generic Symbol : type = GaloisField;
    generic user_symbols : value = natural;
    attribute polynomial = 2 : Symbol;
    constant parity_symbols = polynomial.Order : natural;
    constant net_symbols = user_symbols + parity_symbols :
natural;
};
```

The data type for each `Symbol` must comply with the `GaloisField` interface.

The code size is characterised by a number of `user_symbols` and a code generator polynomial.

The number of `parity_symbols` is determined by the polynomial length, and the `net_symbols` is the sum of the user and parity symbols.

11.1.2.1.1 SystematicRsParams

The parameter set for a Systematic Reed Solomon Codec enforces the use of the Systematic encoder and decoder.

```
record SystematicRsParams : RsParams
{
    constraint : DecodingEntity = SystematicRsDecoder;
    constraint : EncodingEntity = SystematicRsEncoder;
};
```

11.1.2.1.2 TransformRsParams

The parameter set for a Transform Reed Solomon Codec enforces the use of the transform encoder and decoder.

```
record TransformRsParams : RsParams
{
    constraint : DecodingEntity = TransformRsDecoder;
    constraint : EncodingEntity = TransformRsEncoder;
};
```

11.1.2.2 RsDecoder

Graphical usage

Expression usage

out = RsDecoder(in, field)

Description

A Reed Solomon decoder returns the user data applicable to a codeword from amongst the multiplicity of valid encoded codewords which (ignoring erasure locations) has the least number of symbols different from the input data. This is usually the most likely codeword.

There are three main algorithms used as part of the decoding of Reed Solomon codes. These are the Berlekamp-Massey, the Euclid and the Welch-Berlekamp. Where the error conditions are within the constraint, these decoders produce the same result, but need not produce the same result without that constraint. The Welch-Berlekamp can provide a more efficient route to a higher performance than the other two if quality information is provided with symbols, in that erasures may be selected according to those quality measures, and sequentially applied starting with the weakest symbol in order to give the best chance of meeting the decode constraint. This can be done with the other two algorithms, but requires a much greater degree of processing. Maximum likelihood decoding of quality information at bit rather than symbol level is not usually attempted due to the excessive processing required.

Generics

```
generic Params : type = RsParams;
generic Symbol = Params.Symbol : type = GaloisField;
```

The characteristics of an RsDecoder are specified by a type that complies with the `RsParams` interface, and a symbol type that complies with the `GaloisField` interface.

Types

The RsDecoder operates on an array of values describing each symbol. For the purposes of specification this value must support two concepts: the actual symbol and an erasure predicate:

```
record RawSymbol
{
    generic Symbol : type;
    operation is_erased() : boolean;
    operation symbol() : Symbol;
};
```

Actual implementations may maintain erasure context as a boolean threshold, per symbol probability or per-symbol bit probability.

Ports

```
in in : RawSymbol[Params.net_symbols];
out out : Symbol[Params.user_symbols];
out erasures : natural;
out errors : natural;
out probability : real;
```

The configuration must satisfy requirements of the Reed Solomon algorithm

```
constraint : (1 << Symbol.degree) >= Params.net_symbols;
```

The way in which the decoded symbols are derived is not specified, since there are a variety of valid implementations. The decoder is therefore specified with respect to an encoder.

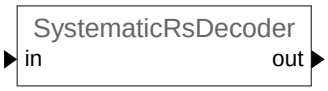
```
let encoded = RsEncoder(out, params);
```

Symbol discrepancies between the encoding of the decoded symbols and the raw inputs are errors.

```
let erroneous_symbols = collect for all i in in
       where (in[i].symbol() != encoded[i]) { i; };
constraint : errors = erroneous_symbols.shape[0];

let erased_symbols = collect for all i in in
       where (in[i].is_erased()) { i; };
constraint : erasures =  erased_symbols.shape[0];

let remnant_symbols = collect for all i in in
   where (!in[i].is_erased() & (in[i].symbol() != encoded[i]))
       { i; };

let remnants = remnant_symbols.shape[0];

constraint : 2 * remnants + erasures <= parity_symbols;
```

11.1.2.2.1 SystematicRsDecoder

Graphical usage



Expression usage

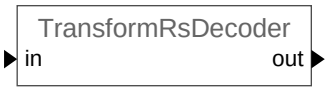
```
out = SystematicRsDecoder(in, params)
```

Description

A systematic Reed Solomon decoder complies with the specification of an RsDecoder for the specific case of an encoding by a SystematicRsEncoder.

11.1.2.2.2 TransformRsDecoder

Graphical usage



Expression usage

```
out = TransformRsDecoder(in, params)
```

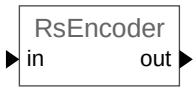
Description

A systematic Reed Solomon decoder complies with the specification of an RsDecoder for the specific case of an encoding by a TransformRsEncoder.

```
out out : Symbol[UserSymbols + ParitySymbols];
```

11.1.2.3 RsEncoder

Graphical usage



Expression usage

out = RsEncoder(in, params)

Description

An RsEncoder is not a unique concept. The distinct systematic and transform behaviours are selected by the appropriate parameters. The RsEncoder therefore delegates its specification to the entity identified by the parameters.

Generics

```
generic Params : type = RsParams;
generic Symbol = Params.Symbol : type = GaloisField;
```

Ports

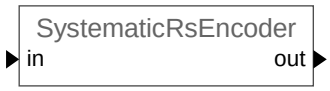
```
in in : Params.Symbol[Params.user_symbols];
in params : Params;
out out : Params.Symbol[Params.net_symbols];
```

Specification

See Encoder.

11.1.2.3.1 SystematicRsEncoder

Graphical usage



Expression usage

```
out = SystematicRsEncoder(in, params)
```

Description

A Reed Solomon encoder adds parity symbols to a vector of input symbols so that a subsequent decode can detect and correct errors. Symbols need not be boolean.

The output of a systematic Reed Solomon encoder comprises a copy of its input vector with parity symbols appended as part of a larger output vector.

Generics

```
generic Params : type = SystematicRsParams;
```

Ports

```
in in : Params.Symbol[Params.user_symbols];
in params : Params;
out out : Params.Symbol[Params.net_symbols];
```

Specification

```
let user_symbols = Params.user_symbols;
let parity_symbols = Params.parity_symbols;
let zero_fill = Ramp(0, parity_symbols, 0);
let u = Polynomial(Concatenator(zero_fill, in));
let p = u % Params.polynomial;
let user_indexes = Ramp(0, user_symbols, 1);
let parity_indexes = Ramp(user_symbols, parity_symbols, 1);
constraint : Index(out, user_indexes) = in;
constraint : Polynomial(out, parity_indexes) = p;
```

11.1.2.3.2 TransformRsEncoder

Graphical usage

Expression usage

```
out = TransformRsEncoder(in, params)
```

Description

A Reed Solomon encoder adds parity symbols to a vector of input symbols so that a subsequent decode can detect and correct errors. Symbols need not be boolean.

The output of a transform Reed Solomon encoder comprises the result of a transform.

Generics

```
generic Params : type = TransformRsParams;
```

Ports

```
in in : Params.Symbol[Params.user_symbols];
in params : Params;
out out : Params.Symbol[Params.net_symbols];
```

Specification

```
let user_symbols = Params.user_symbols;
let parity_symbols = Params.parity_symbols;
let pre_fill = Ramp(0, z, 0);
let post_fill = Ramp(0, parity_symbols, 0);
let u = Polynomial(Concatenator(pre_fill, in, post_fill));
constraint : for all j in net_symbols
    { out[j] = reduce add for all i in net_symbols
        { u[i] * pow(Params.polynomial, i * j); }; };
```

11.2 Filters

Fir

Iir

Viterbi

11.3 Modems

Modulator

Demodulator

Bpsk

Qpsk

Qam

11.4 Transforms

Dct

Dft

Fft

Idct

Idft

Ifft

12 Test aids

12.1 GUI Sources

useful widgets for interactive/logged simulations

12.2 GUI Sinks

useful widgets for interactive/logged simulations

Index

_after 6
_component 6
_inertia 6
_processor 6
_runaway 6
Abs 7
AbsSquared 7
Acos 7
Acosh 7
Add 7, 9
After 26
And 7
angles 3
ArrayOf 41
Asin 7
Asinh 7
Atan 7
Atanh 7
boolean 4
Bundler 14
Case 26
Ceil 7
Clock 27
Commutator 11
Compare 10
complex 4
Complex 7
Composer 15
Concatenator 16
Conj 7
Constructor 18
Cos 7
Cosh 7
CyclicDelay 26
Decomposer 19
Deconstructor 20
Delay 28
dimensions 3
Discarder 26
Discriminator 11
Distributor 11
Div 7
DownSampler 12
EndCase 26
Event 30
Exit 31
Exp 7
FIFO 26
Find 13
Floor 7
Fork 26
Frac 7
Function 7
GaloisField 4
Harmonizer 11
Idiv 7
If 26
Imag 7
Index 12, 21
Int 7
integer 4
LIFO 26
Limit 7
Log10 7
Log2 7
Loge 7
Matmul 7
Max 7
Merge 26
Min 7
Mod 7
Mul 7
natural 4
Neg 7
Nibbler 12
Not 7
Or 7
Parallelizer 11, 23
Partitioner 12
Poll 32
Polynomial 5
Pow 7
PrimitivePolynomial 5
PriorityScheduler 26
quadrant 3
Quant 7
real 4
Real 7
Receiver 26
Recip 7
Recorder 33
Reducer 7
Repeater 12
Reverser 12
RotateLeft 7
RotateRight 7
Round 7
RoundRobinScheduler 26
Sender 26
Serializer 11, 24
ShiftLeft 7
ShiftRegister 26
ShiftRight 7
Sign 7
Sin 7
Sinc 7

Sinh	7	Tanh	7
Sink	34	time	3
Sqrt	7	Time	3
Stack	26	Transposer	12
Stacker	12	Trunc	7
Store	35	types	4
Sub	7	Unnibbler	12
Switch	12, 25	UpSampler	12
Synch	36	When	10, 37
Tan	7	Xor	7