Databoard Specification

Databoard 0.6.0 Specification

DataBoard is a software library built upon a simple but well formulated and expressive type system. The design is a compromise of expression power, advanced functions, and performance.

Contents

Datatypes

Datatype is a type system. There is support for structural and primitive data values, unit types and restrictions.

Primitive datatypes define just a set of valid values:

#	Datatype	Description
0	Boolean	true and false
1	Byte	signed 8-bit integers
2	Integer	signed 32-bit integers
3	Long	signed 64-bit integers
4	Float	32-bit IEEE 754 floating point numbers
5	Double	64-bit IEEE 754 floating point numbers
6	String	Unicode strings of arbitrary length

Derived datatypes are constructed from other datatypes using datatype constructors.

#	Datatype	Description
7	Record	Contains constant set of fields
8	Array	An ordered collection of values
9	Map	An object that maps keys to values
10	Optional	A container that either has or does not have a value

11	Union	A choise between component types.
12	Variant	An object that can contain a value of any type

In textual representation types are written with *type definitions* `type <name> = <type>`. Databoard Type definition file (*.dbt*) is a text file with a list of type definitions. The builtin types are denoted by their names (`Boolean`, `Byte`, `Integer`, `Long`, `Float`, `Double`, `String`, `Variant`)

```
type Name = String
type Length = Integer
```

Complex type definitions and in particular recursive ones can be defined in pieces by giving names to types:

```
type NodeDescription = referable { name : String, children : NodeDescription[] }
```

All other type constructors have form $C(T_1, \ldots, T_k)$, where C is a type constructor and parameters are datatypes. The following constructors are defined:

```
type Example = Optional( BaseType )
```

Parametrised type constructors can be defined as follows:

```
type Tree(A) = | Leaf A | Node referable { left : Tree(A), right : Tree(A) }

type Sample(Value) = { time : Double, value : Value }
```

Annotation

Annotations add meta data to a base type. They never affect to the set of well-formed values of the type but may restrict the set of valid values.

Different primitive types and datatype constructions support different annotations:

types/constructor	annotations supported
Byte, Integer, Long, Float, Double	range, unit
String	pattern, mimeType, length
Array	length
Record	referable

```
type value = Int(range=[1..10000])
type Probability = Double(min=[0..1.0])
type XML = String(mimeType="text/xml")
type Html = String(pattern="^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?", length=[..4096])
```

Built-in types can be annotated. Annotations are written as $T(key_1=value_1, \ldots, key_k=value_k)$, where key_i is an identifier and $value_i$ a value of some built-in type (depending on the annotation).

The following annotations are defined:

- all numerical types
  - unit : String
  - range: Range
- String
  - pattern : String (regular expression pattern of allowed strings, case sensitive)
  - mimeType : String
  - length : Range

```
type Size = Int(range=[1..10000], unit="m")
type Amplitude = Double(range=[-1.0..1.0])
type Frequency = Double(unit="1/s")
type Document = String(mimeType="text/xml")
```

Value Text Notation

Value if (*.dbv*) is a text file that contains a single data value in text format. The type must be known to the reader.

Value definition file is a text file (*.dbd*) that contains a list of value definitions. There is a name, type, and value in a *Value definition* in the following format: `<name> : <type> = <value>`.

It is assumed that the datatype of the value written in textual format is always known in the context. Thus it is not necessary to be able to completely infer the type from the value notation itself.

```
obj1 : Integer = 5
obj2 : { name : String } = { name = "ABC" }
obj3 : Node = { id = "123", parent = obj4 }
```

Strings are written by enclosing the string in double quotes. The special characters in the strings are escaped following Java-specification. [1]

```
"some string"
```

```
"string with special characters such as:\n - \\\n - \"\n"
```

Long strings containing special characters and new lines are enclosed in triple double quotes:

```
"""Long string
spanning multiple
lines"""
```

Characters cannot be escaped in this notation (?).

Integers are specified with a sequence of digits preceded by an optional '-'. *Floating point numbers* can contain also dot and exponent. The exact syntax follows the Java specification for *int* and *double* literals.

```
1
-345
3.1415
1e-10
```

Records

A record type is a sequence of components. Each component has a type and a name. The name is a unrestricted Unicode string. The set of (well-formed/valid) values of the record type is a cartesian product of the (well-formed/valid) values of the component types.

A record type is constructed as $\texttt{\{ f}_1\texttt{ : T}_1\texttt{, ..., f}_k\texttt{ : T}_k\texttt{ \}}$, where $\texttt{f}_i$ is a field name and $\texttt{T}_i$ its datatype. A field name cannot be empty and two field names cannot be equal. Field names are usually written in lowercase.

```
type Color = { red : Double,
               green : Double,
               blue : Double }
```

The fields of a record value are enclosed in curly brackets {} and separated by comma. Each field consists of the field name, equality mark and the value.

```
pink : Color = { red = 1.0, green = 0.4, blue = 0.4 }
```

Long field names are escaped with single quotes. The special characters in the strings are escaped following Java-specification. [2]

```
type Example = { 'long field name' : Double }
value : Example = { 'long field name' = 5.0 }
```

Referable Records

Recursion in Datatypes is based on referable records. Referable record is a record prefixed with keyword `referable`.

```
type Tree = referable { children : Tree[] }
```

Each referable value has a separate value definition. Value definition has name, type and value in the following format: *<name> : <type> = <value>*. Referable values are referred by name.

```
root : Tree = { children = [ node1, node2 ] }
node1 : Tree = { children = [] }
node2 : Tree = { children = [] }
```

Tuple Type

A tuple type is a special case of record type where all components have empty names. The construction is in the following format (T_1, ..., T_k).

```
type Vector = (Integer, Integer, Integer)

vec1 : Vector = (1, 2, 3)
```

When exactly one value is enclosed in parenthesis, the parenthesis are interpreted as grouping not as a tuple. Thus the following two lines are equal:

```
(34)
34
```

Unions

A union type is defined with a sequence of components similarly as a record type. The set of (well-formed/valid) values of the union type is disjoint union of the (well-formed/valid) values of the component types.

A union type is constructed as $\texttt{| n}_1\texttt{ T}_1\texttt{ | ... | n}_k\texttt{ T}_k$, where $\texttt{n}_i$ is a tag name and $\texttt{T}_i$ its datatype. The tag type is optional and is assumed to be `{}`, if left out. Tag names have to be non-empty and distinct. Tag names are usually capitalized.

```
type Color = | RGB (Float, Float, Float)
             | RGBA (Float, Float, Float, Float)
```

Enumerations are also unions. The type names are left out. Name of the type is used as the tag name.

```
type Method = | Disabled | Adaptive | Manual

type CommandResponse = | Success
                       | Error String
```

A tag may have the same name as a builtin datatype

```
type Example = | Double Double | Long Long
```

The value consists of the union tag name followed by a value:

```
result : CommandResponse = Error "The method call failed."
white : Color = RGBA (1,1,1,0)
```

Long union tags are escaped with single quotes. The special characters in the strings are escaped following Java-specification. [3]

```
type Example2 = | 'long union name' (1,1,1)
```

Arrays

An array type is constructed with a base type. Its values are finite sequences of the base type.

The textual notation for the array type construction is `T[]`, where `T` is the base type.

```
type VGA = Double[320][240]
type Names = String[]
```

Minimum and maximum length of the array can be specified as `T[a..]`, `T[..b]`, `T[a..b]` or `T[a]`, where `a` and `b` are integers and $a \leq b$.

- Exact Value `Double[0]`
- Unlimited `Double[]`
- Limited `Double[..100]`, `Double[10..100]`, `Double[10..]`

The values are enclosed in brackets [] and separated by comma.

```
image : Names = ["a", "b", "c"]
```

Maps

A map type is constructed with a base type of key and value. Its values are finite sequence of entries of key-value pairs. Keys may not refer even indirectly to the map itself.

Map type is defined as `Map(K, V)`, where `K` is key datatype, and `V` is value datatype.

```
type TimeSeries = Map( Long(unit="ms"), Double );
type PropertyMap = Map( String, String )
```

Map value is a collection of entries. They are enclosed in curly brackets {} and separated by comma (,). Each entry consists of the key name, equals (=) mark and the value.

```
properties : PropertyMap = map { Name = "Somename", Id = "6.0" }
```

Long field names are escaped with single quotes. The special characters in the strings are escaped following Java-specification. [4]

```
properties : PropertyMap = map { "string key name" = "5.0", "another key name" = "6.0" }
```

Variants

Variant consists of type and a value. If is defined as *`type : value`*.

```
50 : Integer
{x=50, y=50, z=50} : { x:Double, y:Double, z:Double }
(50, 50, 50) : { x:Double, y:Double, z:Double }
```

Type can be omited for strings and booleans

```
"Hello World" : String
"Hello World"
true : Boolean
true
```

Type can also be omited for numbers with the following rule. If there is a full stop (.) then the type is a Double, otherwise an Integer.

```
5.0 : Double
5.0
5 : Integer
5
```

String Binding

Variant string binding is a filename and URL compatible serialization format of variant values. Values have the following encoding:

```
S<string>       String type (without parameters)
                Control characters " : < > | ? * \ / % # [0..31] [128..] are escaped as %<hex><hex>
                " "-space is _
I<integer>      Integer type (without parameters)
L<long>         Long type (without parameters)
B<base64>       All other cases. The string is Base64 encoding of a binary encoded variant.
                Base64 encoding has url and filename safe options enabled.
```

Optional

An optional type is constructed with a base type. Its values are the values of the base type and a special value *null*.

```
type Name = Optional( String )
exmpl1 : Name = "Hei"
exmpl2 : Name = null
```

Record fields of optional type can be omited, if there is no value.

```
type Example = {
         name : Optional (String)
     }
exmpl : Example = {}
exmp2 : Example = { name = "abc" }
```

Sub-typing

Datatypes do not support explicit subtyping (most of its uses can be replaced by using Union-constructor), but it can be defined implicitly: a datatype A is a *subtype* of B, if all valid values of A are also valid values of B. Primitive datatypes and type constructors are defined so that if A is a subtype of B then they have the same set of well-formed values.

Validation

Datatype is a mathematical object that is associated with two sets: *well-formed* and *valid* values, where the former set contains the latter. A datatype is *basic* if its well-formed values are all valid. *Primitive datatypes* are basic data types that are completely defined by their set of valid values. There is only a finite set of primitive datatypes. *Derived data* types are constructed from other datatypes by certain datatype constructors we will define. They may contain also some metadata besides defining the set of valid values.

The distinction between well-formed and valid values can be demonstrated with the following example: "`http://www.simantics.org/`" is a valid value of datatype URI. "`foo::bar`" is not a valid URI, but it is well-formed i.e. it can be represented in all contexts where URIs are used. The number 34 is not a well-formed URI. Similarly, if Probability is defined as a floating point number between 0 and 1, then 1.5 is well-formed but not a valid value of `Probability`.

Comparison

Two data values can be compared and put into order. There is a rule for each type. Values are compared structurally.

Type	Compare-function
RecordType	Field by field in order*
UnionType	by tag, if equals then by value. **DataTypes** have the following order: ArrayType, BooleanType, ByteType, IntegerType, LongType, FloatType, DoubleType, OptionalType, RecordType, StringType, UnionType, VariantType, MapType
NumberType	compare values
ArrayType	compare lengths, then elements
OptionalType	Compare *HasValue*, then *Value*
StringType	Case-sensitive String compare
BooleanType	zero if values are the same; a positive value if the first value represents true and the second false; a negative value if the first is false and send true
VariantType	compare type, then value
MapType	1. Compare Sizes, 2. Eliminate method : pair comparison of highest entries. Take highest key entries of both maps. Compare key, compare value. If equal go to the next entry, else use the compare value of (key then value).

Default Value

There is a default value for all datatypes.

Type	Default Value
RecordType	each field according to type.
UnionType	tag 0 with default value of the composite type
Byte, Integer, Long, Float, Double	0 or min limit if range exists
MapType	no entries

Array	minimum number of entries where each entry has the default value of the composite type.
OptionalType	Compare *HasValue*, then *Value*
String	"" or minimum value if pattern exists
Boolean	false
Variant	{} : void

Hash Function

Hash function produces a 32-bit integer. There is a hash function for each type of value:

Type	Hash-function
Boolean	true=1231, false=1237
Integer	value
Long	lower 32-bits ^ higher 32-bits
Float	*IEEE 754 floating-point "single format" bit layout* as is.
Double	lower 32-bits ^ higher 32-bits of *IEEE 754 floating-point "double format" bit layout*.
Optional	no value = 0, else `hash(value)`
Array	`int result = 1; for (int element : array) result = 31 * result + hash(element);`
Record	`int result = 3; for (field : record) result = 31 * result + hash(field)*;`
Variant	`hash(type) + hash(value)`
Union	`tag + hash(value)`
Map	`int result = 0; for (entry : map) result += hash(key) ^ hash(value);`
Byte	value

```
*) In case of recursion, where the hash-function enters an referable data value a second/consecutive time,
   0 is considered as the hash value.
```

File types

Extension	Description
`.dbb`	Databoard Binary File
`.dbt`	Databoard Type Definition File. Consists of type definitions in text format.
`.dbd`	Databoard Value Definition file. Consists of value definitions in text format.
`.dbv`	Databoard Value File. Contains a single value in text format without type information.

Interfaces

A function type can be constructed as `D -> R`, where `D` is the domain and `R` the range of the function. If the function can throw exceptions, it is denoted as `D -> R throws` E_1`, ...,` E_k, where E_i is a datatype for an exception. Multi-parameter types can be defined using the tuple notation:

```
Double -> Double
() -> String
(Integer,Integer) -> Integer
Integer -> Integer throws IndexOutOfRange
```

A method definition is a combination of name and method type. The format is following, `method M : T`, where `M` is the name and `T` the type of the method. `T` has to be a function type.

```
method getSamples : TimeSegment -> Sample[],
method read : ReadRequest -> Sample,
method getValueBounds : TimeSegment -> Sample[2]
```

Interface is a specification of fields and methods the interface has to have. The interface is defined as a record that contains fields and methods definitions in the curly braces.

```
interface HistoryRecord = {
          records : TimeSeries,

          method getSamples : TimeSegment -> Sample[],
          method read : ReadRequest -> Sample,
          method getValueBounds : TimeSegment -> Sample[2]
      }
```

Interfaces may extend other interfaces. This is denoted as `interface I extends` B_1`, ...,` B_k `{ ... }`.

```
interface MutableHistoryRecord extends HistoryRecord = {
          method write : Sample[] -> {},
          method clear : {} -> {}
      }
```

Binary File Format

Abstract mathematical objects cannot be manipulated with computers if they are not represented somehow. A *serialization* is a definition of how all well-formed values of a certain datatype are represented as byte sequences. A *serialization format* associates some datatypes with a unique serializer.

Values can be serialized into files in binary format. Databoard Binary (.dbb) file contains one single value. The file a concatenation of type and value serialization. It also means that the file is a serialization of of variant.

Binary Serialization Format

There is a *binary serialization* notation which is format used in files and network communication. The same notation is used for both datatype and data value communication. This is possible as datatype is also a **value**; a value of `DataType`. There is a notation for each data value. All numeric values are in Big Endian order (aka Network byte order).

Boolean

Boolean is an `UINT8`, with one of the following values.

Value	Description
`0`	false
`1`	true
`2..255`	*Invalid value*

Numbers

Type	Description
Byte	Int8
Integer	Int32
Long	Int64
Float	Float
Double	Double

Optional

There is a Boolean that describes whether there is an actual value. If false, there is no data to follow, if true, actual value follows.

Field	Description
`hasValue : Boolean`	Tells whether there is a *value*
value	The actual value, available only if hasValue == true.

String

String is a series of bytes encoded as Modified-UTF-8.

Field	Description
`length : Length`	Describes the number of bytes in the string using #Length encoding (1-5 bytes).
data	Modified-UTF-8 encoded String.

The length is encoded as UInt32 of 1 to 5 bytes.

Value	Encoding
`0x00000000..0x0000007F`	*value* (1 byte)
`0x00000080..0x00003FFF`	0x80, value>>6 & 0xff *(2 bytes)*
`0x00004000..0x001FFFFF`	0xC0, value>>5 & 0xff, value>>13 & 0xff *(3 bytes)*
`0x02000000..0x0FFFFFFF`	0xE0, value>>3 & 0xff, value>>12 & 0xff, value>>20 & 0xff *(4 bytes)*
`0x10000000..0xFFFFFFFF`	0xF0, value>>3 & 0xff, value>>11 & 0xff, value>>19 & 0xff, value>>27 & 0xff *(5 bytes)*

Array

Field	Description
`length : UInt32`	Describes the number of elements in the array using #Length encoding. This field is omited if the range of the array is constant.
<elements>	Array elements

Generic Record

Field	Description
<fields>	Component fields in the order specified in datatype.

Referable Record

Field	Description
`recordId : Integer`	Identity that refers in this serialiation to this instance.
<fields>	Component fields in the order specified in datatype.

Generic Union

Field	Description
`tag : Byte, Short or Integer`	Number that indicates that type in the UnionType's *components* array. Type depends on the number of cases.
value	The value of the component type.

Variant

Field	Description
`type : DataType`	Describes the datatype of the following value.
value	The value serialized according to the type

Map

Field	Description
`length : UInt32`	Describes the number of entries in the map using #Length encoding.
<entries>	Map Entries
`key : K`	The key serializes according to the Key type
`value : V`	The value serialized according to the Value type

Entries are in ascending ordered by key (See #Order)

Value Reference

Value Reference is a URI compatible string that represents a in-value path from a structure to a sub-structure. For example, to an element of an array (index) or map (key), or record field (field-name).

There are explicit references and label references. Explicit references are typed, they specify the datatype where the reference is applicable. for example "i-5" is read "Array index 5", or "n-name" is "Record field name".

Label references are more human readable, but must be used in correct context to be usable. For example reference "5" is ambiguous, it can mean array index 5, or map element by key 5:integer, or record field by name "5" - the reference depends on the data where it is applied.

Node Type	Child Reference String Notation
Index Reference (Array, Union, Record)	*i-<index>*
Key Reference (Map)	*k-<key>**
Name Reference (Record, Union)	*n-<field name>***
Component Reference (Optional, Union, Variant)	v
Label Reference (Array, Union, Record, Optional, Variant)	<string>**

```
*) Key is ascii serialized with Variant String encoding
**) Names are escaped using URI escape rules [5]
```

Path separator is /, for example: *nodes/SSINE/value/o/v*

Contracts

History Contract

The word "History" is overloaded; it has been understood as a database, as a process, a recording of an experiment, or a time series of a variable. We use it to mean everything mentioned, a collection of concepts related to handling non real-time data.

Sampling is a produre where samples are collected and recorded from real-time variable context, a DataSource. The output is a *SamplingResult*.

A step is a virtual timecode. It starts at 0 and increments on every sampling. On each step, one or more variables are sampled. There is a time record that contains datasource's timestamps, a corresponding (time)sample for each used virtual timecode. It is used for mapping virtual codes to actual timestamps.

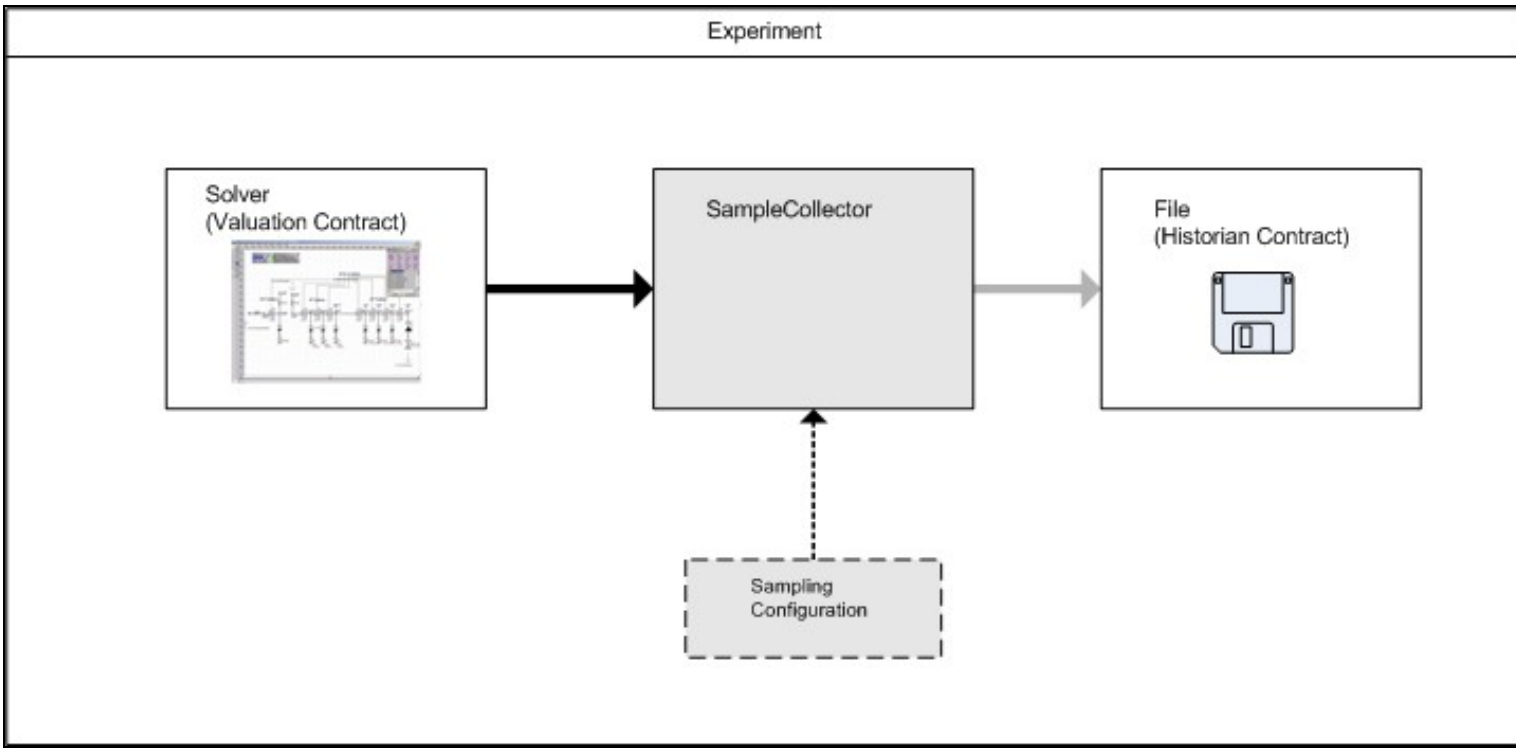
Record has two representation formats: `SPARSE` and `DENSE`. In a DENSE record there is a sample for every step, and in a SPARSE samples may not have been collected on every step.

There is a contract for the data format of recorded variables:

```
// A result of a data capturing session
type RecordingSession = {
```

```
       // An optional Datasource URL
       datasource : Optional( String ),

       // Events that occured in the data source during sampling, a map of Event Id to Event
       events: Map( Integer, Event ),

       // A collection of events that are promoted to milestones, a map of Milestone Id to Event Id
       milestones : Map( Integer, Integer ),

       // All records, a map of NodeId to Recording
       recordings : Map( Variant, Variant )
     }

// The captured data for a single variable
type Recording(T, V) = {
       // Record Identifier
       nodeId : Variant,

       // All labels
       labels : LocalizedTexts,

       // All Segments, segments are time series
       segments : Map(T, V)[]
     }

type Event = {
       // Session unique identifier
       eventId : Integer,

       // Timevalue, a number type e.g. Double(unit="s"), or a date type (See #Time_Types)
       time : Variant,

       // Title
       title : Optional(String),

       // Message
       message : String,

       // NodeId of the sender object, optional
       source : Optional(Variant),

       // Event Type: alarm, action, error, info, debug
       type : String,

       // System Text, generated by the source system, eg. ?YD11D001 started?
       systemText : Optional(String),

       // Comments
       comments : Comment[]
     }

type Comment = {
       user : Optional(String),
       message : String
     }
```

SampleCollector



SampleCollector is a component that captures samples from a real-time data source, eg. simulation or measuring device, and writes them into a `SampleCollection`. `SamplingConfiguration` is an input to the SampleCollector. It describes how to do sampling. There is a `Record` for each *subscribed* variable. Subscription is a describes how and when samples are recorded from a variable.

Samples may be collected:

- on every step
- on change
- on change that exceeds change tolerance dead band
- at intervals

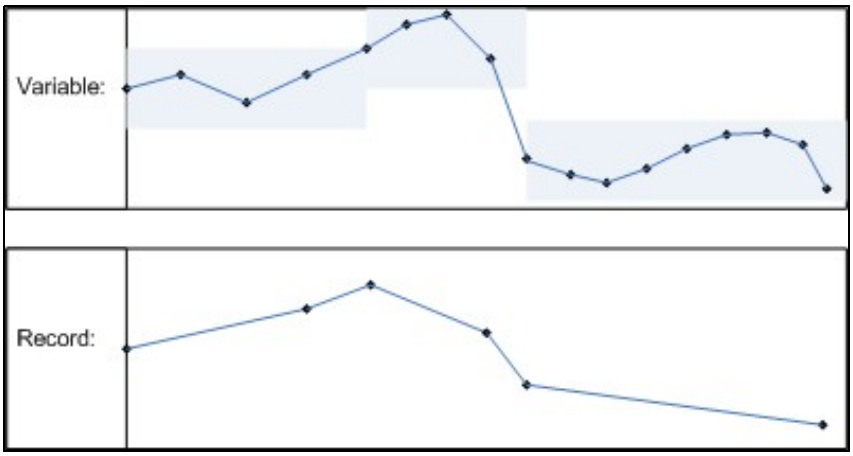
```
type SamplingConfiguration = {
       // Capture events, if true events are captured
       captureEvents : Boolean,

       // Subscribed Variables
       subscriptions : SubscriptionParameters[]
     }

type SubscriptionParameters = {
       variableId : Variant,
       deadband : Optional( Double ),
       interval : Optional( Variant ),
       sampleEveryStep : Boolean
     }
```

If interval is omited, the variable is sampled at every step (this applies to step wise data sources).

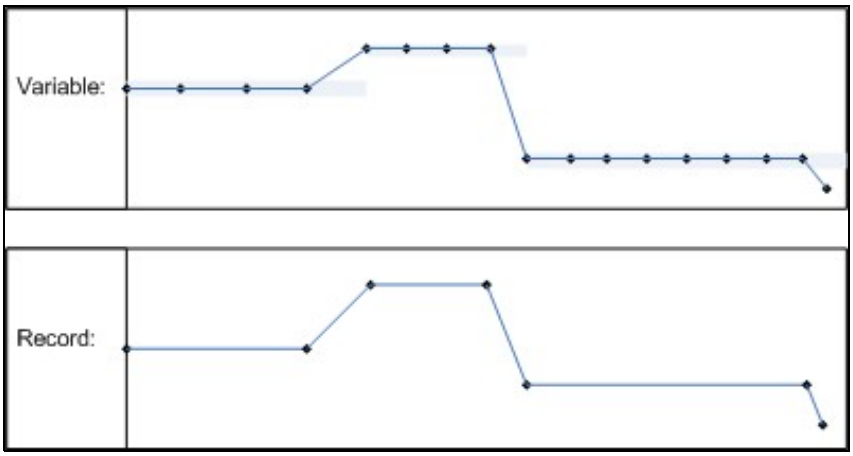
There can be multiple subscriptions for one variable, though they are both written to one record. If sampling from multiple subscriptions create a sample at the same timecode, only one sample is written to the record.

See Dataflows for Simantics layout of data flow components.

Deadband



Often values too small are irrelevant and to conserve space they can be omited. As the value of variable changes, its values are written to a record. If the difference between variable value and the last recorded value does not meet the dead band, the new value is not written to the record. The first and the last value of a deadband segment is always recorded. When deadband property is enabled, the produced record is in `Sparse` presentation format.



The setting `deadband = 0.0` can be used for not-storing redundant samples in the record.

File History

There is one directory per *RecordingSession* and one file for each *Recording*. Recordings are binary files (*.dbb*). RecordingSession is a directory with recordings as individual binary files (*.dbb*). There is also *RecordingSessio.dbv* that contains all the other fields, excluding *recordings*-field, of RecordingSession.

File name corresponds to the id of the recording with the following encodings:

```
  S<string>.dbb   String types
                  control characters " : < > | ? * \ / % [0..31] [128..] are escaped with %<hex><hex>
  I<integer>.dbb  Integer types
  L<long>.dbb     Long types
  B<base64>.dbb    All other cases. The value is binary encoded and presented as single line Base64 string.
                  Base64 encoding has Url and filename safe encoding flags enabled.
```

An example directory of a recording session

```
 history/
   RecordingSession.dbv
   SPA11%5fValve%2fTemperature.dbb
   SPA11%5fPipe%2fPressure.dbb
   I49589585.dbb
   BAAE.dbb
```

Datasource Contract

Flat Datasource

Datasource contract is a databoard presentation format of real-time value producers, such as *experiments*. The contract addresses the following issues: values, address space, identification, and Localizations.

The model consists of nodes. The address space is a tree. Dual structure, a map, enables random access.

Value is optional. Nodes with children and without values are called *folders*.

To support next-to all possible back-end systems, the identifier is a variant. The root id is the Default Value of a Variant type, an empty record. Ids are immutable and unique, two nodes cannot have same identifier. If such is case in the back-end system, a circumventing measure must be used in the implementation. It is typically sufficient, if path is included in the format of the id. Another strategy is to use GUIDs.

```
type Datasource = {
       nodes : Map(Variant, Node)
     }
type Node = {
       id : Variant,
       labels : LocalizedText,
       children : Variant[],
       value : Optional(Variant)
     }
```

Tree Datasource

```
type TreeDatasource = {
       root : TreeNode
     }
type TreeNode = {
       id : Variant,
       labels : LocalizedText,
       children : TreeNode[],
       value : Optional(Variant)
     }
```

Consistency

In accessor interface, the data source is never be represented to the consumer in an inconsistent state. There are two implementation strategies: either to block read & write operations over the inconsistent states, or present the last consistent state. ChangeSet is emited to listeners once consistency is regained.

Repository Contract

Datasource contract is a databoard presentation format for data repositories, such as *history archive*. The contract addresses the following issues: values, address space, identification, and Localizations.

The model consists of nodes. There is dual structure, a tree hierarchy, and a map for random access.

To support next-to-all back-end systems the identifier is variant. The root id is the Default Value of a Variant type, an empty record. Ids are immutable and unique, two nodes cannot have same identifier. If such is case in the back-end system, a circumventing measure must be used in the implementation. It is typically sufficient, if path is included in the format of the id. Another strategy is to use GUIDs. There are three identification serialization formats for the references: text, binary, url.

```
type DataRepository = {
       nodes : Map(Variant, Node)
     }
type Node = {
       id : Variant,
       labels : LocalizedText,
       children : Variant[],
       value : Optional(Variant)
     }
```

Remote Procedure Call

Databoard RPC is a method call interface. Server is an object that handles service requests. The server publishes an #Interface that contains a list of callable methods.

Network Protocol

The protocol is very simple. There are two conversing peers, one is *the client* and the other *the server*. Typically, the server implements has many functions, and client some related call-back procedures. The connection starts with handshake and is then followed by serialization of Request and Response objects.

In handshake, boths peers publish their methods and message size limits. (See Standard Library). Each decide whether to accept the other one's interface, if not the connection is disconnected.

Request

Client sends RequestHeader, followed by a serialization of the message's request argument. The datatype and thus serialization format of the request argument was defined in MethodType which was informed by the server in the handshake.

The server processes the procedure request.

- On procedure success, ResponseHeader is sent, followed by a serialization of ResponseType. ResponseType serialization format was declared in MethodType.
- On procedure failure, ExecutionError_ is sent, followed by a serialization of ErrorType. ErrorType format was declared in MethodType.
- On unexpected error, Exception_ is sent

- On invalid method number, InvalidMethodError is sent
- On request or response message size exceeded, ResponseTooLargeError is sent

Standard Library

There is a standard library of named datatypes. The following types are built-in in Simantics systems.

Datatype

The datatype description of types themselves. This type is used when serializing types in binary file and network connections.

```
type DataType =
     | BooleanType  {}
     | ByteType     { unit : Optional(String), range : Optional(Range) }
     | IntegerType  { unit : Optional(String), range : Optional(Range) }
     | LongType     { unit : Optional(String), range : Optional(Range) }
     | FloatType    { unit : Optional(String), range : Optional(Range) }
     | DoubleType   { unit : Optional(String), range : Optional(Range) }
     | StringType   { pattern : Optional(String), mimeType : Optional(String), length : Optional(String) }
     | RecordType   referable { referable : Boolean, components : Component[], methods : MethodTypeDefinition[] }
     | ArrayType    { componentType : DataType, length : Optional(Range) }
     | MapType      { keyType : DataType, valueType : DataType }
     | OptionalType { componentType : DataType }
     | UnionType    { components : Component[] }
     | VariantType  {}


type Range = { lower : Limit, upper : Limit }
type Limit = Nolimit | Inclusive { value : Double } | Exclusive { value : Double } | InclusiveLong { value : Long } | ExclusiveLong { value :
type Component = { name : String, type : DataType }
type DataTypeDefinition = { name : String, type : DataType }
```

Utility types

UUID represents an universally unique identifier (UUID), a 128-bit value.

```
type UUID = { mostSigBits : Long, leastSigBits : Long }
```

Localized text is a map of user readable text for multiple languages. The key is ISO-639 language code, and value is the text for that language. Default language is *en*, it is highly encouraged to always provide english text in addition to all others.

```
type LocalizedText = Map(String, String)
```

Void type is represented as empty record.

```
type Void = {}
```

URI type is a textual reference.

```
type URI = String
```

Interface

```
type Interface = {
       methodDefinitions : Map(MethodTypeDefinition, {})
     }

type InterfaceDefinition = {
       name : String,
       type : Interface
     }

type MethodType = {
       requestType : DataType,
       responseType : DataType,
       errorType : UnionType
     }

type MethodTypeDefinition = {
       name : String,
       type : MethodType
     }
```

Remote Procedure Call

The following types contain the serializatin format of structures used in Databoard RPC communication protocol.

```
type Handshake = | Version0
type Version0 = {
       recvMsgLimit : Integer,
       sendMsgLimit : Integer,
       methods : MethodTypeDefinition[]
```

```
    }

type Message = | RequestHeader RequestHeader
               | ResponseHeader ResponseHeader
               | ExecutionError_ ExecutionError_
               | Exception_ Exception_
               | InvalidMethodError InvalidMethodError
               | ResponseTooLarge ResponseTooLarge

type RequestHeader = {
       requestId : Integer,
       methodId : Integer
     }

type ResponseHeader = {
       requestId : Integer
     }

type ExecutionError_ = {
       requestId : Integer
     }

type InvalidMethodError = {
       requestId : Integer
     }

type Exception_ = {
       requestId : Integer,
       message : String
     }

type ResponseTooLarge = {
       requestId : Integer
     }
```

Accessor types

ChildReference is a relative reference path to a substructure in a data value or datatype.

```
type ChildReference = | IndexReference     { childReference : Optional(ChildReference), index : Integer }
                      | KeyReference       { childReference : Optional(ChildReference), key : Variant }
                      | NameReference      { childReference : Optional(ChildReference), name : String }
                      | ComponentReference { childReference : Optional(ChildReference) }
                      | LabelReference     { childReference : Optional(ChildReference), label : String }
```

An event contains a modification to the data model.

```
type Event = | ArrayElementAdded     { reference : Optional( ChildReference ), index : Integer, value : Optional( Variant ) }
             | ArrayElementRemoved   { reference : Optional( ChildReference ), index : Integer }
             | MapEntryAdded         { reference : Optional( ChildReference ), key : Variant, value : Optional( Variant ) }
             | MapEntryRemoved       { reference : Optional( ChildReference ), key : Variant }
             | UnionValueAssigned    { reference : Optional( ChildReference ), tag : Integer, newValue : Optional( Variant ) }
             | OptionalValueAssigned { reference : Optional( ChildReference ), newValue : Optional( Variant ) }
             | OptionalValueRemoved  { reference : Optional( ChildReference ) }
             | ValueAssigned         { reference : Optional( ChildReference ), newValue : Optional( Variant ) }
             | InvalidatedEvent      { reference : Optional( ChildReference ) }

type ChangeSet = { events : Event[] }
```

InterestSet describes how and what of a sub-tree is to be monitored.

```
type InterestSet = | BooleanInterestSet  { notification : Boolean, value : Boolean }
                   | ByteInterestSet     { notification : Boolean, value : Boolean }
                   | IntegerInterestSet  { notification : Boolean, value : Boolean }
                   | LongInterestSet     { notification : Boolean, value : Boolean }
                   | FloatInterestSet    { notification : Boolean, value : Boolean }
                   | DoubleInterestSet   { notification : Boolean, value : Boolean }
                   | StringInterestSet   { notification : Boolean, value : Boolean }
                   | RecordInterestSet   { notification : Boolean, notifications : Boolean[], value : Boolean, values : Boolean[] }
                   | ArrayInterestSet    { notification : Boolean, notifications : Integer[], value : Boolean, values : Integer[] }
                   | MapInterestSet      { notification : Boolean, notifications : Variant[], value : Boolean, values : Variant[], component
                   | OptionalInterestSet { notification : Boolean, value : Boolean, componentInterest : InterestSet }
                   | UnionInterestSet    { notification : Boolean, value : Boolean, componentInterests : InterestSet[] }
                   | VariantInterestSet  { notification : Boolean, value : Boolean, componentInterest : InterestSet, completeComponent : Boo
```

Time Types

These great time types are borrowed from JSR-310.

Instant is an instantaneous point on the time-line. It represents nano seconds since epoch (1970-01-01T00:00:00Z) ignoring leap seconds. In order to represent the data a 96 bit number is required. To achieve this the data is stored as seconds, measured using a long, and nanoseconds, measured using an int. The nanosecond part will always be between 0 and 999,999,999 representing the nanosecond part of the second.

```
type Instant = {
    seconds : Long,
    nanoSeconds : Integer(range=[0..999999999])
}
```

Duration is the time between two instants on the time-line. In order to represent the data a 96 bit number is required. To achieve this the data is stored as seconds, measured using a long, and nanoseconds, measured using an int. The nanosecond part will always be between 0 and 999,999,999 representing the nanosecond part of the second. For example, the negative duration of PT-0.1S is represented as -1 second and 900,000,000 nanoseconds.

```
type Duration = {
    seconds : Long,
    nanoSeconds : Integer(range=[0..999999999])
}
```

LocalDate is a date without a time zone in the ISO-8601 calendar system, such as '2007-12-03'.

```
type LocalDate = {
    year : Integer,
    monthOfYear : Integer(range=[1..12]),
    dayOfMonth : Integer(range=[1..31])
}
```

LocalTime is a time without time zone in the ISO-8601 calendar system, such as '10:15:30'. This type stores all time fields, to a precision of nanoseconds. It does not store or represent a date or time zone. Thus, for example, the value "13:45.30.123456789" can be stored in a LocalTime.

```
type LocalTime = {
    hourOfDay : Integer(range=[0..23]),
    minuteOfHour : Integer(range=[0..59]),
    secondOfMinute : Integer(range=[0..59]),
    nanoOfSecond : Integer(range=[0..999999999])
}
```

LocalDateTime is a date-time without a time zone in the ISO-8601 calendar system, such as '2007-12-03T10:15:30'. This type stores all date and time fields, to a precision of nanoseconds. It does not store or represent a time zone. Thus, for example, the value "2nd October 2007 at 13:45.30.123456789" can be stored in an LocalDateTime.

```
type LocalDateTime = {
    year : Integer,
    monthOfYear : Integer(range=[1..12]),
    dayOfMonth : Integer(range=[1..31]),
    hourOfDay : Integer(range=[0..23]),
    minuteOfHour : Integer(range=[0..59]),
    secondOfMinute : Integer(range=[0..59]),
    nanoOfSecond : Integer(range=[0..999999999])
}
```

ZonedDateTime is a date-time with a time zone in the ISO-8601 calendar system, such as '2007-12-03T10:15:30+01:00 Europe/Paris'. This type stores all date and time fields, to a precision of nanoseconds, as well as a time zone and zone offset. Thus, for example, the value "2nd October 2007 at 13:45.30.123456789 +02:00 in the Europe/Paris time zone" can be stored in a ZonedDateTime. The purpose of storing the time zone is to distinguish the ambiguous case where the local time-line overlaps, typically as a result of the end of daylight time.

```
type ZonedDateTime = {
   date : LocalDate,
   time : LocalTime,
   zone : TimeZone
}
```

TimeZones are geographical regions where the same rules for time apply. The rules are defined by governments and change frequently.

Each group defines a naming scheme for the regions of the time zone. The format of the region is specific to the group. For example, the 'TZDB' group typically use the format {area}/{city}, such as 'Europe/London'.

Each group typically produces multiple versions of their data. The format of the version is specific to the group. For example, the 'TZDB' group use the format {year}{letter}, such as '2009b'.

In combination, a unique ID is created expressing the time-zone, formed from {groupID}:{regionID}#{versionID}.

The version can be set to an empty string. This represents the "floating version". The floating version will always choose the latest applicable set of rules. Applications will probably choose to use the floating version, as it guarantees usage of the latest rules.

In addition to the group/region/version combinations, TimeZone can represent a fixed offset. This has an empty group and version ID. It is not possible to have an invalid instance of a fixed time zone.

The purpose of capturing all this information is to handle issues when manipulating and persisting time zones. For example, consider what happens if the government of a country changed the start or end of daylight savings time. If you created and stored a date using one version of the rules, and then load it up when a new version of the rules are in force, what should happen? The date might now be invalid, for example due to a gap in the local time-line. By storing the version of the time zone rules data together with the date, it is possible to tell that the rules have changed and to process accordingly.

TimeZone merely represents the identifier of the zone.

```
type TimeZone = { zoneId : String }
```