Colin Low, Jim Randell, Mike Wray
Hewlett Packard Laboratories
October 1997

Self-Describing Data Representation (SDR)
<draft-low-sdr-00.txt>

Status of this Document

This document expires on April 23 1998

Abstract

This document describes a human-readable, textual syntax for representing self-describing structured data. This representation was designed as a transfer syntax for loosely-coupled distributed applications where one cannot depend on sender(s) and receiver(s) sharing a schema for exchanged data. The syntax is compact, expressive, intuitive, and simple to implement.

1. Introduction

A traditional assumption in distributed systems is that structured data exchanged between applications is marshalled into an efficient

binary representation, and the receiver shares enough state with the sender to unmarshal this representation. This is usually achieved through a shared schema, in the form of shared header files, or an interface description language (IDL) used to generate the marshaling and unmarshaling routines for both sender and receiver.

This assumption is being eroded by the growth of distributed applications on the Internet, where one can no longer make the assumption that sender and receiver have shared state at some time in the past. In some applications sender and receiver are intentionally anonymous [1] and so data must of necessity be free-standing and self-describing.

Application-specific syntaxes which employ self-description do exist (e.g. Mail, News [2], HTTP [3]), but they lack the expressive power required for more general use.

The syntax described here was designed to meet the following goals:

Self-describing. We do not assume the recipient of data is familiar with its schema, and so data values must include additional information to identify the values, and possibly also the representation used for each data value.

Schema Tolerance. In a loosely-coupled and evolving distributed system with millions of participants, it may be difficult or impossible to ensure that all participants use the same schema for interpreting received data. The onus is on the recipient to interrogate the received data to ensure that it contains the minimum information required to make further progress.

Expressiveness. The syntax supports structured data values in the form of maps and lists to any level of containment.

Human Readability. Free-standing data needs to be (within reasonable limits) self-documenting.

Compactness. We wanted a syntax that was efficient to transport and parse.

Transport Independence. Self-describing data can (and should) be decoupled from the method used to transport it - it should be able to "ride piggyback" on any nominated transport. As most transports support the transfer of text, this means that the syntax described here can be employed with CORBA IIOP, ASN.1 Encoding Rules, OSF

DCE, Electronic Mail, TCP, and UDP.

Platform and language independence.

Accessibility and simplicity. Excessive complexity militates against widespread use. Our goal was that an average programmer should be able to write a parser for the syntax in a couple of days.

In order to provide a brief overview and example of the syntax, we have used the syntax to describe information of general interest to users of public transport, in the form of a "bus location update" notification:

```
notification: {

   !An advanced customer service from the Speedy Bus Corporation

   type {
      ...
   },
   content {
      type (omnibus speedy-bus location-update),
      bus-id "23",
      date USDate:"091797",
      time 24hour:"19:36:50",
      latitude (59 43 21),
      longitude (54 23 19),
      vrml "http://www.bus-company.com/vrml/bus.wrl"
   },
   system {
      ...
   }
}
```

As a public service the Speedy Bus Corporation has attached GPS receivers to all its buses, and makes location information available about each bus at regular intervals. Someone receiving this information can interpret it and build applications - for example, it would be straightforward to create a map showing the current positions of all the buses, or use regular information about a single bus to estimate the congestion on a known route.

Anyone receiving the information above can begin to make sense of it - there is no need to contact the bus company for information about how it encodes its location updates. Much of the information

represented is self-evident.

In more detail, the basis for self-description is the map, denoted by curly brackets {}. A map is an unordered set of name/value pairs. In the example above, the outermost map contains three further maps, named 'type', 'system' and 'content'. Because the SDR syntax is schema-tolerant, we don't need to know anything about the information in the 'type' and 'system' maps - we can effectively ignore them. If someone added a fourth map called 'authentication', it would have no impact on existing applications, which could ignore it.

In this application the 'content' map contains application-specific information, such as the position of the bus. We can use as little or as much of the information as we need. Let us suppose that the initial application supplied only the latitude and longitude of the bus. Later, in response to a 3D VRML town plan, the bus company supplies a URL that points to a 3D model of the bus, so that application writers can add the bus dynamically to the town model. The addition of a new 'vrml' element to the 'content' map will not impact older applications, which will continue to use only the position information.

This resilience to extensions to the schema is one of the most powerful features of SDR - loosely-coupled applications implemented on a large scale must tolerate this kind of incremental change and enhancement. The rule we apply is that an SDR parser should be able to accept syntactically correct SDR without prior knowledge of a schema, and present it to an application for interpretation. It is up to the application to decide whether the minimum information required is present.

For example, let us suppose the Reliable Bus Corporation also provides bus location notifications, but has not implemented the pointer to a VRML model of a bus. An application could work with both the Speedy and the Reliable bus notifications. If the 'vrml' field exists, the application could load the VRML model, otherwise it could substitute a default model. Clearly some minimal knowledge about the schema for notifications is necessary, but an application using SDR is insulated from additions and extensions to a schema.

The capability to unmarshal SDR in the absence of a schema leads to new kinds of application capable of carrying out a variety of purely syntactic functions on data-filtering, distribution and rewriting. For example, using the bus notification example above, it is possible to create an application that filters notifications using predicate expressions involving the fields of the notification. This filter

application needs to know nothing about the application (bus location updates) or the meaning of the fields used in predicates (e.g. latitude and longitude).

2. Data Model

SDR provides for the representation of structured data in three ways: map values, list values, and atomic values.

2.1 Map Values

A map is an unordered set of name/value pairs. The map

```
{
     firstname "John",
     lastname "Doe"
}
```

contains two pairs, with braces {} used to delimit the map. The ordering of pairs in a serial representation of a map is undefined - any order of pairs is a valid order.

Any atom (byte sequence) can be used to name a value in a map, subject to the restriction that a name may only be associated with a single value in the same map; that is, names are unique within the map. Values may be of any type - maps are not homogeneous - and can consist of further maps or lists as well as Atomic values.

Map values provide the foundation for self-describing data, and are intended to be heavily used in applications of SDR.

2.2 List Values

List values allow a collection of values to be represented. There is no requirement that the list be homogeneous. The list

```
( 3 "Foobar" { firstname "John" lastname "Doe" } )
```

is a list consisting of the atom 3, the atom "Foobar", and the map value from the previous section. Round brackets () delimit the list.

Lists can be used to represent sets where the number of elements is not known in advance - for example, a list of peoples' names. Lists can also be used to represent ordered data, such as lists of numbers.

Lists should be avoided in cases where an implicit assumption is made

about the interpretation of list elements - for example, the first element is a name, the second element is an age, the third element is the person's sex, and so on. This interpretation requires implicit state shared between the sender and the receiver. That is, the list

```
("John Doe" 35 "male")
```

would be better represented by the map

```
{ name "John Doe",
  age 35,
  sex "male"
}
```

which provides a more robust decoupling between sender and receiver - new elements can be added to the map without affecting the receiver. If the receiver only needs to know the age (for a statistical calculation) then it can extract that without having to interpret the rest of the information in the map.

This difference between maps and lists illustrates the philosophical difference between conventional serial representations of data, and self-describing data.

2.3 Atomic Values

Atomic values are used to represent the data at the leaf nodes of an SDR represenation. An arbitrary sequence of bytes can be represented as an atomic value. Typically any byte sequence represented by an atomic value is treated as an indivisible whole by the system and not subject to further structural breakdown - hence these byte sequences are known as atoms.

2.4 Tags

A value may have a tag associated with it. A tag is an atom associated with a value that may be used at the application level to denote the intended high-level type of the value. For example:

```
Person: {
     firstname "John",
     lastname "Doe"
}
```

indicates a map with the tag 'Person'. Tags are useful in increasing the comprehensibility of the syntax, but more importantly, were

intended to be used in conjunction with data types. For example,

UKDate: "010997"

and

USDate: "010997"

use the same string of numbers to represent different calendar dates (1st September 1997 in former case, 9th January 1997 in the latter).

The astute reader will have noticed that shared types (tags) reintroduce the need for shared state between sender and receiver. Tags are optional. When used sparingly they can be used to enrich the intentionally limited range of atomic types in SDR by imposing application semantics on atomic types (e.g. representing a date as a string).

In many applications tags are unnecessary - there are common notations in programming languages such as 'C' for representing a small number of different types of value such as integers, booleans, strings, and floating point numbers. SDR has a canonical representation for a small number of value types, and values recognised as such (for example, an integer) are supplied with an implicit tag. Further details on the representation of values and tagging are given below.

3. Data Representation

SDR is a human-readable concrete syntax to represent the data values described above as plain text.

3.1 Atoms

There is no single best method to represent an atomic value, and so four different methods of representation are provided which cover most common requirements. An atom can be introduced in four ways: as a token, a string, as counted data, and as quoted data.

Each of these is a way of introducing a sequence of bytes and any of them can be used to denote an atom. It is normally obvious which representation to use - for example, numbers are normally represented as tokens, and text as strings.

3.1.1 Tokens

SDR is based on an octet stream. Certain 8-bit values are reserved which correspond to the ASCII (7-bit) values for the SDR meta-characters (see 3.7).

If a non-empty atom consists entirely of octets equivalent to the following ASCII characters *or* contains octets whose values are greater than 0x7f (hexadecimal) it can be represented directly:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789$%&*+-.@?/_^~;<=>[]'`|
```

Atoms expressed in this limited alphabet are known as tokens. This convention is equivalent to support for UTF8-encoded Unicode.

Tokens are an efficient and readable representation for:

```
boolean values: true, false
integer values: 496 3 -89
floating point values: 1.333 -5.9+e9
names for map values: firstname lastname
tags: USDate UKDate
```

3.1.1.1 Examples

Each of the following is a valid atom introduced as a token and representing the given byte sequence (represented in hex):

```
event             <->  '65/76/65/6E/74'
<                 <->  '3C'
<=                <->  '3C/3D'
=                 <->  '3D'
x[4]              <->  '78/5B/34/5D'
42                <->  '34/32'
return-template   <->  '72/65/74/75/72/6E/2D/74/65/6D/70/6C/
61/74/65'
```

If an atom is not drawn from the limited alphabet it can be represented in one of the three remaining ways: as a string, as quoted data, or as counted data. Each of these ways can introduce arbitrary atoms (i.e. arbitrary byte sequences), whereas the token representation can only be used for a subset of atoms.

3.1.2 Strings

An atom can be introduced by representing it as a C-like string constant. The string is delimited with double quotes (" ") and may

contain octal escape characters as well as the standard escape forms of some other non-printable characters.

Atoms represented as strings consist of all bytes between the quote delimiters with escaped characters introduced by the backslash character (\).

The following escaped forms are recognised in strings and are translated to the given byte (in hex):

```
\b  ->  08 (backspace)
\f  ->  0C (form feed)
\n  ->  0A (line feed)
\r  ->  0D (carriage return)
\t  ->  09 (horizontal tab)
\\  ->  5C (backslash)
\"  ->  22 (double quote)
\'  ->  27 (single quote)
\x where x is an octal number in the range 0-7 (hex: 00-07)
\xx where xx is an octal number in the range 00-77 (hex: 00-3F)
\xxx where xxx is an octal number in the range 000-377 (hex: 00-FF)
```

Any other characters following a backslash are considered a parse error.

3.1.2.1 Examples

Each of the following is a valid atom introduced as a string and representing the given byte sequence (represented in hex):

```
"string"           <->  '73/74/72/69/6E/67'
""                 <->  '' (empty byte sequence)
"forty two"        <->  '66/6F/72/74/79/20/74/77/6F'
"\"pardon?\""      <->  '22/70/61/72/64/6F/6E/3F/22'
"line 1\nline 2"   <->  '6C/69/6E/65/20/31/0A/6C/69/6E/65/20/32'
```

3.1.3 Counted Data

A byte sequence may be introduced as an atom by preceding it with #* (hash, star) followed by a byte count represented in ASCII decimal (characters hex: 30-39) followed by \ (backslash) followed by the bytes themselves.

This format is particularly useful for programatic generation of data, where you know the length of the data you want to introduce, but you do not want to deal with the special quoting rules of

strings.

3.1.3.1 Examples

Each of the following is a valid atom introduced as counted data and representing the given byte sequence (represented in hex). To improve clarity, input characters are delimited by meta characters [and], which are typographic and not part of the Counted Data syntax.

```
[#*10\some bytes]      <->  '73/6F/6D/65/20/62/79/74/65/73'
[#*0\]                 <->  '' (empty byte sequence)
[#*2\  ]               <->  '20/20'
[#*9\"pardon?"]        <->  '22/70/61/72/64/6F/6E/3F/22'
```

3.1.4 Quoted Data

A byte sequence may be introduced as an atom by placing a delimiting sequence that does not occur within it on either side of the sequence. This representation is denoted by preceding it with #< (hash, less-than) followed by a single byte not occurring in the delimiter, the delimiter itself followed by the same single byte, then the atom itself followed by the single byte delimiter and the delimiter string.

3.1.4.1 Examples

Each of the following is a valid atom introduced as quoted data and representing the given byte sequence (represented in hex).

```
#<$END$some bytes$END    ->  '73/6F/6D/65/20/62/79/74/65/73'
#<#x##x                  ->  '' (empty byte sequence)
#<*---*  *---            ->  '20/20'
```

3.2 Tags

3.2.1 Introduction

Tags are a way to describe the intended interpretation of a value in SDR. For example:

```
boolean: true
boolean: 1
```

can both denote the same truth value.

SDR builds compound values out of atoms, and the atoms are syntactically neutral, with no particular meaning intended. In the absence of implicit interpretations of atoms in SDR one would have to use 'int:32' and 'string:"32"' to indicate that the first atom is meant to be an integer and the second a string. In order to avoid this SDR has `implicit tags'. These implicit tags cause untagged atoms to be treated as `the obvious thing' by SDR systems. The following implicit tags have defined meanings: map, list, atom, string, num, int, float.

The implicit tag of '32' is 'int', and the implicit tag of `"32"' is 'string', so we can use these representations without tags when we intend these interpretations.

Since the implicit tag of '"32"' is `string' it is not an equivalent representation to '32', though 'int:"32"' is. The rule is that in using an alternative representation for the bytes of an untagged atom an explicit tag must be supplied if the alternative representation has a different implicit tag.

It is possible that an SDR system may not want to parse integers and floats completely, but still use alternative representations. This is allowed by the use of the `num' tag. This means that the following atomic value is meant to be a number, but has not necessarily been checked for validity. SDR systems should not rewrite untagged atoms using the 'int' or 'float' tags unless they check validity. For example the following are equivalent forms:

```
32  int:"32"  num:"32" int:#*2\32 int:#<<end<32<end
1.414  float:"1.414"  num:"1.414"
4/2    num:"4/2"
"123"  string:123    #<<|<123<|
```

The form integer:"4/2" is not illegal SDR, though it may be uninterpretable as an integer.

A token begining with a digit has implicit tag 'num'. If it is a syntactically valid integer or float the implicit tag 'int' or 'float' may also be used. Any other unquoted token has implicit tag 'token'. An atom introduced as a string using `"' has the implicit tag 'string'.

The implicit tags of maps '{ .. }' and lists '(...)' are 'map' and 'list' respectively. At the moment there are no alternative

representations (up to element re-ordering of maps) so the only equivalent forms are like these:

```
{ x 1, y 2 }   map:{x 1, y 2}   map:{y 2, x 1,}   {y 2, x 1,}
( 12 "abc" )   list:(12 string:#*3\abc)
```

The implicit tag of #* and #< is 'string'. Some equivalences:

```
#*3\abc    #<<|<abc<|  string:abc   "abc"
```

3.2.2 Equivalent Forms

SDR has equivalent ways of saying the same thing, for example { x 1, y 2 } and { y 2, x 1 } mean the same. At least they are intended to mean the same. An SDR parser can see the different order of x and y in the 2 inputs and could make this information available to an application which could then distinguish them.

In order to exclude this behaviour, and make the notion of `meaning the same' precise, we define `SDR-compliance'. Suppose a client program is reading a stream of SDR. We define the client to be SDR-compliant if the input stream could be transformed into any equivalent form without disturbing the functioning of the client.

Similarly, suppose we have an SDR client functioning as a forwarder of SDR streams between clients. The forwarder is allowed to rewrite SDR into any equivalent form. The whole system is `SDR-compliant' if such rewriting does not disturb its functioning.

An atom is equivalent to any form that preserves its bytes and tag (implicit or not).

A map is equivalent to any form that preserves its tag and uses equivalent forms for its keys and values. Permuting the element-pairs and adding or removing the trailing comma are equivalences.

A list is equivalent to any form that preserves its tag and uses equivalent forms for its elements, in the same order.

Values may be introduced by an optional tag, represented by an atom followed by a : (colon) and optional white space, and then the value itself. Both the tag (if present) and an atomic value may be represented in any of the atom formats.

3.2.3 Implicit Tagging

In many applications tagging is unnecessary, and an SDR parser can say something about the types of atoms it recognises without needing an explicit tag. The SDR syntax was designed to facilitate the following implicit tags:

1. If the value is introduced as a token then the implicit tag depends on the content of the value in the following way:

1.1. If the value represents a 64-bit signed integer, expressed either in decimal or twos complement hexadecimal, then it will be assigned the implicit tag 'int'.

1.2. If the value represents an IEEE-754 floating point number, then it will be assigned the implicit tag 'float'.

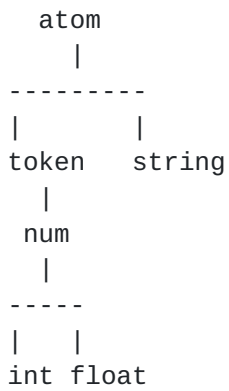
1.3. If the value has at least one byte and the first byte represents either a decimal digit or a sign or radix (+ (plus), - (minus), . (full stop)), then it will be assigned an implicit tag of 'num'.

1.4. Otherwise it will be assigned an implicit tag of 'token'.

2. If the value was introduced as a string then it will be assigned an implicit tag of 'string'.

3. If the value was introduced either as counted data or quoted data then it will be assigned an implicit tag of 'string'.

Implicit tags are structured as follows:



When a value is transmitted, any tag (either explicit or implicit)

must be maintained. For example, an atomic value introduced as [42] may be transmitted as [#*7\int:#*2\42] but not simply [#*2\42] (as it would be received with the implicit tag of 'data'). Similarly a value introduced as [string:1.3e+7] may be transmitted as ["1.3e+7"] but not as [1.3e+7] (as it would be received with the implicit tag of 'float').

3.2.3.1 Examples

Each of the following is a valid atomic value. Each line consists of alternative representations of the same atomic value (i.e. the same tag and the same value). Input characters are delimited by the meta characters [and], which are typographic and not part of the syntax.

```
[string:42]  ["42"]  [#*2\42]  [#*002\42]  [#<x$x42x$]
[37]  [int: 37]  [int: "37"]
[int: "thirty seven"]  [int: #*12\thirty seven]
[token]  [token:"token"]  [token:#*05\token]
```

3.3 List Values

List values are introduced by an optional tag, followed by ((open round bracket) followed by the list elements in order, separated by white space, and terminated with a) (close round bracket).

Untagged list values receive the implicit tag 'list'.

3.3.1.1 Examples

Each of the following is a valid list value.

```
          (one two three four)

          (integer:one integer:two integer: three integer:four)

          (1 (2 2) (3 3 3) (4 four IV 4.0 4+0i))

          template:(
            (insert type (literal (app msg-panel notify)))
            (insert name (literal "WWW"))
            (insert text (format "Update: %s" (substitute (document-info
url))))
            (insert url (substitute (document-info url)))
            (if (test (document-info title) (type atomic))
              (insert text (format "Update: %s - %s" (substitute (document-
               info title)) (substitute (document-info url)))))
          )
```

3.4 Map Values

Map values are introduced by an optional tag, followed by { (open brace) followed by a comma separated sequence of pairs, followed by a } (close brace). Each pair consists of an atom representing the name, followed by whitespace, followed by the value. The value may be an atomic value, a list or a map. If a comma is included after the final maplet in a map it will be ignored.

Untagged map values receive the implicit tag 'map'.

3.4.1.1 Examples

Each of the following is a valid map value.

```
{one 1, two 2, three 3, four 4}

{1 int:one, 2 int:two, 3 int: three, 4 int:four}

notification: {
   type (app wanda document update),
   document-info {
      url "http://keryxsoft.hpl.hp.com/project/web-watcher.html",
      last-modified "Tuesday, 04-Mar-97 09:23:28 GMT",
      checksum {type md5, value 79552c131ee78346de887912534bcc},
      keywords ("Keryx" "Application" "Web" "Notification"),
      visibility (
        {type netmask, pattern 15.0.0.0, mask 255.0.0.0}
      ),
      title "Keryx Web Watcher",
      author-url "mailto:foo@hplb.hpl.hp.com",
      description "Proposal for Keryx Killer App",
      relevance (
         {type (hp logical), value (com hp hpl hplb keryx)}
         {type (geo global), value ("51:30:00N" "02:33:15W")}
      )
   }
}
```

3.5 White Space

The following bytes (given in hex) are considered white space when they occur (other than as part of an atomic value).

```
20  (space)
09  (horizontal tab)
0D  (carriage return)
0A  (line feed)
0C  (form feed)
```

3.6 Comments

When values are read from files comments may be introduced by the character ! (exclamation mark) and continue to the next new-line character. The comment is treated as white space.

3.7 Character Sets

SDR is based on an octet stream. Certain 8-bit values are reserved, corresponding to the ASCII 8-bit values for the characters:

```
(){}#"!
plus the whitespace characters given in 3.5.
```

4. Syntax Description

The syntax for SDR is represented using an extended BNF. Square brackets [] are used to denote an optional item. The character '*' denotes an item that may be repeated zero or more times, and '+' denotes an item that may be repeated one or more times. A choice is denoted by '|'. Literals are quoted: for example, 'foo'.

```
Value          =   [Atom':'] RawValue
RawValue       =   Atom | Compound
Atom           =   Token | String | CountedData | QuotedData
Compound       =   Map | List
Map            =   '{' '}' | '{' [Maplet ',']* Maplet [','] '}'
Maplet         =   Atom Value
List           =   '(' [Value]* ')'
Token          =   RestrictedChar+
String         =   '"' Char* '"'
CountedData    =   '#*' NonNegativeInteger '\' RawBytes
Char           =   Escape | OctalEscape | Ascii
Escape         =   '\n' | '\t' | '\b' | '\r' | '\f' | '\"' | '\'' | '\\'
OctalEscape    =   '\' OctalDigit OctalDigit OctalDigit
RestrictedChar =   one of
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789$%&*+-.
@?/_^~;<=>[]'`|  plus octet values over 0x7F
Ascii          =   Any ASCII character
QuotedData     =   '#<' <single char c> <delimiter string s> <c again>
                   data <c again> <s again>
```

In OctalEscape and in Data types after initial #x, no white-space is permitted.

5. Implementation

An implementation of SDR in Java, with source code, is available as a component of the Keryx Notification Service, available at http://keryxsoft.hpl.hp.com

6. Acknowledgements

This syntax has been critiqued by several people. In particular, Soren Brandt and Anders Kristensen have implemented several variants of the syntax and provided many valuable suggestions.

7. References

[1] The Keryx Notification Service - http://keryxsoft.hpl.hp.com

[2] Standard for the format of Arpa Internet text messages, RFC 822, 1982

[3] Hypertext Transfer Protocol HTTP/1.0, RFC 1945, 1996

8. Author's Address

Colin Low (editor)
Hewlett Packard Laboratories,
Filton Road,
Stoke Gifford,
Bristol BS12 6QZ
UK

Tel: +44 117 9799910
Email: cal@hplb.hpl.hp.com