DFDL Representation Properties

Status of This Memo

This memo provides information to the Grid community regarding the specification of a Data Format Description Language. The specification is currently a draft. It does not define any standards or technical recommendations. Distribution is unlimited.

Abstract

This document provides a description of representation property names for use in DFDL annotations.

Revision History

Latest entry ***at the top*** please			
Version	Author/Contributor	History	Date(yyyy-mm-dd)
002	Mike Beckerle	Got rid of basic vs. advanced distinction. Lots of TBDs still.	2004-11-17
001	Mike Beckerle	Created.	2004-07-29

Contents

1. Glossary

DFDL – Data Format Description Language

The term “byte” herein refers to an 8-bit octet.

Herein where the phrase "must be consistent with" is used, it is assumed that a conforming DFDL implementation must check for the consistency and issue appropriate diagnostic messages.

(TBD: uniform way to talk about suppressibility of warnings/errors?)

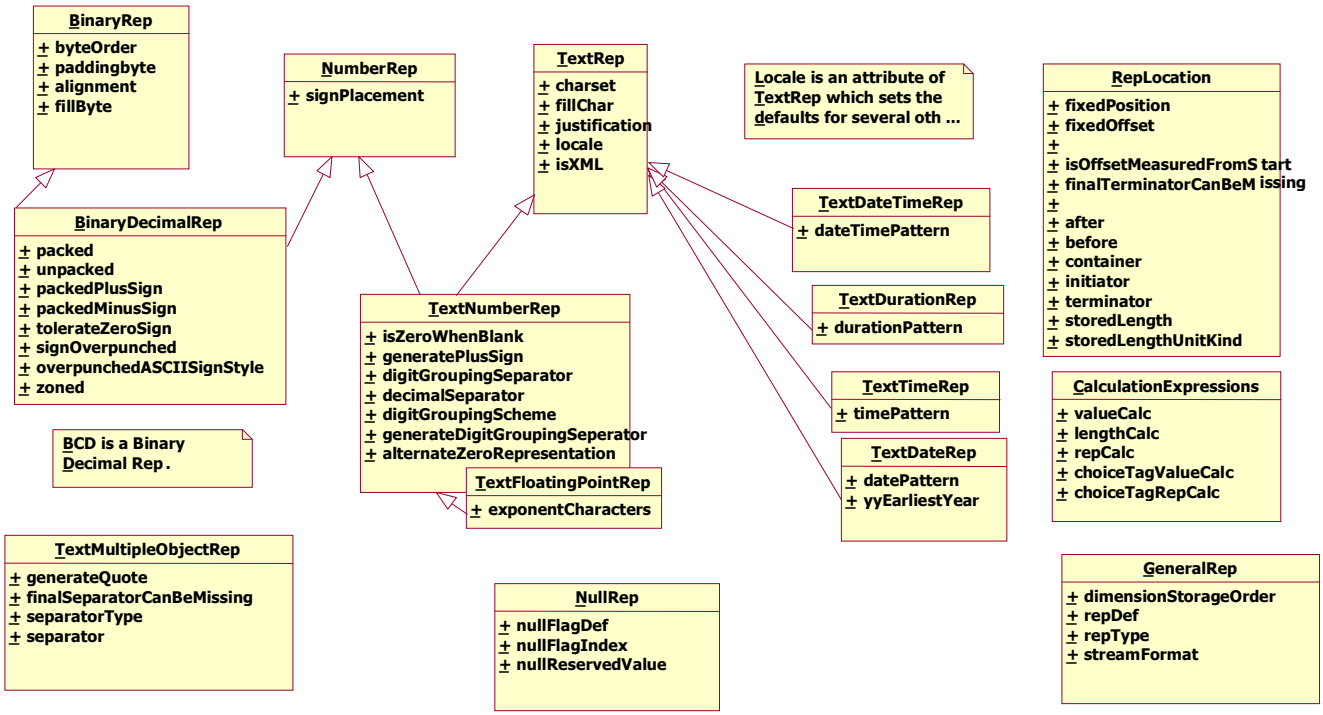
A “DFDL processor” is a program that uses DFDL descriptors in order to process data described by them.

2. Introduction

This document provides a description of representation property names for use in DFDL annotations.

3. Conceptual Model for Properties

The conceptual model which shows the rough organization of the properties is given below.



Note to the reader: THERE ARE NO SUCH CLASSSES. This is all merely conceptual to help mentally organize the properties into logically useful groupings for explanatory reasons.

TBD: These properties are mentioned in Jim Meyer's Examples:

- Root
- Source
- Stream
- representationLevel

TBD: From IBM OMG type model (just ones I could think of -mikeb)

- From IBM: bits in a unit - allow for non 8-bit bytes. E.g., 12-bit chunks
- Ones complement?

The next sections describe the various properties individually:

4. General Properties

Property Name	Description
repType	Allowed values are “binary” or “text”
repDef	Indicates that a separate element XSD is used to describe the representation of this item. This is used when the type is simple, but the representation is complex. E.g., an element may be of type string, but the representation is a sequence of a stored length element followed by the content. You can describe the XSD with annotations for this data structure then reference it as the repDef for the string.
dimensionStorageOrder	Allowed values are “rowMajor” or “columnMajor”. Used for arrays with more than one dimension indicates storage order.
streamFormat	Allowed values are F, FB, V, VB, VBS, none. Default is none.
(TBD: should this be called "recordFormat" ?)	TBD: IBM docs define these formats: D, DB, DBS, DS, F, FB, FBS, FBT, FS, U, V, VB, VBS, VS. Note that "FBS" stands for "Fixed Blocked Standard", where VBS stands for "Variable Blocked Spanning", i.e., the S stands for something different.

5. Text Properties

Property Name	Description
charset	For text format, indicates the character set name. This is an IANA character set name.
fillChar	The positions of the items in a group and the sizes of those items may leave some characters of the representation unpopulated with data; that is, they are unused. When writing such data out this property specifies that they are to be filled with a specific character value. It is crucial to secure software that they not be left to contain whatever was in memory. The default value of fillChar is “ “ (blank). Non-default values (e.g., using the string ‘X’) is mostly used as a debugging technique so that unused space or mistakes in the DFDL descriptor are glaringly obvious when data is examined with simple tools like print statements.

	When a variable-length character set is used, the fillChar must always be a minimum-width character so that there is no question of whether there is enough room for one or not. (Note that when a fixed-length character set is used, the size (in bytes) of the field must be a proper multiple of the character set width, so that there is also no issue of being able to fit the fillChar in or not.)
justification	Values are “left” or “right”. Indicates whether variable-width data should be padded on the left or right. By default, numeric types are right justified, other types (esp. non-numeric strings) are left justified.
locale	ISO-639 language code followed by ISO-3166 country code, e.g., “en_US”. (TBD: more? a full Unix-like LocaleDef?) This setting provides defaults for digitGroupingSeparator, digitGroupingScheme, decimalSeparator, and the text defaults for the locale-sensitive components of date and time formats.
isXML	Boolean indicates if the text is XML, or originated in XML so that it obeys XML syntax rules.

5.1 Text Location and Separation Properties

In textual representations data items can be delimited by characters or strings of characters. There are three different kinds of delimiters: initiators, terminators, and separators. Individual items can have initiators and terminators, aggregates (groups and arrays) have separators.

Some representations can be described in multiple ways. E.g., a text file with 1 record per line can be describe as a data set containing data items which are groups and each one is terminated by an end of line. The same data can be described as a data items with dimension (i.e., a vector) where the items are separated by line endings.

For any delimiters, the value specified is either a single character or a string (usually short) specified in the charset of the item. The characters are translated from the character set of the DFDL document into the charset of the item. For example, a separator=”,” specification on items with charset=”EBCDIC-CP-US” items results in the single byte 0x2C (code for a comma in EBCDIC) regardless of the character set the DFDL description is written in as an XML document.

In case of non-printing characters XML escape sequences can be used to specify the numeric value of the character codes in some cases, special escapes of the form “\uHHHH” and “\UHHHHHHHH” are used for XML-disallowed character codes such as zero. See also the appendix “About Literal Strings”.

Property Name	Description
separator	Value is a TextDelimiter. A separator is positioned between two items to mark their boundary. Comma, tab, and vertical bar “|” separated values are popular file formats. Each line in such a file typically contains a single record whose items are separated by the specified character. Only aggregate types can have separators. (Aggregate types = dimensioned arrays and groups).

separatorType	Valid values are “infix” (the default), “prefix” and “postfix”.
finalSeparatorCanBeMissing	Boolean. Default is false. When using “postfix” separatorType, this Boolean indicates that it is not an error if the last separator of an aggregate type is missing at the end of the containing data item. An example of this might be records separated by line endings. If the last line doesn’t have a CRLF at the end of it, but end-of-file is reached, then this flag indicates that this is not a data format error.
initiator	Value is a TextDelimiter. An initiator delimits the beginning of an item. For example a textual tag such as “[coordinates]” might precede the data lines containing coordinate data.
terminator	Value is a TextDelimiter. A terminator marks the end of an item. For example, a variable-length string as represented in a C program is a vector of characters terminated by the NUL character code (NUL is zero). A file may contain a vector of records where each record is a non-blank line. The vector is terminated by a blank line. A terminator can also be a string such as “—END—“.
finalTerminatorCanBeMissing	Boolean. Default is false. Indicates that it is not an error if the terminator is missing at the end of the data. An example of this might be records terminated by line endings. If the last line doesn’t have a CRLF at the end of it, but end-of-file is reached, then this flag indicates that this is not a data format error.
storedLength	Value is the name of a “peer” item whose value gives the length of the current item.
storedLengthUnitKind	Value is ether “bytes", "characters", or "elements". Default is “elements”. This indicates whether the length is the number of logical items, or the number of representation items. For example, for a string with a stored length, the length could be in bytes of the underlying representation, or it could be in characters. For an array of integers, the stored length could be the number of integers, which could be textually represented, or it could be the number of characters making up the rep, or the number of bytes.

6. Binary Properties

Property Name Description

byteOrder	Valid values are “bigEndian” or “littleEndian”. Applies to binary multi-byte representations. Also applies to text representations when the charset is a fixed-width, multi-byte encoding, such as UTF-16 or UTF-32 which has big and little endian variants.
paddingByte	This property gives a byte integer value used to fill in parts of the representation of an item that are not filled in by the data type. This occurs when you have variable length data types with a maximum length represented as a fixed size buffer of the maximum possible size and a stored length item or delimiter indicating the amount of the buffer actually being consumed by this data item. (Happens all the time in Cobol data.) In this situation the paddingByte value is used to fill in the unused bytes of the representation. This is important for security reasons. This differs from fillByte in that paddingByte is about space within the possible representation of a data item. (TBD: possible generalization to multiple byte sequence. Makes for easy debugging when a pattern like 0xdeadbeef that looks like English words is found in the unused parts of the data area.)
fillByte	The positions of the items in a record and the sizes of those items may leave some bytes of the record unpopulated with data; that is, they are unused. When writing such records out this property specifies that they are to be filled with a specific byte value. It is crucial to secure software that they not be left to contain whatever was in memory. The default value of fillByte is 0 (zero). This byte value is also used to fill any unused bytes due to alignment. Non-zero values (e.g., using the character code for ‘X’) are mostly used as a debugging technique so that unused space or mistakes in the DFDL descriptor are glaringly obvious when examined with simple tools like print statements. This differs from paddingByte in that fillByte is about space that is not occupied by any data item. That is, the need for fillByte arises because data items need not be adjacent. There can be holes left over between the data items in the representation. These are what is filled with the fillByte. (TBD: possible generalization to multiple byte sequence. Makes for easy debugging when a pattern like 0xdeadbeef that looks like English words is found in the unused parts of the data area.)

6.1 Binary Location Properties

Property Name	Description
alignment	This is an integer greater than zero. It gives the alignment required for the beginning of the item. The default value is equal to the largest alignment specified on any item specified recursively within the item, or 1 if none is specified. The value is required to be at least as large as this default. If larger it must be a multiple of this computed default. DFDL implementations must detect inconsistent alignment specifications. Values are usually 1, 2, 4, 8, 16 to match memory word alignment boundaries, 8096 to match page alignment boundaries. However, any integer 1 or greater is allowed.

7. Location Properties (Text or Binary)

These properties apply to any repType. The units they are measured in are bytes for binary and characters for text (codepoints for UTF-16 charset.). These provide multiple ways to express the same thing, the position of a data item. However, this is provided intentionally in order to make capture of a DFDL description easy from a variety of ad-hoc data description formats. For example some ad-hoc data dictionaries are spreadsheets where there is a row per "field" in the data, and one of the cells of that row is the absolute position of the corresponding data field in the data. Other times the data-dictionary/spreadsheet will have a cell containing the field width, and the position is the aggregate width of preceding fields. Rather than requiring translation of these forms into a single common mechanism we provide both absolute positioning and relative positioning.

It is possible to interpret the same storage multiple ways by specifying multiple items as having representations coming from the same regions within the data.

Property Name	Description
fixedPosition	Non-negative integer. If specified it indicates the distance from the beginning of the enclosing data item where the representation of this data item will be found. Units are bytes for binary, characters in the charset for text.
fixedOffset	Integer. Default value 0. If specified it indicates the distance from another data item within the same group where the rep of this data item will be found. Units are bytes for binary, characters in the charset for text. When used with alignment, fixedOffset gives the minimum distance. Additional bytes to provide alignment padding may be needed.
isOffsetMeasuredFromStart	Boolean. Default is false. If specified indicates that the fixedOffset property is measured from the start of the base item, not the end of it.
after	Value is the name of a "peer" element. That is, an element also contained within the same ordered group (aka sequence). This specifies the item that the fixedOffset applies to. The default is the name of the preceding item. This is used to capture relative distances between fields. That is, when fields are specified as having positions measured in distance from another field, not necessarily the beginning of the entire record.
before	Value is the name of a "peer" element. That is, a element also contained within the same ordered group (aka sequence). This specifies the item that the fixedOffset applies to.

7.1 Text Delimiters, Quoting, Escape Sequence

Text delimiters include separators, terminators, and initiators for the representations of data items. A given representation can have one or more delimiters. That is, it can be legal for the items of a group or dimension to be separated by any one of several characters or strings. A good example of this is lines, which can be separated by either the linefeed character alone, or the CRLF two-character sequence. This also illustrates that a text delimiter need not be a single character.

Whenever one specifies a text delimiter, it is possible to specify a quoting scheme by which the text delimiter may appear as content and not be interpreted as a delimiter. A quoting scheme is a surrounding set of markings, usually with quotation marks. Similarly, a text delimiter may be escaped by an escape scheme, which prefixes the delimiter as a way to mark it as content.

In some cases, quoting characters can be escaped, and escape prefixes can be quoted.

- textDelimiter
    - delimiter: string (commonly is just a single character) that is used to separate, initiate, or terminate a data item's representation.

    (TBD: to handle "whitespace" you'd have to put in several terminator declarations. One for LF, one for CRLF, and one for every other legal Unicode way of ending a line. Need a shorthand for "whitespace". One unbiased solution is to use the expressions defined in Unicode regular expressions (http://www.unicode.org/reports/tr18/) such as "\p{Whitespace}". In general we might want to allow the delimiter to be specified as a regexp. In any case in order for this DFDL to be readily translatable into other format description languages we need shorthands for whitespace and newline.)
- quoteScheme: A text delimiter can have zero or more quote schemes. Each specifies the opening and closing quotation marks or brackets that can surround the value of an item in text representation. If a quoteScheme is specified, it contains two strings. The first is the opening quote mark, the second the closing. Inside the quote marks the delimiters may be found unescaped. If multiple QuoteSchemes are specified, then these can be nested in balanced fashion so that one can be used to quote/escape another.

    Example: "123 'abc"doreme"efg'456" has two QuoteSchemes showing how double quotes can be used to quote the interior single quotes which are being used to quote the interior double quotes.
    - open: string (commonly is just a single character) that opens the quoting
    - close: string (commonly is just a single character) that closes the quoting
    - generateOnOutput: Values are "always" or "ifNeeded". Indicates that this particular scheme should be used on output. Default when not specified is that the first quoteScheme (when more than one is specified) is used in "ifNeeded" fashion. When several quoteSchemes are specified only one may have generateOnOutput explicitly specified.
- EscapeScheme: Specifies the prefix characters or strings used to escape occurrences of the quotation characters or the delimiters themselves if they appear inside the value of an item.

    Example: 5, 6.3, 8\,345\,435.21, 22 shows 4 numbers separated by commas. The third uses commas as thousands separators (called digitGroupingSeparators as a property).

Example: 5, “6,7”, “a string with \” (qmarks) embedded”, 42 Here we see an escaped quote mark.

Example: 5, “6\\”,7 Here the middle string contains “6\”. The backslash is used to escape itself.

- prefix: an escape string (usually a single character). Used to prefix an occurance of the delimiter or the open/close of a QuoteScheme. To express an escape for an escape you must use two EscapeSchemes, one referring to the other.
- generateOnOutput: Boolean. Indicates that this particular scheme should be used on output. When several escapeSchemes are specified only one may have the generateOnOutput property specified. Default behavior when not specified for any escapeScheme is as if the first escapeScheme had generateOnOutput specified as “true”.

More Examples TBD.

8. Numbers

Property Name	Description
signPlacement	‘leading’ , ‘trailing’, or ‘surround’ values are supported. Default is leading for text, but trailing for decimal binary forms. Surrounding sign means that “(5)” is equivalent to negative 5. Surrounding sign is only for text representations.
unsigned	Indicates that the rep is unsigned. This is independent of the type itself which might be sign-capable. That is, an Int32 type can have as its rep an unsigned packed decimal number. If an unsigned type (such as UInt32) is given a signed rep, then this is equivalent to asserting that eventhough the rep is signed there will not be any negative values. Think of this as a validation test. The rep is invalid for this logical type if a negative value is found.

Comment [PS1]: Is this adequate? Does the locale define any alternate chars to +, -, () etc? TX has a more general purpose approach, but it could be that its excessive and never used. See TX notes.

8.1 Text Numbers: Integers, Decimal, and Floating Point

Property Name	Description
isZeroWhenBlank	A common data convention for text representation is that if all characters are whitespace then the number should be assumed to be zero even though technically it’s not valid decimal unless it has at least the single digit ‘0’. Generally this is used with fixed length fields; however, for variable length fields if the length is 0, then isZeroWhenBlank indicates that this means the numeric value is zero. (Note: OMG CWM 1.0 , Vol2, Section 3 Cobol calls this “isBlankWhenZero”. In general names here use the parser perspective, not the formatter/writer perspective.) Note that when writing out data we never write out all whitespace to represent zero. We always write out at least the single digit “0”.

generatePlusSign	For text representations create a '+' sign in the output (It's always optional on input.)
digitGroupingSeparator	like POSIX thousands_sep from Locale settings. Default is dependent on the Locale property. (Not on the Locale setting of at the OS level, but the Locale DFDL property.) The digit grouping separators are always optional for reading. Numbers will be interpreted correctly without them being present. (TBD: in some locales whitespace is a common digit grouping separator. Implications are unclear.)
decimalSeparator	like POSIX decimal_point from Locale settings. Default is dependent on the Locale property. (Not on the Locale setting of at the OS level, but the Locale DFDL property.)
digitGroupingScheme	like POSIX 'grouping' from Locale settings. Default is dependent on the Locale property. (Not on the Locale setting of at the OS level, but the Locale DFDL property.)
generateDigitGroupingSeparator	When writing data, this boolean indicates whether digits are grouped or not. Default is false.
exponentCharacters	Allow additional characters such as 'F' and 'G' in addition to 'E' for the exponent marker. (E.g., use 'EFGefg' to allow upper and lower case E, F, and G.)
TBD: rounding mode for binary floating point	When a floating point number is represented as a text in base 10, it is possible for it to be ambiguous what binary floating point number is to be chosen as the representation. That is, it is possible for two different binary floating point numbers to be equidistant from the ideal base 10 number. Both binary numbers if converted back to base 10 would have the exact same base 10 digits; hence, it is necessary to have an property to control which floating point binary number is to be used. The rounding modes are either ceiling, floor, round_inf, trunc_zero which mean to round up, round down, round toward closest infinity, or round toward zero. In the opposite conversion direction (binary to base 10) there's an additional rounding mode: round_even which rounds to the base 10 digit that is even. This equally distributes the rounds with some going up to an even number and some going down to an even number.

Example: In the en_US locale one might encounter 101,234,567.89. In a fr_FR locale, the equivalent format would be 101.234.567,89, and in a Thai locale, 10,12,34,567.89. Notice that in the Thai locale, the "thousands separators", a.k.a., digitGroupingSeparators don't get used to separate thousands; hence, we don't copy the misleading POSIX name.

(TBD: official name of en_US, fr_FR (?), and Thai locales?)

8.2 Decimal Numbers

Commercial applications make heavy use of decimal arithmetic where the precision is larger than what can be captured with today's floating-point representations. Usually this data is fixed precision. That is, the total number of digits and number of fractional digits are fixed. This is sometimes called 'fixed-point' arithmetic; however, it is also critical that the arithmetic itself be done in base 10 with base 10 rounding. There are a variety of commonly used representations for

decimal numbers including both string-based and binary (the packed and unpacked decimal) forms. There is also a relatively new standard for decimal arithmetic and representation, though to date this is hardly used anywhere. The decimal standard is IEEE 854. It is being merged with IEEE 754, the standard for binary floating-point arithmetic. (See http://grouper.ieee.org/groups/754 as a starting point for details.)

There are two approaches to dealing with the complexity of decimal representations. One is to enumerate and separately name every combination that is actually in use. This approach is used by SAS. The SAS Language Reference describes the 'Informats' accepted by SAS. These are a quite comprehensive list of data formats for decimal and date types. This document is not provided online by the SAS Institute, but many organizations have put it online for their users. One such source about the 'Informats' topic is http://www.ncsu.edu/it/sas/help/lrcon/z0920449.htm . This list has dozens of different formats, the differences between them are subtle. The other approach is to tease-apart the different varying aspects that make up the different formats. This has the drawback that many formats can be described that have never existed in history. Nevertheless, we've chosen this latter approach. Using some reusable constructs we can capture definitions which correspond to certain frequently occurring standard combinations of these options. This gives us the best of both approaches.

There are lots of details associated with legacy decimal formats. Here is the list of the properties needed for specification of decimal formats. These are used along with the signPlacement (leading or trailing), unsigned, byteOrder, repType (text/binary), and charset properties to provide full control over the representation of decimal numbers

Property Name	Description
decimalFormat	Valid values are "packed", "unpacked", "zoned", or "text". **packed**: This is a binary format which indicates that the data is stored as two decimal digits per byte. If the number is signed, one nibble is used to store a sign nibble. If the number is unsigned no nibble is used for sign and it is instead used for a digit. For input, hex nibbles A, C, E, and F indicate positive, while B and D indicate negative. (C for positive, and D for negative are most common.) Cobol COMP-3 data is packed decimal. It is possible for an extra nibble to be unused in the representation of a packed decimal number. For example, if there are 4 digits plus a sign, then 5 nibbles are needed. When 3 bytes hold these nibbles, one nibble is unused. When writing data out, this nibble must always be zeroed for security reasons. **unpacked**: each digit is stored as the least significant nibble of a byte. The most significant nibble is always zero. The sign nibble is as for packed decimal but is also stored in the least significant nibble of an additional byte (for signed numbers). Note that unpacked is not a text format. In most character sets the character codes like 0x05 are non-printing characters. In XML the Unicode character with code 0x00 isn't even allowed (as of XML 1.1). This makes the unpacked representation really an infrequently used binary format. Note: unpacked is used by RM/Cobol (see MicroFocus Cobol Compatibility Guide, April 1993, Chapter 8: "Converting RM/COBOL Data Files", page 8-1.) (TBD: need better citation) **zoned**: each digit is stored as a byte. The high nibble contains 0xF for EBCDIC charsets and 0x3 for ASCII-based charsets. The sign is leading

	or trailing, and is usually overpunched. If overpunched, the same nibble values are used as in packed decimal representation. If separate, then bytes 0xC0 and 0xD0 can be used to represent the plus and minus signs respectively, or the characters for '+' and '-' in the character set. Note that 0xC0 is the '{' character in EBCDIC and 0xD0 is the '}' character in EBCDIC. Because of the likelyhood of flawed translation of zoned decimal data from EBCDIC to ASCII, the ASCII byte 0x7B and 0x7D should also be accepted as specifying the plus and minus signs respectively.
	The only difference between zoned decimal and a text string representing the number is that zoned decimal has an implied decimal point location specified by fractionDigits, so periods, commas and other textual characters are not allowed in the representation. More specifically, a zoned decimal number can contain only these characters '0123456789+-{}', in EBCDIC or ASCII characters with 1 byte per character where the braces correspond to plus and minus signs.
	Zoned decimal items are a binary format that is sensitive to character set. On output a zoned decimal type item contains EBCDIC characters if the charset is any EBCDIC variety and ASCII single-byte characters otherwise. If the charset is not an EBCDIC or ASCII character set then use of zoned decimal causes an error. (UTF-8 counts as an "ASCII" character set, but UTF-16 does not.).
	text: Very closely related to zoned, but allows decimal points and digit grouping separators. Text is the default decimal format when the repType property is "text".
packedPlusSign (TBD: rename to "decimalPlusSignNibble")	Specifies a hex nibble to use for the plus sign when writing out signed packed decimal numbers. Defaults to "C". (TBD: Could also be used to validate input data having a specific sign convention.) (TBD: packedPlusSign and packedMinusSign apply even to unpacked representation. RM/Cobol uses different defaults than IBM mainframe Cobol. I.e., 0xB is the plus sign. So, for unpacked data should these conventions just be different?)
packedMinusSign (TBD: rename to "decimalMinusSignNibble")	Specifies a hex nibble to use for the minus sign when writing out signed packed decimal numbers. Defaults to 0xD. (TBD: Could also be used to validate input data having a specific sign convention.) (Note: About unsignedDecimal RM/COBOL COMP-3 UNSIGNED data has a 0xF sign byte anyway. So it would be described in DFDL as signed, i.e., conversion of a Cobol FD for a RM/COBOL system would translate COMP-3 UNSIGNED into a signed packed decimal description in DFDL. Source: MFCobol Conversion document cited elsewhere herein.)
tolerateZeroSign	A common data convention is that if all bytes are zero then the number should be assumed to be zero even though technically it's not valid signed packed (or unpacked) decimal unless it has a positive sign (A,C,E, or F) or negative sign (B or D).
signOverpunched	Boolean. Defaults to true. True means the sign is overpunched. False means the sign is separate. This property determines if the sign is encoded on top of an end digit. This format saves a byte for a separate sign, but lets things that look like '12{' be valid decimal numbers. The default depends on the setting of decimalFormat, and only applies when it is not "packed". The default is false, that is separate sign for "character" decimalFormat. The default is true for zoned and unpacked decimalFormat. The valid characters used for the overpunched signs vary depending on the character set.
	For input of overpunched sign decimal numbers, a single algorithm

	suffices to interpret all known (?) encodings unambiguously without any need for users to specify or know more than the character set being used. If the character set is EBCDIC (or any variant thereof) then bytes with high nibble of 0xF are just digits. Bytes with high nibble of 0xC indicate a plus sign and a digit, and 0xD for minus sign and a digit. If the character set is ASCII (or any variant thereof) then bytes with a high nibble of 0x3 are just digits. For a signed number a byte with high nibble of 0x3, or 0x2 is a plus sign and a digit, a byte with a high nibble of 0x7 is a minus sign and a digit. The byte 0x7B (character '{'} is a plus sign and the digit 0 (zero). The bytes 0x41 to 0x49 (characters 'ABCDEFGHI') are a plus sign and the digits 1 to 9. The byte 0x7D (character ')') is a minus sign and the digit 0. The bytes 0x4A to 0x52 (characters 'JKLMNOPQR') are a minus sign and the digits 1 to 9 respectively. Unfortunately, when writing out this overpunched sign representation, if the character set is EBCDIC or a variant, then the convention is unambiguous. If the character set is ASCII, then one must specify a value for a property (overpunchedASCIISignStyle) to control what convention is used for the sign encoding.
overpunchedASCIISignStyle	Valid values are "ebcdicConvertedStyle", "MFMSStyle", "CARealiaStyle". This property is primarily needed for output to resolve ambiguities. (TBD: On input this could also be used for input validation if you know the style that the numbers should be in.) The exact behavior, i.e., what characters are used to represent overpunched positive and negative digits, varies by character set and Cobol compiler and system. By far the most common convention is the EBCDIC one described here used on IBM mainframe systems. In EBCDIC character sets, overpunched characters '}JKLMNOPQR' represent a negative sign and digits 0 to 9. The characters '{ABCDEFGHI' represent a positive sign and digits 0 to 9. In ASCII character sets, there are several conventions and this is what we need this property for, to choose which one is to be used for output of data. EBCDIC converted style: The overpunched character is the ASCII equivalent of the EBCDIC above. In other words, characters '}JKLMNOPQR' still represent negative sign and digits 0 to 9 even though when looking at the bytes in hex this makes no sense whatsoever because the "}" character has ASCII code 0x7B. MicroFocus and Microsoft Cobol Style: ASCII characters 'pqrstuvwxy' represent negative sign and digits 0 to 9. ASCII characters '0123456789' represent a positive sign and the corresponding digit. (Sign nibble for '+' is 0x3, which is the high nibble of these character codes unmodified.) CA Realia Compiler Style: ASCII characters 'pqrstuvwxy' represent negative sign and digits 0 to 9. ASCII characters from code 0x20 to 0x29 are used for positive sign and the corresponding decimal digit. These characters include the space (' ') for zero, characters '!"#$%&' for 1 through 6, the single quote character "'" for 7, and the parenthesis '()' for 8 and 9. (TBD: any default? The default should be selected based on what style is found when data is read; however, what if a program isn't doing any reading? Then the default really is ambiguous. For a flag this obscure it is unpleasant to make it required. My guess is that ebcdicConvertedStyle is the most common, but that's a guess. -mikeb 2003-08-20) (TBD: other conventions may be out there. So, this set may get extended over time. This set is all I could find in my own documentation and by

searching the Internet with Google)

.

8.2.1 Example: Decimal big-endian, zoned, trailing overpunched sign

Let's consider this example using the EBCDIC character set (example from http://www.discinterchange.com/TechTalk_signed_fields_.html) :

This is how the value +123 would be stored with the sign in the LSD (least significant digit):

The 1 is stored as the EBCDIC character "1", which is 0xF1.

The 2 is stored as the EBCDIC character "2", which is 0xF2.

The 3, however, is converted to a binary value of 3, which is 03 hex, then OR'd with the positive sign code of 0xC0, resulting in the byte 0xC3.

The resulting item in hex is: F1, F2, C3. The proper way to interpret the C3 is as two entities: the sign of C0 and the value of 03. However, it also happens to be the code for the letter "C", so if you view the item in EBCDIC character mode, you will see "12C".

The value of +120 would likewise become F1, F2, C0 (00 OR'd with C0). Since the character assigned to the value C0 is a left brace, "{", when viewed in EBCDIC character mode, you will see "12{".

9. Nullability Properties

Property Name	Description
nullReservedValues	This property specifies a reserved value that is used to indicate null for in-band null representations. This property can appear more than once. In which case the set of all the values is the set of reserved null values that are all equivalent in indicating that the input value is null. On output the first one is generated to indicate null.
nullFlagDef	The value is the name of a element which holds the null indicator for this item. Usually this precedes the item; however, it is not required to precede the item. The nullReservedValues are then values for the nullFlagDef item to have if it is to represent null state for this item. TBD: path? Not just name?
nullFlagIndex	If the nullFlagDef indentifies an array, then this property gives an index within the array of the element which contains the nullFlag. (Note: Use case for this is Orchestrate DataSet format which stores null bits in a vector after the group. However, this additionally requires bit-arrays, i.e., support for bits as a data type.)

10. Date, Time, and Duration Properties

Handling date and time is a thankless job. There is an enormous variety of data formats in wide use for representing dates and times. Helpful standards like ISO 8601 standard date and time formats have been very late entrants to the world of data processing.

The proper capture of date and time information in DFDL involves not only describing the information stored in the data, but also describing much contextual information that is needed to understand the data but is not stored. For example, if the data is in a file and it contains dates and times in the format yyyy/mm/dd, hh:mm:ss, then we must know what time zone is assumed and also what calendar system is implied for the date. It is likely that data in this format is in the Gregorian calendar, which is by far the most important calendar for commercial purposes, but the time zone is completely ambiguous.

Consider this example item: “06/07/08 24:19:11.876”. The “06/07/08” is presumably a date, but what parts are what? Let’s assume we determine that the format is “DD/MM/YY”. It’s now ambiguous what year was intended by the “08” is that 2008, or 1908, so we need to know what year basis is used to disambiguate 2-digit years. Let’s assume it was 1908. A couple of other things are also ambiguous, though defaults are probably correct for most commercial data. These are the calendar being used, and the era. The calendar is probably Gregorian if the application is commercial; however, the last country to convert to the Gregorian calendar was Turkey (1926-01-01) prior to that Greece (1924-03-24) (source www.ssa.gov). Clearly there are many people still alive today whose birth dates are ambiguous because of this. The era is certainly CE (Common Era, a.k.a., AD in the en_US locale) and this should be the default setting; however, it should be possible to be specific, and there should at least be future extensibility to other calendars and their eras.

What about the time part “24:19:11.876”. Even if we assume the notation is hours, minutes and seconds, does this value, ‘24’, mean this time is invalid or is a 1-based hour-numbering being used here. (Some places/people count hours starting from 01:00 to 24:59.)

Date and time formats vary widely from one locale to another. Months and days of the week have different names of course, with a variety of accepted abbreviations, but the difficulties go further: For example, in Italian, the era which in English is notated “AD” or “CE” is notated “DC” (D for “dopo”), and the era notated in English as “BC” or “BCE” is notated “AC” (A for “avanti”). Of course either the English or the native Italian language form could be used.

Note also that 60 is valid for seconds, not just 0 to 59. This is not of commercial importance, but it is needed to accommodate leap-seconds which are added every few years. For example, the following is a valid ISO datetime: “1998-12-31T23:59:60+00:00”.

10.1 Default Formats for Dates, Times, and Duration

DFDL depends on ISO 8601 format for dates, times, and durations. This is the same as the standard XML syntax for dates and times.

Certainly for all data being created today, and for representing all dates and times since 1926-01-01, ISO 8601 textual formats are a highly recommended representation. For example these are two ISO 8601 date and time strings:

- 1926-01-01T00:00:00.000+00:00
- 1926-W01-1T00:00:00.000+00:00

(Note: This is the first truly unambiguous millisecond of time where we don't have to know whether the locale is Turkey or not. All times prior must be qualified as to what calendar applies.)

ISO 8601 also specifies durations unambiguously:

- "P18Y9M4DT11H9M8S" (18 years, 9 months, 4 days, 11 hours, 9 minutes and 8 seconds.
- "P2W" (2 weeks).

You can also specify durations between two times, and can anchor a duration to being before or after a specific time.

- Period with Specific Start: "19930214T131030/P18Y9M4DT11H9M8S"
- Period with Specific End: "P18Y9M4DT11H9M8S/19930214T131030"
- Specific Start and Specific End: "19930214T131030/19930214T131031" (one second)

10.2 Date, Time, and Duration Properties

Property Name	Description
locale	This property applies to textual data in general, but is repeated here to highlight that in some locales, time is measured in hours from 1 to 24, not 0 to 23.
yyEarliestYear	When converting two digit year representations into actual year values this property is used. It's value is the earliest year to be represented. E.g., if yyEarliestYear=1970, then the two digits "76" refer to the year 1976, but the digits "66" refer to 2066.
datePattern	See the Section: Date, Time, and Duration Patterns below.
timePattern	See the Section: Date, Time, and Duration Patterns below.
dateTimePattern	See the Section: Date, Time, and Duration Patterns below.
durationPattern	See the Section: Date, Time, and Duration Patterns below.

Note that the datePattern, timePattern, dateTimePattern, or durationPattern can be specified multiple times. When reading data these are used as alternatives for parsing the representation data. The order of these properties is the order that they are attempted. The first one that successfully parses the data is used. When data is being written out, all but the first pattern are ignored and data is written in the representation format specified by the first pattern.

(TBD: which one for writing? - the order for unambiguous reading may not put the desired output format first in the list. Perhaps need a specific indicator.)

10.3 Date, Time, and Duration Patterns

These pattern properties are consistent with the ICU software library (International Components for Unicode) which has been the basis of the Java language date/time features. This list of attributes and the pattern languages described here are intended to be consistent with the Java standard.

Textual representations are parsed by specifying a pattern string. The patterns show how to parse the various run-time information content components of Date, Time, or Duration information from the text. The information content values that can be parsed out of the representation include:

- Era : Common Era or Before Common Era
- Year
- Month
- DayOfMonth
- WeekOfYear
- DayOfWeek
- Hour
- Minute
- Second
- Millisecond
- Timezone: hours and minutes delta from Coordinated Universal Time (abbreviated UTC). Also known as Zulu time (Z) or Greenwich Mean Time (GMT).

For durations the set includes:

- Years
- Months
- Days
- Hours
- Minutes

- Seconds
- Milliseconds

(This table is taken from the ICU web site, and should be consistent with Java 1.4 also, but extended with the Kanji examples using XML entities.)

The date/time and duration textual format is specified by means of a string pattern. The count of pattern letters determines the format. In this pattern, letters are reserved as pattern letters:

Symbol	Meaning	Presentation	Example
G	era designator	(Text)	AD (Locale specific)
y	year	(Number)	1996
M	month in year	(Text and Number)	July and 07 (Locale specific)
d	day in month	(Number)	10
h	hour in am/pm (1~12)	(Number)	12
H	hour in day (0~23)	(Number)	0
m	minute in hour	(Number)	30
s	second in minute	(Number)	55
S	millisecond	(Number)	978
E	day in week	(Text)	Tuesday (Locale specific)
D	day in year	(Number)	189
F	day of week in month	(Number)	2 (2nd Wed in July)
w	week in year	(Number)	27
W	week in month	(Number)	2
a	am/pm marker	(Text)	pm (Locale specific. TBD?)
k	hour in day (1~24)	(Number)	24
K	hour in am/pm (0~11)	(Number)	0
Z (TBD: Java Internationalization book lists this as “z”, not “Z”)	time zone	(Text)	Pacific Standard Time (Locale Specific)
‘	escape for text		
“	single quote		‘

The 'Presentation' is interpreted as follows:

Text: Four or more, use full form, <4, use short or abbreviated form if it exists. (for example, "EEEE"" produces " Monday", "EEE" produces "Mon" in the US locale. In the FR locale, for the same value, "EEEE" produces "Lundi")

Number: The minimum number of digits. Shorter numbers are zero-padded to this amount (for example, if "m" produces "6", "mm" produces "06"). Year is handled specially; that is, if the count of 'y' is 2, the Year will be truncated to 2 digits. (for example, if "yyyy" produces "1997", "yy" produces "97".)

Both Text and Number: Three or over, use text, otherwise use number. (for example, "M" produces "1", "MM" produces "01", "MMM" produces "Jan", and "MMMM" produces "January". The text chosen is sensitive to the locale selected.)

Note: Any characters in the pattern that are not in the ranges of ['a'..'z'] and ['A'..'Z'] will be treated as quoted text. For instance, characters like ':', '.', ' ', '#' and '@' will appear in the resulting time text even if they are not enclosed within single quotes.

Format Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss Z"	1996.07.10 AD at 15:08:56 PDT
"EEE, MMM d, "yy"	Wed, July 10, '96
"h:mm a"	8:08 PM
"hh 'o"clock' a, ZZZZ"	09 o'clock AM. Eastern Standard Time
"K:mm a, Z"	9:34 AM, PST
"yyyyy.MMMMM.dd GGG hh:mm aaa"	1996.July.10 AD 12:08 PM
"yyyy年mm月dd日"	2003年08月27日
"yyyy年mm月dd日"	2003年08月27日

Note in the above the expansion of Unicode character entities is specifically allowed.

(TBD: It's a shame if we can't have a pattern that corresponds to ISO 8601 format. To do this we need to add pattern support for time zone presented numerically as in "00:00:00+06:00", need a way to specify that we want the "+06:00" not "CST" or "MDT" or "Central Standard Time". Need to designate a pattern letter for this. This is actually needed to process web data such as email, and possibly web logs of various kinds.)

11. References and Notes

- Orchestrate Schema Format (http://www.ascential.com/) - Not publicly available.

- Mercator Type Trees (http://www.mercator.com/) - Not publicly available.
- XML 1.0 http://www.w3.org/TR/REC-xml
- XML 1.1 http://www.w3.org/TR/xml11/
- Unicode (now at version 4.0) http://www.unicode.org/
- IANA character set encoding names: (http://www.iana.org/assignments/character-sets)
- XML Schema: http://www.w3.org/XML/Schema
- ISO COBOL standard: http://www.cobolstandard.info/wg4/wg4.html
- MicroFocus Cobol 1993 Manual (http://www.microfocus.com/) (online at http://docs.hp.com/cgi-bin/doc3k/BB243390044.15145/1)
- SAS: http://www.sas.com/ (also SAS 'Informats' http://www.ncsu.edu/it/sas/help/lrcon/z0920449.htm)
- POSIX/Unix Locale http://www.opengroup.org/onlinepubs/007908799/xbd/locale.html
- ICU: International Components for Unicode http://oss.software.ibm.com/icu/userguide/index.html
- ISO 8601 standard for dates and times: http://www.iso.ch/iso/en/prods%2Dservices/popstds/datesandtime.html
- The U.S. Social Security Administration Gives date of conversion of Greece (actually the Orthodox Catholic Church in Greece) to the Gregorian calendar as 1924-03-24. (see http://policy.ssa.gov/poms.nsf/lnx/0200307180). Turkey converted in 1926, but they converted the days in 1917 and converted the year in 1926.
- Leap Seconds: see http://tycho.usno.navy.mil/leap.html or search the web!
- Java Internationalization (ISBN 0-596-00019-7)
- IBM OS/390 DFSMS: Using Data Sets. IBM publication SC26-7339-01, Second Edition, December 2000. (online at: http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DGT1D411/CCONTENTS?SHELF=EZ239126&DN=SC26-7339-01&DT=20001014144419)

12. ToDo

Besides TBDs in the text:

Terminator - TBD: Can it also be a binary value used to delimit the end of an array? For example, an array of integers terminated by two final integers both zero. (Unix process environment data structures are terminated this way. So if you wanted to describe a Unix core dump in DFDL this is one way you could handle the environment part.)