Version 3 Data Types
PART II
Unabridged Specification

Chair/Editor: Gunther Schadow
Regenstrief Institute for Health Care
Editor: Paul V. Biron
Kaiser Permanente, Southern California

Preface

This document is one of two parts specifying the HL7 Version 3 Data Types on an abstract layer, independent of representation.

- Part I explains Version 3 Data Types in a style that can be readily understood by interface analysts and programmers.
- Part II provides a rigorous definition of Version 3 Data Types in a style that is suited for those readers with a strong academic background in Computer Science and Mathematics.

Each part can stand on its own and is addressed to different kinds of audience and slightly different purpose. Both parts are normative; therefore both parts must remain consistent in their description of Version 3 Data Types. Due to its more thorough nature, Part II takes precedence over Part I in cases of conflict or unclear interpretation. Casual readers who need a quick orientation into the matter should read Part I. However, for any serious implementation work understanding the additional detail provided in Part II is usually required.

This standard is further accompanied by one or more Implementable Technology Specifications (ITS) to specify the concrete representation of the Version 3 Data Types.

Acknowledgements

This standard is the result of two years of intense work through e-mail, telephone conferences and meeting discussions. Gunther Schadow (Regenstrief Institute for Health Care) chaired this task force, and is the main author of this document. Major contributions are from Mark Tucker (Regenstrief Institute), Paul V. Biron (Kaiser Permanente), George Beeler (Mayo), and Stan Huff (Intermountain Health Care), as well as Mike Henderson (Kaiser Permanente), Anthony Julian (Mayo), Joann Larson (Kaiser Permanente), Mark Shafarman (Oacis Healthcare Systems), Wes Rishel (Gartner Group), and Robin Zimmerman (Kaiser Permanente). Acknowledgements for their critical review and infusion of ideas go to Bob Dolin (Kaiser Permanente), Clem McDonald (Regenstrief Institute), Kai Heitmann (HL7 Germany), Rob Seliger (Sentillion), and Harold Solbrig (Mayo). Vital support came from the members of the task force, Laticia Fitzpatrick (Kaiser Permanente), Matt Huges, Randy Marbach (Kaiser Permanente), Larry Reis (Wizdom Systems), Carlos Sanroman (Kaiser Permanente), Greg Thomas (Kaiser Permanente). Thanks James Case (University of California, Davis), Norman Daoust (Partners HealthCare Systems), Irma Jongeneel (HL7 The Netherlands), Michio Kimura (HL7 Japan), John Molina (SMS), Richard Ohlmann (McKessonHBOC), Dawid Rowed (HL7 Australia), and Klaus Veil (Macquarie Health Corp., HL7 Australia), for sharing their expertise in critical questions. This work was made possible by the Regenstrief Institute for Health Care.

Contents

1 Introduction

1.1.1 What is a Data Type?

Every data element has a data type. Data types define the meaning (semantics) of data values that can be assigned to a data element. Meaningful exchange of data requires that we know the definition of values so exchanged. This is true for complex "values" such as business messages as well as for simpler values such as character strings or integer numbers.

According to ISO 11404, a data type is "a set of distinct values, characterized by properties of those values and by operations on those values." A data type has *intension* and *extension*. Intensionally, the data type defines the properties exposed by every data value of that type. Extensionally, data types have a set of data values that are of that type (the type's "value set").

Semantic properties of data types are what ISO 11404 calls "properties of those values and […] operations on those values." A semantic property of a data type is referred to by a name and has a value for each data value. The value of a data value's property must itself be a value defined by a data type – no data value exists that would not be defined by a data type.

Data types are thus the basic building blocks used to construct any higher order meaning: messages, computerized patient record documents, or business objects and their transactions. What, then, is the difference between a data type and a message, document, or business object? **Data type values stand for themselves, the value is all that counts, neither identity nor state or changing of state is defined for a data value.** Conversely in business objects, we track state and identity; the properties of an identical object might change between now and later. Not so with data values: a data value and its properties are constant. For example, number 5 is always number 5, there is no difference between this number 5 and that number 5 (no identity distinguished from value), number 5 never changes to number 6 (no change of state). One can think of data values as immutable objects where identity does not matter (identity and equality are the same.)[1]

1.1.2 Representation of Data Values

Data values can be represented through various symbols but the data value's meaning is not bound to any particular representation.

For example, cardinal numbers (non-negative integers) are defined – intensionally – as a data type where each value has a successor value, where zero is the successor of no other cardinal value. Based on this definition we can define addition, multiplication, and other mathematical operations. Whatever representation reflects the rules we stated in the intensional definition of the cardinal data type is a valid representation of cardinal numbers. Examples for valid cardinal number representations are decimal digit strings, bags of glass marbles, or scratches on a wall. The number two is represented by the word "five" by the Arabic number "5" or the Roman number "V". The representation does not matter as long as it conforms to the semantic definition of the data type.

Another example, the Boolean data type is defined by its extension, the two distinct values *true* and *false* and the rules of negation and combining these values in conjunction and disjunction. The representation of Boolean values can be the words "true" and "false," "yes" and "no," the numbers 0 and 1, any two signs that are distinct from each other. The representation of data types does not matter as long as it conforms to the semantic definition of the data type.

This specification defines the semantics, the meaning of the HL7 data types. **This specification is about semantics only, independent from representational and operational concerns or specific implementation technologies.** Additional standards for representing the data values defined here are

[1] The HL7 Message Development Framework defines "update modes" for fields in a message. Note that because data values have neither identity nor state nor changing of state, these update modes do not apply for the properties of data values. Data values and their properties are never updated. A field of an object (e.g., a message) can be updated in which case the field's value is replaced by another value. But the value itself is never updated.

being defined for various technological approaches. These standards are called "Implementable Technology Specification" (ITS.) Those ITS define how values are represented so that they conform to the semantic definitions of this specifications, this may include syntaxes for character or binary representations, and computer procedures to act on the representation of data values. The meaning of these ITS representations communicated, generated, and processed in computer programs, is defined based on this standard, the semantic data type specification.

1.1.3 Properties of Data Values

Data values have properties defined by their data type. The "fields" of "composite data types" are the most common example of such properties. However, more generally one should think of a data value's property as logical predicates or as mathematical functions; in simpler but still correct terms, properties are questions one can ask about a data value to receive another data value as an answer.

A property is referred to by its name. For example, the data type *integer* may have a property named "sign." A property has a domain, which is the set of possible "answer" values. The set of possible "answer" values is defined by the property's data type, but the domain of a property may be a subset of the data type's value set.

A property may also have arguments, additional information one must supply with a question to get an answer. For example, an important property of an integer number is that one integer *plus* another integer results in another integer, so the plus property of one integer needs an argument: the other integer.

Whether semantic properties have arguments is not a fundamentally relevant distinction. A data type's semantic property without arguments is not necessarily a "field" of a "composite" data type. For example, for integer values, we can define the property *is-zero* that has the Boolean value *true* when the number is zero and *false* when the number is not zero. This does not mean that *is-zero* must be an explicit component of any integer representation.

A data type's semantic property with arguments has no specific operational notions such as "procedure call," "passing arguments," "return values," "throwing exceptions," etc. These are all concepts of computer systems implementation of data types – but these operational notions are irrelevant for the semantics of data types.

This specification is about semantics of data types only. Neither is it about value representation syntax (not even an abstract syntax), nor is it about an operational interface to the data values.

1.1.4 Need for the Abstraction

Why does this specification make such a big issue about its being abstract from representation syntax as well as operational implementation?

HL7 needs this kind of abstract semantic data type specification for a very practical purpose. One important design feature of HL7 version 3 is its openness towards representation and implementation technologies. All HL7 version 3 specifications are supposed to be done in a form independent from specific representation and implementation technologies. HL7 acknowledges that, while at times some representation and implementation technologies may be more popular than others, technology is going to change – and with changing technology, representations of data values will change. HL7 standards are primarily targeted to healthcare domain information, independent from the technology supporting this information. HL7 expects that specifications defined independent from today's technology will continue to be useful, even after the next technological "paradigm shift".

The issue of data types is closer to implementation technology than most other HL7 information standards – and therein lays a certain danger that we define data types too dependent on current implementation technologies.

The majority of HL7 standards is about complex business objects. Complex business objects with many informational attributes can be specified as abstract syntax, where components are eventually

defined in terms of data types. Conversely, defining data types in terms of abstract syntax is of little use because the components of such abstract syntax constructs would still have to have data types.[2]

Why doesn't this specification define a set of primitive data types based on which composite data types could be defined simply as abstract syntax?

Any concrete implementation of the HL7 standards must ultimately use the built-in data types of their implementation technology. Therefore, we need a very flexible mapping between HL7 abstract data types and those data types built into any specific implementation technology. With a semantic specification, an Implementable Technology Specification (ITS) can conform simply by stating a mapping between the constructs of its technology and the HL7 version 3 data type semantics. Whether a data type is primitive of composite is irrelevant from a semantic perspective, and the answer may be different for different implementation technologies.

For example, this standard specifies a character string as a data type with many properties (e.g., charset, language, etc.) However, in many Implementation Technologies, character strings are primitive first class data types. We encourage that these native data types be used rather than a structure that slavishly represents all the semantic properties as "components." This specification only requires that the properties defined for data values can somehow be inferred from whatever representation is chosen, it does not matter how these values are represented. Whether "primitive" or "composite", with few or many "components", as "fields" or "methods" – this is all irrelevant.

For another example, a decimal representation, a floating-point register and a scaled integer are all possible native representations of real numbers for different implementation technologies. Some of these representations have properties that others do not have. Scaled integers, for instance, have a fixed precision and a relatively small range. Floating-point values have variable precision and a large range, but floating-point values lose any information about precision. Decimal representations are of variable precision and maintain the precision information (yet are slow to processing.) The data type semantics must be independent from all these accidental properties of the various representations, and must define the essential properties that any technology should be able to represent.

1.1.5 Need for an HL7 Data Type Standard

Why does HL7 need its own data type standard? Why can't HL7 simply adopt a standard defined by some other body?

As noted in the previous section, all HL7 implementation technologies have some data type system, but there are differences among the data type systems between implementation technologies. In addition, many implementation technologies' data type systems are not powerful enough to express the concepts that matter for the HL7 application layer.

For example, few implementation technologies provide the concepts of physical quantities, precision, ranges, missing information, and uncertainty that are so relevant in scientific and health care computing.

On the other hand, implementation technologies do make distinctions that are not relevant from the abstract semantics viewpoint, e.g., fixed point vs. floating-point real numbers; 8, 16, 32, or 64-bit integers; date vs. timestamp.

A number of data type systems have been used as input to this specification. These include the type systems of many major programming languages, including BASIC, Pascal, MODULA-2, C, C++, JAVA, ADA, LISP and SCHEME. This also includes type systems of language-independent implementation technologies, such as Abstract Syntax Notation One (ASN.1), Object Management Group's (OMG) Interface Definition Language (IDL) and Object Constraint Language (OCL), SQL 92 and SQL 3, the ISO 11404 language independent data types, and XML Schema Part 2 data types. Health care standards related data types have been considered as well, among these HL7 version 2.x, types used by CEN TC 251 messages and Electronic Health Record Architecture (EHCRA) and DICOM.

[2] This is the reason why the ISO Abstract Syntax Notation 1 (ASN.1) is not an appropriate formalism for semantic data type specifications.

1.1.6 Forms of Data Type Definitions

This specification defines data types in several forms, using textual description, UML diagrams, tables, and a formal definition.

1.1.6.1 Formal Data Type Definition Language

A formal definition of data types is used in order to clarify the semantics of the proposed types as unambiguously as possible. This data type definition language is described in detail in Section 1.3. Formal languages make crisp essential statement and are therefore accessible to some formal argument of proof or rebuttal. However, the terseness of such formal statements may also be difficult to understand by humans. Therefore, all the important inferences from the formal statements are also included as plain English statements.

1.1.6.2 Tables of Properties

For a quick overview at the beginning of many data types this specification contains tables listing what is called "primary" properties. "Primary" properties are a somewhat fuzzy notion of those properties that are more likely to be thought of as "fields" when the data type where implemented as a record ("composite data type"). These tables only exist to facilitate an overview of the content and purpose of data types. While their content is part of the normative specification, the fact that a property is or is not listed in these tables has no significance. There is no requirement that the properties listed in these tables be represented as fields, and these tables are **not** abstract syntax definitions.

Property tables are not shown for all data types. Again, this does not mean that those data types have no properties. It also does not mean that those data types are "primitive" data types as per this specification. The property tables are used as a helpful summary only, and are not used when they would confuse more than they would help.

Each row of the property tables describes one property with the following columns:

Name – the name of the property as of the formal definition. For some data types, the name field of the first property may be empty. This may happen in those data types that are defined as extension of other data types and when it is not useful for the summary of the child to show any properties of the parent.

Type – the data type of that property.

Status – indicates the "importance" of the property. Status values include:

Default – a default value is a typical value of this property (when the status of .the property is "fixed," the default specifies that predetermined value.) In an interface, a property takes on that default value if a value for that property is not otherwise determined in the representation received by that interface. Default values are what is implicitly understood. A default may be determined at various levels.

Table 1: Levels of Determination for Defaults

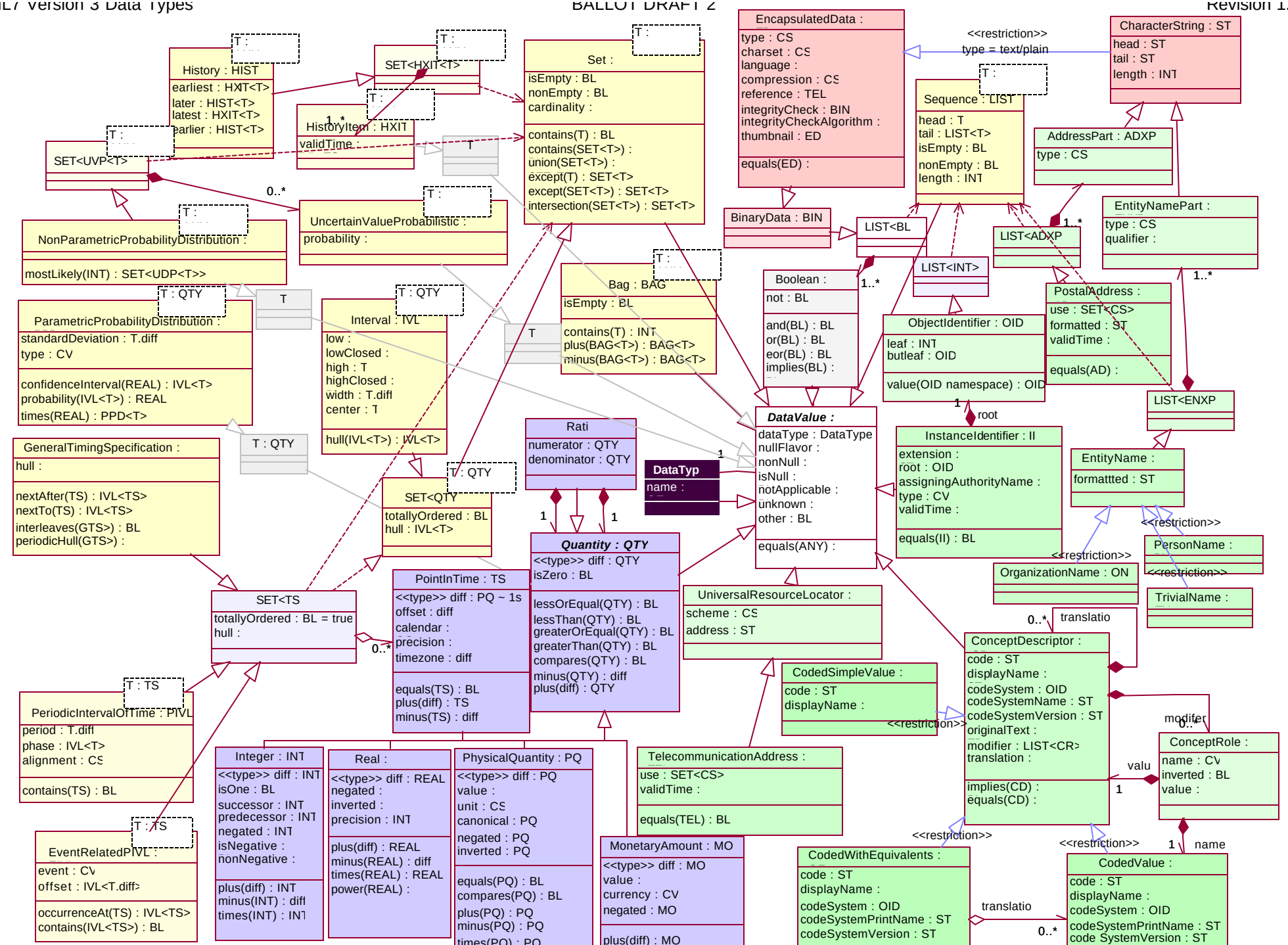
Status	Definition
immediate	The default is known and mentioned in this specification. The default column then contains that a literal for that default value (not the word "immediate").
NULL	No default is defined.
ITS	The default is determined by the Implementable Technology Specification (ITS) either immediately or through some ITS-specified rule.
CONTEXT	The default is determined depending on the context of a data value. For example, it may be determined by some other data in the same message.

Constraint – for coded properties, this column contains the named domain (as per the HL7 domain specifications.) For other properties, this column contains other constraints – these constraints are fully specified in the formal data type definition.

Definition – a short text describing the meaning of the property.

1.1.6.3 Unified Modeling Language (UML) Diagrams

The Unified Modeling Language (UML) is used for a graphical presentation of how data types relate. Data types are shown as UML classes. The name compartment contains the long name of the data type followed by a colon and the standard abbreviation. Properties of types without arguments are shown in the UML attribute compartment. Properties with arguments are shown in the UML operations compartment. Generalization links indicate extension and restriction relationships. Aggregations are an *additional* representation of properties, when the relation between data types through that property is important. Generic types are shown as UML parameterized classes, with UML realization links relating their instantiations.



1.2 Overview of Data Types

Table 2: Overview of HL7 Version 3 Data Types

Name	Symbol	Description
Boolean	BL	The Boolean type stands for the values of two-valued logic. A Boolean value can be either true or false.
Encapsulated Data	ED	Data that is primarily intended for human interpretation or for further machine processing outside the scope of this specification. This includes unformatted or formatted written language, multi-media data, or structured information in as defined by a different standard (e.g., XML-signatures.) Instead of the data itself, an ED may contain only a reference (see TEL.) Note that the ST data type is a specialization of the ED data type when the ED media type is text/plain.
Character String	ST	Text data, primarily intended for machine processing (e.g., sorting, querying, indexing, etc.) Used for names, symbols, and formal expressions.) Note that the ST data type is a specialization of the ED data type when the ED media type is text/plain.
Coded Simple Value	CS	Coded data, consists of a code and display name. The code system and code system version is fixed by the context in which the CS value occurs. CS is used for coded attributes that have a single HL7-defined value set.
Coded Value	CV	Coded data, consists of a code, display name, code system, and original text. Used when a single code value must be sent.
Coded With Equivalents	CE	Coded data, consists of a coded value (CV) and, optionally, coded value(s) from other coding systems that identify the same concept. Used when alternative codes may exist.
Concept Descriptor	CD	Coded data, is like a CE with the extension of modifiers. Modifiers for codes have an optional role name and a value. Modifiers allow one to express, e.g., “FOOT, LEFT” as a postcoordinated term built from the primary code “FOOT” and the modifier “LEFT”.
Coded With Category	CC	A specific restriction of the CD used to communicate a possibly local code with another code that specifies the category of the communicated concept in an HL7-defined standard code.
Instance Identifier	II	An identifier to uniquely identify an individual instance. Examples are medical record number, order number, service catalog item number, etc. Based on the ISO Object Identifier (OID)
Telecommunication Address	TEL	A telephone number or e-mail address specified as a URL. In addition, this type contains a time specification when that address is to be used, plus a code describing the kind of situations and requirements that would suggest that address to be used (e.g., work, home, pager, answering machine, etc.)
Postal Address	AD	For example, a mailing address. Typically includes street or post office Box, city, postal code, country, etc.
Entity Name	EN	A name of a person, organization, place, or thing. Can be a simple character string or may consist of several name parts that can be classified as given name, family name, nickname, suffix, etc.
Person Name	PN	A name of a person. Person names usually consist of several name parts that can be classified as given, family, nickname etc. PN is a restriction of EN.
Organization Name	ON	A name of an organization. ON name parts are typically not distinguished, but may distinguish the suffix for the legal standing of an organization (e.g. “Inc.”, “Co.”, “B.V.”, “GmbH”, etc.) from the name itself. ON is a restriction of EN.
Trivial Name	TN	A restriction of EN that is equivalent with a plain character string (ST). Typically used for the names of things, where name parts are not distinguished.
Integer Number	INT	Positive and negative whole numbers typically the results of counting and enumerating. The standard imposes no bounds on the size of integer numbers.
Real Number	REAL	Fractional numbers. Typically used whenever quantities are measured, estimated, or computed from other real numbers. The typical representation is decimal, where the number of significant decimal digits is known as the precision.
Physical Quantity	PQ	A dimensioned quantity expressing the result of measurement. It consists of a real number value and a physical unit. Physical quantities are often constrained to a certain dimension by specifying a unit representing the dimension (e.g. m, kg, s, kcal/d, etc.) However, physical quantities should not be constrained to any particular unit (e.g., should not be constrained to centimeter instead of meter or inch.)
Monetary Amount	MO	The amount of money in some currency. Consists of a value and a currency denomination (e.g., U.S.$, Pound sterling, Euro, Indian Rupee.)

Ratio	RTO	A quantity explicitly including both a numerator and a denominator (e.g. 1:128.) Only in the rare cases when the numerator and denominator must stand separate should the Ratio data type should be used. Normally, the REAL, PQ, or MO data types are more appropriate.
Point in Time	TS	A time stamp.
Set Collection	SET<T>	An unordered collection of unique values of any type T.
List Collection	LIST<T>	A sequence of values of any type T.
Bag Collection	BAG<T>	An unordered set of values of any type T where each value can occur more than once (rare.)
Interval	IVL<T>	Ranges (intervals) of values of type T. An interval is a set of consecutive values of any quantity data type, such as, integer, real number, point in time, physical quantity, monetary amount, and ratio.) Intervals should be preferred instead of two attributes expressing a start and an end separately.
History	HIST<T>	A collection of data where each element is tagged with a valid-time interval.
Uncertain value using probabilities	UVP<T>	A nominal value with a probability number indicating the level of certainty for the value to apply in the given context.
Non-parametric probability distribution	NPPD<T>	A collection of alternative uncertain values. Used to represent frequency distributions (histograms) but also other weighed alternatives (e.g., utility distributions in preferences.
Parametric probability distribution	PPD<T>	A probability distribution used to indicate certainty (accuracy) of a quantitative value. Allows specifying a distribution type and applicable parameters. All distribution types have the parameters mean and standard distribution. The mean is the value that would be reported if no probability distribution were available.
General Timing Specification	GTS	One or more time intervals used to specify the timing of events. Every event spans one time interval (occurrence interval). A repeating event is timed through a sequence of such occurrence intervals. Such timings are often specified not directly as a sequence of intervals but as a rule, e.g., "every other day (Mon – Fri) between 08:00 and 17:00 for 10 minutes."

1.3 Introduction to the Formal Data Type Definition Language

Important Disclaimer: This is not an API specification. While this formal language might resemble some programming language or interface definition language, it is not intended to define the details of programs and other means of implementation. The formal definitions are normative part of this specification, but this particular language needs not be implemented or used in conformant systems; nor need all the semantic properties be implemented or used by conformant systems. The internal working of systems, their way to implement data types, their **functionality and services is entirely out of scope of this specification. The formal definition only specifies the meaning** of the data values through making statements how one would theoretically expect these values to relate and behave.

This formal data type definition language[3] specifies:

- type name and short name;
- named values of a fully enumerated extension;
- semantic properties, unary, binary, and higher order properties;
- invariants, i.e. constraints over the properties.

[3] The data type definition language employed here is a conclusion of experiments and experience with various alternatives. These alternatives include data type definition tables and the use of the Object Management Group's (OMG) Interface Definition Language (IDL). The disadvantage of the data type definition tables was that they gave the wrong impression of this specification being a specification of abstract syntax rather than semantics. Conversely, the disadvantage with IDL was that IDL gave the wrong impression of this specification being an application programming interface (API) definition.
The resulting data type definition language borrows significantly from IDL, the Object Constraint Language (OCL), JAVA, C++, and the parser generation tools LEX and YACC. It is inspired by features and style of these languages but amalgamating and augmenting these languages into precisely what is needed for this data type specification. The goal was a language that is minimal, and self-consistent. Also, as the main purpose of this language is to define data types it tries to get by without any built-in data types.

- allowable type conversions;
- syntax of character string value literals (if any);

Definition of a data type occurs in two steps. First, the data type is declared. The declaration claims a name for a new data type with a list of names, types, and signatures of the new type's semantic properties. This declares, not defines the type. The definition occurs in both logic statements about what is always true about this type's values and their properties (invariant statements.)

1.3.1 Declaration

Every data type is declared in a form that begins with the keyword **`type`**. For example, the following is the header of a declaration for the data type Boolean that has the short name alias BL and extends (specializes) the data type ANY.[4]

```
type Boolean alias BL extends ANY
    values(true, false)
{
            BL          not;
            BL          and(BL x);
};
```

The Boolean data type declaration also contains a **`values`**-clause that declares the Boolean's complete set of values (its extension) as named entities. These named values are also valid character string literals. None of the other data types defined in this specification has a finite value set, which is why the **`values`**-clause is unique to the Boolean. In the marked-up formal language, value names use Italics font.

The block in curly braces following the header contains declarations of the semantic properties that hold for every value of the data type. A semicolon terminates each property declaration; and another semicolon after the closing curly brace terminates the data type declaration.

A property declaration mentions from left to right: (1) the data type of the property's value domain, the property name, and (3) an optional argument list. The argument list of a property is enclosed in parentheses containing a sequence of argument declarations. Each argument is declared by the data type name and argument name. Semantic properties without arguments do not use an empty argument list.[5]

The **`extends`**-clause has the usual meaning of a specialization relationship known from the object-oriented method.[6] Specialization means (a) inheritance of properties from the genus to the species, and (b) substitutability of values of the species type for variables of the genus type. In addition, however, this data type definition language specifies two variants of specialization: extension (**`extends`**) and restriction (**`restricts`**). Extension indicates that additional properties are being defined for the specialized type. Restriction indicates that the inherited properties are being constrained.

[4] As can be seen, the **`type`** keyword is in place of IDL's and Java's **`interface`** and C++ amd Java's **`class`** keyword. The alias clause is unique to this specification as we do have the need for extremely short data type mnemonics in addition to more descriptive names. The **`extends`** clause is the same as JAVA's, which is preferred over C++ or IDL's colon clause as its meaning is more obvious.

[5] Note that the IDL's notion of input and output arguments and IDL's, JAVA's and C++'s notion of return values and exceptions are all irrelevant concepts for this specification. The semantics of data types is not about procedure calls and parameter passing or normal and abnormal returns of control from a procedure body. Instead, each semantic property is conceptualized as a function that maps a value and optional arguments to another value. This mapping is not "computed" or "generated" it logically exists and we do not need to "call" such a function to actualize the mapping.

[6] "Extends" means "refines" or "specializes and adds properties." This kind of "extension" (specialization) has nothing to do with the "extensional" (vs. "intensional") definitions of data types.

An example for inheritance is: when ANY has the property isNull and BL extends ANY then BL also has this property isNull even though isNull is not listed explicitly in the property declaration of BL. An example for substitutability is: when a property is declared as of a data type ANY and BL extends ANY then a value of such property may be of type BL. In other words, substitutability is the same as subsumption of all values of type BL being also values of type ANY.[7]

The **`type`**-declaration may be qualified by the keyword **`abstract`** and **`protected`**. An abstract type is a type where no value can be just of this type without belonging to a concrete specialization of the abstract. A protected type is a type that is used inside this specification but no property outside this specification should be declared of a protected type.[8] (We also use the qualifier **`private`** at one point. Private types are only specified for the sake of formal definition of other types and are not used in any form outside this specification.)

1.3.2 Invariant Statements

The declaration of semantic properties, their names, data types, and arguments provide only clues as to what the new data type might be about. The true definition lies in the invariant statements. Invariant statements are logical statements that are true at all times.

Throughout this specification, invariant statements are provided in a formal syntax but are also written in plain English. The advantage of the formal syntax is that it can be interpreted unambiguously, and that it is strongly typed. The advantage of plain English statements is that they are more understandable, especially to those untrained in reading formal languages.

The formal syntax does help to sharpen the decisiveness of this specification. In some cases, however, the full semantics of a type are beyond what can be fully expressed in such invariant statements. The combination of both plain and formal language helps to make this specification more clear.

Invariant statements are formed using the **`invariant`** keyword that declares one or more variables in the same form as an argument list of a property. The invariant statement can contain a **`where`** clause that constrains the arguments for the entire invariant body. The invariant body is enclosed in curly braces. It contains a list of assertions that must all be true.

```
invariant(BL x) where x.nonNull {
  x.and(true).equals(x);
};
```

The semantics of the invariant statement is a logic predicate with a universal quantifier ("for all").

The above invariant statement can be read in English as "For all Boolean values x, where x is non-NULL it holds that x AND *true* equals x." All properties should be named such that one can read the assertions like English sentences.[9]

[7] The restriction variant of specialization deserves explanation. It is generally touted that inheritance should not retract properties that have been defined for the genus. This is still true for the restriction as properties are not actually retracted but constrained to a smaller value set. This may mean constraining properties to NULL, if NULL was an allowed value for that property in the parent type. In any case, logically, restriction is a specialization, with inheritance and substitutability. Furthermore extends and restricts are not hard opposites as a specialized type may both extend and constrain; the two keywords are mainly used to be comprehensible to a human reader.

[8] Note the meaning of *protected* is a little different from the accessibility qualifiers (public, package, protected, private) as known from JAVA and C++. The protection used here is not about hiding the type information or baring properties defined by a protected type from access outside of this specification "package." It mainly is a strong recommendation not to declare attributes or other features of such protected types. Protected types should be used as "wrapped" in other types. The protected type is still directly accessible within the "wrap," no notion of "delegated properties" exists.

[9] The invariant statement syntax and semantics is similar to the OCL "inv" clause. We did not use OCL in this specification, however, for several reasons. (1) OCL syntax has a Smalltalk style that does not fit the C++/Java style of the data type definition language. (2) OCL has many primitive constructs and data types, while this specification avoids many primitives. (3) In part because of the richness in primitive constructs, OCL is fairly complex, more than is needed in this specification.

The argument list of an invariant statement need not be specified if no such argument is needed.

```
invariant {
  true.not.equals(false);
  false.not.equals(true);
};
```

1.3.2.1 Assertion Expressions

Assertions in invariant statements are expressions built with the semantic properties of defined data types. Assertion expressions must have a Boolean value (*true* or *false*.)[10] No primitive data types, or operations, pre-exist the definition of any data type. The only preexisting features of the assertion expression language are:[11]

- character strings representing utterances in the data type definition language;
- the notion of an assertion being successful (*true*) or failing (*false*);
- the invariant statement: **`invariant`**`(...)` **`where`** `... {...};`
- the universal quantifier expression form **`forall`**`(...)` **`where`** `... {...};` synonymous to the invariant statemen;
- the existence quantifier expression form **`exists`**`(...)` **`where`** `... {...};`
- the implicit conjunction (logical AND) between the semicolon-separated assertions: $assertion_1$`;` $assertion_2$`; ... ;` $assertion_n$`;`
- variables and declarations in the invariant argument list;
- the property reference using the period: *x*`.`*property*;
- implicit and explicit type conversion: `(`*T*`)`*x*;
- parentheses to override the priorities of the conversion and property resolution operators: `(`*T*`)`*x*`.`*property* versus `((`*T*`)`*x*`).`*property*.

1.3.2.2 Nested Quantifier Expressions

Within assertion expressions, nested quantifier statements can be formed similar to invariant statements. In fact, the universal quantifier built using the **`forall`** keyword is the same as the invariant statement. The universal quantifier can be used in a nested expression when the complexity of the problem requires it, such as in the following example:

[10] This construct is somewhat cyclical, there is a preexisting notion of Boolean values even though the Boolean is a type defined just like any other type. In addition, since this data type definition language is written in character strings, the notion of character strings pre-exists the definition of the character string type. These two types, character string and Boolean are therefore exceptional, but on the surface, they are defined just like any other data type. Since this data type specification language is not meant to be implemented, the cyclicality is not a real issue. Even if this language was implemented, one can use a "bootstrapping" technique as is common, e.g., for compilers that compile themselves.

[11] Most of these syntactic features are in the spirit of the JAVA language, use of argument lists, curly braces to enclose blocks, semicolon to finish a statement, and the period to reference value properties. The double colon :: as used by C++ or IDL to distinguish between member-references and value-references are not used (as in Java). Unlike Java but like C++ and IDL, every statement is ended by a semicolon, including type declarations. Implicit type conversion is also retained from C++.


```
invariant(SET<T> x, y) where x.nonNull {
  x.subset(y).equals(
      forall(T element) where x.contains(element) {
        y.contains(element);
      });
};
```

The existence quantifier has the meaning as in common propositional logic. For example, the following invariant means: "SET values *x* and *y* intersect if and only if there exists an element *e* that is contained in both sets *x* and *y*."

```
invariant(SET x, y) where x.nonNull {
  x.intersects(y).equals(
      exists(T e) {
        x.contains(e);
        y.contains(e);
      });
};
```

The existence quantifier may have a where-clause, however, there is no difference whether an assertion is made as a where-clause or in the body of the existence quantifier. Conversely, for universal quantifiers, the where-clause weakens the assertion since the body now only applies for values that meet the criterion in the where-clause.

1.3.3 Type Conversion

This specification defines certain allowable conversions between data types. For example, there is a pair of conversions between the Character String (ST) and Encode Data (ED). This means that if a one expects an ED value but actually has an ST value instead, one can turn the ST value into an ED.[12]

Three kinds of type conversions are defined: promotion, demotion, and character string literals. Type conversions can be implicit or explicit. Implicit type conversion occurs when a certain type is expected (e.g. as an argument to a statement) but a different type is actually provided. If the type provided has a conversion to the type expected the conversion should be done implicitly.

ITS Note: an Implementation Technology Specification will have to specify how implicit type conversions are supported. Some technologies support it directly others do not; in any case, processing rules can be set that specify how these conversions are realized.

An explicit conversion can be specified in an assertion expression using the converted-to type name in parenthesis before the converted value. For example the following is an explicit type conversion in the where clause of an invariant statement.

```
invariant(ED x) where ((ST)x).nonNull { ... };
```

The type conversion has lower priority than the property resolution period. Thus "`(T)a.b`" converts the value of the property *b* of variable *a* to data type *T* while "`((T)a).b`" converts the value of variable *a* to *T* and then references property *b* of that converted value.

[12] These type conversions add necessary flexibility to support inter-version compatibility and localization. Note: HL7 v2.x used to have implicit type conversions as a side effect of its delimiter-based syntax. It was thus possible for the specification to define additional components to a field, or change the data type of a field (e.g., ID to CE) and still maintain backward compatibility.

Implicit type conversions in the assertion expressions are performed where possible. If a property's formal argument is declared of data type *T*; but the expression used as an actual argument is of type *U*; and if *U* does not extend *T*; and if *U* defines a conversion to *T*, that conversion from *T* to *U* takes effect.

1.3.3.1 Demotion

A demotion is a conversion with a net loss of information. Generally, this means that a more complex type is converted into a simple type.

An example for a demotion is the conversion from Interval (IVL) to a simple Quantity (QTY), e.g. the center of the interval. In the data type definition language, a demotion is declared using the keyword **`demotion`** and the data type name to which to demote:

```
type Interval alias IVL {
  ...
  demotion   QTY;
  ...
};
```

The specification of demotions shall indicate what information is lost and what the major consequences of losing this information are.

1.3.3.2 Promotion

A promotion is a conversion where new information is generated. Generally, this means that a simpler type is converted into a more complex type.

For example, we allow any Quantity (QTY) to be converted to an Interval (IVL). However, IVL has more semantic properties than QTY, low and high boundary. Thus, the conversion of QTY to IVL is a promotion. The additional properties of QTY not present in IVL must assume new values, default values, or computed values. The specification of the promotion must indicate what these values are or how they can be generated.

A promoting conversion from type QTY to type IVL is defined as a semantic property of data type QTY using the keyword promotion and the data type name to which to promote:

```
type Quantity alias QTY {
  ...
  promotion  IVL;
  ...
};
```

Typically, a promotion is defined from a simple type to a more complex type. Also typically, the simple type is declared earlier in this document than a more complex type. Declaring all promotions to complex types in the simple type would thus involve forward references and would be confusing to the reader. Therefore, an alternative syntax allows promotions to be defined in the more complex type. This is indicated by naming the type from which to promote in an argument list behind the type to which to promote.


```
type Interval alias IVL {
  ...
  promotion  IVL        (QTY x);
  ...
};
```

1.3.4 Literal Form

A literal is a character string representation of a data value. Literals are defined for many types, simple types and types that are more complex. A literal is a type conversion from and to a specially formatted Character String (ST).

Not every conversion from and to an ST is a literal conversion (e.g., the ED/ST conversion is not a literal.) A literal for a data type should be able to represent the entire value set of a data type (the ED/ST conversion can not represent the entire value set of ED.)

The purpose of having literals is so that one can write down values in a short human readable form. For example, literals for the types Integer (INT) and Real (REAL) are strings of sign, digits, possibly a decimal point, etc. The more important Interval types (IVL<REAL>, IVL<PQ>, IVL<TS>) have literal representations that allow one to use, e.g., "<5" to mean less than 5, which is much more readable than a fully structured form of the Interval. For some of the more advanced data types such as intervals, general timing specification, and parametric probability distribution we expect that the literal form may be the only form seen for representing these values until users have become used to the underlying conceptualizations.

Each literal conversion has its own syntax (grammar,) often aligned with what people find intuitive. This syntax may therefore not be completely straightforward from a computer's perspective.[13]

ITS Note: Character string based Implementable Technology Specifications (ITS) of these abstract data types may or may not choose the literals defined here as a their representations for these data types. For the XML ITS we expect that some of the literals defined here be used.

1.3.4.1 Declaration

In the data type definition language we declare a literal form as a property of a data type using the keyword literal followed by the data type name ST, since the literal is a conversion to and from the ST data type.

```
type IntegerNumber alias INT {
  ...
  literal   ST;
  ...
};
```

1.3.4.2 Definition

The actual definition of the literal form occurs outside the data type declaration body using an attribute grammar. An attribute grammar is a grammar that specifies both syntax and semantics of language structures. The syntax is defined in essentially the Backus-Naur-Form (BNF).[14]

[13] The different grammars of literals are not meant to be combined into one overall HL7 value expression grammar. Although attempt have been made to resolve potential ambiguities between the literals of different types where they would be harmful, some of these ambiguities still remain. For example "1.2" can be a valid literal for both Object Identifier (OID) and a Real Number.

For example, consider the following simple definition of a data type for cardinal numbers (positive integers.) This type definition depends only the Boolean data type (BL) and has a character string literal declared:

```
type CardinalNumber alias CARD {
            BL          isZero;
            BL          equals(CARD x);
            CARD        successor;
            CARD        plus(CARD x);
            CARD        timesTen;
  literal   ST;
};
```

1.3.4.2.1 Syntax Definition

The literal syntax and semantics is first exposed completely and then described in all detail.

```
CARD.literal ST {
  CARD
  : CARD digit          { $.equals($1.timesTen.plus($2); }
  | digit               { $.equals($1); };

  CARD digit
  : "0"                 { $.isZero; }
  | "1"                 { $.equals(0.successor); }
  | "2"                 { $.equals(1.successor); }
  ...
  | "8"                 { $.equals(7.successor); }
  | "9"                 { $.equals(8.successor); }
};
```

Every syntactic rule consists of the name of a symbol, a colon and the definition (so called *production*) of the symbol. A production is a sequence of symbols. These other symbols are also defined in the grammar, or they are terminal symbols. Terminal symbols are character strings written in double quotes or string patterns (called *regular expressions*.) Thus the form:

```
  CARD : CARD digit | digit;
```

means, that any cardinal number symbol is a cardinal number symbol followed by a digit or just a digit. The vertical bar stands for a disjunction (logical OR.) A syntactic rule ends with a semicolon.

Every symbol has exactly one value of a defined data type. The data type of the symbol's value is declared where the symbol is defined:

[14] The BNF variant used here is similar to the YACC parser and LEX lexical analyzer generator languages but is simplified and made consistent to the syntax and declarative style of this data type definition language. The differences are that all symbols have exactly one attribute, their value strongly typed as one of the defined data types. Each symbol's type is declared in front of the symbol's definition (e.g.: `INT digit : "0" | "1" | … | "9";`). The start symbol has no name but just a type (e.g., `INT : digit | INT digit;`). A data type name can occur as a symbol name meaning a literal of that data type.


```
CARD digit : “0” | “1” | “2” | ... | “8” | “9”;
```

means that the symbol *digits* has a value of type CARD. The start-symbol is the data type itself and does not need a separate name.

1.3.4.2.2 Semantics Definition

The semantics of the literal expression is specified in semantic rules enclosed in curly braces for each of the defined productions of a symbol:

symbol **:** *production*$_1$ **{** *rule*$_1$ **}** **|** *production*$_2$ **{** *rule*$_2$ **}** **|** … **|** *production*$_n$ **{** *rule*$_n$ **};**

A semantic rule is simply a semicolon-separated list of Boolean assertion expressions of the same kind as those used in invariant statements. However, there are special variables defined in the semantic rule that all begin with a dollar character (e.g., $, $1, $2, $3, …) The simple $ stands for the value of the currently defined symbol; while $1, $2, $3, etc. stand for the values of the parts of the semantic rule's associated production. For example, in

```
CARD
: CARD digit          { $.equals($1.timesTen.plus($2); }
| digit               { $.equals($1); };
```

the first production "`CARD digit`" has a semantic rule that says: the value `$` of the defined symbol equals the value `$1` of the first symbol `CARD` times ten plus the value `$2` of the second symbol `digit`.[15]

1.3.4.2.3 Terminal Symbols

A terminal symbol can be specified as a string pattern, so-called *regular expression*. The regular expression syntax used here is the classic syntax invented by Aho and used in AWK, LEX, GREP, and PERL. Regular expressions appear between two slashes **/**…**/**. In a regular expression pattern every character except **[ ] ^ $. / : ( ) \ | ? * + { }** matches itself. The other characters that are actually used in this specification are defined in Table 3.

Table 3: Special Characters for Regular Expressions

Pattern	Definition
[…]	Specifies a character class. For example, `/[A-Za-z]/` matches the characters of the upper and lower case English alphabet.
[^ …]	Specifies a character class negatively. For example, `/[^BCD]/` matches any character except B, C, and D.
…?	The preceding pattern is optional. For example, `/ab?c/` matches "`ac`" and "`abc`".
…*	The preceding pattern may occur zero or many times. For example, `/ab*c/` matches "`ac`", "abc", "`abbc`", "`abbbc`", etc.
…+	The preceding pattern may occur one or more times. For example, `/ab+c/` matches "`abc`","abbc", "`abbbc`", but not "`ac`".
… {*n*,*m*}	The preceding pattern may occur *n* to *m* times where *n* and *m* are cardinal numbers $0 \le n \le m$. For example, `/ab{2,4}c/` matches "`abbc`", "`abbbc`", and "`abbbbc`".
… | …	The pattern on either side of the bar may match. For example, `/ab|cd/` matches "`abd`" and "acd" but not "`abcd`".
(…)	The pattern in parentheses is used as one pattern for the above operators. For example, `/a(bc)*/` matches "a", "`abc`", "`abcbc`", "`abcbcbc`", etc.
… : …	The left pattern matches if followed by the right pattern, but the right pattern is not consumed by a

[15] Note that the equals property (defined for all data types, see Section 1.4.2.3) is a relation, a test for equality, not an assignment statement. One can not assign a value to another value. Unlike YACC and LEXanalyzers, this data type definition language is purely declarative it has no concept of assignment. For this reason, the grammar rules define both parsing and building literal expressions.

	match. For example, `/ab:c/` matches "`abc`" but not "`ab`", however, the value of a symbol thus matched is "`ab`" and the "`c`" is left over for the next symbol. The colon is a slight deviation from the conventional slash / but the slash is also conventionally used to enclose the entire pattern and may occur as a character to match – three meanings is one too many.
... \ ...	Matches the following character literally, i.e. escapes from any special meaning of that character. For example, `/a\+b/` matches "`a+b`".
... \/ ...	Matches the slash as a character. For example, `/a\/bc/` macthes "`a/bc`".

1.3.5 Generic Data Types

Generic data types are incomplete type definitions. This incompleteness is signified by one or more *parameters* to the type definition. Usually parameters stand for other types. Using parameters, a generic type might declare semantic properties of other not fully specified data types. For example, the generic data type Interval is declared with a parameter *T* that can stand for any Quantity data type (QTY). The components *low* and *high* are declared as being of type *T*.

```
template<QTY T>
type Interval<T> alias IVL<T> {
            T          low;
            T          high;
};
```

Instantiating a generic type means completing its definition. For example, to instantiate an Interval, one must specify of what *base data type* the interval should be. This is done by *binding* the parameter *T*. To instantiate an Interval of Integer numbers, one would bind the parameter *T* to the type Integer. Thus, the incomplete data type Interval is completed to the data type *Interval of Integer*.

For example the following type definition for MyType declares a property named "multiplicity" that is an interval of the cardinal number data type used in the above examples.

```
type MyType alias MT {
            IVL<CARD>  multiplicity;
};
```

1.3.5.1 Generic Collections

Generic data types for collections are being used throughout this specification. The most important of them are

Set (SET<T>.) A set contains elements in no particular order and without duplicate elements. The SET<T> data type requires all elements of a set to be of the same data type.

Sequence (LIST<T>.) A sequence is a collection of values in an arbitrary but particular order. A sequence has a head and a tail, where the head is an element and the tail is the sequence without its head.

Interval (IVL<T>.) An interval is a continuous subset of an ordered type.

These and other generic types are fully defined in Section 1. These generic data types and their properties are being used in this specification early on. For the best understanding of this specification knowledge about the set, sequence and interval is important and the reader is advised to refer to Section 1 when coming across a generic type being used to define another type.

1.3.5.2 Generic Type Extensions

Generic data type extensions are generic types with one parameter type that the generic type extends. In the formal data type definition language, generic type extensions follow the pattern:


```
template<ANY T> type GenericTypeExtensionName extends T { ... };
```

These generic type extensions inherit properties of their base type and add some specific feature to it. The generic type extension is a specialization of the base type, thus a value of the extension data type can be used instead of its base data type.[16]

ITS Note: values of extended types can be substituted for their base type. However, an ITS may make some constraints as to what extensions to accommodate. Particularly, extensions need not be defined for those components carrying the values of data value properties. Thus, while any data value can be annotated outside the data type specification, an ITS may not provide for a way to annotate the value of a data value property.

1.4 Data Type and Data Value

1.4.1 Data Type (type)

This section defines the fundamental properties of all data types and all data values. The type DataType is a meta-type declared in order to allow the formal definitions to speak about the data type of a value. Any data type defined in this specification is a value of the type DataType.

```
protected type DataType extends DataValue {
            CE          name;
};
```

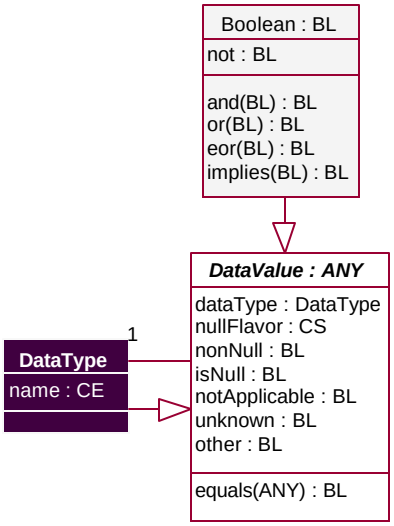


Figure 1: Fundamental data types

1.4.1.1 Name : CE

A data type name is a code with equivalents (CE, see Section 2.4.4.3). The short alias name, if defined, is the main code value, in which case the long name is an equivalent translation in the CE value.

1.4.2 Data Value (ANY)

The type DataValue defines the basic properties of every data value. This is an abstract type, meaning that no value can be just a data value without belonging to any concrete type. Every concrete type is a specialization of this general abstract DataValue type.

```
abstract type DataValue alias ANY {
            DataType    dataType;
            BL          nonNull;
            CS          nullFlavor;

            BL          isNull;
            BL          notApplicable;
            BL          unknown;
            BL          other;

            BL          equals(ANY x);
};
```


1.4.2.1 dataType : Data Type

Every data value is of a data type. The data value implicitly carries the information about its own type. Thus, given a data value in an HL7 message, one can inquire about its data type.

```
invariant(ANY x) {
  x.dataType.nonNull;
};
```

1.4.2.2 Exceptional Values (NULL-Values)

Exceptional values express missing information and possibly the reason why the information is missing. Exceptional values are also called NULL-values, and the exception is called the "flavor" of NULL.

Thus, every data value is either a proper value or it is NULL. If the value is NULL, the *nullFlavor* property is non-NULL. If the value is not NULL, its *nullFlavor* property is NULL (not applicable.)

```
invariant(ANY x) {
  x.nonNull.equals(x.nullFlavor.isNull);
  x.isNull.equals(x.nonNull.not);
};
```

Table 4: Flavors of NULL

Concept	Symbol	Implies	Definition
no information	NI		No information whatsoever can be inferred from this exceptional value. This is the most general exceptional value. It is also the default exceptional value.
not applicable	NA	NI	No proper value is applicable in this context (e.g., last menstrual period for a male.)
unknown	UNK	NI	A proper value is applicable, but not known.
not asked	NASK	UNK	This information has not been sought (e.g., patient was not asked)
asked but unknown	ASKU	UNK	Information was sought but not found (e.g., patient was asked but didn't know)
temporarily unavailable	NAV	ASKU	Information is not available at this time but it is expected that it will be available later.
other	OTH		The actual value is not an element in the value domain of a variable. (e.g., concept not provided by required code system.)[17]
positive infinity	PINF	OTH	Positive infinity of numbers.
negative infinity	NINF	OTH	Negative infinity of numbers.
not present	*NP*		*Value is not present in a message. NP is a presentation layer concept only, **not** and application layer concept! All values not*

[16] Generic type extensions are sometimes called "mixins", since their effect is to *mix* certain properties *in*to the preexisting data type.

[17] There is a fine difference in coded data types between NULL/*other* and "*coded with extensibility*" (CWE.) The CWE concept is defined elsewhere (*HL7 Development Framework*). CWE vocabulary domains include any pertinent local coding system. Since CWE domains include every locally defined concept, there is hardly any case where a concept is not within that value domain. Thus, NULL/*other* hardly ever occurs for CWE fields outside of applications. However, an interface that cannot interpret the local code used for a not otherwise coded concept will still map such local-coded value to NULL/*other*, because it might not be able or willing to expand its interpretable value domain.
For example, if the standard domain for administrative gender contains only the concepts *male* and *female*, and the concept *intersex* needs coding, *intersex* might be coded using a local code that extends the gender code. However, a receiving system that does not know about that non-standard code for *intersex* will map the unknown code to NULL/other. Alternatively, the sending system could have used NULL/other instead of its local code in the first place. For CWE fields, the local code is allowed, for CNE (coded, non-extensible) fields NULL/*other* is the only legal way.

present in a message must be replaced by the applicable default value, or no-information (NI) if no other default value is defined.

Note the distinction between value domain and vocabulary domain. A vocabulary domain is a value domain for coded values, but not all value domains are vocabulary domains. The null flavor other is used whenever the actual value is not in the required value domain, this may be, for example, when the value exceeds some constraints that are defined too restrictive (e.g., age less than 100 years.)

Some of these null flavors are defined as named properties that can be used as simple predicates for all data values. This is done to simplify the formulation of invariants in the remainder of this specification. Note the difference between semantic properties and representational "components" of data values. An ITS must only represent those components that it needs to *infer* the semantic properties. The null-flavor predicates nonNull, isNull, notApplicable, unknown, and other can all be inferred from the nullFlavor property.

```
invariant(ANY x) {
  x.notApplicable.equals(x.nullFlavor.implies(NA));
  x.unknown.equals(x.nullFlavor.implies(UNK));
  x.other.equals(x.nullFlavor.implies(OTH));
};
```

When a property, RIM attribute, or message field is called *mandatory* this means that any non-NULL value of the type to which the property belongs must have a non-NULL value for that property, in other words, a field may not be NULL, providing that its container (object, segment, etc.) is present.

In other HL7 specifications the label "mandatory" is used, while this specification formulates the mandatory constraint explicitly. For example, the following invariant says that the dataType property is *mandatory* for any data value that is non-NULL.

```
invariant(ANY x) where x.nonNull {
  x.dataType.nonNull;
}
```

ITS Note: NULL-flavors are applicable to any property of a data value or a higher-level object attribute. Where the difference of null flavors is not significant, ITS are not required to represent them. If nothing else is noted in this specification, ITS need not represent general NULL-flavors for data-value properties.

1.4.2.3 Equality

Any two data values can be tested for equality. Equality is a reflexive, symmetric, and transitive relation. Only values of the same data type can be equal.

```
invariant(ANY x, y, z)
  where x.nonNull.and(y.nonNull).and(z.nonNull)
{
  x.equals(x);                                          /* reflexivity */
  x.equals(y).equals(y.equals(x));                      /* symmetry */
  x.equals(y).and(y.equals(z)).implies(x.equals(z))     /* transitivity */
  x.equals(y).implies(x.dataType.equals(y.dataType);
}
```


How equality is determined must be defined for each data type. If nothing else is specified, two data values are equal if they are indistinguishable, that is, if they differ in none of their semantic properties. A data type can "override" this general definition of equality, by specifying its own equals relationship. This overriding of the equality relation can be used to exclude semantic properties from the equality test. If a data type excludes semantic properties from its definition of equality, this implies that certain properties (or aspects of properties) that are not part of the equality test are not essential to the meaning of the value.

For example the physical quantity has the two semantic properties (1) a real number and (2) a coded unit of measure. The equality test, however, must account for the fact that, e.g., 1 meter equals 100 centimeter; independent equality of the two semantic properties is too strong a criterion for the equality test. Therefore, physical quantity must override the equality definition.

Note: with data values, no distinction exists between equality and identity. Equality is a static property between two values, and values never change.

2 Basic Types

2.1 Boolean (BL)

The Boolean type stands for the values of two-valued logic. A Boolean value can be either "true" or "false". With any data value potentially being NULL, the two-valued logic is effectively extended to a three-valued logic as shown in the following truth tables:

Table 5: Truth tables for Boolean logic with NULL values

NOT	
true	*false*
false	*true*
NULL	NULL

AND	*true*	*false*	NULL
true	*true*	*false*	NULL
false	*false*	*false*	*false*
NULL	NULL	*false*	NULL

OR	*true*	*false*	NULL
true	*true*	*true*	*true*
false	*true*	*false*	NULL
NULL	*true*	NULL	NULL

```
type Boolean alias BL extends ANY
    values(true, false)
{
            BL          and(BL x);
            BL          not;

  literal   ST;

            BL          or(BL x);
            BL          eor(BL x);
            BL          implies(BL x);
};
```

The literal form of the Boolean is determined by the named values specified in the values clause.

2.1.1.1 Negation

Negation of a Boolean turns *true* into *false* and *false* into *true* and is NULL for NULL values.

```
invariant(BL x) {
  true.not.equals(false);
  false.not.equals(true);
  x.isNull.equals(x.not.isNull);
};
```

2.1.1.2 Conjunction

Conjunction (AND) is associative and commutative, with *true* as a neutral element. *False* AND any Boolean value is *false*. These rules hold even if one or both of the operands are NULL.

```
invariant(BL x) {
  x.and(true).equals(x);
  x.and(false).equals(false);
};
```


If both operands for AND are NULL, the result is NULL.

```
invariant(BL x, y) where x.isNull.and(y.isNull) {
  x.and(y).isNull;
};
```

2.1.1.3 Disjunction

The disjunctions OR and exclusive OR can be specified in terms of negation and conjunction. The disjuntion *x* OR *y* is *false* if and only if *x* is *false* and *y* is *false*. The exclusive-OR constrains OR such that *x* and *y* may not both be *true*.

```
invariant(BL x, y) {
  x.or(y).equals(x.not.and(y.not).not);
  x.eor(y).equals(x.or(y).and(x.and(y).not));
};
```

2.1.1.4 Implication

The logical implication is important to make invariant statements. An implication is a rule of the form IF *condition* THEN *conclusion*. Logically the implication is defined as the disjunction of the negated condition and the conclusion, meaning that when the condition is true the conclusion must be true to make the overall statement true.

```
invariant(BL condition, conclusion) {
  condition.implies(conclusion).equals(condition.not.or(conclusion));
};
```

The implication is not reversible and does not specify what is true when the condition is false (*ex falso quodlibet*).

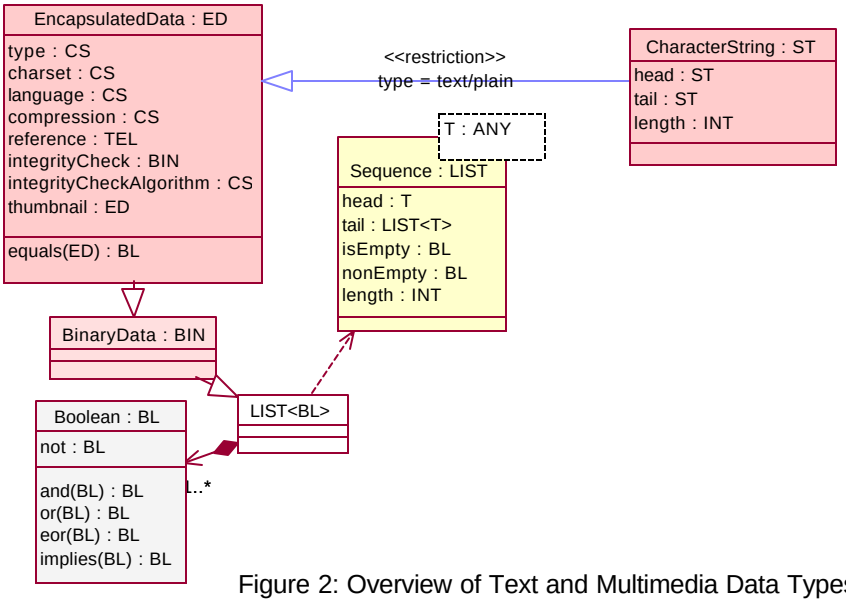


Figure 2: Overview of Text and Multimedia Data Types

2.2 Encapsulated Data (ED)

The encapsulated data (ED) type can convey any data. ED is a wrapper around a block of binary data. Therefore, binary data must be defined first.

Table 6: Summary of Primary Properties of Encapsulated Data (ED)

Name	Type	Status	Default	Constraint	Definition
	BIN	mandatory	NULL		The binary data.
type	CS	mandatory	*text/plain*	IANA: MIME media types	Identifies the encoding of the data and a method to interpret the data.
charset	CS	implied	**ITS**	IANA: charset	Where applicable, specifies the character set and character encoding used.
language	CS	implied	**CONTEXT**		Where applicable, specifies the language of text data.
compression	CS	optional	NULL		Indicates whether the raw byte data is compressed, and what compression algorithm was used.
reference	TEL	optional	NULL		A telecommunication address that resolves to the binary data.
integrityCheck	BIN	optional	NULL		A short binary value representing a cryptographically strong checksum over the binary data.
integrityCheckAlgorithm	CS	fixed	*SHA-1*		Specifies the algorithm used to compute the integrityCheck value.
thumbnail	ED	optional	NULL		An abbreviated rendition of the full data.

2.2.1 Binary Data (BIN)

Binary data is a sequence of uninterpreted bits. A bit is identical with a Boolean value. Thus, all binary data is – semantically – a sequence of Boolean values. The binary data type is protected; it should not be used directly but only inside the encapsulated data (ED).

```
protected type BinaryData alias BIN extends LIST<BL>;
```

ITS Note: the representation of arbitrary binary data is the responsibility of an ITS. How the ITS accomplishes this depends on the underlying Implementation Technology (whether it is character-based or binary) and on the so represented data. Semantically character data is represented as binary data, however, a character-based ITS should not convert character data into arbitrary binary data and then represent binary data in a character encoding. Ultimately even character-based implementation technology will communicate binary data.

An empty sequence is not considered binary data but counts as a NULL-value. In other words, non-NULL binary data contains at least one bit.

```
invariant(BIN x) where x.nonNull {
  x.nonEmpty;
  x.length.greaterThan(0);
};
```

2.2.2 Properties of Encapsulated Data (ED)

The encapsulated data (ED) type can convey any data. ED is based on binary data (BIN), however, in order for that binary data to convey meaning, it must be decoded and further interpreted. ED adds to BIN a mechanism to specify the method of interpretation of the binary data. Encapsulated data may be

a plain character string, formatted text, or any of several kinds of multimedia data. The kind of encoding is conveyed in three properties:

type – specifies the protocol, or application used to decode and interpret the data (also called the "media type" when referring to multi-media data.)

charset – identifies the character set and character encoding for character-based "media."

compression – data may be given in a compressed form in which case compression identifies the compression algorithm used.

Encapsulated data can be present in two forms, inline or by reference. Inline data is communicated or moved as part of the encapsulated data value, whereas by-reference data may reside at a different (remote) location. The data is the same whether it is located inline or remote.

```
type EncapsulatedData alias ED extends BIN {
            CS          type;
            CS          charset;
            CS          language;
            CS          compression;

            TEL         reference
            BIN         integrityCheck;
            CS          integrityCheckAlgorithm;

            ED          thumbnail;

            BL          equals(ED x);
};
```

2.2.2.1 type : CS

Identifies the encoding of the data and identifies the method to interpret or render the data. The domain of the encapsulated data's type property are the MIME media types, defined by the *Internet Assigned Numbers Authority* (IANA).

The encapsulated data's type is a mandatory property, i.e., every non-NULL instance of encapsulated data must have a defined type property.

```
invariant(ED x) where x.nonNull {
  x.type.nonNull;
};
```

The IANA defined domain of media types is established by the Internet standard RFC 2046 [ftp://ftp.isi.edu/in-notes/rfc2046.txt]. RFC 2046 defines the media type to consist of two parts:

top level media type, and

media subtype.

However, this specification treats the entire media type as one atomic code symbol in the form defined by IANA, i.e., top level type followed by a slash "/" followed by media subtype. Currently defined media types are registered in a database [http://www.isi.edu/in-notes/iana/assignments/media-types] maintained by IANA. Currently more than 160 different MIME media types are defined, with the list growing rapidly. In general, all those types defined by the IANA may be used.

To prevent the interoperability-problems associated with this diversity, this specification prefers certain media types to others. This is to define a greatest common denominator on which interoperability is not only possible, but that is powerful enough to support even advanced multimedia communication needs.

Table 7 below assigns a status to certain MIME media types, where the status means one of the following:

required

Every HL7 application must support at least the required media types if it supports a given kind of media. One required media-type for each kind of media exists. Some media types are required for a specific purpose, which is then indicated as "*required for …*"[18]

recommended

Other media types are recommended for a particular purpose. For any given purpose there should be only very few additionally recommended media types and the rationale, conditions and assumptions of such recommendations must be made very clear.

indifferent

This status means, HL7 does neither forbid nor endorse the use of this media type. All media types not mentioned here by default belong into the *indifferent* category. Since there is one required and several recommended media types for most practically relevant use cases, media types of this status should be used very conservatively.

deprecated

Deprecated media types should not be used, because these media types are flawed, because there are better alternatives, or because of certain risks. Such risks could be security risks, for example, the risk that such a media type could spread computer viruses. Not every flawed media type is marked as deprecated, though. A media type that is not mentioned, and thus considered *other* by default, may well be flawed.

Table 7: Use of MIME media types

Media Type	Status	Use Case
text/plain	required **default**	For any plain text. This is the default and is equivalent to a character string (ST) data type.
application/x-cda-level-1	required *for HL7 CDA documents*	For XML documents according to the HL7 CDA Level 1 standard.
text/x-hl7-ft	recommended *for compatibility*	For compatibility, this represents the HL7 v2.x FT data type. Its use is recommended only for backward compatibility with HL7 v2.x systems.
text/html	recommended	For marked-up text according to the Hypertext Mark-up Language. HTML markup is sufficient for typographically marking-up most written-text documents. HTML is platform independent and widely deployed.
application/pdf	recommended	The Portable Document Format is recommended for written text that is completely laid out and read-only. PDF is a platform independent, widely deployed, and open specification with freely available rendering tools.
text/sgml text/xml	other	For structured character based data. There is a risk that general SGML/XML is too powerful to allow a sharing of general SGML/XML documents between different applications.
text/rtf	other	The Rich Text Format is widely used to share word-processor documents. However, RTF does have compatibility problems, as it is quite dependent on the word processor. May be useful if word

[18] The set of required media types, however, is very small so that no undue requirements are forced on HL7 applications, especially legacy systems. In general, no HL7 application would be forced to support any given kind of media other than written text. For example, many systems just do not want to receive audio data, because those systems can only show written text to their users. It is a matter of application conformance statements to say: "I will not handle audio". Only if a system claims to handle audio media, it must support the required media type for audio.

		processor edit-able text should be shared.
application/msword	deprecated	This format is very prone to compatibility problems. If sharing of edit-able text is required, text/plain, text/html or text/rtf should be used instead.
audio/basic	required *for audio*	This is a format for single channel audio, encoded using 8bit ISDN mu-law [PCM] at a sample rate of 8000 Hz. This format is standardized by: CCITT, Fascicle III.4 –Recommendation G.711. *Pulse Code Modulation (PCM) of Voice Frequencies*. Geneva, 1972.
audio/mpeg	recommended *for CD quality audio*	MPEG-1 Audio layer-3 is an audio compression algorithm and file format defined in ISO 11172-3 and ISO 13818-3. MP3 has an adjustable sampling frequency for highly compressed telephone to CD quality audio.
audio/k32adpcm	recommended *for audio compression*	ADPCM allows compressing audio data. It is defined in the Internet specification RFC 2421 [ftp://ftp.isi.edu/in-notes/rfc2421.txt]. Its implementation base is unclear.
image/png	required *for images*	Portable Network Graphics (PNG) [http://www.cdrom.com/pub/png] is a widely supported lossless image compression standard with open source code available.
image/gif	other	GIF is a popular format that is universally well supported. However GIF is patent encumbered and should therefore be used with caution.
image/jpeg	required *for high color images*	This format is required for high compression of high color photographs. It is a "lossy" compression, but the difference to lossless compression is almost unnoticeable to the human vision.
image/g3fax	recommended *for FAX*	This is recommended only for fax applications.
video/mpeg	required *for video*	MPEG is an international standard, widely deployed, highly efficient for high color video; open source code exists; highly interoperable.
video/x-avi	deprecated	The AVI file format is just a wrapper for many different codecs; it is a source of many interoperability problems.
model/vrml	recommended *for 3D models*	This is an openly standardized format for 3D models that can be useful for virtual reality applications such as anatomy or biochemical research (visualization of the steric structure of macromolecules)
application/x-dicom	other	For HIS/PACS interface applications that need to encapsulate DICOM data in HL7 data.

2.2.2.2 charset : CS

For character-based encoding types, this property specifies the character set and character encoding used. The charset is defined according to Internet RFC 2278, *IANA Charset Registration Procedures*, [http://www.isi.edu/in-notes/rfc2278.txt].

The charset domain is maintained by the *Internet Assigned Numbers Authority* (IANA) [http://www.isi.edu/in-notes/iana/assignments/character-sets]. The IANA source specifies names and multiple aliases for most character sets. For the HL7's purposes, use of multiple alias names is not allowed. The standard name for HL7 is the one marked by IANA as "preferred for MIME." If IANA has not marked one of the aliases as "preferred for MIME" the main name shall be the one used for HL7.

Table 8 lists a few of the IANA defined character sets that are of interest to current HL7 members. The definitions of the "status" column is as given for Table 7.

Table 8: Select Character Set Codes as defined by IANA

Code	Status	Description
US-ASCII	required	ANSI X3.4-1968
UTF-8	required *for Unicode*	8 bit Unicode Transfer Format [RFC 2279]. This is the default character set (ISO 10646/Unicode) and encoding for XML and natively supported by Java. It is backward compatible to 7-bit US-ASCII.
ISO-10646-UCS-2	deprecated	Unicode ISO 10646, the 16 bit per character Basic Multilingual Plane. Unicode has a special protocol to specify the byte order, which must be followed. To avoid byte ordering problems (and – for the western part of the world – to conserve bandwidth) the UTF-8 encoding should be used.
ISO-10646-UCS-4	deprecated	Unicode ISO 10646, the full code-set (32-bit per character.) Unicode has a special protocol to specify the byte order, which must be followed. To avoid byte ordering problems (and – for the western part of the world – to

		conserve bandwidth) the UTF-8 encoding should be used.
UTF-7	indifferent	7 bit Unicode Transfer Format [RFC 2152]. This is a Unicode encoding that is sure to be safe for older communication links or file formats that remove the 7th bit of each transferred byte.
ISO-8859-1	indifferent	ISO 8859 Latin-1 character set is native on western European (and U.S.) Microsoft Windows installations and on many Unix/X-Windows systems.
ISO-8859-2	indifferent	ISO 8859 Latin-2 character set for the Slavic languages of Central Europe (Polish, Czech).
ISO-8859-5	indifferent	ISO 8859 Cyrillic character set for the languages Russian, Bulgarian, Byelorussian, Macedonian, Serbian and Ukrainian.
JIS-2022-JP	indifferent	ISO 2022 is a character-encoding framework in which multilingual code-pages can be switched in and out. JIS-2022-JP, is ISO 2022 as released as a Japanese Information Standard and as the Internet specification *Japanese Character Encoding for Internet Messages* [RFC 1468].
EBCDIC	indifferent	Extended binary-coded decimal interchange code. A coded character set of 256 8-bit characters commonly used by IBM mainframes.

2.2.2.3 language : CS

For character based information the language property specifies the language of the text.[19] The principles of the code domain of this attribute is specified by RFC 1766, *Tags for the Identification of Languages* [http://www.isi.edu/in-notes/rfc1766.txt]. It is a set of pre-coordinated pairs of one 2-letter ISO 639 language code and one 2-letter ISO 3166 country code.[20]

Language tags do not modify the meaning of the characters found in the text; they are only an advice on if and how to present or communicate the text.[21]

ITS Note: representation of language tags to text is highly dependent on the ITS. An ITS should use the native way of language tagging provided by its target implementation technology. Some may have language information in a separate component, e.g., XML has the xml:lang tag for strings. Others may rely on language tags as part of the binary character string representation, e.g., ISO 10646 (Unicode) and its "plane-14" language tags.

The language tag should not be mandatory if it is not mandatory in the implementation technology. Semantically, language tagging of strings follows a default-logic. If nothing else is specified the local language is assumed. If a language is set for an entire message or document, that language is the default. If any information element or value that is superior in the syntax hierarchy specifies a language, that language is the default for all subordinate text values.

If language tags are present in the beginning of the encoded binary text (e.g., through Unicode's plane-14 tags) this is the source of the language property of the encapsulated data value.

2.2.2.4 compression : CS

Indicates whether the raw byte data is compressed, and what compression algorithm was used.

Table 9: Compression Algorithms

Name	Code	Status	Description and Comment
deflate	DF	required	The "deflate" compressed data format as specified in RFC 1951 [ftp://ftp.isi.edu/in-notes/rfc1951.txt].
gzip	GZ	other	A compressed data format that is compatible with the widely used GZIP utility as specified in RFC 1952 [ftp://ftp.isi.edu/in-notes/rfc1952.txt] (uses the *deflate* algorithm.)
zlib	ZL	other	A compressed data format that also uses the *deflate* algorithm. Specified as RFC 1950 [ftp://ftp.isi.edu/in-notes/rfc1950.txt]

[19] The need for a language code for text data values is documented in RFC 2277, *IETF Policy on Character Sets and Languages* [http://www.isi.edu/in-notes/rfc2277.txt]. Further background information can be found in *Using International Characters in Internet Mail* [http://www.imc.org/mail-i18n.html], a memo by the Internet Mail Consortium.

[20] RFC 1766 is the HL7-approved coding system for all reference to human languages, in data types and elsewhere.

[21] For this reason, a system or site that does not deal with multilingual text or names in the real world can safely ignore the language property.

compress	Z	deprecated	Original UNIX compress algorithm and file format using the LZC algorithm (a variant of LZW). Patent encumbered and less efficient than *deflate*.

Compression may not be allowed for encapsulated data depending on the attribute or component that is declared encapsulated data. Character strings (see Section 2.3) may never be compressed.

2.2.2.5 reference : TEL

A telecommunication address (TEL), such as a URL for HTTP or FTP, which will resolve to precisely the same binary data that could as well have been provided as inline data.

The semantic value of an encapsulated data value is the same, regardless whether the data is present inline data or just by-reference. However, an encapsulated data value without inline data behaves differently, since any attempt to examine the data requires the data to be downloaded from the reference.

An encapsulated data value may have both inline data and a reference. The reference must point to the same data as provided inline.

By-reference encapsulated data may not be allowed depending on the attribute or component that is declared encapsulated data. Character strings (see Section 2.3) must always be inline.

2.2.2.6 integrityCheck : BIN

The integrity check is a short binary value representing a cryptographically strong checksum that is calculated over the binary data. The purpose of this property, when communicated with a reference is for anyone to validate later whether the reference still resolved to the same data that the reference resolved to when the encapsulated data value with reference was created.

The integrity check is calculated according to the integrity check algorithm. By default, the *Secure Hash Algorithm-1* (SHA-1) shall be used. The integrity check is binary encoded according to the rules of the integrity check algorithm.

The integrity check is calculated over the raw binary data that is contained in the data component, or that is accessible through the reference. No transformations are made before the integrity check is calculated. If the data is compressed, the Integrity Check is calculated over the compressed data.

2.2.2.7 integrityCheckAlgorithm : CS

Specifies the algorithm used to compute the integrityCheck value.[22]

Table 10: Integrity Check Algorithm

Name	Code	Description
Secure Hash Algorithm – 1	SHA-1	This algorithm is defined in FIPS PUB 180-1: *Secure Hash Standard*. As of April 17, 1995.

2.2.2.8 thumbnail : ED

A thumbnail is an abbreviated rendition of the full data.[23] A thumbnail requires significantly fewer resources than the full data, while still maintaining some distinctive similarity with the full data. A

[22] The cryptographically strong checksum algorithm *Secure Hash Algorithm-1* (SHA-1) is currently the industry standard. It has superseded the MD5 algorithm only a couple of years ago, when certain flaws in the security of MD5 were discovered. Currently the SHA-1 hash algorithm is the default and required only choice for the integrity check algorithm. However, there is no assurance that SHA-1 will not be superseded at anytime when its flaws will be discovered.

[23] Originally, the term thumbnail refers to an image in a lower resolution (or smaller size) than another image. However, the thumbnail concept can be metaphorically used for media types other than images. For example, a movie may be represented by a shorter clip; an audio-clip may be represented by another audio-clip that is shorter, has a lower sampling rate, or a lossy compression.

thumbnail is typically used with by-reference encapsulated data. It allows a user to select data more efficiently before actually downloading through the reference.

Thumbnails may not be allowed depending on the attribute or component that is declared encapsulated data. Character strings (see Section 2.3) never have thumbnails, and a thumbnail may not itself contain a thumbnail.

```
invariant(ED x) where x.thumbnail.nonNull {
  x.thumbnail.thumbnail.isNull;
};
```

ITS Note: the ITS should consider the case where the thumbnail and the original both have the same properties of type, charset and compression. In this case, these properties need not be represented explicitly for the thumbnail but might be "inherited" from the main encapsulated data value to its thumbnail.

2.2.2.9 Equality

Two values of type Encapsulated Data are equal if and only if their type and referenced data are equal. For those ED values with compressed data or remote data, only the de-referenced and uncompressed data counts for the equality test. The compression and reference property themselves are excluded from the equality test, as is the thumbnail and the language property. If the ED.type is character based and the charset property is not equal, the charset property must be resolved through mapping of the data between the different character sets.

The integrity check algorithm and integrity check is excluded from the equality test. However, since equality of integrity check value is strong indication for equality of the data, the equality test can be practically based on the integrity check, given equal integrity check algorithm properties.

2.3 Character String (ST)

The character string is a restricted encapsulated data type (ED), whose type property is fixed to *text/plain*, and whose data must be inlined and not compressed. Thus, the properties compression, reference, integrity check, algorithm, and thumbnail are not applicable. The character string data type is used when the appearance of text does not bear meaning, which is true for formalized text and all kinds of names.

Table 11: Summary of Primary Properties of Character String (ST)

Name	Type	Status	Default	Constraint	Definition
	BIN	mandatory	NULL		The binary data of the character string.
type	CS	fixed	*text/plain*	IANA: MIME media types	
charset	CS	implied	**ITS**	IANA: charset	Specifies the character set and character encoding used.
language	CS	implied	**CONTEXT**		Specifies the language of text data.

The character string (ST) data type interprets the encapsulated data as character data (as opposed to bits), depending on the charset property of the encapsulated data type.

```
type CharacterString alias ST restricts ED {
            INT        length;
            ST         head;
            ST         tail;
};
```



```
invariant(ST x) where x.nonNull {
  x.type.equals("text/plain");
  x.compression.notApplicable;
  x.reference.notApplicable;
  x.integrityCheck.notApplicable;
  x.integrityCheckAlgorithm.notApplicable;
  x.thumbnail.notApplicable;
}
```

ITS Note: because many of the properties of the encapsulated data are bound to a default value, an ITS need not represent these properties at all. In fact, if the character encoding is also fixed, the ITS only represents the encoded character data.

The character string inherits the properties head, tail, and length from BIN (via ED). These properties head, tail, and length, are redefined so that the character string appears as a sequence of entities each of which uniquely identifies one character from the joint set of all characters known by any language of the world.[24] The properties head, tail, and length therefore refer to character, string, and character counts respectively, rather than bits and bit counts.

The head of a string is a string of only one character. A character string must at least have one character or else it is NULL. The length of a character string is the number of characters in the string. A zero-length string is an exceptional value (NULL), not a proper character string value.

```
invariant(ST x) where x.nonNull {
  x.head.nonEmpty;
  x.head.tail.isEmpty;

  x.tail.isEmpty.implies(x.length.equals(1));
  x.tail.nonEmpty.implies(x.length.equals(x.tail.length.successor));
};
```

The length of a string is the number of characters, not the number of encoded bytes. Byte encoding is an ITS issue and is not relevant on the application layer.

2.3.1.1 Literal Form

Two variations of character string literals are defined, a token form and a quoted string.[25] The token form consists only of the lower case and upper case English alphabet, the ten decimal digits and the

[24] ISO/IEC 10646-1: 1993 defines a character as "A member of a set of elements used for the organisation, control, or representation of data." ISO/IEC TR 15285 – *An operational model for characters and glyphs*. Discusses the problems involved in defining characters. Notably, characters are abstract entities of information, independent of type font or language. The ISO 10646 (UNICODE [http://www.unicode.org]) – or in Japan, JIS X0221 – is a globally applicable character set that uniquely identifies all characters of any language in the world.
In this specification, ISO 10646 serves as a semantic model for character strings. The important point is that for semantic purposes, there is no notion of separate character sets and switching between character sets. Character set and character encoding are ITS layer considerations. The formal definition gives indication to this effect because each character is by itself an ST value that has a charset property. Thus, the binary encoding of each character is always understood in the context of a certain character set. This does not mean that the ITS should represent a character string as a sequence of full blown ED values. What it means is that on the application layer the notion of character encoding is irrelevant when we deal with character strings.

[25] A character string literal is a conversion from a character string to another data type. Obviously, character string literals for character strings is a cyclical if not redundant feature. This literal form, therefore, mainly specifies how character strings are parsed in the data type specification language.

underscore. The quoted string can contain any character between double-quotes. The double quotes prevent a character string from being interpreted as some other literal. The token form allows keywords and names to be parsed from the data type specification language.

```
ST.literal ST {
  ST : /"[^]*"/        { $.equals($1); }          /* quoted string */
     | /[a-zA-Z0-9_]+/ { $.equals($1); };         /* token form */
};
```

ITS Note: since character string literals are so fundamental to implementation technology, most ITS will specify some modified character string literal form. However, ITS designers must be aware of the interaction between the character string literal form and the literal forms defined for other data types. This is particularly critical if the other data type's literal form is structured with major components separated by break-characters (e.g., real number, physical quantity, set, and list literals, etc.)

2.4 Concept Descriptor (CD)

A concept descriptor represents any kind of concept. The CD refers to a concept usually by citing a code defined in a coding system. A given concept may be expressed in multiple terms where each term is a translation or re-encoding of the meaning in another code system. In addition compositional code systems are supported. In exceptional cases, the concept descriptor may not contain a code but only free text describing that concept. The CD is typically used through one of its restrictions described in Section 2.4.3.

Table 12: Summary of Primary Properties of Concept Descriptor (CD)

Name	Type	Status	Default	Constraint	Definition
code	ST	mandatory	NULL		The plain code symbol
displayName	ST	auxilary	NULL		A name or title for the code, under which the sending system shows the code value to its users
codeSystem	OID	mandatory	**CONTEXT**		Specifies the code system that defines the code
codeSystemName	ST	auxilary	NULL		A common name of the coding system
codeSystemVersion	ST	optional	NULL		If applicable, a version descriptor defined specifically for the given code system
originalText	ED	auxilary	NULL		The text or phrase used as the basis for the coding
modifier	LIST<CR>	optional	NULL		Specifies additional codes that modify the meaning of this concept descriptor
translation	SET<CD>	optional	NULL		A set of other concept descriptors that translate this concept descriptor into other code systems.

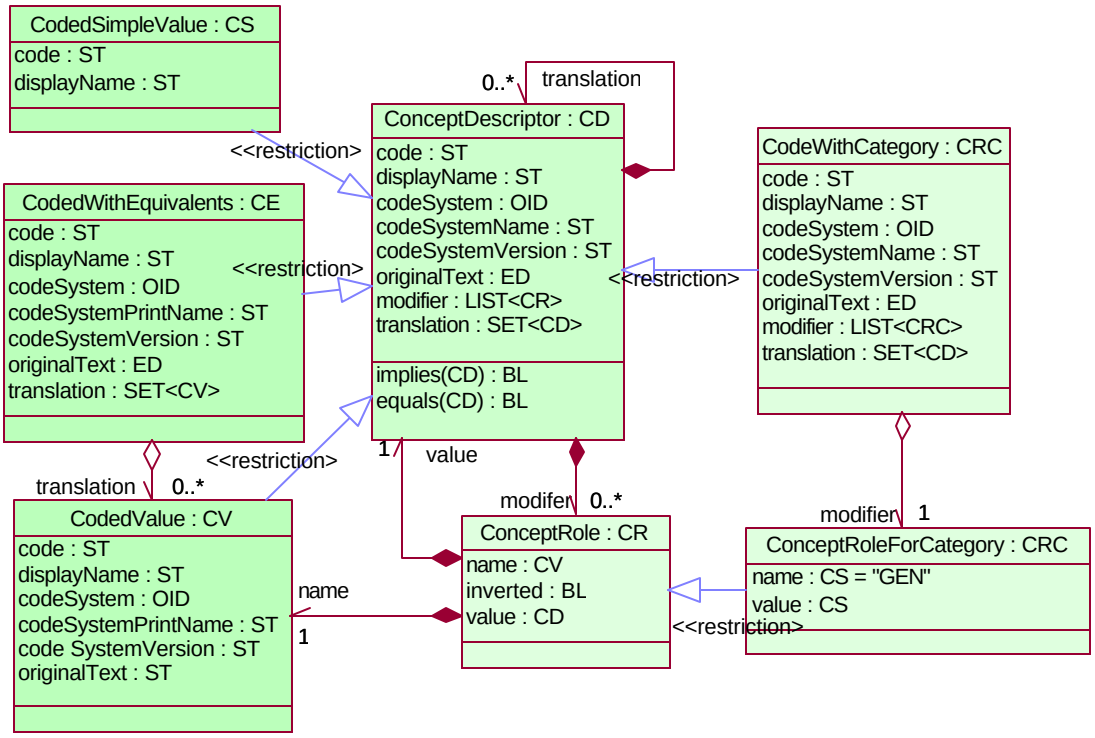


Figure 3: The Concept Descriptor information model. The concept descriptor is mostly used in one of its restricted or "profiled" forms, CS, CE, CV, and CC.

```
type ConceptDescriptor alias CD extends ANY {
            ST          code;
            ST          displayName;
            OID         codeSystem;
            ST          codeSystemName;
            ST          codeSystemVersion;
            ED          originalText;
            LIST<CR>    modifier;
            SET<CD>     translation;

            BL          equals(CD x);
            BL          implies(CD x);

  demotion  ED;
};
```

2.4.1.1 code : ST

This is the plain code symbol defined by the code system. For example, "784.0" is the code symbol of the ICD-9 code "784.0" for headache.

A non-exceptional CD value has a non-NULL code citing a valid code from an identified coding system. Conversely, a CD value without the code or with a code not from the cited coding system is an exceptional value (NULL of flavor *other*).

```
invariant(CD x) where x.nonNull {
  x.code.nonNull;
};
```

2.4.1.2 codeSystem : OID

Specifies the code system that defines the code. Code systems shall be referred to by ISO Object Identifiers (OID). The OID allows unambiguous reference to standard HL7 codes, other standard code systems, and local codes. HL7 shall assign an OID to each of its code tables as well as to external standard coding systems that are being used with HL7. Local sites can use their OID to construct a globally unique local coding system identifier.

Under HL7's branch, 2.16.840.1.113883, the sub-branches 5 and 6 contain HL7 standard and external code system identifiers respectively. The HL7 Vocabulary Technical Committee maintains these two branches.

A non-exceptional CD value (i.e. a CD value that has a non-null code property) has a non-NULL code system specifying the system of concepts that defines the code. In other words whenever there is a code there is also a code system.

ITS Note: although every non-NULL CD value has a defined code system, in some circumstances, the external representation of the CD value needs not explicitly mention the code system. For example, when the context mandates one and only one code system to be used specifying the code system explicitly would be redundant. However, in that case the code system property assumes that context-specific default value and is not NULL.

```
invariant(CD x) where x.code.nonNull {
  x.codeSystem.nonNull;
};
```

An exceptional CD of NULL-flavor "other" indicates that a concept could not be coded in the coding system specified. Thus, for these coding exceptions, the code system that did not contain the appropriate concept must be provided in the code system property.

Some code domains are qualified such that they include the portion of any pertinent local coding system that does not simply paraphrase the standard coding system (*coded with extensibility*, CWE.) If a CWE qualified field actually contains such a local code, the coding system must specify the local coding system from which the local code was taken. However, for CWE domains the local code is a valid member of the domain, so that local codes in CWE domains constitute neither an error nor an exceptional (NULL/other) value in the sense of this specification.

```
invariant(CD x) where x.other {
  x.code.isNull;
  x.codeSystem.nonNull;
};
```

2.4.1.3 codeSystemName : ST

This is a common name of the coding system referred to by the codeSystem OID. The code system name is optional and has no function in communication. The purpose of a code system name is to assist an unaided human interpreter of a code value to interpret the code system OID. It is suggested –

though not absolutely required – that ITS provide for code system name fields in order to annotate the OID for human comprehension.

HL7 systems must not functionally rely on the code system name. The code system name can never modify the meaning of the code system OID value and can not exist without the OID value.

```
invariant(CD x) {
  x.codeSystemName.nonNull.implies(x.codeSystem.nonNull);
};
```

2.4.1.4 codeSystemVersion : ST

Specifies a version of the code system as a plain character string. HL7 shall specify how these version strings are formed for each external code system. If HL7 has not specified how version strings are formed for a particular coding system, version designations have no defined meaning for such coding system.

Different versions of one code system must be compatible. Whenever a code system changes in an incompatible way, it will constitute a new code system, not simply a different version, regardless of how the vocabulary publisher calls it.

For example, the publisher of ICD-9 and ICD-10 calls these code systems, “revision 9” and “revision 10” respectively. However, ICD-10 is a complete redesign of the ICD code, not a backward compatible version. Therefore, for the purpose of this data type specification, ICD-9 and ICD-10 are different code systems, not just different versions. By contrast, when LOINC updates from revision "1.0j" to "1.0k", HL7 would consider this to be just another version of LOINC, since LOINC revisions are backwards compatible.

```
invariant(CD x) {
  x.codeSystemVersion.nonNull.implies(x.codeSystem.nonNull);
};
```

2.4.1.5 displayName : ST

The display name is a name or title for the code, under which the sending system typically or actually shows the code value to its users. It is included both as a courtesy to an unaided human interpreter of a code value and as a documentation of the name used to display the concept to the user. The display name has no functional meaning; it can never exist without a code; and it can never modify the meaning of the code.

Note: display names may not alter the meaning of the code value. Therefore, display names should not be presented to the user on a receiving application system without ascertaining that the display name adequately represents the concept referred to by the code value. Communication must not simply rely on the display name. The display name’s main purpose is to support debugging of HL7 protocol data units (e.g., messages.)

```
invariant(CD x) {
  x.displayName.nonNull.implies(x.code.nonNull);
};
```

2.4.1.6 translation : SET<CD>

The translation property of a concept descriptor y holds a set X of other concept descriptors $x_i \in X$ that translate the concept descriptor y into different code systems. Each element $x_i \in X$ was translated from the concept descriptor y. Each translation x_i may also contain translations. Thus, when a code is

translated multiple times the information about which code served as the input to which translation will be preserved.

Note: the translations are quasi-synonyms of one real-world concept. Every translation in the set is supposed to express the same meaning "in other words." However, exact synonymy rarely exists between two structurally different coding systems. For this reason, not all of the translations will be equally exact.

2.4.1.7 originalText : ED

This is the text or phrase used as the basis for the coding. The original text exists in a scenario where an originator of the information does not assign a code, but where the code is assigned later by a coder (post-coding.) In the production of a concept descriptor, original text may thus exist without a code.

Although the concept descriptor's value property is NULL, original text may still exist for the CD value. Any CD value with the code property of NULL signifies a coding exception. In this case, the text property is a name or description of the concept that was not coded. Such exceptional CD may contain translations. Such translations directly encode the concept described in the original text property.

Neither display name nor original text is part of the information a receiving system must automatically recognize. An information producer is responsible for the proper coding of all information in the value attribute, for any information consumer may safely ignore the display name and original text attributes.

A concept descriptor can be demoted into an ED value representing only the original text of the CD value.

```
invariant(CD x) where x.text.nonNull {
  ((ED)x).equals(x.text);
};
```

2.4.1.8 modifier : LIST <CR>

Specifies additional codes that modify the meaning of this concept descriptor. The primary code and all the modifiers together make up one concept. A concept descriptor with modifiers is also called a *code phrase*.

Modifiers can only be used according to well-defined rules of post-coordination. A concept descriptor may only have modifiers if the code system defines the use of such modifiers or if there is a third code system that specifies how other code systems may be combined.

For example, SNOMED allows constructing concepts as a combination of multiple codes. SNOMED RT defines a concept "cellulitis (morphologic abnormality)" (M-41650) a role "associated topography" (G-C505) and another concept "left foot (body structure)" (T-D9720). SNOMED-RT allows one to combine these codes in a code phrase:

```
<finding code="M-41650"
         codeSystem="&SNOMED-RT;"
         displayName="cellulites (morphologic abnormality)">
  <modifier code="T-D9720"
            displayName="left foot">
    <name  code="G-C505"  displayName="associated topography" />
  </modifier>
</finding>
```

In this example, there is one code system, SNOMED-RT that defines all the primary code and the modifiers and how these are used, which is why in our example representation the codeSystem does not need to be mentioned for the modifier name and value (the codeSystem is inherited from the primary code.)

Another common example is the U.S. Health Care Financing Administration (HCFA) procedure codes. HCFA procedure codes (HCPCS) are based on CPT-4 and add additional modifiers to it. For example, the patient with above finding (plus peripheral artherial disease, diabetes mellitus, and a chronic skin lesion at the left great toe) may have an amputation of that toe. The CPT-4 concept is "Amputation, toe matatarsophalangeal joint" (28820) and a HCPCS modifier needs to be added to indicate "left foot, great toe" (TA). Thus we code:

```
<procedure code="28820"
         codeSystem="&CPT-4;"
         displayName="Amputation, toe matatarsophalangeal joint">
  <modifier code="TA"
            codeSystem="&HCPCS;"
            displayName="left foot, great toe" />
</procedure>
```

In this example, the code system of the modifier (HCPCS) is different than the code system of the primary code (CPT-4.) It is only because there are well-defined rules that define how these codes can be combined, that the modifier may be used. Note also, that the role name is optional, and for HCPCS codes there are no distinguished role names.

The order of modifiers is preserved, particularly for the case where the coding system allows post-coordination but defines no role names. (e.g., some ICD-9CM codes, or the old SNOMED "multiaxial" coding.)

2.4.2 Concept Role (CR)

The concept role is used to hold code modifiers with optionally named roles. Both modifier roles and values must be defined by the coding system. For example, if SNOMED RT defines a concept "leg", a role relation "has-laterality", and another concept "left", the concept role relation allows to add the modifier "has-laterality: left" to a primary code "leg" to construct the meaning "left leg".

Table 13: Summary of Primary Properties of Concept Role (CR)

Name	Type	Status	Default	Constraint	Definition
value	CD	mandatory	NULL		Specifies the code system that defines the code
name	CV	optional	NULL		Specifies the manner in which the value modifies the meaning.
inverted	BL	mandatory	*false*		Indicates that the sense of the role name is reversed

The use of modifiers is strictly governed by the code system used. The CD does not permit using code modifiers with code systems that do not provide for modifiers (e.g. pre-coordinated systems, such as LOINC, ICD-10 PCS.) The rules of the modifier use must be governed by the code system (e.g., recent SNOMED RT revision, GALEN.)

```
protected type ConceptRole alias CR extends ANY {
            CV          name;
            BL          inverted;
            CD          value;
};
```

2.4.2.1 name : CV

Specifies the manner in which the value contributes to the meaning of a code phrase. For example, if SNOMED RT defines a concept "leg", a role relation "has-laterality", and another concept "left", the concept role relation allows to add the modifier "has-laterality: left" to a primary code "leg" to construct the meaning "left leg". In this example "has-laterality" is the CR.name.

If a coding system allows postcoordination but no role names the name attribute can be NULL. The name attribute must not itself have modifiers.

```
invariant(CR x) where x.nonNull {
  x.name.modifier.isNull;
};
```

2.4.2.2 value : CD

This is the concept that modifies the primary code of a code phrase through the role relation. For example, if SNOMED RT defines a concept "leg", a role relation "has-laterality", and another concept "left", the concept role relation allows to add the modifier "has-laterality: left" to a primary code "leg" to construct the meaning "left leg". In this example "left" is the CR.value.

This component is of type concept descriptor and thus can be in turn have modifiers. This allows modifiers to nest. Modifiers can only be used as far as the underlying code system defines them. It is not allowed to use any kind of modifiers for code systems that do not explicitly allow and regulate such use of modifiers.

```
invariant(CR x) where x.nonNull {
  x.value.nonNull;
};
```

2.4.2.3 inverted : BL

Indicates if the sense of the role name is inverted. This can be used in cases where the underlying code system defines inversion but does not provide reciprocal pairs of role names. By default, inverted is *false*.

For example, a code system may define the role relation "causes" besides the concepts "Streptococcus pneumoniae" and "Pneumonia". If that code system allows its roles to be inverted, one can construct the post-coordinated concept "Pneumococcus pneumonia" through "Pneumonia – causes, inverted – Streptococcus pneumoniae."

Roles may only be inverted if the underlying coding systems allows such inversion. Notably, if a coding system defines roles in inverse pairs or intentionally does not define certain inversions, the appropriate role code (e.g. "caused-by") must be used rather than inversion. It must be known whether the inverted property is true or false, if it is NULL, the role cannot be interpreted.

```
invariant(CR x) where x.nonNull {
  x.inverted.nonNull;
};
```

ITS Note: the property "inverted" should be conveyed in an indicator attribute, whose default value is *false*. That way the inverted indicator does not have to be sent when the role is not inverted.

2.4.3 Comparing Concept Descriptors

The main use of concept descriptors is for the purpose of indexing, querying and decision-making based on a coded value. A semantically unambiguous specification of coded values therefore requires a clear definition of what equality of concept descriptor values means and how CD values should be compared.

2.4.3.1 Equality

The equality of two concept descriptor values is determined solely based upon the code and coding system. The code system version is excluded from the equality test.[26] If modifiers are present, the modifiers are included in the equality test. Translations are not included in the equality test.[27] Exceptional concept descriptor values are not equal even if they have the same NULL-flavor or the same original text.[28]

```
invariant(CD x, y) x.nonNull.and(y.nonNull) {
  x.equals(y).equals(x.code.equals(y.code)
                .and(x.codeSystem.equals(y.codingSystem))
                .and(x.modifier.equals(y.modifier)));
};
```

Some code systems define certain style options to their code values. For example, the U.S. National Drug Code (NDC) has a dash and a non-dash form. An example for the dash form may be 1234-5678-90 when the non-dash form is 01234567890. Another example for this problem is when certain ISO or ANSI code tables define optional alphanumeric and numeric forms of two or three character lengths all in one standard.

In the case where code systems provide for multiple representations, HL7 shall make a ruling about which is the preferred form. HL7 shall document that ruling where that respective external coding system is recognized. HL7 shall decide upon the preferred form based on criteria of practicality and common use. In absence of clear criteria of practicality and common use, the safest, most extensible, and least stylized (the least decorated) form shall be given preference.[29]

2.4.3.2 Implication

Naturally, concepts can be narrowed and widened to include or exclude other concepts. Many coding systems have an explicit notion of concept specialization and generalization. The HL7 vocabulary principles also provide for concept specialization for HL7 defined value sets. The *implies*-property is a predicate that compares whether one concept is a specialization of another concept, and therefore implies that other concept.

When writing predicates (e.g., conditional statements) that compare two codes, one should usually test for implication not equality of codes.

For example, in Table 22 the "telecommunication use" concepts: work (W), home (H), primary home (HP), and vacation home (HV) are defined, where both HP and HV imply H. When selecting any home phone number, one should test whether the given use-code *c implies* H. Testing for *c equals* H would only find unspecified home phone numbers, but not the primary home phone number.

[26] The code system versions do not count in the equality test since by definition a code symbol must have the same meaning throughout all versions of a code system. Between versions, codes may be retired but not withdrawn or reused.

[27] Translations are not included in the equality test of concept descriptors for safety reasons. An alternative would have been to consider two CD values equal if any of their translations are equal. However, some translations may be equal because the coding system of that translation is very coarse-grained. More sophisticated comparisons between concept descriptors are application considerations that are not covered by this specification.

[28] NULL-values are exceptional values, not proper concepts. It would be unsafe to equate two values merely on the basis that both are exceptional (e.g., not codable or unknown.) Likewise there is no guarantee that original text represents a meaningful or unique description of the concept so that equality of that original text does not constitute concept equality. The reverse is also true: since there is more than one possible original text for a concept, the fact that original text differs does not constitute a difference of the concepts.

[29] This ruling at design-time is necessary to prevent HL7 interfaces from being burdened by code literal style conversions at runtime. This is notwithstanding the fact that some applications may require mapping from one form into another if that application has settled with the representation option that was not chosen by HL7.

Operationally, implication can be evaluated in one of two ways. The code system literals may be designed such that one single hierarchy is reflected in the code literal itself (e.g., ICD-9.) Apart from such special cases, however, a terminological knowledge base and an appropriate subsumption algorithm will be required to evaluate implication statements. For post-coordinated coding systems, designing such a subsumption algorithm is a non-trivial task.[30]

2.4.4 Restrictions for the Concept Descriptors

Use of the full concept descriptor data type is exceptional. It requires a conscious decision and documented rationale. In all other cases, one of the CD restrictions shall be used.[31]

All CD restrictions constrain certain properties of the CD. Properties may be constraint to the extent that only one value may be allowed for that property, in which case mentioning the property becomes redundant. Constraining a property to one value is referred to as suppressing that property. Although, conceptually a suppressed property is still semantically applicable, it is safe for an HL7 interface to assume the implicit default value without testing.

2.4.4.1 Coded Simple Value (CS) restricts CD

The Coded Simple Value (CS) is a restriction of the concept descriptor (CD). The CS suppresses all properties of the CD, except for code and display name. The code system and code system version is fixed by the context in which the CS value occurs. Original text is not applicable to CS values.

Table 14: Summary of Primary Properties of Coded Simple Value (CS)

Name	Type	Status	Default	Constraint	Definition
code	ST	mandatory	NULL		The plain code symbol
displayName	ST	auxiliary	NULL		A name or title for the code, under which the sending system shows the code value to its users
codeSystem	OID	fixed	**CONTEXT**		Specifies the code system that defines the code
codeSystemName	ST	fixed	**CONTEXT**		A common name of the coding system

```
type CodedSimpleValue alias CS restricts CD {
             ST         code;
             ST         displayName;
};

invariant(CS x) {
  x.codeSystem.equals(CONTEXT.codeSystem);
  x.codeSystemVersion.equals(CONTEXT.codeSystemVersion);
  x.codeSystemName.equals(CONTEXT.codeSystemName);
```

[30] This is one reason why the CD.modifiers for post-coordination are to be used sparingly and with caution. An additional problem of post-coordinated coding is that a general rule for equality may not exist at all.
[31] The advantage of the concept descriptor data type is its expressiveness, however, if all of its features, such as coding exceptions, text, translations and modifiers are used at all times, implementation and use become very difficult and unsafe. Therefore, the CD type is most often used in a restricted form with reduced features.


```
  x.originalText.isNull;
  x.translation.isNull;
  x.modifier.notApplicable;
};
```

CS can only be used in either of the following cases:

1) for a coded attribute which has a single HL7-defined code system, and where code additions to that value set require formal HL7 action (such as harmonization.) Such coded attributes that are designated "structural" codes must be assigned the CS restriction.

2) for a *technical* property in this specification that is assigned to a single code system defined either in this specification or defined outside HL7 by a body that has authority over the concept and the maintenance of that code system.

For example, since the ED type subscribes to the MIME design, it trusts IETF to manage the media type. This includes that this specification subscribes to the extension mechanism built into the MIME media type code (e.g., "application/x-myapp").

For CS values, the designation of the domain qualifier will always be CNE (*coded, non-extensible*) and the context determines unambiguously which HL7 value set applies.[32]

2.4.4.2 Coded Value (CV) restricts CD

The Coded Value (CV) is a restriction of the concept descriptor (CD). The CV suppresses the CD properties *translation* and *modifier*, which are both not applicable. The CV also constrains the original text to a character string (ST) instead of the more general encapsulated data (ED) type.

Table 15: Summary of Primary Properties of Coded Value (CV)

Name	Type	Status	Default	Constraint	Definition
code	ST	mandatory	NULL		The plain code symbol
displayName	ST	auxilary	NULL		A name or title for the code, under which the sending system shows the code value to its users
codeSystem	OID	mandatory	**CONTEXT**		Specifies the code system that defines the code
codeSystemName	ST	auxilary	NULL		A common name of the coding system
codeSystemVersion	ST	optional	NULL		If applicable, a version descriptor defined specifically for the given code system
originalText	ST	auxilary	NULL		The text or phrase used as the basis for the coding

[32] This is not withstanding the fact that an external referenced domain, such as the IETF MIME media type may include an extension mechanism. These extended MIME type codes would not be considered "extensions" in the sense of violating the CNE provision. The CNE provision is only violated if an attempt is made in using a different code system (by means of the CD.codeSystem property), which is not possible with the CS data type.


```
type CodedValue alias CV restricts CD {
            ST          code;
            OID         codeSystem;
            ST          codeSystemName;
            ST          codeSystemVersion;
            ST          displayName;
            ST          originalText;
};

invariant(CV x) {
  x.translation.isNull;
  x.modifier.notApplicable;
};
```

This type is used when any reasonable use case will require only a single code value to be sent. Thus, it should not be used in circumstances where multiple alternative codes for a given value are desired. This type may be used with both the CNE (*coded, non-extensible*) and the CWE (*coded, with extensibility*) domain qualifiers.

2.4.4.3 Coded With Equivalents (CE)

The data type "Coded with Equivalents" (CE) is a restriction of the concept descriptor (CD). The CE suppresses the CD modifier property, which is not applicable. The CE also restricts the translation property such that the translation is a set of CV values. CV values may not themselves contain translations.

Table 16: Summary of Primary Properties of Coded with Equivalents (CE)

Name	Type	Status	Default	Constraint	Definition
code	ST	mandatory	NULL		The plain code symbol
displayName	ST	auxilary	NULL		A name or title for the code, under which the sending system shows the code value to its users
codeSystem	OID	mandatory	**CONTEXT**		Specifies the code system that defines the code
codeSystemName	ST	auxilary	NULL		A common name of the coding system
codeSystemVersion	ST	optional	NULL		If applicable, a version descriptor defined specifically for the given code system
originalText	ED	auxilary	NULL		The text or phrase used as the basis for the coding
translation	SET<CV>	optional	NULL		A set of other concept descriptors that translate this concept descriptor into other code systems.


```
type CodedWithEquivalents alias CE restricts CD {
            ST          code;
            ST          displayName;
            OID         codeSystem;
            ST          codeSystemName;
            ST          codeSystemVersion;
            ED          originalText;
            SET<CV>     translation;
};

invariant(CE x) {
  x.modifier.notApplicable;
};
```

The CE type is used when the use case indicates that alternative codes may exist and where it is useful to communicate these. The CE type provides for a primary code value, plus a set of alternative or equivalent representations.

2.4.4.4 Coded With Category (CC)

The data type "Coded with Category" (CC) is a specific profile of using the concept descriptor (CD) used for certain coded attributes that have such large domains that multiple coding systems including local codes are used.[33] In these cases HL7 has established a mechanism where HL7 defined a set of coarse-grained categories (e.g. "material", "animal", "chemical", "place") under which external concept repertoires are assigned (e.g., an IUPAC code under "chemical".) Since there will be local codes communicated for such fields, the concept and it's HL7 defined category may not be known to a receiver of such local code. The CC data type is defined to communicate a fine-grained code together with one HL7 defined coarse-grained category code.

Table 17: Summary of Primary Properties of Coded with Category (CC)

Name	Type	Status	Default	Constraint	Definition
code	ST	mandatory	NULL		The plain code symbol
displayName	ST	auxilary	NULL		A name or title for the code, under which the sending system shows the code value to its users
codeSystem	OID	mandatory	CONTEXT		Specifies the code system that defines the code
codeSystemName	ST	auxilary	NULL		A common name of the coding system
codeSystemVersion	ST	optional	NULL		If applicable, a version descriptor defined specifically for the given code system
originalText	ED	auxilary	NULL		The text or phrase used as the basis for the coding
modifier	LIST<CRC>	conditional	CONTEXT	Only if the codeSystem is not registered with HL7	For one modifier that tells the HL7-defined category of the coded concept.
translation	SET<CV>	optional	NULL		A set of other concept descriptors that translate this concept descriptor into other code systems.

[33] There are two attributes in the RIM 1.0, Entity.type_cd and Act.type_cd.


```
type CodedWithCategory alias CC restricts CD {
             ST          code;
             ST          displayName;
             OID         codeSystem;
             ST          codeSystemName;
             ST          codeSystemVersion;
             ED          originalText;
             LIST<CRC>   modifier;
             SET<CV>     translation;
};

invariant(CC x)
    where x.nonNull {
     .and(x.codeSystem.value(2.16.840.1.113883.3).nonEmpty) {
  x.modifier.head.nonNull;
  x.modifier.tail.isEmpty;
};
```

The data type code-role for category (CRC) is a restriction of the CR data type constraining both the name and the value to CS and binding the name to a fixed code "has-generalization" (GEN).

Table 18: Summary of Primary Properties of Concept Role For Category (CRC)

Name	Type	Status	Default	Constraint	Definition
value	CS	mandatory	CONTEXT	CONTEXT	An HL7 defined code for the category of the concept.
name	CS	fixed	NULL	"GEN"	Fixed to "has-generalization" (GEN).

```
protected type CodeRoleForCategory alias CRC restricts CR {
             CS          name;
             CS          value;
};

invariant(CRC x) where x.nonNull {
  x.name.codeSystem.equals(2.16.840.1.113883.5.⟨to be assigned⟩);
  x.name.code.equals("GEN");
  x.value.nonNull;
  x.value.codeSystem.equals(2.16.840.1.113883.5.⟨to be assigned⟩);
  x.inverted.equals(false);
};
```


2.5 Instance Identifier (II)

The Instance Identifier (II) data type is used to uniquely identify an instance, thing or object. Examples are object identifier for HL7 RIM objects, medical record number, order id, service catalog item id, etc. Instance identifiers are defined based on ISO object identifiers.

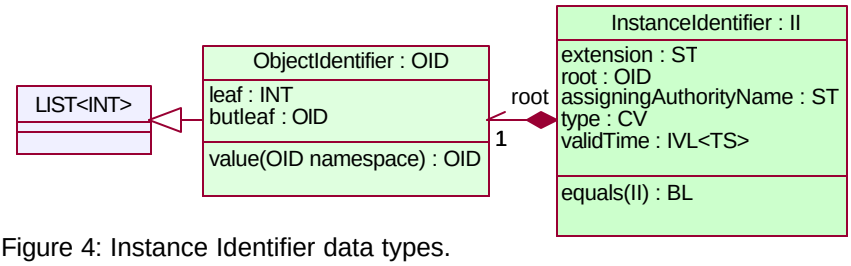


Figure 4: Instance Identifier data types.

Table 19: Summary of Primary Properties of Instance Identifier (II)

Name	Type	Status	Default	Constraint	Definition
root	OID	mandatory	NULL		A unique identifier that guarantees the global uniqueness of the instance identifier. The root alone may be the entire instance identifier, an extension value is not needed.
extension	ST	optional	NULL		An identifier that is unique in the namespace designated by the root
assigningAuthorityName		auxiliary			A name or mnemonic for the assigning authority of this identifier if applicable
validTime	IVL<TS>	optional			If applicable, specifies during what time the identifier is valid

2.5.1 ISO Object Identifier (OID)

The ISO Object Identifier is defined by ISO/IEC 8824:1990(E) clause 28.

> **28.9** The semantics of an object identifier value are defined by reference to an **object identifier tree**. An object identifier tree is a tree whose root corresponds to [the ISO/IEC 8824 standard] and whose vertices [i.e. nodes] correspond to administrative authorities responsible for allocating arcs [i.e. branches] from that vertex. Each arc from that tree is labeled by an object identifier component, which is [an integer number]. Each information object to be identified is allocated precisely one vertex (normally a leaf) and no other information object (of the same or a different type) is allocated to that same vertex. Thus an information object is uniquely and unambiguously identified by the sequence of [integer numbers] (object identifier components) labeling the arcs in a path from the root to the vertex allocated to the information object.
>
> **28.10** An object identifier value is semantically an ordered list of object identifier component values. Starting with the root of the object identifier tree, each object identifier component value identifies an arc in the object identifier tree. The last object identifier component value identifies an arc leading to a vertex to which an information object has been assigned. It is this information object, which is identified by the object identifier value. [...]

From ISO/IEC 8824:1990(E) clause 28

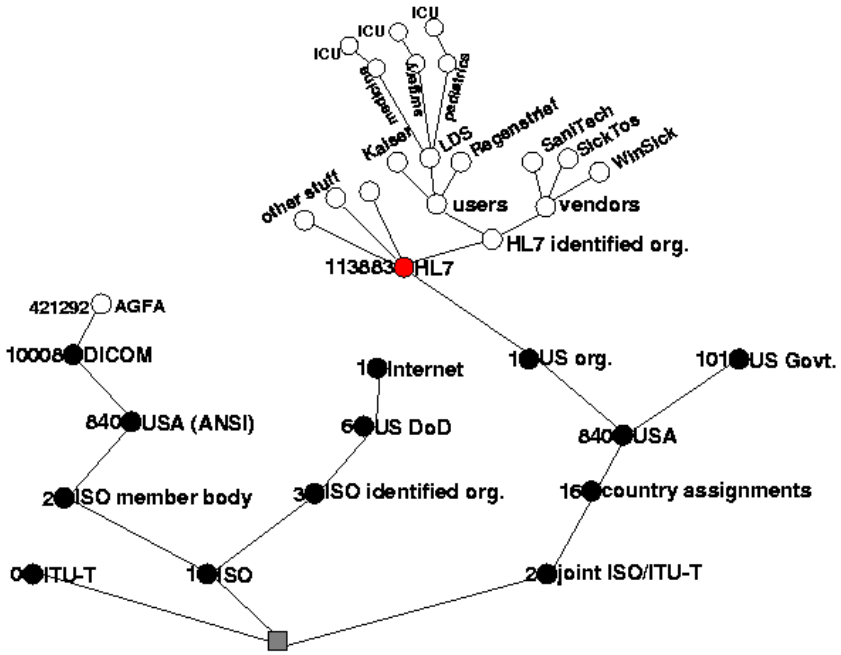


Figure 5: Example for a tree of ISO object identifiers. HL7's OID is 2.16.840.1.113883.

```
type ObjectIdentifier alias OID extends LIST<INT> {
            INT        leaf;
            OID        butleaf;
            OID        value(namespace OID);
  literal   ST;
};
```

HL7 shall establish an OID registry and assign OIDs in its branch for HL7 users and vendors upon their request. HL7 shall also assign OIDs to public identifier-assigning authorities both U.S. nationally (e.g., the U.S. State driver license bureaus, U.S. Social Security Administration, HIPAA Provider ID registry, etc.) and internationally (e.g., other countries Social Security Administrations, Citizen ID registries, etc.)

When assigning OIDs to third parties or entities, HL7 shall investigate whether an OID is already assigned for such entities through other sources. It this is the case, HL7 shall record such OID in a catalog, but HL7 shall not assign a duplicate OID in the HL7 branch. If possible, HL7 shall notify a third party when an OID is being assigned for that party in the HL7 branch.

Though HL7 shall exercise diligence before assigning an OID in the HL7 branch to third parties, given the lack of a global OID registry mechanism, one cannot make absolutely certain that there is no preexisting OID assignment for such third-party entity. Also, a duplicate assignment can happen in the future through another source. If such cases of supplicate assignment become known to HL7, HL7 shall make efforts to resolve this situation. For continued interoperability in the meantime, the HL7 assigned OID shall be the preferred OID used.

While most owners of an OID will "design" their namespace sub-tree in some meaningful way, there is no way to generally infer any meaning on the parts of an OID. HL7 does not standardize or require any namespace sub-structure. An OID owner, or anyone having knowledge about the logical structure of part of an OID, may still use that knowledge to infer information about the associated object; however, the techniques cannot be generalized.

An HL7 interface must not rely on any knowledge about the substructure of an OID for which it cannot control the assignment policies.

2.5.1.1 Literal Form

```
OID.literal ST {
    OID : INT "." OID   { $.head.equals($1);
                          $.tail.equals($3); }
        | INT           { $.head.equals($1);
                          $.tail.isEmpty; }
}
```

For Implementation Technologies that do not have native support for ISO OIDs, the ITS representations for OIDs may be a character string literal rather than a recursive data structure. The character string literal is more concise and most of the time OIDs will only be compared for equality but not analyzed further.

For compatibility with the DICOM standard, the literal form of the OID should not exceed 64 characters. (see DICOM part 5, section 9).

2.5.1.2 Structured Form: Sequence of Integers versus Value and Namespace

According to ISO/IEC 8824 an object identifier is a sequence of object identifier component values, which are integer numbers. These component values are ordered such that the root of the object identifier tree is the head of the list followed by all the arcs down to the leaf representing the information object identified by the OID. The fact that OID extends LIST<INT> represents this path of object identifier component values from the root to the leaf.

The **leaf** and "**butleaf**"properties take the opposite view. The leaf is the last object identifier component value in the list, and the "butleaf" property is all of the OID *but* the *leaf.* In a sense, the leaf is the identifier value and all of the OID but the leaf refers to the namespace in which the leaf is unique and meaningful.

However, what part of the OID is considered **value** and what is **namespace** may be viewed differently. In general, any OID component sequence to the left can be considered the namespace in which the rest of the sequence to the right is defined as a meaningful and unique identifier value. The value-property with a namespace OID as its argument represents this point of view.[34]

```
invariant(OID x) x.nonNull {
  x.nonEmpty;
  x.tail.isEmpty.implies(x.leaf.equals(x.tail));
  x.tail.nonEmpty.implies(x.leaf.equals(x.tail.leaf);

  x.tail.isEmpty.implies(x.butleaf.isNull);
  x.tail.nonEmpty.implies(x.butleaf.head.equals(x.head)
                    .and(x.butleaf.tail.equals(x.butleaf(x.tail))));

  forall(OID v; OID n) where v.equals(x.value(n)) {
    n.isEmpty.implies(v.equals(x));
    n.nonEmpty.implies(v.equals(x.value(n.tail)));
  };
};
```

[34] The value/namespace view on ISO object identifiers has important semantic relevance. It represents the notion of identifier value versus identifier assigning authority (= namespace), which is common in healthcare information systems in general, and HL7 v2.x in particular.

2.5.2 Properties of the Instance Identifier

```
type InstanceIdentifier alias II extends ANY {
            ST          extension;
            OID         root;
            ST          assigningAuthorityName;
            CV          type;
            IVL<TS>     validTime;

            BL          equals(II x);
};
```

2.5.2.1 root : OID

The root of an instance identifier guarantees the global uniqueness of the identifier. The root alone may be the entire unique identifier; an extension value is not needed.[35]

In the presence of a non-null extension, the root is commonly interpreted as the "assigning authority", that is, it is supposed that the root OID somehow refers to an organization that assigns identifiers sent in the extension. However, the root does not have to be an organizational OID, it can also be an OID specifically registered for an identifier scheme.

```
invariant(II x) where x.nonNull {
  root.nonNull;
};
```

2.5.2.2 extension : ST

The extension is a character string that is unique in the namespace designated by the root. If a non-NULL extension is exists, the root specifies a namespace (sometimes called "assigning authority" or "identifier type".) The extension property may be NULL in which case the root OID is the complete unique identifier.

It is recommended that systems use the OID scheme for external identifiers of their communicated objects. The extension property is mainly provided to accommodate legacy alphanumeric identifier schemes.

Some identifier schemes define certain style options to their code values. For example, the U.S. Social Security Number (SSN) is normally written with dashes that group the digits into a pattern "123-12-1234". However, the dashes are not meaningful and a SSN can just as well be represented as "123121234" without the dashes.

In the case where identifier schemes provide for multiple representations, HL7 shall make a ruling about which is the preferred form. HL7 shall document that ruling where that respective external identifier scheme is recognized. HL7 shall decide upon the preferred form based on criteria of practicality and common use. In absence of clear criteria of practicality and common use, the safest, most extensible, and least stylized (the least decorated) form shall be given preference.[36]

[35] DICOM objects are identified by OID only. For the purpose of DICOM/HL7 integration, it would be awkward if HL7 required the extension to be mandatory and to consider the OID only as an assigning authority. Since OID values are simpler and do not contain the risks of containing meaningless decoration, we do encourage systems to use simple OID identifiers as external references to their objects.

[36] This ruling at design-time is necessary to prevent HL7 interfaces from being burdened by identifier literal style conversions at runtime. This is notwithstanding the fact that some applications may require mapping from one form into another if that application has settled with the representation option that was not chosen by HL7.

HL7 may also decide to map common external identifiers to the value portion of the II.root OID. For example, the U.S. SSN could be represented as 2.16.840.1.113883.4.1.123121234. The criteria of practicality and common use will guide HL7's decision on each individual case.

2.5.2.3 assigningAuthorityName : ST

This is a name or mnemonic for the assigning authority of this identifier if applicable. This name is provided solely for the convenience of unaided humans interpreting an II value. The assigning authority name need not be unique or globally meaningful.

> Note: no automated processing must depend on the assigning authority name to be present in any form.

The assigning authority name is not the name for the individually identified object, but for the namespace, that immediately contains that object identifier. Two cases exist. 1) If the extension property is non-NULL, the root OID identifies the assigning authority; hence the assigning authority name is a name or mnemonic for the entire root OID. 2) If the extension is NULL, the assigning authority name is the name or mnemonic of the namespace property of the OID value.

2.5.2.4 validTime : IVL<TS>

The identifier is valid in this optional time-range. By default, the identifier is valid indefinitely. Any specific interval may be undefined on either side indicating unknown effective or expiry time.

> Note: identifiers for information objects in computer systems should not have restricted valid times, but should be globally unique at all times. The identifier valid time is provided mainly for real-world identifiers, whose maintenance policy may include expiry (e.g., credit card numbers.)

The II type conforms to the history item data type extension (Section 0). This means that the data types HXIT<II> and II are the same.

2.5.2.5 Equality

Two instance identifiers are equal if and only if their root and extension properties are equal.

```
invariant(II x, y) where x.nonNull.and(y.nonNull) {
  x.equals(y).equals(x.root.equals(y.root)
                .and(x.extension.equals(y.extension)));
}
```

2.6 Telecommunication Address (TEL)

A telecommunication address is a locator for some resource (information or services) mediated by telecommunication equipment. The semantics of a telecommunication address is that a communication entity responds to that address (the responder.) and therefore can be contacted by a communication initiator.

The responder of a telecommunication address may be an automatic service that can respond with information (e.g., FTP or HTTP services.) In such case a telecommunication address is a reference to that information accessible through that address. A telecommunication address value can thus be resolved to some information (in the form of encapsulated data, ED.)

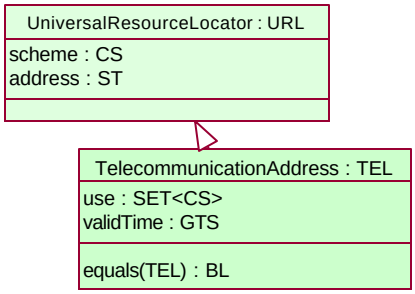


Figure 6: Telecommunication Address and URL data types.

Table 20: Summary of Primary Properties of Telecommunication Address (TEL)

Name	Type	Status	Default	Constraint	Definition
	URL	mandatory	NULL		
use	SET<CS>	optional	NULL	TelecommunicationAddressUse	A code advising a system or user which telecommunication address in a set of like addresses to select for a given telecommunication need
validTime	GTS	optional	NULL		Identifies the periods of time during which the telecommunication address can be used.

The telecommunication address is an extension of the Universal Resource Locator (URL) that specifies as an Internet standard RFC 1738 [http://www.isi.edu/in-notes/rfc1738.txt]. The URL specifies the protocol and the contact point defined by that protocol for the resource. Notable use cases for the telecommunication address data type are for telephone and fax numbers, e-mail addresses, Hypertext references, FTP references, etc.

2.6.1 Universal Resource Locator (URL)

This data type is defined as an Internet standard RFC 1738 [ftp://ftp.isi.edu/in-notes/rfc1738.txt].

> Just as there are many different methods of access to resources, there are several schemes for describing the location of such resources.
>
> The generic syntax for URLs provides a framework for new schemes to be established using protocols other than those defined in this document.
>
> URLs are used to “locate” resources, by providing an abstract identification of the resource location. Having located a resource, a system may perform a variety of operations on the resource, as might be characterized by such words as “access”, “update”, “replace”, “find attributes”. In general, only the “access” method needs to be specified for any URL scheme.

From RFC 1738

```
protected type UniversalResourceLocator alias URL extends ANY {
            CS         scheme;
            ST         address;
  literal   ST;
};
```

2.6.1.1 Literal Form

```
URL.literal  ST {
    URL : /[a-z0-9+.-]+/ “:” ST       { $.scheme.equals($1);
                                        $.address.equals($3); }
};
```

2.6.1.2 scheme : CS

The URL scheme identifies the protocol used to access the resource. URL schemes are registered by the Internet Assigned Numbers Authority (IANA) [http://www.iana.org], however IANA only registers URL schemes that are defined in Internet RFC documents. In fact there are a number of URL schemes

defined outside RFC documents, part of which is registered at the World Wide Web Consortium (W3C).[37]

Similar to the MIME media types, HL7 makes suggestions about URL schemes classifying them as *required*, *recommended*, *other*, and *deprecated*. Any scheme not mentioned has status *other*.

Table 21: URL Schemes

Code	Status	Definition
tel	required	A voice telephone number [draft-antti-telephony-url-11.txt].
fax	required	A telephone number served by a fax device [draft-antti-telephony-url-11.txt].
mailto	required	Electronic mail address [RFC 2368].
http	required	Hypertext Transfer Protocol [RFC 2068].
ftp	required	The File Transfer Protocol (FTP) [RFC 1738].
file	other	Host-specific local file names [RCF 1738]. Note that the file scheme is not interoperable. It only works with site-agreements for specific local files or shared file systems (NFS or AFS global file system.)
telnet	other	Reference to interactive sessions [RFC 1738]. Some sites, (e.g., laboratories) have TTY based remote query sessions that can be accessed through telnet.
modem	other	A telephone number served by a modem device [draft-antti-telephony-url-11.txt].

Note that this specification explicitly limits itself to URLs. Universal Resource Names (URN) are not covered by this specification. URNs are a kind of identifier scheme for other than accessible resources. This specification is only concerned with accessible resources, which belong into the URL category.

2.6.1.3 address : ST

The address is a character string whose format is entirely defined by the URL scheme.

2.6.1.4 Telephone and FAX Numbers

Note that there is no special data type for telephone numbers, telephone numbers are telecommunication addresses and are specified as a URL.

The telephone number URL is defined in the Internet RFC 2806 [http://www.isi.edu/in-notes/rfc2806.txt] *URLs for Telephone Calls*. Its definition is summarized in this subsection. This summary does not override or change any of the Internet specification's rulings.

The voice telephone URLs begin with "`tel:`" and fax URLs begin with "`fax:`"

The address part of the URL contains the telephone number in accordance with the ITU-T Recommendation E.123 *Telephone Network and ISDN Operation, Numbering, Routing and Mobile Service: Notation for National and International Telephone Numbers* (1993.) While HL7 does not add or withdraw from the URL specification, the preferred subset of the URL address syntax is given as follows:

```
proctected type TelephoneURL restricts URL {
  literal ST {
    URL : /(tel)|(fax)/ ":" address          { $.scheme.equals($1);
                                               $.address.equals($3); };
```

[37] The data type of the scheme property is still CS and for HL7 purposes, the scheme property is a CNE domain. This seems odd, because we just said that the URL scheme domain is not strictly defined anywhere. However we cannot allow extension of the URL scheme using the HL7 mechanism of local alternative code systems, which is why technically the URL scheme is a CS data type.


```
    ST address : "+" phoneDigits
    ST phoneDigits : digitOrSeparator phoneDigits | digitOrSeparator
    ST digitOrSeparator : digit | separator;
    ST digit : /[0..9]/;
    ST separator : /[().-]/;
  };
};
```

The global absolute telephone numbers starting with the "+" and country code are preferred. Separator characters serve as decoration but have no bearing on the meaning of the telephone number. For example: "`tel:+13176307960`" and "`tel:+1(317)630-7960`" are both the same telephone number; "`fax:+49308101724`" and "`fax:+49(30)8101-724`" are both the same fax number.

2.6.2 Properties of Telecommunication Address

A given telecommunication address value may have limited validity through time and may be tagged by a use code to indicate under what circumstances a specific telecommunication address may be preferred among a set of alternatives.

```
type TelecommunicationAddress alias TEL extends URL {
            GTS         validTime;
            SET<CS>     use;

            BL          equals(TEL x);
};
```

2.6.2.1 validTime : GTS

Identifies the periods of time during which the telecommunication address can be used. For a telephone number, this can indicate the time of day in which the party can be reached on that telephone. For a web address, it may specify a time range in which the web content is promised to be available under the given address.

The TEL data type where validTime is constrained to a simple interval of time (IVL<TS>) conforms to the history item data type extension (HXIT, see Section 4.1.1). Thus, HXIT<TEL> is a simple restriction of TEL.

2.6.2.2 use : SET<CS>

A code advising a system or user which telecommunication address in a set of like addresses to select for a given telecommunication need.

Table 22: Telecommunication Address Use Code

Concept	Code	Implies	Definition
home	H		A communication address at a home, attempted contacts for business purposes might intrude privacy and chances are one will contact family or other household members instead of the person one wishes to call. Typically used with urgent cases, or if no other contacts are available.
primary home	HP	H	The primary home, to reach a person after business hours.
vacation home	HV	H	A vacation home, to reach a person while on vacation.
work place	WP		An office address. First choice for business related contacts during business hours.
answering service	AS		An automated answering machine used for less urgent cases and if

		the main purpose of contact is to leave a message or access an automated announcement.
emergency contact	EC	A contact specifically designated to be used for emergencies. This is the first choice in emergencies, independent of any other use codes.
pager	PG	A paging device suitable to solicit a callback or to leave a very short message.
mobile contact	MC	A telecommunication device that moves and stays with its owner. May have characteristics of all other use codes, suitable for urgent matters, not the first choice for routine business.

The telecommunication use code is not a complete classification for equipment types or locations. Its main purpose is to suggest or discourage the use of a particular telecommunication address. There are no easily defined rules that govern the selection of a telecommunication address.

2.6.2.3 Equality

Two telecommunication address values are considered equal if both their URLs are equal. Use code and valid time are excluded from the equality test.

```
invariant(TEL x, y) x.nonNull.and(y.nonNull) {
  x.equals(y).equals(((URL)x).equals((URL)y));
}
```

2.7 Postal Address (AD)

The postal address data type is used to communicate mailing and home or office addresses. The main use of such data is to allow printing mail labels, or to allow a person to physically visit that address.

The postal address data type is not supposed to be a container for additional information that might be useful for finding geographic locations (e.g., GPS coordinates) or for performing epidemiological studies. Only those parts of addresses that are conventional for designating mailboxes or home or office addresses are part of the address data type. HL7 has other and better ways to handle global positioning or census units.

Table 23: Summary of Primary Properties of Postal Address (AD)

Name	Type	Status	Default	Constraint	Definition
	LIST<ADXP>	mandatory	NULL		The address data
use	SET<CS>	optional	NULL	AddressUse	A code advising a system or user which address in a set of like addresses to select for a given purpose
validTime	GTS	optional	NULL		Identifies the periods of time during which the address can be used.

The postal address data type is essentially a sequence of address part values. Addresses are conceptualized as text with added mark-up. The mark-up may break the address into lines and may describe in detail the role of each address part if it is known. Address parts occur in the address in the order in which they would be printed on a mailing label. The model is similar to HTML or XML markup of text.

2.7.1 Address Part (ADXP)

An address part is essentially a character string that may have a type-tag signifying its role in the address. Typical parts that exist in about every address are street, house number, or post box, ZIP code, city, country but other roles may be defined regionally, nationally, or on an enterprise level (e.g. in military addresses). Addresses are usually broken up into lines, indicated by special line-break tokens.

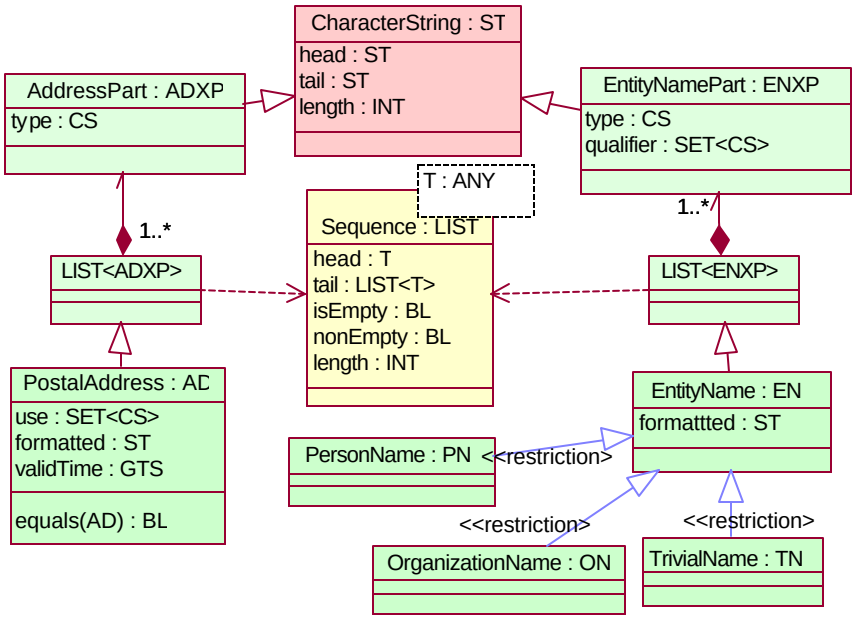


Figure 7: Data types for Postal Address and Entity Names (Person, Organization, and Trivial Names) are all based on extensions of a character string.

Table 24: Summary of Primary Properties of Address Part (ADXP)

Name	Type	Status	Default	Constraint	Definition
	ST	mandatory	NULL		The address part data
type	CS	optional	NULL	AddressPartType	Indicates whether an address part is the street, city, country, postal code, post box, etc.

```
protected type AddressPart alias ADXP extends ST {
            CS          type;
};
```

2.7.1.1 type : CS

Indicates whether an address part is the street, city, country, postal code, post box, etc. If the type is NULL the address part is unclassified and simply appears on the label as is.

Table 25: Address Part Type Code

Concept	Code	Definition
delimiter	DEL	Delimiters are printed without framing white space. If no value component is provided, the delimiter appears as a line break.
country	CNT	Country
state or province	STA	A sub-unit of a country with limited sovereignty in a federally organized country.
city	CTY	City
postal code	ZIP	A postal code designating a region defined by the postal service.
street name	STR	Street name or number.
house number	HNR	The number of a house or lot alongside the street. Also known as "primary street number", but does not number the street but the house.
direction	DIR	direction (e.g., N, S, W, E)
additional locator	ADL	This can be a unit designator, such as apartment number, suite number, or floor. There may be several unit designators in an address (e.g., “3rd floor, Appt. 342”.) This

		can also be a designator pointing away from the location, rather than specifying a smaller location within some larger one (e.g., Dutch "t.o." means "opposite to" for house boats located across the street facing houses.)
post box	POB	A numbered box located in a post station.

2.7.2 Properties of Postal Addresses

Addresses are essentially sequences of address parts, but add a "use" code and a valid time range for information about if and when the address can be used for a given purpose. The property "formatted" has a character string value with the address formatted in lines and with proper spacing.[38]

```
type PostalAddress alias AD extends LIST<ADXP> {
            GTS         validTime;
            SET<CS>     use;

            BL          equals(AD x);

            ST          formatted;
};
```

2.7.2.1 validTime : GTS

This General Time Specification (GTS) identifies the periods of time during which the address can be used. Typically, this is used to refer to different addresses for different times of the year or to refer to historical addresses.

The AD data type where validTime is constrained to a simple interval of time (IVL<TS>) conforms to the history item data type extension (HXIT, see Section 4.1.1). Thus, HXIT<AD> is a simple restriction of AD.

2.7.2.2 use : SET<CS>

This is a code advising a system or user which address in a set of like addresses to select for a given purpose.

Table 26: Address Use Code

Concept	Code	Implies	Definition
visit address	RES		Used primarily to visit an address.
mail address	PST		Used to send mail.
invoice address	INV	PST	An address at which to send invoices
temporary address	TMP		A temporary address, may be good for visit or mailing. Note that an address history can provide more detailed information.
bad address	BAD		A flag indicating that the address is bad, in fact, useless.
home	H		A private (home) address.
primary home	HP	H	The primary home.
vacation home	HV	H	A vacation home, to reach a person while on vacation.
work place	WP		An office address.

An address without specific use code might be a default address useful for any purpose, but an address with a specific use code would be preferred for that respective purpose.

[38] Remember that semantic properties are bare of all control flow semantics. The property *formatted* could be implemented as a "procedure" that would "return" the formatted address, but it would not usually be a variable to which one could assign a formatted address. However, HL7 does not define applications but only the semantics of exchanged data values. Hence, the semantic model abstracts from concepts like "procedure", "return", and "assignment" but speaks only of *property* and *value*.

2.7.2.3 Equality

Two address values are considered equal if both their address part lists are equal. Use code and valid time are excluded from the equality test.

```
invariant(AD x, y) x.nonNull.and(y.nonNull) {
  x.equals(y).equals(((LIST<ADXP>)x).equals((LIST<ADXP>)y));
}
```

2.7.2.4 Formatting Addresses

This address data type's main purpose is to capture postal addresses so that one can visit that address or send mail to it. Humans will look at addresses in printed form, such as on a mailing label. The address data type defines precise rules of how its data is formatted.[39]

Addresses are ordered lists of address parts. Each address part is printed in the order of the list from left to right and top to bottom (or in any other language-specific reading direction.) Every address part value is printed. Most address parts are framed by white space. The following six rules govern the setting of white space.

1) White space never accumulates, i.e. two subsequent spaces are the same as one. Subsequent line breaks can be reduced to one. White space around a line break is not significant.
2) Literals may contain explicit white space, subject to the same white space reduction rules. There is no notion of a literal line break within the text of a single address part.
3) Leading and trailing explicit white space is insignificant in all address parts, except for delimiter (DEL) address parts.
4) By default, an address part is surrounded by implicit white space.
5) Delimiter (DEL) address parts are not surrounded by any implicit white space.
6) Leading and trailing explicit white space is significant in delimiter (DEL) address parts.

This means that all address parts are generally surrounded by white space, but white space does never accumulate. Delimiters are never surrounded by implicit white space and every white space contributed by preceding or succeeding address parts is discarded, whether it was implicit or explicit.

Examples. The following shows examples of addresses in an XML encoded form, where the XML tag is the address part role and the data content is the address part value. The use of XML in these examples does not preempt any XML implementation technology specification, it is solely for the purpose of this example.

1050 Wishard Blvd. RG 5th floor,
Indianapoli, IN 46240.

has the following three valid encodings

```
<AD purpose="RES">
   1050 Wishard Blvd, RG 5th floor<DEL/>
   Indianapolis, IN 46240
</AD>
```

```
<AD purpose="RES">
   <STR>1050 Wishard Blvd</STR><ADL>RG 5th floor</ADL><DEL/>
```

[39] These rules for formatting addresses are part of the semantics of addresses because addresses are primarily defined as text displayed or printed and consumed by humans. Other uses (e.g., epidemiology) are secondary – although not forbidden, the AD data type may not serve these other use cases very well, and HL7 will define better ways to handle these use cases. Note that these formatting rules are *not* ITS issues, since this formatting applies to presentations for humans whereas ITS specifications are presentations for computer interchange.


```
   <CTY>Indianapolis</CTY><STA>IN</STA><ZIP>46240</ZIP>
</AD>

<AD purpose="RES">
   <HNR>1050</HNR><STR>Wishard Blvd</STR><ADL>RG 5th
floor</ADL><DEL/>
   <CTY>Indianapolis</CTY><STA>IN</STA><ZIP>46240</ZIP>
</AD>
```

the second encoding in this example is more specific about the role of the address parts than the first one. The first form would result from a system that only stores addresses as line 1, line 2, etc. The second form is the typical form seen in the U.S., where street address is sometimes separated, and city, state and ZIP code are always separated. However, in the U.S. the house number is not usually separated from the street address, where in Germany many systems keep house number as separate fields (third example.)

This example shows the strength of the mark-up approach to addresses. A typical German system that stores house number and street name in separate fields would print the address with street name first followed by the house number. For U.S. addresses, this would be wrong as the house number in the U.S. is written before the street name. The marked-up address allows keeping the natural order of address parts and still understanding their role.

2.8 Entity Name (EN)

An entity name data value specifies a name of a person, organization, place or thing. Examples for entity name values are “Jim Bob Walton, Jr.”, “Health Level Seven, Inc.”, “Lake Tahoe”, etc. An entity name may be as simple as a character string or may consist of several entity name parts (ENXP), such as, “Jim”, “Bob”, “Walton”, and “Jr.”, “Health Level Seven” and “Inc.”, “Lake” and “Tahoe”.

Table 27: Summary of Primary Properties of Entity Name (EN)

Name	Type	Status	Default	Constraint	Definition
	LIST<ENXP>	mandatory	NULL		The name data

2.8.1 Entity Name Part (ENXP)

An entity name part is a character string token that may have a type code signifying the role of the part in the whole entity name. Typical name parts that exist in about every name are given names, and family names, titles, etc.

Table 28: Summary of Primary Properties of Entity Name Part (ENXP)

Name	Type	Status	Default	Constraint	Definition
	ST	mandatory	NULL		The entity name part data
type	CS	optional	NULL	EntityNamePartType	Indicates whether the name part is a given name, family name, prefix, suffix, etc.
qualifier	SET<CS>	optional	NULL	EntityNameQualifier	A set of codes each of which specifies a certain subcategory of the name part in addition to the main name part type

```
protected type EntityNamePart alias ENXP extends ST {
            CS          type;
            SET<CS>     qualifier;
};
```

2.8.1.1 type : CS

Indicates whether the name part is a given name, family name, prefix, suffix, etc. Not every name part must have a type code, if the type code is unknown, not applicable, or simply undefined this is expressed by a null value (type.isNull). For example, a name may be "Rogan Sulma" and it may not be clear which one is a first name or which is a last name, or whether Rogan may be a title.

Table 29: Name Part Type

Name	Code	Definition
family	FAM	Family name, this is the name that links to the genealogy. In some cultures (e.g. Eritrea) the family name of a son is the first name of his father.
given	GIV	Given name (don't call it "first name" since this given names do not always come first)
prefix	PFX	A prefix has a strong association to the immediately following name part. A prefix has no implicit trailing white space (it has implicit leading white space though). Note that prefixes can be inverted.
suffix	SFX	A suffix has a strong association to the immediately preceding name part. A prefix has no implicit leading white space (it has implicit trailing white space though). Suffixes cannot be inverted.
delimiter	DEL	A delimiter has no meaning other than being literally printed in this name representation. A delimiter has no implicit leading and trailing white space.

2.8.1.2 qualifier : SET<CS>

The qualifier is a set of codes each of which specifies a certain subcategory of the name part in addition to the main name part type. For example, a given name may be flagged as a nickname, a family name may be a pseudonym or a name of public records

Table 30: Name Part Qualifier

Name	Code	Definition
Name change classifiers describe how a name part came about. More than one value allowed.		
birth	BR	A name that a person had shortly after being born. Usually for family names but may be used to mark given names at birth that may have changed later.
unmarried	MD	A name that a person (either sex) had immediately before her/his first marriage. Usually called "maiden name", this concept of maiden name is only for compatibility with cultures that keep up this traditional concept. In most cases maiden name is equal to birth name. If there are adoption or deed polls before first marriage the maiden name should specify the last family name a person acquired before giving it up again through marriage.
chosen	CH	A name that a person assumed because of free choice. Most systems may not track this, but some might. Subsumed in the concept of "chosen" are pseudonym (alias), and *deed poll*. The difference in civil dignity of the name part is given through the R classifier below. I.e. a deed poll creates a chosen name of record, whereas a pseudonym creates a name not noted in civil records.
adoption	AD	A name that a person took on because of being adopted. Adoptions may happen for adults too and may happen after marriage. Whether adoption name or the birth name is considered the "maiden" name is not fully defined and may, as always, simple depend on the discretion of the person or a data entry clerk.
spouse	SP	The name assumed from the partner in a marital relationship (hence the "M"). Usually the spouse's family name. Note that no inference about gender can be made from the existence of spouse names.
Affix types. Usually only one value per affix.		
voorvoegsel	VV	A Dutch "voorvoegsel" is something like "van" or "de" that might have indicated nobility in the past but no longer so. Similar prefixes exist in other languages such as Spanish, French or Portugese.
academic	AC	Indicates that a prefix like "Dr." or a suffix like "M.D." or "Ph.D." is an academic title.
professional	PR	Primarily in the British Imperial culture people tend to have an abbreviation of their professional organization as part of their credential suffices.
nobility	NB	In Europe and Asia, there are still people with nobility titles (aristocrats.) German "von" is generally a nobility title, not a mere voorvoegsel. Others are "Earl of" or "His Majesty King of..." etc. Rarely used nowadays, but some systems do keep track of this.
legal status	LS	For organizations a suffix indicating the legal status, e.g., "Inc.", "Co.", "AG", "GmbH", "B.V." "S.A.", "Ltd." etc.

Additional qualifiers. More than one value allowed.		
nick	NK	Indicates that the name part is a nickname. Not explicitly used for prefixes and suffixes, since those inherit this flag from their associated significant name parts. Note that most nicknames are given names although it is not required.
callme	CL	A callme name is (usually a given name) that is preferred when a person is directly addressed.
record	RE	This flag indicates that the name part is known in some official record. Usually the antonym of nickname. Note that the name purpose code "license" applies to all name parts or a name, whereas this code applies only to name name part.
initial	IN	Indicates that a name part is just an initial. Initials do not imply a trailing period since this would not work with non-Latin scripts. Initials may consist of more than one letter, e.g., "Ph." could stand for "Philippe" or "Th." for "Thomas".
weak	WK	Used only for prefixes and suffixes (affixes). A weak affix has a weaker association to its main name part than a genuine (strong) affix. Weak prefixes are not normally inverted. When a weak affix and a strong affix occur together, the strong affix is closer to its associated main name part than the weak affix.
invisible	HD	Indicates that a name part is not normally shown. For instance, traditional maiden names are not normally shown. "Middle names" may be invisible too.

Note: an entity may have multiple names as defined through the RIM class Entity_name, which is outside the scope of this specification.

2.8.2 Properties of Entity Name

Entity names have no additional properties adding information to the sequence of entity name parts. The property "formatted" has a character string value with the formatted person name.[40]

```
type EntityName alias EN extends LIST<PNXP> {
            ST         formatted;
};
```

2.8.2.1 Formatting Entity Names

The entity name data type's main purpose is to capture names so that one can understand the parts and render them correctly on labels, addresses, badges, etc. Humans will look at names in printed form, such as on a mailing label. This entity name data type therefore defines precise rules of how its data is formatted.[41]

Entity names are ordered lists of entity name parts. Each entity name part is printed in the order of the list from left to right (or in any other language-specific reading direction.) Every entity name part (except for those marked "invisible") is printed. Most entity name parts are framed by whitespace. The following six rules govern the setting of whitespace.

1) White space never accumulates, i.e. two subsequent spaces are the same as one.
2) Literals may contain explicit white space subject to the same white space reduction rules.
3) Except for *prefix*, *suffix* and *delimiter* name parts, every name part is surrounded by implicit white space. Leading and trailing explicit whitespace is insignificant in all those name parts.
4) Delimiter name parts are not surrounded by any implicit white space. Leading and trailing explicit whitespace is significant in in delimiter name parts.

[40] Remember that semantic properties are bare of all control flow semantics. The property *formatted* could be implemented as a "procedure" that would "return" the formatted name. It would not usually be implemented as a variable to which one could assign a formatted person name. However, HL7 does not define applications but only the semantics of exchanged data values. Hence, the semantic model abstracts from concepts like "procedure", "return", and "assignment" but speaks only of *property* and *value*.

[41] These rules for formatting names are part of the semantics of names because the name parts have been designed with the important use case of displaying and rendering on labels. Note that these formatting rules are *not* ITS issues, since this formatting applies to presentations for humans whereas ITS specifications are presentations for computer interchange.

5) Prefix name parts only have implicit leading white space but no implicit trailing white space. Trailing explicit whitespace is significant in prefix name parts.

6) Suffix name parts only have implicit trailing white space but no implicit leading white space. Leading explicit whitespace is significant in suffix name parts.

This means that all entity name parts are generally surrounded by whitespace, but whitespace does never accumulate. Delimiters are never surrounded by implicit white space, prefixes are not followed by implicit white space and suffixes are not preceded by implicit white space. Every whitespace contributed by preceding or succeeding name parts around those special name parts is discarded, whether it was implicit or explicit.

Examples. The following shows examples of entity names in an XML encoded form, where the XML tag is the entity name part type and the data content is the entity name part value. The use of XML in these examples does not preempt any XML implementation technology specification; it is solely for the purpose of this example.

A very simple encoding of "John W. Doe" would be:

```
<EN>
   <GIV>John</GIV>
   <GIV>W.</GIV>
   <FAM>Doe</FAM>
</EN>
```

none of the special qualifiers need to be mentioned if they are unknown or irrelevant. The next example shows extensive use of multiple given names, prefixes, suffixes, for academic degrees, nobility titles, *vorvoegsels* ("van"), and professional designations.

```
<EN>
   <PFX Q="AC">Dr. phil.  </PFX>
   <GIV>Regina</GIV><GIV>Johanna</GIV><GIV>Maria</GIV>
   <PFX Q="NB">Gräfin_</PFX><PFX Q="VV">von_</PFX>
   <FAM Q="MD">Hochheim</FAM><DEL>-</DEL><FAM
Q="SP">Weilenfels</FAM>
   <SFX Q="PR WK">NCFSA</SFX>
</EN>
```

The next example is an organization name, "Health Level Seven, Inc." in simple string form:

```
<EN>Health Level Seven, Inc.</EN>
```

and as a fully parsed name

```
<EN>Health Level Seven<DEL>, </DEL><SFX Q="LS">Inc.</SFX></EN>
```

2.8.3 Restrictions of Entity Name

Three restrictions to Entity Name are defined in order to allow making specific constraints for certain kinds of entities, trivial name (TN), person name (PN), and organization name (ON).

2.8.3.1 Trivial Name (TN)

The trivial name (TN) is an entity name that consists of only one name part without any name part type or qualifier. The TN, and its single name part are therefore equivalent to a simple character string. This equivalence is expressed by a defined demotion to ST and promotion from ST.


```
type TrivialName alias TN extends LIST<ST> {
             ST         formatted;
 demotion    ST;
 promotion   TN         (ST x);
};

invariant(TN x) where x.nonNull {
  x.head.nonNull;
  x.tail.isEmpty;
  x.formatted.equals(x.head);
};

invariant(ST x) {
  ((TN)x).head.equals(x);
};
```

2.8.3.2 Person Name (PN)

Since most of the functionality of entity name is in support of person names, the person name (PN) is only a very minor restriction on the entity name part qualifier.

```
type PersonName alias PN extends LIST<PNXP> {
             ST         formatted;
};

protected type PersonNamePart alias PNXP extends ST {
             CS         type;
             SET<CS>    qualifier;
};

invariant(PNXP x) where x.nonNull {
  x.qualifier.contains(“LS”).not;
};
```

2.8.3.3 Organization Name (ON)

A name for an organization, such as “Health Level Seven, Inc." An organization name consists only of untyped name parts, prefixes, suffixes, and delimiters.

```
type OrganizationName alias ON extends LIST<ONXP> {
             ST         formatted;
};
```

```
protected type OrganizationNamePart alias ONXP extends ST {
            CS          type;
            SET<CS>     qualifier;
};

invariant(ONXP x) where x.nonNull {
  x.type.implies("FAM").not;
  x.type.implies("GIV").not;
};
```

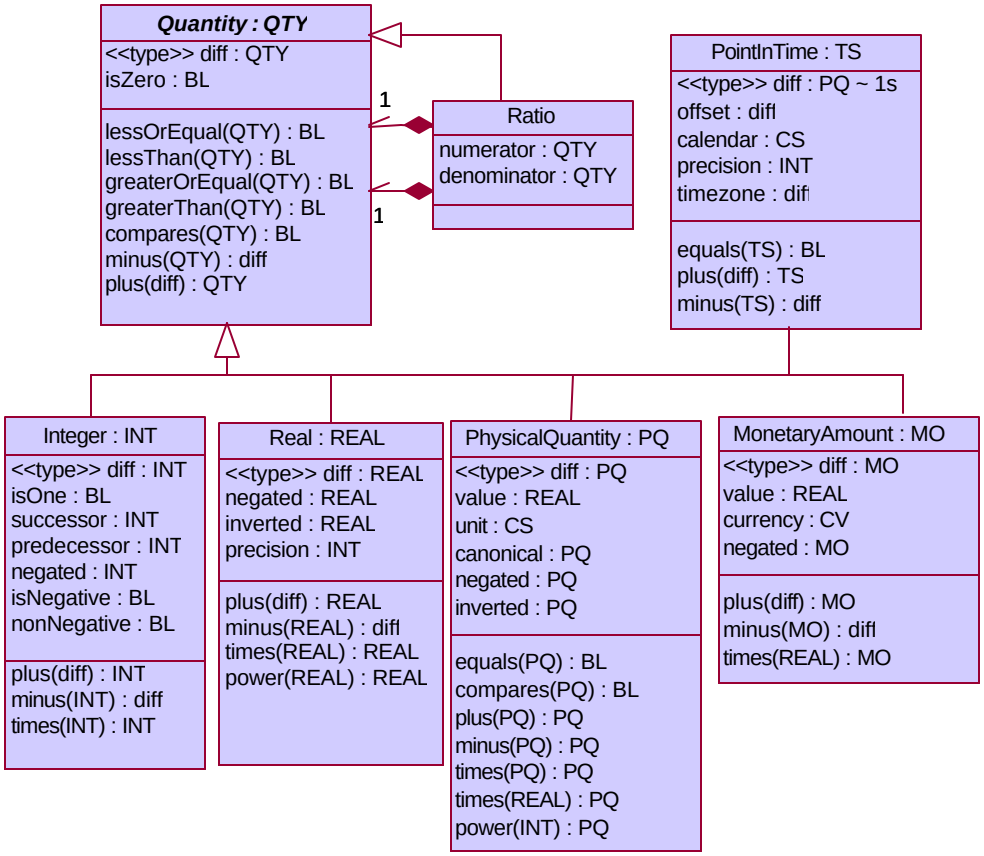


Figure 8: Quantity Data Types

2.9 Abstract Type Quantity (QTY)

The quantity data type is an abstract generalization for all data types (1) whose value set has an order relation (less-or-equal, ≤) and (2) where difference is defined in all of the data type's totally ordered value subsets. The quantity type abstraction is needed in defining certain other types, such as the interval and the probability distribution.

```
abstract type Quantity alias QTY extends ANY {
            BL          lessOrEqual(QTY x);
            BL          compares(QTY x);

  type      QTY         diff;
            diff        minus(QTY x);
            QTY         plus(diff x);
            BL          isZero;

            BL          lessThan(QTY x);
            BL          greaterOrEqual(QTY x);
            BL          greaterThan(QTY x);
};
```

2.9.1.1 Ordering

An ordered set is a set with an order relation (e.g., less-or-equal, ≤). An order relation is asymmetric and transitive.

A totally ordered set is an ordered set where all pairs of elements have a defined order (e.g., the integer and real numbers are totally ordered.)

A partially ordered set is an ordered set where not all pairs of elements are comparable through the order relation (e.g., a tree structure or the set of physical quantities is a partially ordered set.) Two data values *x* and *y* of an ordered type are comparable (*x*.compares(*y*)) if the less-or-equal relation holds in either way ($x \le y$ or $y \le x$).

A partial order relation generates totally ordered subsets whose union is the entire set (e.g., the set of all length is a totally ordered subset of the set of all physical quantities.)

For example, a tree structure is partially ordered, where the root is considered less or equal to a leaf, but there may not be an order among the leafs. Also, physical quantities are partially ordered, since an order exists only among quantities of the same dimension (e.g., between two lengths, but not between a length and a time.) A totally ordered subset of a tree is a path that transitively connects a leaf to the root. The physical dimension of time is a totally ordered subset of physical quantities.

```
invariant (QTY x, y, z)
    where x.nonNull.and(y.nonNull).and(z.nonNull) {
  x.lessOrEqual(x);                                 /* reflexive */
  x.lessOrEqual(y)                                  /* asymmetric */
     .implies(y.lessOrEqual(x)).not();
  x.lessOrEqual(y).and(y.lessOrEqual(z))            /* transitive */
     .implies(x.lessOrEqual(z))
```

```
  x.lessThan(y).equals(x.lessOrEqual(y).and(x.equals(y).not));
  x.greaterOrEqual(y).equals(y.lessOrEqual(x));
  x.greaterThan(y).equals(y.lessThan(x));

  x.compares(y).equals(x.lessOrEqual(y).or(y.lessOrEqual(x)));
};
```

2.9.1.2 Difference

A difference is defined in an ordered set if it is semantically meaningful to state that ***D*** is the difference between the values *x* and *y*. This difference ***D*** must be meaningful independently from the values *x* and *y*. This independence exists if for all values *u* one can meaningfully derive a value *v* such that ***D*** would also be the difference between *u* and *v*. The judgment for what is *meaningful* can not be defined formally.[42]

The *diff*-property is a data type that can express the difference between two values for which the ordering relation is defined (i.e., two elements of a common totally ordered subset.) For example, the difference data type of integer number is integer number, but the difference type of point in time is a physical quantity in the dimension of time. A difference data type is a totally ordered data type.

The difference between two values *x* minus *y* must be defined for all *x* and *y* in a common totally ordered subset of the data type's value set. Zero is the difference between a value and itself.

```
invariant(QTY x, y) where x.compares(y) {
  x.minus(y).nonNull;
  x.minus(x).isZero;
  x.plus(y.minus(x)).equals(y);
};
```

2.10 Integer Number (INT)

Integer numbers are precise numbers that are results of counting and enumerating. Integer numbers are discrete, the set of integers is infinite but countable. No arbitrary limit is imposed on the range of integer numbers. Two exceptional values are defined for the positive and negative infinity.

```
type IntegerNumber alias INT extends QTY {
            BL          isOne;
            INT         successor;
            INT         plus(diff x);
            INT         times(INT x);
```

[42] The quantity data type abstraction corresponds to the notion of difference scales in contrast to ordinal scales and ratio scales (Guttman and Stevens). A data type with only the order requirement but not the difference requirement would be an ordinal. Ordinals are not currently defined with a special data type. Instead, ordinals are usually coded values, where the underlying code system specifies ordinal semantics. This ordinal semantics, however, is not reflected in the HL7 data type semantics at this time.


```
  type       INT        diff;
             diff       minus(INT x);
             INT        predecessor;
             INT        negated;
             BL         isNegative;
             BL         nonNegative;

  literal    ST;
};
```

2.10.1.1 Algebraic Operations

Since the integer number data type includes all of the semantics of the mathematical integer number concept, the basic operations plus (addition) and times (multiplication) are defined. These operations are defined here as characterizing operations in the sense of ISO 11404, and because these operations are needed in other parts of this specifications, namely the semantics of the literal form.

The traditional recursive definitions of addition and multiplication are due to Grassmann:[43]

```
invariant(INT x, o, i) where x.nonNull.and(o.isZero()) {
  x.lessThan(x.successor);
  x.plus(o).equals(x);
  x.plus(y.successor).equals(x.plus(y).successor);
  x.times(o).equals(o);
  x.times(y.successor).equals(x.times(y)).plus(x);
};
```

The inverse element, negation, and predecessor are defined as follows

```
invariant(INT x) where x.nonNull {
  x.plus(x.negated).isZero;
  x.successor.predecessor.equals(x);
  x.nonNegative.equals(0.lessOrEqual(x));
  x.isNegative.equals(x.nonNegative.not);
};
```

2.10.1.2 Literal Form

The literal form of an integer is a simple decimal number, i.e. a string of decimal digits.

[43] H. Grassman. *Lehrbuch der Arithmetik*. 1861. We prefer Grassman's original axioms to the Peano axioms, because Grassman's axioms work for all integers, not just for natural numbers. Also, "it is rather well-known, through Peano's own acknowledgment, that Peano borrowed his axioms from Dedekind and made extensive use of Grassmann's work in his development of the axioms." (Hao Wang. The Axiomatization of Arithmetic. *J. Symb. Logic*; 1957:22(2); p. 145.)

```
INT.literal ST {
  INT digit : "0"             { $.isZero; }
            | "1"             { $.equals(0.successor); }
            | "2"             { $.equals(1.successor); }
            | "3"             { $.equals(2.successor); }
            | "4"             { $.equals(3.successor); }
            | "5"             { $.equals(4.successor); }
            | "6"             { $.equals(5.successor); }
            | "7"             { $.equals(6.successor); }
            | "8"             { $.equals(7.successor); }
            | "9"             { $.equals(8.successor); };

  INT uint : digit            { $.equals($1); }
           | uint digit       { $.equals($1.times(9.successor).plus($2)); };

  INT : uint                  { $.equals($1); }
      | “+” uint              { $.equals($2); }
      | “-” uint              { $.equals($2.negated); };
};
```

2.11 Real Number (REAL)

Mathematically, real numbers are the superset of integer numbers, rational numbers, and irrational numbers. Real numbers are needed beyond integers whenever quantities of the real world are *measured*, *estimated*, or computed from other real numbers.

Note: This specification defines the real number data type in the broadest sense possible. However, it does not imply that any conforming ITS or implementation must be able to represent the full range of Real numbers, which would not be possible in any finite implementation. HL7's current use cases for the Real number data type are measured and estimated quantities and monetary amounts. These use cases can be handled with a restricted Real value space, rational numbers, and even just very limited decimals (scaled integers.) However, we declare the representations of the real value space as floating points, rationals, scaled integers, or digit strings) and their limitations to be out of the scope of this specification.

This specification offers two choices for a number data type. The choice is made as follows: Any number attribute is a real if it is not known for sure that it is an integer. A number is an integer if it is *always* counted, typically representing an ordinal number. If there are conceivable use cases where such a number would be estimated or averaged, it is not always an integer and thus should be using the Real data type.

```
type RealNumber alias REAL extends QTY {
  type      REAL      diff;
            diff      minus(REAL x);
            REAL      plus(diff x);
            REAL      negated;
```



```
            REAL       times(REAL x);
            REAL       inverted;
            REAL       power(REAL x);

  literal   ST;
            INT        precision;

  demotion  INT;
  promotion REAL       (INT x);
  promotion PQ;
  promotion RTO;
};
```

2.11.1.1 Algebraic Operations

The algebraic operations are specified here as characterizing operations in the sense of ISO 11404, and because these operations are needed in other parts of this specifications. Here, the specification of the real numbers is based on the homomorphism (the type conversion that preserves integer arithmetic) between integer and real numbers:

```
invariant(INT n, m) where n.nonNull.and(m.nonNull) {
  ((REAL)n.plus(m)).equals(((REAL)n).plus((REAL)m));
  ((REAL)n.times(m)).equals(((REAL)n).times((REAL)m));
};
```

Unlike the integer numbers, the real numbers semantics are not inductively defined but only their algebraic properties described.

```
invariant(REAL x, y, z)
    where x.nonNull.and(y.nonNumm).andz.nonNull) {
  /* ADDITION */
  x.plus(0).equals(x)                              /* neutral element */
  x.plus(x.negated).equals(0)                      /* inverse element */
  x.plus(y).plus(z).equals(x.plus(y.plus(z)));     /* associative */
  x.plus(y).equals(y.plus(x))                      /* commutative */

  /* MULTIPLICATION */
  x.times(0).equals(0);
  x.times(1).equals(x);                            /* neutral element */
  x.times(x.inverted).equals(1)                    /* inverse element */
  0.inverted.isNull;                               /* … except for zero */
  x.times(y).times(z).equals(x.times(y.times(z)));/* associative */
  x.times(y).equals(y.times(x));                   /* commutative */
  x.times(y.plus(z))                               /* distributive */
    .equals(x.times(y).plus(x.times(z));
```

```
  /* EXPONENTIATION */
  x.power(0).equals(1);
  x.power(1).equals(x);
  x.power(y).power(z).equals(x.power(y.times(z)));
  x.power(y).times(x.power(z)).equals(x.power(y.plus(z)));
  x.power(y).inverted.equals(x.power(y.negated));
  x.power(y).power(y.inverted).equals(x);
};
```

2.11.1.2 Literal Form

The syntax and semantics of real number literals is defined below. In summary, a real number is represented in decimal form with optional + or – sign, and optional decimal point, and optional exponential notation using a case insensitive "e" between the mantissa and the exponent. The number of significant digits must conform to the precision property.

```
REAL.literal ST {
  REAL : mantissa                    { $.equals($1); }
       | mantissa /[eE]/ INT         { $.equals($1
                                          .times(10.power($3)); };

  REAL mantissa
       : /0*/ 0                      { $.isZero; $.precision.equals(1); }
       | /0*/ "." /0*/               { $.isZero; $.precision.equals(
                                                   $3.length.successor); }
       | /0*/ "." /0*/ fractional    { $.equals($4);
                                       $.precision.equals($4.precision); }
       | integer                     { $.equals($1); }
       | integer "." fractional      { $.equals($1.plus($2));
                                       $.precision.equals($1.precision
                                         .plus($3.precision)); };

  REAL integer
       : uintval                     { $.equals($2); }
       | "+" uintval                 { $.equals($1.times($2)); }
       | "-" uintval                 { $.equals($1.times($2).negated); };

  REAL uintval : /0*/ uint           { $.equals($2); };

  REAL uint : digit                  { $.equals($1);
                                       $.precision.equals(1); }
            | uint digit             { $.equals($1.times(10).plus($2));
                                       $.precision.equals(
                                         $1.precision.successor; };
```



```
  REAL fractional
       : digit                          { $.equals($1.times(10.inverted));
                                          $.precision.equals(1); }
       | digit fractional               { $.equals(
                                            $1.plus($2.times(10.inverted));
                                          $.precision.equals(
                                            $1.precision.successor); };

  INT digit : /[0-9]/                   { $.equals($1); }
};
```

Examples of real literals are for two-thousand are `2000`, `2000.`, `2e3`, `2.0e+3`, `+2.0e+3`.

Note that the literal form does not carry type information. For example, "2000" is a valid representation of both a real number and an integer number. No trailing decimal point is used to disambiguate from integer numbers. An ITS that uses this literal form must recover the type information from other sources.

2.11.1.3 precision : INT

The precision property indicates the quality of the approximation of a decimal real number representation. Precision is the number of significant decimal digits in that decimal representation. The precision attribute is the precision of a decimal digit representation, *not the precision or accuracy of the real number value.* Precision does not play a role in deciding whether two real number values are equal.

The purpose of the precision property for the real number data type is to faithfully capture the whole information presented to humans in a number. The amount of decimal digits shown conveys information about the uncertainty (i.e., precision and accuracy) of a measured value.

Note: the precision of the representation is independent from uncertainty (precision accuracy) of a measurement result. If the uncertainty of a measurement result is important, one should send uncertain values as defined in Section 4.4.

The rules for what digits are significant are as follows:

All non-zero digits are significant.

All zeroes to the right of a significant digit are significant.

When all digits in the number are zero the zero-digit immediately left to the decimal point is significant (and because of rule 2, all following zeroes are thus significant too.)

Note, these rules of significance differ slightly from the more casual rules taught in school. Notably trailing zeroes before the decimal point are consistently regarded significant here. Elsewhere, e.g., 2000 is ambiguous as to whether the zeroes are significant. This deviation from the common custom is warranted for the purpose of unambiguous communication.

Examples:

2000	has 4 significant digits.
2e3	has 1 significant digit, used if one would naturally say "2000" but precision is only 1.
0.001	has 1 significant digits.
1e-3	has 1 significant digit, use this if one would naturally say "0.001" but precision is only 1.
0	has 1 significant digit.

0.0 has 2 significant digits.

000.0 has 2 significant digits.

0.00 has 3 significant digits.

4.10 has 3 significant digits.

4.09 has 3 significant digits.

4.1 has 2 significant digits.

The precision of the representation *should* match the uncertainty of the value. However, precision of the representation and uncertainty of the value are separate independent concepts. Refer to Section 4.4.2 for details about uncertain real numbers.

For example "0.123" has 3 significant digits *in the representation*, but the *uncertainty of the value* may be in any digit shown or not shown, i.e., the uncertainty may be 0.123±0.0005, 0.123±0.005 or 0.123±0.00005, etc. Note that external representations *should* adjust their representational precision with the uncertainty of the value. However, since the precision in the digit string is granular to ±0.5 the least significant digit, while uncertainty may be anywhere between this raster, 0.123±0.005 would also be an adequate representation for the value between 0.118 and 0.128.

ITS Note: on a character based Implementation Technology the ITS need not represent the precision as an explicit attribute if numbers are represented as decimal digit strings. In that case, the ITS must abide by the rules of an unambiguous determination of significant digits. A number representation must not produce more or less significant digits than were originally in that number. Conformance can be tested through round-trip encoding – decoding – encoding.

2.12 Ratio (RTO)

A ratio quantity is a quantity constructed through division of a numerator quantity with a denominator quantity. Ratios are *different from* rational numbers, i.e., in ratios common factors in the numerator and denominator never cancel out. A ratio of two real or integer numbers is not automatically reduced to a real number.

Table 31: Summary of Primary Properties of Ratio (RTO)

Name	Type	Status	Default	Constraint	Definition
numerator	QTY	mandatory	(INT) 1		The numerator of the ratio.
denominator	QTY	mandatory	(INT) 1	$\neq 0$	The denominator of the ratio

The purpose of the ratio data type is to support certain quantities produced by laboratories, such as *titers* (e.g., "1:128"). Ratios are not simply "structured numerics", blood pressure measurements (e.g. "120/60") are not ratios.

Note: This data type is not defined to generally represent rational numbers. In this Ratio data type, it is *not* correct to cancel out common factors in numerator and denominator. For example, if a ratio is recorded as 2:8, it should not be reduced to 1:4.

```
type Ratio alias RTO extends QTY {
             QTY        numerator;
             QTY        denominator;

  demotion   REAL;
  demotion   PQ;
};
```


2.12.1.1 numerator (QTY)

This is the numerator quantity. The default is the integer number 1 (one.)

2.12.1.2 denominator (QTY)

This is the denominator quantity. The default is the integer number 1 (one.) The denominator must not be zero.

```
invariant(RTO x) where x.nonNull {
  x.denominator.isZero().not();
};
```

2.12.1.3 Literal Form

The syntax and semantics of ratio literals is defined below. In summary, a ratio literal form exists for all ratios where both numerator and denominators have literal forms. A ratio is simply the numerator literal a colon as separator followed by the denominator literal. When the colon and denominator are missing, the integer number 1 is assumed as the denominator.

```
RTO.literal ST {
  RTO : QTY                          { $.numerator.equals($1);
                                       $.denominator.equals((INT)1); };
      | QTY ":" QTY                  { $.numerator.equals($1);
                                       $.denominator.equals($3); };
};
```

2.13 Physical Quantity (PQ)

A physical quantity is a dimensioned quantity expressing the result of a measurement act.

Table 32: Summary of Primary Properties of Physical Quantity (PQ)

Name	Type	Status	Default	Constraint	Definition
value	REAL	mandatory	NULL		The magnitude of the quantity measured in terms of the unit
unit	CS	mandatory	*1*	UCUM	The unit of measure

```
type PhysicalQuantity alias PQ extends QTY {
             REAL       value;
             CS         unit;

             BL         equals(PQ x)
             BL         lessOrEqual(PQ x);
             BL         compares(PQ x);
             PQ         canonical;
```

```
  type       PQ         diff
             diff       minus(PQ x);
             PQ         plus(diff x);
             PQ         negated;

             PQ         times(REAL x);
             PQ         times(PQ x);
             PQ         inverted;
             PQ         power(INT x);

  literal    ST;
  demotion   REAL;
};
```

2.13.1.1 value : REAL

This is the magnitude of the quantity measured in terms of the unit.

2.13.1.2 unit : CS

This is the unit of measure. The unit of measure must be specified in the *Unified Code for Units of Measure* (UCUM) [http://aurora.rg.iupui.edu/UCUM].

Note that equality of physical quantity does not require the values and units to be equal independently. Value and unit is only how we represent physical quantities. For example, 1 m equals 100 cm. Although the units are different and the values are different, the physical quantities are equal! Therefore one should never expect a particular unit for a physical quantity but instead provide automated conversion between different comparable units.

2.13.1.3 Equality, Ordering and the Canonical Form

Physical quantities semantically are the results of measurement acts. Although physical quantities are represented as pairs of value and unit, semantically, a physical quantity is more than that. To find out whether two physical quantities are equal, it is not enough to compare equality of their two values and units independently. For example, semantically 100 cm equals 1 m although neither values nor units are equal. To define equality we introduce the notion of a canonical form.

Every physical quantity has a canonical form. The canonical form is a physical quantity expressed as a pair of value and unit such that each dimension in a given unit system has one and only one canonical value-unit pair. Defining the canonical form is not subject of this specification, only asserting that such a canonical form exists for every physical quantity. A physical quantity is equal to its canonical form.

For example, for a unit system based on the Système International (SI) one can define the canonical form as (a) the product of only the base units; (b) without prefixes; where (c) only multiplication and exponents are used (no division operation); and (d) where the seven base units appear in a defined ordering (e.g., m, s, g…) Thus, 1 mm Hg would be expressed as 133322 m^{-1} s^{-2} g. As can be seen, the rules how to build the canonical form of units may be quite complex. However, for the semantic specification it doesn't matter how the canonical form is built, or what specific canonical form is chosen, only that *some* canonical form *could* be defined.

Two physical quantities are equal if each their values and their units of their canonical forms are equal.

Two physical quantities compare each other (and have an ordering and difference) if the units of their canonical forms are equal.


```
invariant(PQ x, y) where x.nonNull.and(y.nonNull) {
  x.canonical.equals(x);
  x.equals(y).implies(x.compares(y));

  x.equals(y).equals(x.canonical.value.equals(y.canonical.value)
               .and(x.canonical.unit.equals(y.canonical.unit)));

  x.compares(y).equals(x.canonical.unit.equals(y.canonical.unit));
};
```

2.13.1.4 Algebraic Operations

Algebraic operations are defined for physical quantities because they are characterizing operations in the sense of ISO 11404 and because this specification makes use of them when defining the literal form.

Any two physical quantities can be multiplied. The quotient of two comparable quantities is comparable to the unity (the unit **1**).

```
invariant(PQ x, y, z)
    where x.nonNull.and(y.nonNull).and(z.nonNull) {
  x.compares(y).implies(x.times(y.inverted).compares(1));
  x.times(1).equals(x);                          /* neutral element */
  x.times(x.inverted).equals(1);                 /* inverse element */
  x.times(y).times(z).equals(x.times(y.times(z))); /* associative */
  x.times(y).equals(y.times(x));                 /* commutative */
};
```

A physical quantity can be multiplied with a real number to form a scaled quantity. A scaled quantity is comparable to its original quantity. If two quantities Q_1 and Q_2 compare each other, there exists a real number r such that $r\,\mathbf{1} = Q_1 / Q_2$.

```
invariant(PQ x, y; REAL r)
    where x.nonNull.and(y.nonNull).and(r.nonNull) {
  x.times(r).value.equals(x.value.times(r));
  x.times(r).compares(x);
};
```

A physical quantity Q that compares the unity – i.e. the unit **1** (one) – can be converted to a real number r such that $r\,\mathbf{1} = Q$.

```
invariant(PQ x) where x.nonNull.and(x.compares(unity) {
  unity.times((REAL)x).equals(x));
};
```

A physical quantity can be raised to an integer power.

```
invariant (PQ x; INT n) where x.nonNull {
  x.power(0).equals(1);
  n.greaterThan(0).implies(
      x.power(n).equals(x.times(x.power(n.predecessor))));
  n.lessThan(0).implies(
      x.power(n).equals(x.power(n.negated).inverted);
}
```

Two physical quantities that compare each other can be added.

```
invariant (PQ x, y, z)
    where x.compares(y).and(y.compares(z)) {
  x.plus(y).plus(z).equals(x.plus(y.plus(z)));     /* associative */
  x.plus(x.times(0)).equals(x)                     /* neutral elem. */
  x.plus(x.negated).equals(x.times(0))             /* inverse elem. */
  x.plus(y).equals(y.plus(x))                      /* commutative */

  forall(PQ w) with w.nonNull {
    w.times(x.plus(y))                             /* distributive */
       .equals(w.times(x).plus(w.times(y)));
  };

  forall(REAL r) where r.nonNull {
    x.plus(y).times(r)                             /* distributive */
       .equals(x.times(r).plus(y.times®));
  };
};
```

2.13.1.5 Literal Form

The literal form for a physical quantity is a real number literal followed by optional white space and a character string representing a valid code in the *Unified Code for Units of Measure*.

```
PQ.literal ST {
  PQ : REAL unit        { $.value.equals($1);
                          $.unit.equals($2); }

  CS unit : ST          { $.value.equals($1);
                          $.codeSystem.equals(2.16.840.1.113883.6.8); };
};
```

For example 20 minutes is "`20 min`".

2.14 Monetary Amount (MO)

A monetary amount is a quantity expressing the amount of money in some currency. Currencies are the units in which monetary amounts are denominated in different economic regions. While the

monetary amount is a single kind of quantity (money) the exchange rates between the different units are variable. This is the principle difference between physical quantity and monetary amounts, and the reason why currency units are not physical units.

Table 33: Summary of Primary Properties of Monetary Amount (MO)

Name	Type	Status	Default	Constraint	Definition
value	REAL	mandatory	NULL		The magnitude of the monetary amount in terms of the currency unit.
currency	CS	mandatory	NULL	ISO 4217	The currency unit

```
type MonetaryAmount alias MO extends QTY {
            REAL       value;
            CS         currency;
  type      MO         diff
            MO         plus(diff x);
            diff       minus(MO x);
            MO         negated;
            MO         times(REAL x);
  literal   ST;
  type      MO         diff;
};
```

2.14.1.1 value : REAL

This is the magnitude of the monetary amount in terms of the currency unit.

Note: monetary amounts are usually precise to 0.01 (one cent, penny, paisa, etc.) For large amounts, it is important not to store monetary amounts in floating point registers, since this may lose precision. However, this specification does not define the internal storage of real numbers as fixed or floating point numbers.

The precision attribute of the real number type is the precision of the decimal representation, not the precision of the value. The real number type has no notion of uncertainty or accuracy. For example, “1.99 USD” (precision 3) times 7 is “13.93 USD” (precision 4) and should not be rounded to “13.9” to keep the precision constant.

2.14.1.2 currency : CS

The currency unit as defined in ISO 4217.

Table 34: Select ISO 4217 currency codes

Country	Currency	Code
Argentina	Argentine Peso	ARS
Australia	Australian Dollar	AUD
Austria	Austrian Schilling	ATS
Belgium	Belgian Franc	BEF
Brazil	Brazilian Real	BRL
Canada	Candian Dollar	CAD
Chile	Unidades de Formento	CLF
China	Yuan Renminbi	CNY

Cuba	Cuban Peso	CUP
European Union	Euro	EUR
European Union	ECU (until 1998-12-31)	XEU
Finland	Markka	FIM
France	French Franc	FRF
Germany	Deutsche Mark	DEM
India	Indian Rupee	INR
Israel	Shekel	ILS
Japan	Yen	JPY
Korea (south)	Won	KRW
Luxembourg	Luxembourg Franc	LUF
Mexico	Mexican Nuevo Peso	MXN
Netherlands	Netherlands Guilder	NLG
New Zealand	New Zealand Dollar	NZD
Norway	Norwegian Krone	NOK
Philippines	Philippine Peso	PHP
Russian Federation	Russian Ruble	RUR
Singapore	Singapore Dollar	SGD
South Africa	Rand	ZAR
Spain	Spanish Peseta	ESP
Sweden	Swedish Krona	SEK
Switzerland	Swiss Franc	CHF
Thailand	Baht	THB
Taiwan	New Taiwan Dollar	TWD
United Kingdom	Pound Sterling	GBP
United States	US Dollar	USD

2.14.1.3 Algebraic Operations

Equality of two monetary amounts – unlike physical quantities – is determined as the joint equality of their value and currency properties independently. (This is according to the general definition of equality as defined in Section 1.4.2.3.) If the currencies are not equal, the amounts can not be compared. Conversion between the currencies is outside the scope of this specification. In practice, foreign exchange rates are highly variable not only over long and short amounts of time, but also depending on location and access to currency trade markets.

```
invariant(MO x, y) where x.nonNull.and(y.nonNull) {
  x.equals(y).equals(x.currency.equals(y.currency)
               .and(x.value.equals(y.value)));

  x.currency.equals(y.currency).not.implies(x.lessOrEqual(y).isNull);
};
```

Two monetary amounts can be added if they are denominated in the same currency.

```
invariant(MO x, y) where x.nonNull.and(y.nonNull)
     .and(x.currency.equals(y.currency)) {
  x.plus(y).value.equals(x.value.plus(y.value));
  x.plus(y).currency.equals(x.currency);
};
```


Any monetary amount can be multiplied with a real number.

```
invariant(MO x; REAL r) where x.nonNull.and(r.nonNull) {
  x.times(r).value.equals(x.value.times(r));
  x.times(r).currency.equals(x.currency);
};
```

2.14.1.4 Literal Form

The literal form for a monetary amount is a real number literal followed by optional white space and a character string representing a valid code.

```
MO.literal ST {
  MO : value currency          { $.value.equals($1);
                                 $.currency.equals($2); }

  REAL value : REAL            { $.value.equals($1); }

  CS currency : ST             { $.currency.value.equals($1);
                                 $.currency.codeSystem
                                    .equals(2.16.840.1.113883.6.9); }
};
```

For example U.S. $176,000 is "`176000 USD`".

2.15 Point In Time (TS)

A point in time is a scalar defining a point on the axis of natural time. A point in time is most often represented as a calendar expression. Semantically, however, time is independent from calendars. The semantic properties of point in time are best described by their relationship to elapsed time (measured as a physical quantity in the dimension of time.) A point in time plus an elapsed time yields another point in time. Inversely, a point in time minus another point in time yields an elapsed time. As a kind of quantity, points in time are a difference-scale quantity, where no absolute zero-point exists, where only differences are defined but no ratios. (For example, no point in time is – absolutely speaking – "twice as late" as another point in time.)

Given some arbitrary zero-point, one can express any point in time as an elapsed time measured from that offset. Such an arbitrary zero-point is called an epoch. This epoch-offset form is used as a semantic representation here, without implying that any system would have to implement the TS data type in that way. Systems that do not need to compute distances between points in time will not need any other representation than a calendar expression literal.

```
type PointInTime alias TS extends QTY {
            PQ          offset;
            CS          calendar;
            INT         precision;
            PQ          timezone;

            BL          equals(TS x);
            TS          plus(PQ x);
            PQ          minus(TS x);
```



```
  literal    ST;
  type       PQ         diff;
};
```

2.15.1.1 offset : PQ

The time elapsed since any constant epoch, measured as a physical quantity in the dimension of time (i.e., comparable to one second.) It is not necessary for this specification to define a canonical epoch; the semantics is the same for any epoch, as long as it is constant. Two point-in-time values are equal if and only if their offsets (relative to the same epoch) are equal.

```
invariant(TS x, y) where x.nonNull.and(y.nonNull) {
  x.offset.compares(1 s);
  x.equals(y).equals(x.offset.equals(y.offset));
};
```

ITS Note: the offset property may be treated as a purely semantic property that is not represented in any way other than the calendar literal expression. However, an ITS may just as well choose to define a constant epoch and represent point-in-time values as elapsed time offsets relative to that epoch. However, an ITS using an epoch-offset representation would still need to communicate the calendar code and the precision of a calendar representation once other calendars are supported.

2.15.1.2 calendar : CS

A code specifying the calendar used in the literal representation of this point in time.[44]

Table 35: Calendar Codes

Name	Code	Definition
Gregorian	GREG	The Gregorian calendar is in effect in the most countries of Christian influence since approximately 1582. This calendar superceded the Julian calendar.

The purpose of this attribute is mainly to faithfully convey what has been entered or seen by a user in a system originating such a point-in-time value. The calendar property also advises any system rendering a point-in-time value into a literal form of which calendar to use. However, this is only advice; any system that renders point-in-time values to users may choose to use the calendar and literal form demanded by its users rather than the calendar mentioned in the calendar property. Hence, the calendar property is not constant in communication between systems, the calendar is not part of the equality test.

A calendar is a concept of measuring time in various cycles. Such cycles are years, months, days, hours, minutes, seconds, and weeks. Some of these cycles are synchronized and some are not (e.g., weeks and months are not synchronized.) After "rolling the time axis" into these cycles (see Figure 9,) a calendar expresses a point in time as a sequence of integer counts of cycles, e.g., for year, month, day, hour, etc. The calendar is rooted in some conventional start point, called the "epoch."

For the purpose of defining the literal form based on the calendar two private data types, Calendar (CAL) and CalendarCycle (CLCY,) are defined. These calendar data types exist only for defining this specification. These private data types may not be used at all outside this specification.

[44] At this time, no other calendars than the Gregorian calendar are defined. However, the notion of a calendar as an arbitrary convention to specify absolute time is important to properly define the semantics of time and time-related data types. Furthermore, other calendars might be supported *when needed* to facilitate HL7's use in other cultures.

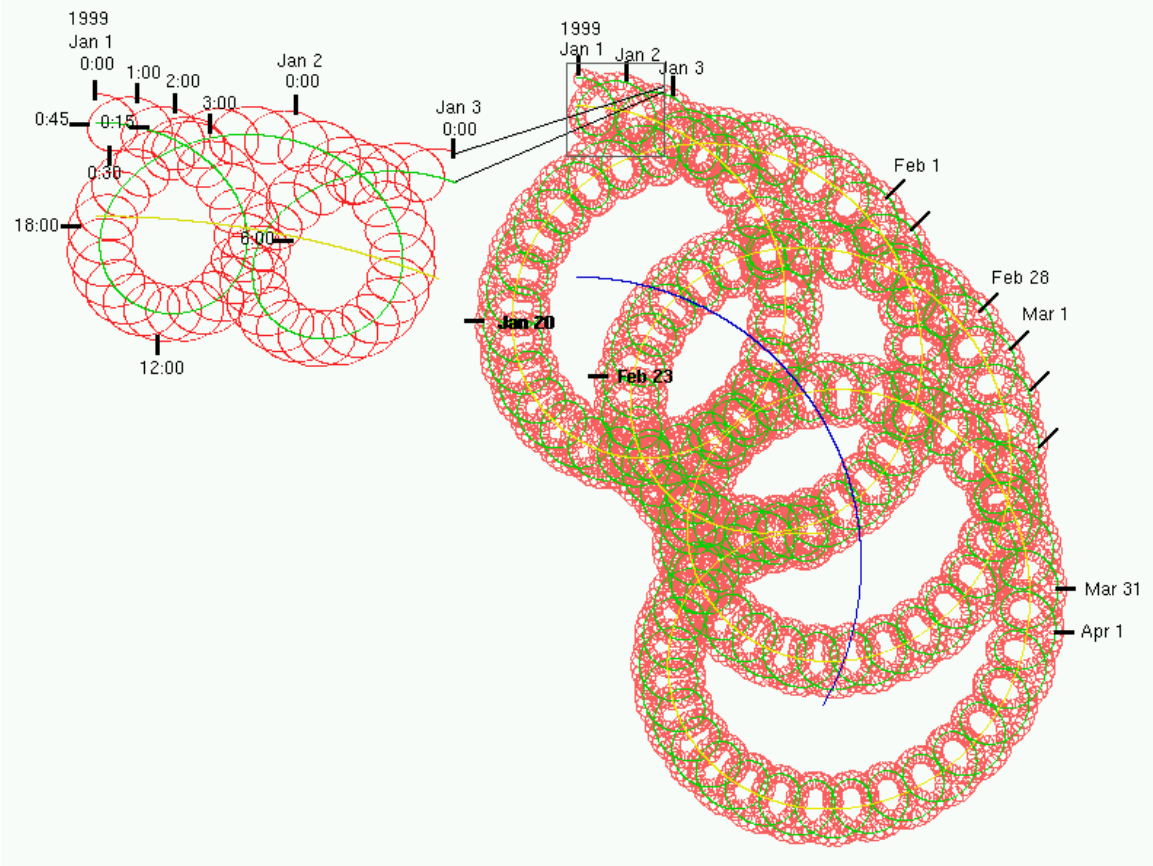


Figure 9: A calendar "rolls" the time axis into a complex convolute according to the calendar periods year (blue), month (yellow), day (green), hour (red), etc. The cycles need not be aligned, for example, the week (not shown) is not aligned to the month.[45]

Calendar is defined as a set of calendar cycles, and has a name and a code. The head of the Calendar is the largest CalendarCycle appearing right most in the calendar expression. The epoch is the beginning of that calendar, i.e., the point in time where all calendar cycles are zero.

```
private type Calendar alias CAL extends SET⟨CLCY⟩  {
            CV          name;
            CLCY        head;
            TS          epoch;
};

invariant(CAL c) where c.nonNull {
  c.name.nonNull;
  c.contains(c.head);
};
```

A calendar cycle defines one group of decimal digits in the calendar expression. A calendar cycle has a name and two codes, a one-letter code and a two-letter code. The property *ndigits* is the number of

[45] Imagine a special clock that measures those cycles, where the pointers are not all stacked on a common axis but each pointer is attached to the end of the pointer measuring the next larger cycle.

decimal digits occupied in the calendar expression. The property *start* specifies where counting starts (i.e., at 0 or 1.) The *next* property is the next lower cycle in the order of the calendar expression. The max(*t*) property is the maximum number of cycles at time *t* (max depends on the time *t* to account for leap years and leap seconds.) The property value(*t*) is the integer number of cycles shown in the calendar expression of time *t*. The property sum(*t*, *n*) is the sum of *n* calendar cycles added to the time *t*.

```
private type CalendarCycle alias CALCY extends ANY {
            CE          name;
            INT         ndigits;
            INT         start;
            CALCY       next;
            INT         max(TS);
            TS          sum(TS t, REAL r);
            INT         value(TS t);
};

invariant(CALCY c) where c.nonNull {
  c.name.nonNull;
  c.start.equals(0).or(c.start.equals(1));
  c.digits.greaterThan(0);
};
```

The calendar definition can be shown as in Table 36 for the modern Gregorian calendar. The calendar definition table lists a calendar cycle in each row. The calendar units are dependent on each other and defined in the value column. The sequence column shows the relationship through the next property. The other columns are as in the formal calendar cycle definition.[46]

Table 36: Calendar Periods for the Modern Gregorian Calendar

Name	Code		Counter			Period Duration		
	one	two	seq.	digits	start	condition		value
year	Y	CY	1	4	0			MY12
month of the year	M	MY	2	2	1	MY01,03,05,07,08,10,12	→	DM31
						MY04,06,09,11	→	DM30
						MY02 Y/4 Y/100	→	DM28
						MY02 Y/4	→	DM29
						MY02	→	DM28
month (continuous)		CM			0	*continuous*		MY
week (continuous)	W	CW			0			CD7
week of the year		WY		2	1	*continuous*		DW7
day of the month	D	DM	3	2	1			HD24
day (continuous)		CD			0			CH24
day of the year		DY		3	1			HD24
day of the week (begins with Monday)	J	DW		1	1			HD24
hour of the day	H	HD	4	2	0			MH60

[46] At present, the CalendarCycle properties sum and value are not formally defined. The computation of calendar digits involves some complex computation which to specify here would be hard to understand and evaluate for correctness. Unfortunately, no standard exists that would formally define the relationship between calendar expressions and elapsed time since an epoch. ASN.1, the XML Schema Data Type specification and SQL92 all refer to ISO 8601, however, ISO 8601 does only specify the syntax of Gregorian calendar expressions, but not their semantics. In this standard, we define the syntax and semantics formally, however, we presume the semantics of the sum-, and value-properties to be defined elsewhere.

hour (continuous)		CH			0			CN60
minute of the hour	N	NH	5	2	0	*UTC leap second*	→	SN61
							→	SN60
minute (continuous)		CN			0			CS60
second of the minute	S	SN	6	2	0			CS1
second (continuous)		CS			0	*basis*		

2.15.1.3 precision : INT

The purpose of the precision property for the point in time data type is to faithfully capture the whole information presented to humans in a calendar expression. The number of digits shown conveys information about the uncertainty (i.e., precision and accuracy) of a measured point in time. Although, the precision of a calendar expression is not a good measure for the uncertainty of the value, the precision of the calendar expression *should* match the accuracy of the measurement.

Note: the precision of the representation is independent from uncertainty (precision accuracy) of a measurement result. If the uncertainty of a measurement result is important, one should send uncertain values as defined in Section 4.4.

The precision property is dependent on the calendar. A given precision value relative to one calendar does not mean the same in another calendar with different periods.

For example "20000403" has 8 significant digits *in the representation*, but the *uncertainty of the value* may be in any digit shown or not shown, i.e., the uncertainty may be to the day, to the week, or to the hour. Note that external representations *should* adjust their representational precision with the uncertainty of the value. However, since the precision in the digit string depends on the calendar and is granular to the calendar periods, uncertainty may not fall into that grid (e.g., 2000040317 is an adequate representation for the value between 2000040305 and 2000040405.)

ITS Note: on a character based Implementation Technology the ITS need not represent the precision as an explicit attribute if point in time values are represented as literal calendar expressions. A point in time representation must not produce more or less significant digits than were originally in that value. Conformance can be tested through round-trip encoding – decoding – encoding.

2.15.1.4 time zone : PQ

The time zone is specified as the difference between the local time in that time zone and Universal Coordinated Time (UTC, formerly called Greenwich Mean Time, GMT). The time zone is a physical quantity in the dimension of time (i.e., comparable to one second.) A zero time zone value specifies UTC. The time zone value does not permit conclusions about the geographical longitude or a conventional time zone name.

For example, 200005121800-0500 may be eastern standard time (EST) in Indianapolis, IN, or central daylight savings time (CDT) in Decatur, IL. Furthermore in other countries having other latitude the time zones may be named differently.

```
invariant(TS x, y) where x.nonNull.and(y.nonNull) {
  x.timezone.compares(1 s);
};
```

When the time zone is NULL (unknown), "local time" is assumed. However, "local time" is always local to some place, and without knowledge of that place, the time zone is unknown. Hence, a local time can not be converted into UTC. The time zone should be specified for all point in time values in order to avoid a significant loss of precision when points in time are compared. The difference of two local times where the locality is unknown has an error of ±12 hours.

In administrative data context, some time values do not carry a time zone. For a date of birth in administrative data, for example, it would be incorrect to specify a time zone, since this may

effectively change the date of birth when converted into other time zones. For such administrative data the time zone is NULL (not applicable.)

2.15.1.5 Addition and Subtraction

A point in time plus an elapsed time (i.e., physical quantity in the dimension of time) is a point in time. Inversely, the difference between two points in time is an elapsed time.

```
invariant(TS x, PQ t)
    where x.nonNull.and(t.compares(1 s)) {
  x.plus(t).offset.equals(x.offset.plus(t));
  x.minus(y).offset.equals(x.offset.plus(y.offset.negated));
};
```

2.15.1.6 Literal Form

Point-in-time literals are simple calendar expressions, as defined by the calendar definition table. By default, the western (Gregorian) calendar shall be used (Table 36).

For the default Gregorian calendar the calendar expression literals of this specification conform to the constrained ISO 8601 that is defined in ISO 8824 (ASN.1) under clause 32 (generalized time) and to the HL7 version 2 TS data format.

Calendar expression literals are sequences of integer numbers ordered according to the "Counter/ord." column of Table 36. Periods with lower order numbers stand to the left of periods with higher order numbers. Periods with no assigned order number cannot occur in the calendar expression for points in time.

The "Counter/digits" column of Table 36 specifies the exact number of digits for the counter number for any period.

Thus, Table 36 specifies that western calendar expressions begin with the 4-digit year (beginning counting at zero); followed by the 2-digit month of the year (beginning counting at one); followed by the 2-digit day of the month (beginning with one); followed by the 2-digit hour of the day (beginning with zero); and so forth. For example, "`200004010315`" is a valid expression for April 1, 2000, 3:15 am.

A calendar expression can be of variable precision, omitting parts from the right.

For example, "`20000401`" is precise only to the day of the month.

The last calendar unit may be written as a real number, with the number of integer digits specified, followed by the decimal point and any number of fractional digits.

For example, "`20000401031520.34`" means April 1, 2000, 3:15 and 20.34 seconds.

When other calendars will be used in the future, a prefix "`GREG:`" can be placed before the western (Gregorian) calendar expression to disambiguate from other calendars. Each calendar shall have its own prefix. However, the western calendar is the default if no prefix is present.

In the modern Gregorian calendars (and all calendars where time of day is based on UTC,) the calendar expression may contain a time zone suffix. The time zone suffix begins with a plus (+) or minus (–) followed by digits for the hour and minute cycles. UTC is designated as offset "`+00`" or "`-00`"; the ISO 8601 and ISO 8824 suffix "`Z`" for UTC is not permitted.


```
TS.literal ST {
  TS : cal timestamp($1)              { $.equals($2); }
     | timestamp(GREG)                { $.equals($1); };


  TS timestamp(Calendar C)
  : cycles(C.head, C.epoch) zone(C)   { $.equals($1.minus($2)); }
                                        $.timezone.equals($2); }
  | cycles(C.head, C.epoch)           { $.equals($1);
                                        $.timezone.unknown; };

  Calendar cal
  : /[a-zA-Z_][a-zA-Z0-9_]*:/         { $.equals($1); };

  TS cycles(CalendarCycle c, TS t)
  : cycle(c, t) cycles(c.next, $1)    { $.equals($2); }
  | cycle(c, t) "." REAL.fractional   { $.equals(c.sum($1, $3));
                                        $.precision.equals(
                                          t.precision.plus($3.precision)); }
  | cycle(c, t)                       { $.equals($1); };

  TS cycle(CalendarCycle c, TS t)
  : /[0-9]{c.ndigits}/                { $.equals(c.sum(t, $1));
                                        $.precision.equals(
                                          t.precision.plus(c.ndigits)); };

  PQ zone(Calendar C)
  : “+” cycles(C.zonehead, C.epoch)   { $.equals($2.minus(C.epoch)); }
  | “-” cycles(C.zonehead, C.epoch)   { $.equals(C.epoch.minus($2)); };
}
```

3 Generic Collections

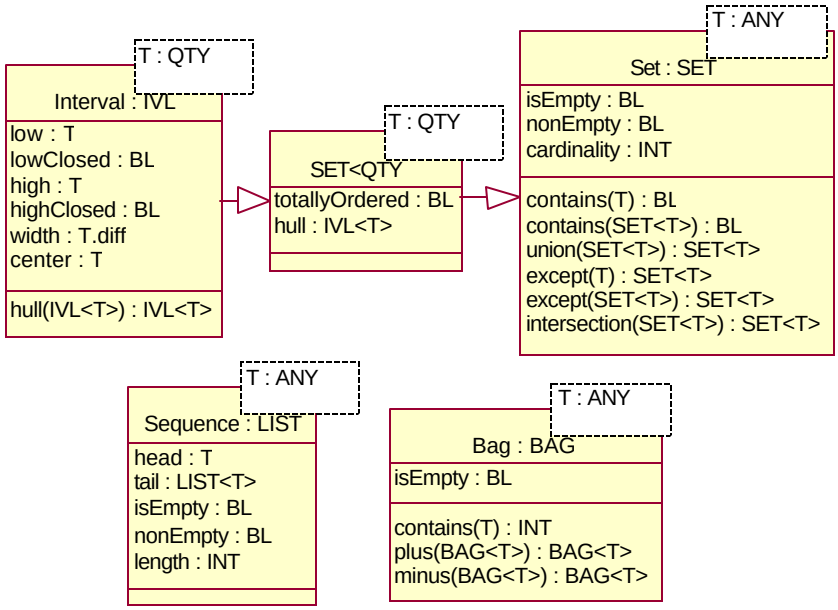


Figure 10: Generic Collection Data Types

This section defines data types that can "collect" other data values, Set, Sequence, Bag and Interval.[47] These collection types are defined as generic (parameterized) types. The concept of generic types is described in Section 1.3.5.

3.1 Set (SET)

A set is a value that contains other values of a certain data type as its elements. The elements are contained in no particular ordering. All elements in the set are distinct, the same element value can not be contained more than once in the set.

```
template<ANY T>
type Set<T> alias SET<T> extends ANY {
            BL          contains(T element);
            BL          isEmpty;
            BL          nonEmpty;
            BL          contains(SET<T> subset);
            INT         cardinality;

            SET<T>      union(SET<T> otherset);
            SET<T>      except(T element);
            SET<T>      except(SET⟨T⟩ otherset);
            SET<T>      intersection(SET<T> otherset);
```

[47] In some programming languages, "collection types" are understood as containers of individually enumerated data items, and thus, an interval (low – high) would not be considered a collection. Such narrow interpretation of "collection" however is heavily representation/implementation dependent. From a data type semantics viewpoint, it doesn't matter whether an element of a collection "is actually contained in the collection" or not. There is no need for all elements in a collection to be individually enumerated.


```
  literal    ST;
  promotion  SET<T>     (T x);
};
```

3.1.1.1 Element

The primitive semantic property of a set is the *contains*-relation of elements in the set. On this semantic primitive, all other properties are defined. A set may only contain distinct non-NULL elements. Exceptional values (NULL-values) can not be elements of a set.

```
invariant(SET<T> x) where x.nonNull {
  forall(T e) where x.contains(e) { x.nonNull; };
};
```

3.1.1.2 Cardinality and Empty Set

The empty set is a set without any elements. The empty set is a proper set value, *not* an exceptional (NULL) value. The cardinality of a set is the number of distinct elements in the set.

```
invariant(SET<T> x) where x.nonNull {
  x.nonEmpty.equals(exists(T e) { x.contains(e); });
  x.isEmpty.equals(nonEmpty.not);

  exists(T e) where x.contains(e) {
    x.cardinality.equals(x.except(e).cardinality.successor);
  };
};
```

The cardinality definition is not sufficient since it doesn't converge for uncountably infinite sets (REAL, PQ, etc.) and it doesn't terminate for infinite sets. In addition, the definition of integer number type in this specification is incomplete for these cases, as it doesn't account for infinities. Finally the cardinality value is an example where it would be necessary to distinguish the cardinality $\aleph_0$ ($aleph_0$) of countably infinite sets (e.g., INT) from $\aleph_1$ ($aleph_1$), the cardinality of uncoutable sets (e.g., REAL, PQ).

3.1.1.3 Subset

A subset of a superset is a set where each element in the subset is also an element in the superset.

```
invariant(SET<T> superset, subset; T element)
    where superset.nonNull.and(subset.nonNull).and(element.nonNull) {
  superset.contains(subset)
    .equals(subset.contains(element).implies(superset.contains(element)));
};
```

3.1.1.4 Union

A union of two sets *X* and *Y* is the set *Z* where *e* is an element of *Z* if and only if *e* is also an element of *X* or an element of *Y*.

```
invariant(SET<T> x, y, z)
    where x.nonNull.and(y.nonNull).and(z.nonNull) {
  x.union(y).equals(z)
    .equals(forall(T e) {
              z.contains(e).equals(x.contains(e).or(y.contains(e)));
            });
};
```

3.1.1.5 Difference

The difference (*X* except *Y*) of two sets is the set *Z*, where *e* is an element of *Z* if and only if *e* is an element of *X* and not an element of *Y*.

```
invariant(SET<T> x, y, z)
    where x.nonNull.and(y.nonNull).and(z.nonNull) {
  x.except(y).equals(z)
    .equals(forall(T e) {
              z.contains(e).equals(x.contains(e).and(y.contains(e).not));
            });
};
```

The difference between a set *X* and an element *d* (*X* except *d*) is the set *Z*, where *e* is an element of *Z* if and only if *e* is an element of *X* and *e* is not equal to *d*.

```
invariant(SET<T> x, z; T d)
    where z.nonNull.and(z.nonNull).and(d.nonNull) {
  x.except(d).equals(z)
    .equals(forall(T e) {
              z.contains(e).equals(x.contains(e).and(d.equals(e).not));
            });
};
```

3.1.1.6 Intersection

The intersection between two sets *X* and *Y* is the set *Z* where *e* is an element of *Z* if and only if it is contained in both of the sets *X* and *Y*.

```
invariant(SET<T> x, y, z)
    where x.nonNull.and(y.nonNull).and(z.nonNull) {
  x.intersection(y).equals(z)
    .equals(forall(T e) {
              z.contains(e).equals(x.contains(e).and(y.contains(e)));
            });
};
```


3.1.1.7 Literal Form

When the element type T has a literal form, the set SET<T> has a literal form, wherein the elements of the set are enumerated within curly braces and separated by semicola.

```
SET<T>.literal ST {
  SET<T> : "{" elements "}"          { $.equals($2); };
  SET<T> elements
        : elements ";" T             { $.except($2).equals($1); }
        | T                          { $.contains($1);
                                       $.except($1).isEmpty; };
};
```

Note: this literal form for sets is only practical for relatively small enumerable sets; this does not mean, however, that all sets are relatively small enumerations of elements.

For example,

`{1; 3; 5; 7; 19}` is a set of integer numbers or real numbers;

`{1.2 m; 2.67 m; 17.8 m}` is a set of discrete physical quantities;

`{apple; orange; banana}` is a set of character strings.

ITS Note: a character-based ITS *should* choose a different literal form for sets if the Implementation Technology has a more native literal form for such collections.

3.1.1.8 Promotions of any Values to Sets

A data value of type T can be promoted into a trivial SET<T> with that data value as its only element.

```
invariant(T x) {
  ((SET<T>)x).contains(x);
  ((SET<T>)x).except(x).isEmpty;
};
```

3.1.2 Sets of Quantities, Total Ordering, and Convex Hull

Sets of quantities may be totally ordered sets when there is an order relationship defined between any two elements in the set. Note that "ordered set" does not mean the same as Sequence (LIST). For example, the set {3; 2; 4; 88; 1} is an ordered set. The ordering of the elements in the set notation is still irrelevant, but elements can be compared to establish an order (1; 2; 4; 88).

Totally ordered sets have convex hull. A convex hull of a totally ordered set *S* is the smallest interval that is a superset of *S*. This concept is going to be important later on.

```
type Set<QTY> alias SET<QTY> {
            BL          totallyOrdered;
            IVL<T>      hull;
};
```

```
invariant(SET<QTY> s) where s.nonNull {
  s.totallyOrdered.equals(forall(QTY x, y) where s.contains(x)
                                                .and(s.contains(y)) {
                            x.compares(y); });
};

invariant(SET<QTY> s) where s.totallyOrdered {
  s.hull.contains(s);
  forall(T e) where s.contains(e) {
    s.hull.low.lessOrEqual(e);
    e.lessOrEqual(s.hull.high);
  };
};
```

Note that hull is defined if and only if the *actual* set is a totally ordered set. The data type of the elements itself need not be totally ordered. For example, the data type PQ is only partially ordered (since only quantities of the same kind can be compared), but a SET<PQ> may still be totally ordered (if it contains only comparable quantities.) For example, the convex hull of {4 s, 20 s, 55 s} is [4 s;55 s]; the convex hull of {“apples”; “oranges”; “bananas”} is undefined because the elements have no order relationship among them; and the convex hull of {2 m; 4 m; 8 s} is likewise undefined, because it is not *totally* ordered (seconds are not comparable with meters.)

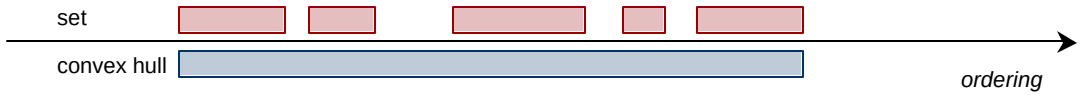


Figure 11: Convex Hull of a Totally Ordered Set

3.2 Sequence (LIST)

A sequence is an ordered collection of discrete values.

```
template<ANY T>
type Sequence<T> alias LIST<T> extends ANY {
             T          head;
             LIST<T>    tail;
             BL         isEmpty;
             BL         nonEmpty;
             INT        length;

  literal    ST;
  promotion  LIST<T>    (T x);
};
```

A non-empty sequence has a head and a tail. An empty sequence has length zero. Notice the difference between empty-sequence and NULL. The empty sequence is a proper sequence, not a NULL-value.


```
invariant(LIST<T> x) where x.isEmpty {
  x.head.isNull;
  x.tail.isNull;
  x.length.isZero;
};

invariant(LIST<T> x) {
  x.nonEmpty.equals(x.isEmpty.not);
}
```

The length of a sequence is the number of elements in the sequence. NULL elements are counted as regular sequence elements.

```
invariant(LIST<T> x) where x.nonEmpty {
  x.length.equals(x.tail.length.successor);
};
```

Two lists are equal if and only if they are both empty, or if both their head and their tail are equal.

```
invariant(LIST<T> x, y) where x.isEmpty.and(y.isEmpty) {
  x.equals(y);
}

invariant(LIST<T> x, y) where x.nonEmpty.and(y.nonEmpty) {
  x.equals(y).equals(x.head.equals(y.head)
                .and(x.tail.equals(y.tail)));
};
```

3.2.1.1 Literal Form

When the element type T has a literal form, the sequence LIST<T> has a literal form. List elements are enumerated, separated by semicolon, and enclosed in parentheses.

```
LIST<T>.literal ST {
  LIST<T>
  : "(" elements ")"                { $.equals($2); }
  | "(" ")"                         { $.isEmpty; };

  LIST<T> elements
         : T ";" elements           { $.head.equals($1);
                                      $.tail.equals($3); }
         | T                        { $.head.equals($1);
                                      $.tail.isEmpty; };
};
```

For example,

`(1; 3; 5; 7; 19)` is a sequence of integer numbers or real numbers;

`(1.2 m; 2.67 m; 17.8 m)` is a sequence of discrete physical quantities;

`(apple; orange; banana)` is a sequence of character strings.

ITS Note: a character-based ITS *should* choose a different literal form for sequences if the Implementation Technology has a more native literal form for such collections.

3.2.1.2 Promotions of any Values to Sequences

A data value of type T can be promoted into a trivial sequence LIST<T> with that data value as its only element.

```
invariant(T x) {
  ((LIST<T>)x).head.equals(x);
  ((LIST<T>)x).tail.isEmpty;
};
```

3.3 Bag (BAG)

A bag is an unordered collection of elements where each element can be contained more than once in the bag. The bag is defined only briefly here for completeness, since bags are a commonly recognized collection type.

```
template<ANY T>
type Bag<T> alias BAG<T> extends ANY {
             INT        contains(T kind);
             BL         isEmpty;
             BAG<T>     plus(BAG<T>);
             BAG<T>     minus(BAG<T>);

  promotion  BAG<T>     (T x);
};
```

ITS Note: a bag can be represented in two ways. Either as a simple enumeration of elements, including repeated elements, or as a "compressed bag" whereby the content of the bag is listed in pairs of element value and number. A histogram showing absolute frequencies is a bag represented in compressed form. The bag is therefore useful to communicate raw statistical data samples.

3.3.1.1 Elements

The semantic primitive for bags is the *contains*-function that maps element values to non-negative integer numbers, where zero means that the element value is not contained in the bag. An empty bag is distinguished from an exceptional bag value (the NULL bag.)

```
invariant(BAG<T> x; T e) where x.nonNull.and(e.nonNull) {
  x.contains(e).nonNegative;
  x.isEmpty.equals(x.contains(e).isZero);
};
```


3.3.1.2 Addition and Subtraction

Bags can be added, meaning that the *contains*-values for each element are added. Bags can and subtracted, meaning that the *contains*-values are subtracted. Note that bags can not carry deficites, i.e., the minimal *contains*-value is zero.

```
invariant(BAG<T> x, y, z) where x.nonNull.and(y.nonNull) {
  x.plus(y).equals(z)
    .equals(forall(T e) where e.nonNull {
              z.contains(e).equals(x.contains(e).plus(y.contains(e)));
            });

  x.minus(y).equals(z)
    .equals(forall(T e) where e.nonNull {
              exists(INT n)
                  where n.equals(x.contains(e).minus(y.contains(e)) {
                n.nonNegative.equals(z.contains(e));
                n.isNegative.equals(z.contains(e).isZero);
              };
            });
}
```

3.3.1.3 Promotions of any Values to Bags

A data value of type T can be promoted into a trivial bag BAG<T> with that data value as its only element.

```
invariant(T x) {
  ((BAG<T>)x).contains(x).equals(1);
  forall(T y) { ((BAG<T>)x).contains(y).implies(x.equals(y)) };
};
```

3.4 Interval (IVL)

An interval is a set of consecutive values of any ordered data type. An interval is thus a contiguous subset of its base data type. Any ordered type can be the basis of an interval. It does not matter whether the base type is discrete or continuous. If the base data type is only partially ordered, all elements of the interval must be elements of a totally ordered subset of the ordered data type.

For example, physical quantities are considered ordered. However the ordering of physical quantities is only partial; a total order is only defined among comparable quantities (quantities of the same physical dimension.) While intervals between 2 and 4 meter exists, there is no interval between 2 meters and 4 seconds.

Intervals are sets and have all the properties of sets. However, union and differences of intervals may not be intervals any more, since the elements of these union and difference sets might not be contiguous. Intersections of intervals are always intervals.

```
template<QTY T>
type Interval<T> alias IVL<T> extends SET<T> {
            T          low;
            BL         lowClosed;
            T          high;
            BL         highClosed;
            T.diff     width;
            T          center;

            IVL<T>     hull(IVL<T> x);

  literal   ST;
  promotion IVL<T>     (T x);
  demotion  T;
};
```

3.4.1.1 low : T

This is the low boundary of the interval.

```
invariant(IVL<T> x; T e) where x.nonNull.and(x.contains(e)) {
  x.low.lessOrEqual(e);
};
```

3.4.1.2 high : T

This is the upper boundary of the interval.

```
invariant(IVL<T> x; T e) where x.nonNull.and(x.contains(e)) {
  e.lessOrEqual(x.high);
};
```

3.4.1.3 width : T.diff

The width is the difference between high and low boundary. The purpose of distinguishing a width property is to handle all cases of incomplete information symmetrically. In any interval representation only two of the three properties high, low, and width need to be stated and the third can be derived.

When both boundaries are known, width can be derived as high minus low. When one boundary and the width is known, the other boundary is also known. When no boundary is known, the width may still be known. For example, one knows that an activity takes about 30 minutes, but one may not yet know when that activity is started.

```
invariant(IVL<T> x) {
  x.low.lessOrEqual(x.high);
  x.width.equals(x.high.minus(x.low));
};
```


3.4.1.4 center : T

The center is defined of finite intervals and is then the arithmetic mean of the interval (low plus high divided by 2). The purpose of distinguishing the center as a semantic property is for conversions of intervals to point values. This is most relevant when intervals are used to express uncertainty.

```
invariant(IVL<T> x) where x.low.nonNull.and(x.high.nonNull) {
  x.center.equals(x.low.plus(x.width.times(0.5))));
};

invariant(IVL<T> x) where x.low.isNull.or(x.high.isNull) {
  x.center.notApplicable;
};
```

3.4.1.5 lowClosed : BL

Indicates whether the interval is closed or open at the low boundary. For a boundary to be closed, a finite boundary must be provided, i.e. unspecified or infinite boundaries are always open.

```
invariant(IVL<T> x) where x.nonNull {
  x.low.nonNull.implies(x.lowClosed.equals(x.contains(x.low)));
  x.low.isNull.implies(x.lowClosed.not);
};
```

3.4.1.6 highClosed : BL

Indicates whether the interval is closed or open at the high boundary. For a boundary to be closed, a finite boundary must be provided, i.e. unspecified or infinite boundaries are always open.

```
invariant(IVL<T> x) where x.nonNull {
  x.high.nonNull.implies(x.highClosed.equals(x.contains(x.high)));
  x.high.isNull.implies(x.highClosed.not);
};
```

3.4.1.7 Literal Form

The literal form for the interval data type is defined such that it is as intuitive to humans a possible. Five different forms are defined:[48]

1) the interval form using square brackets, e.g., “[3.5; 5.5[”;

2) the dash-form, e.g., “3.5–5.5”;

[48] The presence of so many options deserves explanation. In principle, the interval form together with the width-only form would be sufficient. However, the interval form is felt alien to many in the field of medical informatics. One important purpose of the literal forms is to eradicate non-compliance through making compliance easy, without compromising on the soundness of the concepts.
Furthermore, the different literal forms all have strength and weaknesses. The interval and center-width forms’ strength is that they are most exact, showing closed and open boundaries. The interval form’s weakness, however, is that infinite boundaries require special symbols for infinities, not necessary in the “comparator” form. The center-width form cannot specify intervals with an infinite boundary at all. The “comparator” form, however, can only represent single-bounded intervals (i.e., where the other boundary is infinite or unknown.) The dash form, while being the weakest of all, is the most intuitive form for double bounded intervals.

3) the “comparator” form, using relational operator symbols, e.g., “<5.5”;

4) the center-width form, e.g., “4.5[2.0[”.

5) the width-only form using square brackets, e.g., “[2.0[”.

```
IVL<T>.literal ST {
  IVL<T> range
  : interval                  { $.equals($1); }
  | dash                      { $.equals($1); }
  | comparator                { $.equals($1); }
  | center_width              { $.equals($1); }
  | width                     { $.equals($1); };

  IVL<T> interval
  : open T “;” T close;       { $.low.equals($2);
                                $.high.equals($4);
                                $.lowClosed.equals($1);
                                $.highClosed.equals($5); };
  BL open : “[”               { $.equals(true); }
          | “]”               { $.equals(false); };
  BL close : “]”              { $.equals(true); }
           | “[”              { $.equals(false); };

  IVL<T> width
  : open T.diff close         { $.width.equals($2);
                                $.lowClosed.equals($1);
                                $.highClosed.equals($3); };

  IVL<T> center_width
  : T width                   { $.center.equals($1);
                                $.width.equals($2.width);
                                $.lowClosed.equals($2.lowClosed);
                                $.highClosed.equals($2.highClosed); };

  IVL<T> dash : T “-” T;      { $.low.equals($2);
                                $.high.equals($4);
                                $.lowClosed.equals(true);
                                $.highClosed.equals(true); };
```



```
  IVL<TS> comparator
  : "<"  T                    { $.high.equals(T);
                                $.high.closed(false);
                                $.low.negativelyInfinite; }
  | ">"  T                    { $.low.equals(T);
                                $.low.closed(false);
                                $.high.positivelyInfinite; }
  | "<=" T                    { $.high.equals(T);
                                $.high.closed(true);
                                $.low.negativelyInfinite; }
  | ">=" T                    { $.low.equals(T);
                                $.low.closed(true);
                                $.high.positivelyInfinite; };
};
```

Table 37: Examples of interval literals

literal	low		high		alternate	
	closed	low	high	closed	center	width
3.5-5.5	true	3.5	5.5	true	4.5	2.0
[3.5;5.5]	true	3.5	5.5	true	4.5	2.0
[3.5;5.5[	true	3.5	5.5	false	4.5	2.0
4.5[2.0]	true	3.5	5.5	true	4.5	2.0
4.5[2.0[	true	3.5	5.5	false	4.5	2.0
<5.5	false	–∞	5.5	false	N/A	∞
>3.5	false	3.5	∞	false	N/A	∞
>=3.5	true	3.5	∞	false	N/A	∞
<=5.5	false	–∞	5.5	true	N/A	∞
]-inf;5.5]	false	–∞	5.5	true	N/A	∞
[3.5;+inf[	true	3.5	∞	false	N/A	∞
];5.5]	false	UNK	5.5	true	UNK	UNK
[3.5;[	true	3.5	UNK	false	UNK	UNK
-3.5-3.5	true	–3.5	3.5	true	0.0	7.0
-5.5--3.5	true	–5.5	–3.5	true	–4.5	2.0
[-5.5;-3.5]	true	–5.5	–3.5	true	–4.5	2.0
-4.5[2.0]	true	–5.5	–3.5	true	–4.5	2.0
<-3.5	false	–∞	–3.5	false	N/A	∞
>-5.5	false	–5.5	∞	false	N/A	∞
[3.5;3.5]	true	3.5	3.5	true	3.5	0
[2.5]	true	UNK	UNK	true	UNK	2.5
[2.5[	true	UNK	UNK	false	UNK	2.5

3.4.1.8 Conversion Between Point Values and Intervals

A quantity type T can be promoted into a trivial interval IVL<T> where low and high boundaries are equal and boundaries closed.

```
invariant(T x) {
  ((IVL<T>)x).low.equals(x);
  ((IVL<T>)x).high.equals(x);
  ((IVL<T>)x).highClosed;
  ((IVL<T>)x).lowClosed;
};
```

An interval IVL<T> can be demoted to a simple quantity type T. If both boundaries are finite, the conversion yields the center of the interval. If one boundary is infinite, conversion yields the other boundary. If both boundaries are infinite, the conversion to a point value is not applicable.

```
invariant(IVL<T> x) where x.nonNull {
  x.low.nonNull.and(x.high.nonNull).implies(((T)x).equals(x.center));
  x.high.nonNull.and(x.low.isNull).implies(((T)x).equals(x.high));
  x.low.nonNull.and(x.high.isNull).implies(((T)x).equals(x.low));
  x.low.isNull.and(x.high.isNull).implies(((T)x).notApplicable);
};
```

3.4.1.9 Convex Hull

A convex hull or "interval hull" of two intervals is the least interval that is a superset of its operands. This concept will play an important role later on.

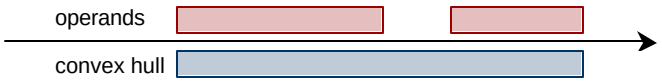


Figure 12: Convex Hull of two Intervals

```
invariant(IVL<T> h, IVL<T> i, j) where h.equals(i.hull(j)) {
  i.low.lessOrEqual(j.low).implies(h.low.equals(i.low));
  j.low.lessOrEqual(i.low).implies(h.low.equals(j.low));
  i.high.lessOrEqual(j.high).implies(h.high.equals(j.high));
  j.high.lessOrEqual(i.high).implies(h.high.equals(i.high));
};
```

3.4.2 Interval of Physical Quantities (IVL<PQ>)

An interval of physical quantities is constructed from the generic interval type. However, recognizing that the unit can be factored from the boundaries, we add additional semantics and a separate literal form. The additional view of an interval of physical quantities is an interval of real numbers with one unit.

```
type Inteval<PQ> alias IVL<PQ> {
            IVL<REAL>  value;
            CS         unit;
};
```

The unit applies to both low and high boundary.

```
invariant(IVL<PQ> x) where x.nonNull {
  x.value.nonNull;
```



```
  x.low.value.equals(x.value.low);
  x.low.unit.equals(x.unit);
  x.lowClosed.equals(x.value.lowClosed);
  x.high.value.equals(x.value.high);
  x.high.unit.equals(x.unit);
  x.highClosed.equals(x.value.highClosed);
};
```

The special literal form is simply an interval of real numbers a space and the unit.

```
IVL<PQ>.literal ST {
  IVL<PQ>
  : IVL<REAL> " " unit        { $.value($1); $.unit.equals($3); }
  | IVL<REAL>                 { $.equals($1); };

  CS unit : ST                { $.value.equals($1);
                                $.codeSystem(2.16.840.1.113883.3.2); };
};
```

For example: "`[0;5] mmol/L`" or "`<20 mg/dL`" are valid literal forms of intervals of physical quantities. The generic interval form, e.g., "`[50 nm; 2 m]`" is also allowed.

3.4.3 Interval of Point in Time (IVL<TS>)

The generic interval data type defines the interval of points in time too. However, there are some special considerations about literal representations and conversions of intervals of point in time, which are specified in this section.

```
type Inteval<TS> alias IVL<TS> {
  literal    ST
  promotion  IVL<TS>    (TS x);
};
```

3.4.3.1 Promotion of TS to IVL<TS>

A TS can be promoted to an IVL<TS> whereby the low boundary is the TS value itself, and the width is inferred from the precision of the TS and the duration of the least significant calendar period specified. The high boundary is open. For example, the TS literal "`200009`" is converted to an IVL<TS> with low boundary `200009` and width 30 days, which is the interval "`[200009;200010[`".

3.4.3.2 Literal Form

The literal form for interval of point in time is exceptional.

- The "dash form" is not allowed for intervals of point in time
- A "hull form" is defined instead

In order to avoid syntactic conflicts with the timezone and slightly different usage profiles of the ISO 8601 that occur on some ITS platforms, **the dash form of the interval is not permitted for IVL<TS>**. The interval-form using square brackets is preferred.

Example: May 12, 1987 from 8 to 9:30 PM is "`[198705122000;198705122130]`".

> Note: The precision of a stated interval boundary is irrelevant for the interval. One might wrongly assume that the interval "`[19870901;19870930]`" stands for the entire September 1987 until end of the day of September 30. However, this is not so! The proper way to denote an entire calendar cycle (e.g., hour, day, month, year, etc.) in the interval notation with is to use an open high boundary. For example, all of September 1987 is denoted as "`[198709;198710[`".[49]

The "hull-form" of the literal is defined as the convex hull (cf. Section 3.4.1.9) of interval-promotions from two time stamps.

```
IVL<TS> hull : TS ".." TS; { $.equals(((IVL<TS>)$1).hull((IVL<TS>)$3)); };
```

For example, "`19870901..19870930`" is a valid literal using the hull form. The value is equivalent to the interval form "`[19870901;19871001[`".[50]

The hull-form further allows an abbreviation, where the higher timestamp literal does not need to repeat digits on the left that are the same as for the lower timestamp literal. The two timestamps are right-aligned and the digits to the left copied from the lower to the higher timestamp literal. This is a simple string operation and is not formally defined here.

Example: May 12, 1987 to May, 23, 1987 is "`19870512..23`". However, note that May 12, 1987 to June 2, 1987 is "`19870512..0602`", and not "`20000512..02`".

[49] This statement seems to directly contradict the ruling about the promotion of TS to IVL<TS>. However, there is no contradiction. The precision of a boundary does not have any relevance, but the precision of a simple timestamp (not as an interval boundary) is relevant, when that timestamp is promoted to an interval.

[50] The hull form appears superfluous for the simple interval all by itself. However, the hull form will become important for the periodic interval notation as it shortens the notation and (perhaps arguably) makes the notation of more complex timing structures more intuitive.

4 Generic Type Extensions

Generic type extensions are generic types with one parameter type, and that extend (specialize) their parameter type. In the formal data type definition language, generic type extensions follow the pattern: **template**<ANY T> **type** *GenericTypeExtensionName* **extends** T { ... }; These generic type extensions inherit most properties of their base type and add some specific feature to it. The generic type extension is a specialization of the base type, thus a value of the extension data type can be used instead of its base data type.

ITS Note: values of extended types can be substituted for their base type. However, an ITS may make some constraints as to what extensions to accommodate. Particularly, extensions need not be defined for those components carrying the values of data value properties. Thus, while any data value can be annotated outside the data type specification, and ITS may not provide for a way to annotate the value of a data value property.

At this time HL7 does not permit use of generic type extensions, except where explicitly enabled (in this or another HL7 specification) for such use cases where this advanced functionality is important.[51]

4.1 History (HIST) and History Item (HXIT)

This generic data type is used to collect an entire history of any other data value. A history is a non-empty set of data values that conform to the history item (HXIT) type, i.e., data values that have a valid-time property. The history information is not limited to the past; expected future values can also appear.

```
template<ANY T>
type History<T> alias HIST<T> extends SET<HXIT<T>> {
            HXIT<T>    earliest;
            HIST       exceptEarliest;
            HXIT<T>    latest;
            HIST       exceptLatest;
  demotion  HXIT<T>;
};
```

The earliest history item is the item in the set whose valid time's low boundary (validity start time) is less or equal (i.e. before) that of any other history item in the set. Likewise, the latest history item is the item in the set whose valid time's high boundary (validity end time) is greater or equal (i.e. after) that of any other history item in the set.

The semantics does not principally forbid the time intervals to overlap. However, if two history items have the same low (high) boundary in the valid time interval, it is undefined which one is considered the earliest (latest).

Except-earliest is the derived history that has the earliest item excluded. Except-latest is the derived history that has the latest item excluded.

```
invariant(HIST x) where x.nonNull {
  x.nonEmpty;
```

[51] This specification imposes a self-restraint upon itself to allow existing systems a graceful transition. However, the formal specification keeps the generic type extensions as substitutable for their base types. This self-restraint may be omitted in the future. New implementations are advised to accommodate some generalizable support for these generic data type extensions.


```
  forall(HXIT<T> e) where x.contains(e) {
    x.earliest.validTime.low.lessOrEqual(e.validTime.low);
    x.latest.validTime.high.greaterOrEqual(e.validTime.high);
  };

  x.exceptEarliest.equals(x.except(x.earliest));
  x.exceptLatest.equals(x.except(x.latest));

  ((T)x).equals(x.latest);
};
```

A type conversion exists between an entire history HIST<T> and a single history item HXIT<T>. This conversion takes the latest data from the history. The purpose of this conversion is to allow an information producer to produce a history of any value instead of sending just one value. An information-consumer, who does not expect a history but a simple value, will convert the history to the latest value.

Note from the definition of history item (HXIT) below, that HXIT<T> semantically extends T. This means, that the information-consumer expecting a T but given an HXIT<T> will not recognize any difference (substitutability of specializations.)

ITS Note: the order of history items in the lists should be backwards in time.

4.1.1 History Item (HXIT)

This generic data type extension tags a time range to its base data value. The time range is the time in which that data was, is, or is expected to be valid. If the base type T does not possess a valid time property, the HXIT<T> adds that property to the base type. If, however, the base type T does have a valid time property, that property can be mapped to the valid time property of the HXIT<T>.[52]

```
template<ANY T>
type HistoryItem<T> alias HXIT<T> extends T {
            IVL<TS>    validTime;
};
```

4.1.1.1 validTime : IVL <TS>

The time interval during which the given information was, is, or is expected to be valid. The interval can be open or closed infinite or undefined on either side.

4.2 Uncertain Value – Probabilistic (UVP)

This is a generic data type extension to specify one uncertain value tagged with a probability. The probability expresses the information producer's belief that the given value holds. How the probability number was arrived at is outside the scope of this specification.

[52] Note that data types are specifications of abstract properties of values. This specification does not mandate how these values are represented in an ITS or implemented in an application. Specifically, it does not mandate how the represented components are named or positioned. In addition, the semantic generalization hierarchy may be different from a class hierarchy chosen for implementation (if the implementation technology has inheritance.) Keep the distinction between a type (interface) and an implementation (concrete data structure, class) in mind. The ITS must contain a mapping of ITS defined features of any data type to the semantic properties defined here.

Probabilities are subjective and (as any pieces of data) apply in a context. The context of any data item is the data structure in which that item appears. While the context dependence is important for any information, it is critical to understand the context dependency of probabilities: when new information is found the probability might change. Thus, for any message (document, or other information representation) the information – and particularly the probabilities – reflect what the information producer believed was appropriate at the given time and for the given purpose for which the message (document) was created.

Since probabilities are subjective measures of belief, they can be stated without being "correct" or "incorrect" *per se*, let alone "precise" or "imprecise". Notably, one does not have to entertain experiments to measure a frequency of some outcome in order to specify a probability. In fact, whenever statements about individual people or events are made, it is not possible to confirm such probabilities with "frequentists" experiments.

```
template<ANY T>
type UncertainValueProbabilistic<T> alias UVP<T> extends T {
            REAL       probability;
};
```

The type T is not formally constrained. In theory, discrete probabilities can only be stated for discrete data values. Thus, generally UVP<REAL> and UVP<PQ> values should not be stated. However, by definition a discrete value set is one that is finite or countably infinite, and abiding by this definition any measured value or real number recorded with digits is discrete. Thus, the distinction between discrete and continuous values is not practical for our purpose. Indeed, even though integer numbers are discrete (countably infinite) estimating a single integer number and tagging it with a probability is not reasonable. Most textbook on statistics treat estimations of integers or ordinals as real numbers when defining the estimated value of a random sample X as the sum of $x_i \cdot p(x_i)$ over all $x_i \in X$.

4.2.1.1 probability : REAL

This is the probability assigned to the value. The probability is a real number between 0 and 1. If the probability is unstated (NULL), an UVP<T> is indistinguishable from a simple data value T.

```
invariant(UVP<T> x) where x.nonNull.and(x.probability.nonNull) {
  ((IVL<REAL>)[0;1]).contains(x.probability);
};
```

There is no "default probability" that one can assume when the probability is unstated. Therefore, it is impossible to make any semantic difference between an UVP<T> without probability and a simple T. UVP<T> does not mean "uncertain", and a simple T does not mean "certain". In fact, the probability of the UVP<T> could be 0.999 or 1, which is quite certain, where a simple T value could be a very vague guess.

4.3 Non-Parametric Probability Distribution (NPPD)

This is a generic data type to specify a value as a non-empty set of uncertain values forming a probability distribution (histogram.) All the elements in the set are considered alternatives and are rated each with its probability expressing the belief (or frequency) that each given value holds.

The purpose of the non-parametric probability distribution is chiefly to support statistical data reporting as it occurs in measurements taken from many subjects and consolidated in a histogram. This occurs in epidemiology, verterinary medicine, laboratory medicine, but also in cost controlling and business process engineering.

Semantically, the information of a stated value exists in contrast to the complement set of unstated possible values. Thus, semantically, a non-parametric probability distribution contains *all* possible values and assigns probabilities to each of them.

ITS Note: even though semantically the NPPD assigns probabilities to all possible values, not all values need to be represented explicitly. Those possible values that are not mentioned in a NPPD data structure will have the rest-probability distributed equally over all unmentioned values. For example, if the value set is {A; B; C; D} but the NPPD value states just {(B; 0.5); (C; 0.25)} then the rest-probability is 1 – 0.75 = 0.25 which is distributed evenly over the complement set: {(A; 0.125); (D; 0.125)}. Semantically, the NPPD is the union of the stated probability distribution and the unstated complement with rest-probability distributed evenly.

```
template<ANY T>
type NonParametricProbabilityDistribution⟨T⟩
    alias NPPD<T> extends SET<UDP<T>> {
            SET<UDP<T>> mostLikely(INT n);
};
```

Just as with UVP, the type T is not formally constrained, even though there are reasonable and unreasonable use cases. Typically one would use the non-parametric probability distributions for unordered types, if only a "small" set of possible values is assigned explicit probabilities, or if the probability distribution cannot (or should not) be approximated with parametric methods. For other cases, one may prefer parametric probability distributions.

```
invariant(NPPD<T> x) where x.nonNull {
  x.nonEmpty;

  x.contains(x.mostLikely(n));
  x.mostLikely(n).
  forall(UVP<T> d, e; SET<UVP<T>> m; INT n)
      where x.contains(d)
       .and(m.equals(x.mostLikely(n)))
       .and(m.contains(e)) {
    e.greaterOrEqual(d).or(m.contains(d));
  };
};
```

4.4 Parametric Probability Distribution (PPD)

A parametric probability distribution is a generic data type extension specifying an uncertain value of a quantity data type using a distribution function and its parameters. Aside from the specific parameters of the distribution, a mean (expected value) and standard deviation is always given to help maintain interoperability if receiving applications can not deal with a certain probability distribution.

```
template<QTY T>
type ParametricProbabilityDistribution<T> alias PPD<T> extends T {
            T.diff     standardDeviation;
            CS         type;
```



```
            IVL<T>     confidenceInterval(REAL p);
            REAL       probability(IVL<T> x);

            PPD<T>     times(REAL x);
};
```

Since a PPD<T> extends the base type T, a simple T value is the mean (expected value or first moment) of the probability distribution. Applications that can not deal with distributions will take the simple T value neglecting the uncertainty. That simple value of type T is also used to standardize the data for computing the distribution.

Probability distributions are defined over integer or real numbers and normalized to a certain reference point (typically zero) and reference unit (e.g., standard deviation = 1). When other quantities defined in this specification are used as base types, the mean and the standard deviation are used to scale the probability distribution. For example, if a PPD<PQ> for a length is given with mean 20 ft and a standard deviation of 2 in, the normalized distribution function $f(x)$ that maps a real number x to a probability density would be translated to $f'(x')$ that maps a length x' to a probability density as $f'(x') = f((x' - m) / s)$.

Where applicable, the PPD specification conforms to the ISO *Guide to the Expression of Uncertainty in Measurement* (GUM) as reflected by NIST Technical Note 1297, *Guidelines for Evaluating and Expressing the Uncertainty of NIST Measurement Results*. The PPD specification does not describe how uncertainty is to be evaluated but only how it is expressed. The concept of "standard uncertainty" as set forth by the ISO GUM corresponds to the "standard deviation" property of the PPD.

4.4.1.1 standardDeviation : T.diff

The standard deviation of the probability distribution. The standard deviation is used to normalize the data for computing the distribution function. Applications that can not deal with probability distributions can still get an idea about the confidence level by looking at the standard deviation.

The standard deviation of a probability distribution over a type T is of a related type T.diff that can express differences between values of type T. If T is REAL or INT, T.diff is also REAL or INT respectively. However if T is a point in time (TS), T.diff is a physical quantity (PQ) in the dimension of time.

The standard deviation is what ISO GUM calls "standard uncertainty."

4.4.1.2 type : CS

This code specifies the type of probability distribution. Possible values are as shown in the attached table. The NULL value (unknown) for the type code indicates that the probability distribution type is unknown. In that case, the standard deviation has the meaning of an informal guess.

Table 38 lists the defined probability distributions. Many distribution types are defined in terms of special parameters (e.g., the parameters α and β for the γ-distribution, number of degrees of freedom for the t-distribution, etc.) For all distribution types, however, the mean and standard deviation are defined. The PPD data type is specified with the parameters mean and standard distribution only. The definition column in Table 38 contains the relationship between the special parameters and the mean μ and standard deviation σ.

ITS Note: an ITS does not need to represent any of the specialized parameters for the distribution types. As it turns out, all of these specialized parameters can be calculated from the mean and standard deviation.

Table 38: Probability Distribution Types

Name	Code	Definition
unknown	(NULL)	Used to indicate that the mean is estimated without any closer consideration of its probability distribution. In this case, the meaning of the standard deviation is not crisply defined. However, interpretation should be along the lines of the normal distribution, e.g., the interval covered by the mean ±1 standard deviation should be at the level of about two thirds confidence.
uniform	U	The uniform distribution assigns a constant probability over the entire interval of possible outcomes, while all outcomes outside this interval are assumed to have zero probability. The width of this interval is $2\sigma\sqrt{3}$. Thus, the uniform distribution assigns the probability densities $f(x) = (2\sigma\sqrt{3})^{-1}$ to values $\mu - \sigma\sqrt{3} \geq x \leq \mu + \sigma\sqrt{3}$ and $f(x) = 0$ otherwise.
normal (Gaussian)	N	This is the well-known bell-shaped normal distribution. Because of the central limit theorem, the normal distribution is the distribution of choice for an unbounded random variable that is an outcome of a combination of many stochastic processes. Even for values bounded on a single side (i.e. greater than 0) the normal distribution may be accurate enough if the mean is "far away" from the bound of the scale measured in terms of standard deviations.
log-normal	LN	The logarithmic normal distribution is used to transform skewed random variable X into a normally distributed random variable $U = \log X$. The log-normal distribution can be specified with the properties mean μ and standard deviation σ. Note however that mean μ and standard deviation σ are the parameters of the raw value distribution, not the transformed parameters of the lognormal distribution that are conventionally referred to by the same letters. Those log-normal parameters μ_{log} and σ_{log} relate to the mean μ and standard deviation σ of the data value through $\sigma_{log}^2 = \log(\sigma^2/\mu^2 + 1)$ and $\mu_{log} = \log \mu - \sigma_{log}^2/2$.
γ (gamma)	G	The gamma-distribution used for data that is skewed and bounded to the right, i.e. where the maximum of the distribution curve is located near the origin. The γ-distribution has a two parameters α and β. The relationship to mean μ and variance σ^2 is $\mu = \alpha\beta$ and $\sigma^2 = \alpha\beta^2$.
exponential	E	Used for data that describes extinction. The exponential distribution is a special form of γ-distribution where $\alpha = 1$, hence, the relationship to mean μ and variance σ^2 are $\mu = \beta$ and $\sigma^2 = \beta^2$.
χ^2 (chi square)	X2	Used to describe the sum of squares of random variables which occurs when a variance is estimated (rather than presumed) from the sample. The only parameter of the χ^2-distribution is ν, so called the *number of degrees of freedom* (which is the number of independent parts in the sum). The χ^2-distribution is a special type of γ-distribution with parameter $\alpha = \nu/2$ and $\beta = 2$. Hence, $\mu = \nu$ and $\sigma^2 = 2\nu$.
t (Student)	T	Used to describe the quotient of a normal random variable and the square root of a χ^2 random variable. The t-distribution has one parameter ν, the degrees of freedom. The relationship to mean μ and variance σ^2 are: $\mu = 0$ and $\sigma^2 = \nu / (\nu - 2)$
F	F	Used to describe the quotient of two χ^2 random variables. The F-distribution has two parameters ν_1 and ν_2, which are the numbers of degrees of freedom of the numerator and denominator variable respectively. The relationship to mean μ and variance σ^2 are: $\mu = \nu_2 / (\nu_2 - 2)$ and $\sigma^2 = (2\nu_2(\nu_2 + \nu_1 - 2)) / (\nu_1(\nu_2 - 2)^2(\nu_2 - 4))$.
β (beta)	B	The beta-distribution is used for data that is bounded on both sides and may or may not be skewed (e.g., occurs when probabilities are estimated.) Two parameters α and β are available to adjust the curve. The mean μ and variance σ^2 relate as follows: $\mu = \alpha / (\alpha + \beta)$ and $\sigma^2 = \alpha\beta / ((\alpha + \beta)^2(\alpha + \beta + 1))$.

The three distribution-types *unknown* (NULL), *uniform* and *normal* must be supported by every system that claims to support PPD. All other distribution types are optional. When a system interpreting a PPD representation encounters an unknown distribution type, it maps this type to the unknown (NULL) distribution-type.

4.4.1.3 Literal Form

The parametric probability distribution has a literal form. The general syntax is as follows:


```
PPD<T>.literal ST {
  PPD<T> : T "(" type T.diff ")"      { ((T)$).equals($1);
                                        $.type.equals($3);
                                        $.standardDeviation.equals($4); };

  CV type : ST                        { $.value.equals($1);
                                        $.system.equals(); };
};
```

Examples: an example for a PPD<REAL> is "`1.23(N0.005)`" for a normal distribution of a real number around 1.23 with a standard deviation of 0.005. An example for a PPD<PQ> is "`1.23 m (5 mm)`" for a distribution of unknown type around the length 1.23 meter with a standard deviation of 5 millimeter. An example for a PPD<TS> is "`2000041113(U4 h)`" for a uniform distribution around April 11, 2000 at 1pm with standard deviation of 4 hours.

4.4.2 Probability Distribution over Real Numbers (PPD<REAL>)

The parametric probability distribution of real numbers is fully defined by the generic data type.

```
type ParametricProbabilityDistribution⟨REAL⟩ alias PPD<REAL>;
```

However, there are some special considerations about literal representations and conversions of probability distributions over real numbers, which are specified in this section.

4.4.2.1 Converting a real number (REAL) to an uncertain real number (PPD<REAL>)

When converting a REAL into a PPD<REAL>, the standard deviation is calculated from the REAL value's order of magnitude and precision (number of significant digits). Let x be a real number with precision n. We can determine the order of magnitude e of x as $e = \log_{10} |x|$ where e is rounded to the next integer that is closer to zero (special case: if x is zero, e is zero.) The value of least significant digit l is then $l = 10^{e-n}$ and the standard deviation σ is $\sigma = l / 2$.

Table 39: Examples of standard deviations computed from precision p and order of magnitude e

Representation	x	e	p	$e - p + 1$	l	σ
0	0	(0)	1	0	1	0.5
1	1	0	1	0	1	0.5
2	2	0	1	0	1	0.5
9	9	0	1	0	1	0.5
10	10	1	2	0	1	0.5
100	100	2	3	0	1	0.5
1e+1	10	1	1	1	10	5
1e+2	100	2	1	2	100	50
10e+1	100	2	2	1	10	5
1.1	1.1	0	2	-1	0.1	0.05
10.1	10.1	1	3	-1	0.1	0.05
1.1e+2	110	2	2	1	10	5
1.1e-2	0.011	-2	2	-3	0.001	0.0005
1.1e-4	0.00011	-4	2	-5	0.00001	0.000005
10.1e-4	0.00101	-3	3	-5	0.00001	0.000005
0.1e-1	0.01	-2	1	-2	0.01	0.005
0.01e-1	0.001	-3	1	-3	0.001	0.0005
0.01e-2	0.0001	-4	1	-4	0.0001	0.00005
0.00	0	(0)	3	-2	0.01	0.005

4.4.2.2 Concise Literal Form for PPD<REAL>

Besides the generic literal form of the PPD defined in Section 4.4.1.3, a concise literal form is defined for PPD over real numbers. This concise literal form is defined such that the standard deviation can be expressed in terms of the least significant digit in the mantissa. This literal is defined as an extension of the REAL literal:

```
PPD<REAL>.literal ST {
  PPD<REAL> mantissa
  : REAL.mantissa "(" type T.diff ")" { ((T)$).equals($1);
                                        $.type.equals($3);
                                        $.standardDeviation.equals($4); }
  | REAL.mantissa                     { $.equals($1);
                                        $.type.equals($3);
       $.standardDeviation.equals($1.leastSignificantDigit.times(0.5)); };

  CS type : ST                        { $.value.equals($1);
       $.system.equals(2.16.840.1.113883.5.1019); };
};
```

Examples: "`1.23e-3 (U5e-6)`" is a the unifom distribution around 1.23×10^{-3} with 5×10^{-6} standard deviation in generic literal form. "`1.230(U5)e-3`" is the same value in concise literal form.

4.4.3 Parametric Probability Distributions over Physical Quantities (PPD<PQ>)

A parametric probability distribution over physical quantities is constructed from the generic PPD type. However, recognizing that the unit can be factored from the boundaries, we add additional semantics and a separate literal form. The additional view of a probability distribution over physical quantities is a probability distribution over real numbers with one unit.

```
type ParametricProbabilityDistribution<PQ> alias PPD<PQ> {
            PPD<REAL>  value;
            CS         unit;
};
```

The unit applies to both mean and standard deviation.

```
invariant(PPD<PQ> x) where x.nonNull {
  x.value.nonNull;

  ((REAL)x.value).equals(((PQ)x).value);
  x.unit.equals(((PQ)x).unit);
  x.value.standardDeviation.equals(x.standardDeviation.value);
  x.standardDeviation.unit.equals(x.unit);
};
```


4.4.3.1 Concise Literal Form for PPD<PQ>

A concise literal form for probability distributions of physical quantities is defined based on the concise literal form of PPD<REAL> (cf. Section 4.4.2.2) where REAL is the value. This literal is defined as an extension of the PQ literal.

```
PPD<PQ>.literal ST {
  PPD<PQ> : PPD<REAL> " " unit        { $.value.equals($1);
                                        $.unit.equals($3); }
};
```

Examples: "`1.23e-3 m (N5e-6 m)`" is the normal-distributed length of 1.23×10^{-3} m with 5×10^{-6} m standard deviation in generic literal form. "`1.230(N5)e-3 m`" is the same value in concise literal form. "`1.23e-3(N0.005e-3) m`" is also valid; it is the concise literal form for PPD<PQ> combined with the generic literal form for PPD⟨REAL⟩.

4.4.4 Probability Distribution over Time Points (PPD<TS>)

The parametric probability distribution over time points is fully defined by the generic data type.

```
type ParametricProbabilityDistribution<TS> alias PPD<TS>;
```

The standard deviation is of type TS.diff, which is a duration (a physical quantity in the dimension of time.)

4.4.4.1 Converting a point in time (TS) to an uncertain point in time (PPD<TS>)

When converting a TS into a PPD<TS>, the standard deviation is calculated from the TS value's order of magnitude and precision (number of significant digits) such that two standard deviations span the maximal time range of the digits not specified. For example, in 20000609 the unspecified digits are hour of the day and lower. All these digits together span a duration of 24 hours, and thus, the standard deviation s is s = 12 h from 20000609000000.0000… up to 20000609999999.9999… (= 20000610)

This rule is different from real numbers in that the range of uncertainty lies above the time value specified. This is to go with the common sense judgment that June 9th spans all day of June 9th with noon as the center, not midnight.

5 Timing Specification

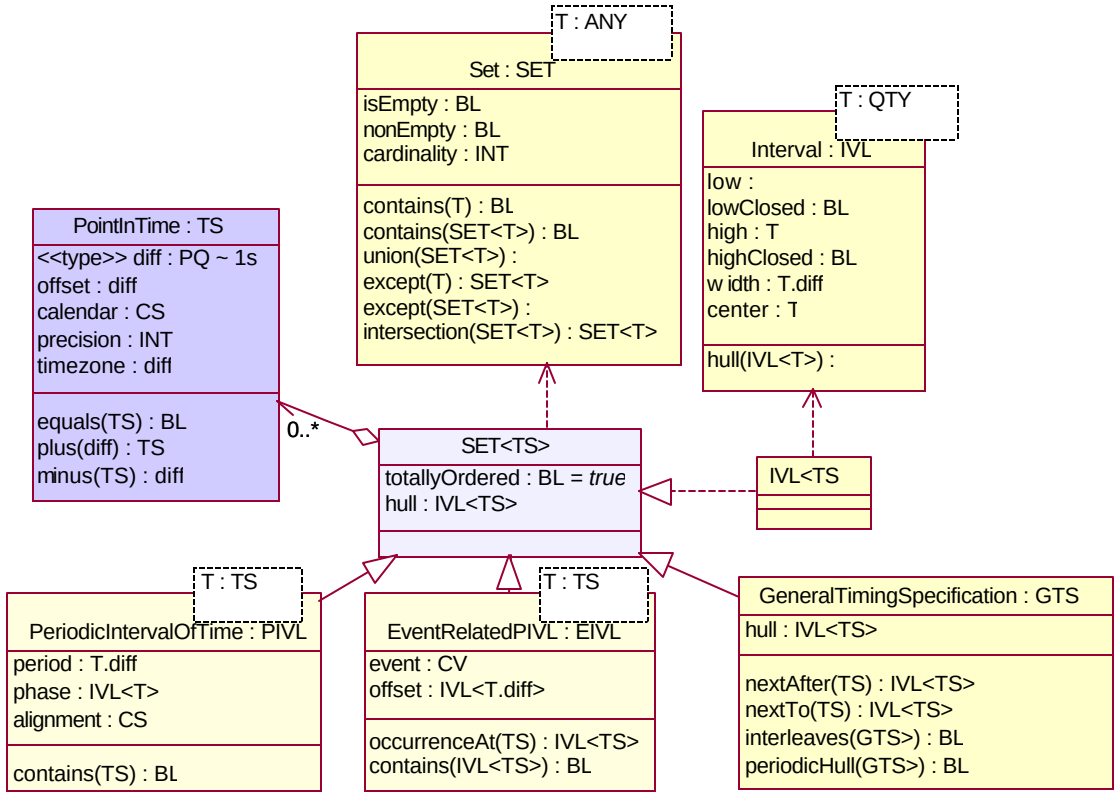


Figure 13: Overview of Timing Specification Data Types

The timing specification suite of generic data types is used to specify the complex timing of events and actions such as they occur in order management and scheduling systems. It also supports the cyclical validity patterns that may exist for certain kinds of information, such as phone numbers (evening, daytime), addresses (so called "snowbirds," residing in the south during winter and north during summer) and office hours.

The timing specification data types include point in time (TS) and the interval of time (IVL<TS>), and add to it other kinds of collection types that are specifically suited to specify repeated schedules. These additional collections include periodic interval, event-related periodic interval, and finally the generic timing specification types itself. All these timing types are semantically sets of time points SET<TS>, describing the time distribution of repeating states or events.

5.1 Periodic Interval of Time (PIVL)

The periodic interval of time specifies an interval of time that recurs periodically. Periodic intervals have two properties, phase and period. The phase specifies the interval prototype that is repeated every period.

Table 40: Summary of Primary Properties of Periodic Interval of Time (PIVL<TS>)

Name	Type	Status	Default	Constraint	Definition
phase	IVL<TS>	mandatory			A prototype of the repeating interval, may anchor the periodic interval sequence at a certain point in time.
period	PQ	mandatory			A time duration specifying the

	(~1 s)			frequency at which the periodic interval repeats.
alignment	CS	optional		Specifies an alignment of the repetition to a calendar (e.g., to distinguish every 30 days from "the 5th of every month".)
institutionSpecifiedTime	BL	optional	*false*	Indicates whether the exact timing is up to the party executing the schedule (e.g., to distinguish "every 8 hours" from "3 times a day".)

For example, "every eight hours for two minutes" is a periodic interval where the interval's width equals two minutes and the period at which the interval recurs equals eight hours.

The phase also marks the anchor point in time for the entire series of periodically recurring intervals. The recurrence of a periodic interval has no beginning or ending, but is infinite in both future and past.

```
template<TS T>
protected type PeriodicInterval<T> alias PIVL<T> extends SET<T> {
            T.diff     period;
            IVL<T>     phase;
            CS         alignment;
            BL         institutionSpecifiedTime;

            BL         contains(TS);
  literal   ST;
};
```

A periodic interval is fully specified when both the period and the phase are fully specified. The interval may be only partially specified where either only the width or only one boundary is specified.

For example: "every eight hours for two minutes" specifies only the period and the phase's width but no boundary of the phase. Conversely, "every eight hours starting at 4 o'clock" specifies only the period and the phase's low boundary but not the phase's high boundary. "Every eight hours for two minutes starting at 4 o'clock" is fully specified since the period, and both the phase's low boundary and width are specified (low boundary and width implies the high boundary.)

The periodic interval of time is a generic data type PIVL<T> where the type parameter T is restricted to the point in time (TS) data type and it's extensions. The parametric probability distribution of point in time (PPD<TS>) is an extension of point in time and therefore can be used to form periodic intervals of probability distributions of point in time (PIVL<PPD<TS>>) values (uncertain periodic interval.)

Oftentimes repeating schedules are only approximately specified. For instance "three times a day for ten minutes each" does not usually mean a period of precisely 8 hours and does often not mean exactly 10 minutes intervals. Rather the distance between each occurrence may vary as much as between 3 and 12 hours and the width of the interval may be less than 5 minutes or more than 15 minutes. An uncertain periodic interval can be used to indicate how much leeway is allowed or how "timing-critical" the specification is.

5.1.1.1 Period : T.diff

The period specifies how frequently the periodic interval recurs. The period is a physical quantity in the dimension of time (TS.diff.) For an uncertain periodic interval (PIVL<PPD<TS>>) the period is a probability distribution over elapsed time (PPD<PQ>). A non-NULL period exists for every non-NULL periodic interval.

```
invariant(PIVL<T> x) where x.nonNull {
  x.period.nonNull;
};
```

5.1.1.2 Phase : IVL<TS>

The phase specifies the interval prototype that is repeated every period. The phase also marks the anchor point in time for the entire series of periodically recurring intervals. The recurrence of a periodic interval has no begin or end but is infinite in both future and past. A phase must be specified for every non-NULL periodic interval. The width of the phase must be less or equal the period.

```
invariant (PIVL<T> x) where x.nonNull {
  x.phase.nonNull;
  x.phase.width.lessOrEqual(x.period);
};
```

5.1.1.3 Alignment : CS

A periodic interval may be specified aligned to the calendar underlying the phase. A non-aligned periodic interval recurs independently from the calendar. An aligned periodic interval is synchronized with the calendar.

The domain of this code is the calendar cycle code.

For example, "every 5th of the month" is a calendar aligned periodic interval. The period spans 28 to 31 days depending on the calendar month. Conversely, "every 30 days" is an independent period that will fall on a different date each month.

The calendar alignment specifies a calendar cycle to which the periodic interval is aligned. The even flow of time will then be partitioned by the calendar cycle. The partitioning is called the calendar "grid" generated by the aligned-to calendar cycle. The boundaries of each occurrence interval will then have equal distance from the earliest point in each partition. In other words, the distance from the next lower grid-line to the beginning of the interval is constant.

For example, with "every 5th of the month" the alignment calendar cycle would be month of the year (MY.) The even flow of time is partitioned in months of the year. The distance between the beginning of each month and the beginning of its occurrence interval is 4 days (4 days because day of month (DM) starts counting with 1.) Thus, as months differ in their number of days, the distances between the recurring intervals will vary slightly, so that the interval occurs always on the 5th.

5.1.1.4 Institution Specified Time

A Boolean property indicating whether the exact timing is up to the person or organization executing the time plan. For example, with a schedule "three times a day" the average time between repetitions is 8 hours, however, with institution specified time indicator true, the timing could follow some rule made by the executing person or organization ("institution"), that. e.g., three times a day schedules are executed at 7 am, noon, and 7 pm.

5.1.1.5 Periodic Intervals as Sets

The essential property of a set is that it contains elements. For non-aligned periodic intervals, the *contains*-property is defined as follows. A point in time *t* is contained in the periodic interval of time if and only if there is an integer *i* for which *t* plus the period times *i* is an element of the phase interval.


```
invariant (PIVL<TS> x, TS t) where x.nonNull.and(x.alignment.isNull) {
  x.contains(t).equals(exists(INT i) {
       x.phase.contains(t.plus(x.period.times(i)));
     });
};
```

For calendar-aligned periodic intervals the *contains* property is defined using the calendar-cycle's sum(*t*, *n*) property that adds *n* such calendar cycles to the time *t*.

```
invariant (PIVL<TS> x, TS t, CalendarCycle c)
    where x.nonNull.and(c.equals(x.alignment)) {
  x.contains(t).equals(exists(INT i) {
        x.phase.contains(c.sum(t, i));
     });
};
```

5.1.1.6 Literal Form

Generic Literal Form. The generic literal form for periodic intervals of time is as follows:

⟨*phase* : IVL<T>⟩ **/** ⟨*period* : T.diff⟩ [**@** ⟨*alignment*⟩] [**IST**].

```
PIVL<T>.literal ST {
  PIVL<T>
  : S2                          { $.equals($1); }
  | S2 "IST"                   { $.phase.equals($1.phase);
                                  $.period.equals($1.period);
                                  $.institutionSpecified.equals(true); };

  PIVL<T> S2
  : S1                          { $.equals($1); }
  | S1 "@" "(" PQ ")"          { $.phase.equals($1.phase);
                                  $.period.equals($1.period);
                                  $.alignment.equals($4); };

  PIVL<T> S1
  : IVL<T> "/" T.diff          { $.phase.equals($1);
                                  $.period.equals($3); }
  |        "/" T.diff          { $.period.equals($2); };
};
```

For example, "[200004181100;200004181110]/(7 d)@DW" specifies every Tuesday from 11:00 to 11:10 AM. Conversely, "[200004181100;200004181110]/(1 mo)@DM" specifies every 18th of the month 11:00 to 11:10 AM.

See Table 36 for calendar-period codes defined for the Gregorian calendar. There are 1-character and 2-character symbols. The 2-character symbols are preferred for the alignment period identifier.

Calendar Pattern Form. This form is used to specify calendar-aligned timing more intuitively using "calendar patterns." The calendar pattern syntax is (semi-formally) defined as follows:

⟨*anchor*⟩ [⟨*calendar digits*⟩ [**..** ⟨*calendar digits*⟩]] **/** ⟨*number* : INT⟩ [**IST**]

A calendar pattern is a calendar date where the higher significant digits (e.g., year and month) are omitted. In order to interpret the digits, a period identifier is prefixed that identifies the calendar period of the left-most digits. This calendar period identifier *anchors* the calendar digits following to the right.

See Table 36 for calendar-period codes defined for the Gregorian calendar. There are 1-character and 2-character symbols. The 1-character symbols are preferred for the calendar pattern anchor.

For example: "`M0219`" is February 19 the entire day every year. This periodic interval has the February 19 of any year as its phase (e.g., "`[19690219;19690220[`"), a period of one year, and alignment month of the year (M). The alignment calendar-cycle is the same as the anchor (e.g., in this example, month of the year.)

The calendar digits may also omit digits on the right. When digits are omitted on the right, this means the interval from lowest to highest for these digits. For example, "`M0219`" is February 19 the entire day; "`M021918`" is February 19, the entire hour between 6 and 7 PM.

In absence of a formal definition for this, the rules for parsing a calendar pattern are as follows (example is "`M021918..21`")

1. Read the anchoring period identifier (e.g. "`M`")
    a. the PIVL's alignment is equal to this calendar period (e.g. month of the year)
    b. use the current point in time and format a literal exact to the next higher significant calendar period from the anchoring calendar period (e.g. year, "`2000`", constructing "`2000021918`"), this is the "stem literal"
2. Read this constructed literal (e.g., "`2000021918`") into a TS value and convert that value to an IVL<TS> according to Section 3.4.3.1 (e.g., "`[2000021918;2000021919[`") this is the "low interval."
3. If the hull-operator token "`..`" follows, read the following calendar digits (e.g., "`21`")
    a. Right-align the stem literal and the calendar digits just read

        "`2000021918`"
        "`  21`"

    b. and copy all digits from the stem literal that are missing to the left of the calendar digits just read (e.g., yields "`2000021921`".)
    c. Read this constructed literal (e.g., "`2000021918`") into a TS value and convert that value to an IVL<TS> according to Section 3.4.3.1 (e.g., "`[2000021921;2000021922[`") this is the "high interval."
    d. The phase interval is the convex hull of the low interval and the high interval (e.g., "`[2000021918;2000021922[`").
4. If the hull-operator was not present, the phase is simply the low interval.

Interleave. A calendar pattern followed by a slash and an integer number *n* indicates that the given calendar pattern is to apply every n^{th} time.

For example: "`D19/2`" is the 19^{th} of every second month.

A calendar pattern expression is evaluated at the time the pattern is first enacted. At this time, the calendar digits missing from the left are completed using the earliest date matching the pattern (and following a preceding pattern in a combination of time sets).

For example: "D19/2" is the 19th of every second month. If this expression is evaluated on March 14, 2000 the phase is completed to: "[20000319;20000320[/(2 mo)@DM" and thus the two-months cycle begins with March 19, followed by May 19, etc. If the expression were evaluated by March 20, the cycle would begin at April 19, followed by June 19, etc.

If no calendar digits follow after the calendar period identifier, the pattern matches any date. The integer number following the slash indicates the length of the cycle. The phase interval in these cases has only the width specified to be the duration of the anchoring calendar-cycle (e.g., in this example 1 day.)

For example: "CD/2" is every other day, "H/8" is every 8th hour, for the duration of one hour.

Institution Specified Time. Both, a generic periodic interval literal and a calendar pattern may be followed by the three letters "IST" to indicate that within the larger calendar cycle (e.g., for "hour of the day" the larger calendar cycle is "day") the repeating events are to be appointed at institution specified times. This is used to specify such schedules as "three times a day" where the periods between two subsequent events may vary well between 4 hours (between breakfast and lunch) and 10 hours (over night.)

Table 41: Examples for literal expressions for periodic intervals of time

Generic Form	Calendar Pattern Form	Description
[198709;198710[/(1 a)@MY	M09	September, the entire month, every year (note that in the year 1987 in the generic form is irrelevant since the periodic interval recurs every year past and future.)
[19870915;19870916[/(1 a)@DM	M0915	September 15, the entire day, every year
[1987091516;1987091517[/(1 a)@DM	M091516	September 15 at 4 PM, the entire hour, every year
[198709151630;198709151710]/(1 a)@DM	M09151630..1710	September 15 at 4:30 5:10 PM, every year
[1987091516;[/(1 a)@DM		September 15 at 4 PM, end time explicitly unknown, every year
[198709151630;198709151631[/(1 a)@DM	M09151630	September 15 at 4:30 PM, the entire minute, every year
[1987091516;1987091517[/(1 mo)@DM	D1516..17	every 15th day of the month at 4 to 5 PM
[1987091516;1987091517[/(1 mo)		September 15, 1987 from 4 to 5 PM and then every 730.5 hours continuously (this example has little practical value beyond comparing the unaligned with the aligned form in the preceding row.)
[1987091516;1987091517[/(1 mo)@HD		September 15, 1987 from 4 to 5 PM and then every 30.4375 days, but aligned to the hour of the day.
[1 mo]/(2 mo)@MY	M/2	every other month of the year; (Jan, Mar, …) vs. (Feb, Apr, …) is undefined
[198701;197502[/(2 mo)@MY	M01..12/2	every other month of the year, Jan, Mar, …
[198702;197503[/(2 mo)@MY	M02..12/2	every other month of the year, Feb, Apr, …
[19870401;19870930[/(1 a)@DM	M04..09	April 1until (and including) September 30
19870401-0930/(1 a)@DM	M0401..0930	April 1 to September 30 (the generic form uses the dash-form for the phase interval)
[20001202;20001203[/(1 wk)@DW	J6	every Saturday
[20001202;20001203[/(2 wk)@DW	J6/2	every other Saturday
[20001202;20001203[/(3 wk)@DW	J6/3	every third Saturday
[1 d]/(2 d)@DW	J/2	every other day of the week; (Mon, Wed, Fri, …) vs. (Tue, Thu, Sat, …) is undefined
[20001204;20001205[/(2 d)@DW	J2..6/2	every other day of the week (Tue, Thu, Sat, Tue, Thu, Sat, …)
[20001204;20001205[/(2 d)	D/2	every other day (Tue, Thu, Sat, Mon, Wed, Fri, Sun, Tue, …)
[19870601;19870606[/(1 wk)@DW	J1..5	Monday to Friday every week
[19870601;19870608[/(2 wk)	W/2	every other week (continuous)
[19870101;19870105[/(2 wk)@WY	WY/2	every other week of the year (a blunt example on the impact of the calendar alignment: the phase interval spans only 4 days and yet it represents an entire week in the calendar alignment “week of the year”.)
[19870406;19870413[/(1 a)@WY	WY15	the 15th calendar week of every year
[19870105;19870112[/(1 mo)@WM	WM2	the second week of the month, every month
[19870508;19870509[/(1 a)@DY	DY128	the 128th day of the year, every year
[10 min]/(2 d)		every other day for 10 minutes (only width of repeating interval is known)
[1 h]/(8 h)	H/8	every eighth hour (each time a 60 minutes interval)
[1 h]/(8 h) IST	H/8 IST	three times a day at institution specified times (each time a 60 minutes interval)
/(8 h) IST		three times a day at institution specified times. Nothing about the repeating interval is known i.e., this includes only a period (frequency), while the phase is left undefined

5.2 Event-Related Periodic Interval of Time (EIVL)

The event-related periodic interval of time allows specifying a periodic interval of time based on activities of daily living, important events that are time-related but not fully determined by time.

For example, "one hour after breakfast" specifies the beginning of the interval at one hour after breakfast is finished. Breakfast is assumed to occur before lunch but is not determined to occur at any specific time.

```
template<TS T>
protected type EventRelatedPeriodicInterval<T> alias EIVL<T> extends SET<T>{
            CV          event;
            IVL<T.diff> offset;

            IVL<T>      occurrenceAt(TS eventTime);
            BL          contains(TS);
  literal   ST;
};
```

5.2.1.1 Event : CV

A code for a common (periodical) activity of daily living based on which the event related periodic interval is specified. Such events qualify for being adopted in the domain of this attribute for which all of the following is true:

- the event commonly occurs on a regular basis,
- the event is being used for timing activities, and
- the event is not entirely determined by time.

Table 42: Event Codes for Event-Related Periods

Code	Definition
HS	the hour of sleep (e.g., H18-22)
AC	before meal (from lat. *ante cibus*)
PC	after meal (from lat. *post cibus*)
IC	between meals (from lat. *inter cibus*)
ACM	before breakfast (from lat. *ante cibus matutinus*)
ACD	before lunch (from lat. *ante cibus diurnus*)
ACV	before dinner (from lat. *ante cibus vespertinus*)
PCM	after breakfast (from lat. *post cibus matutinus*)
PCD	after lunch (from lat. *post cibus diurnus*)
PCV	after dinner (from lat. *post cibus vespertinus*)
ICM	between breakfast and lunch
ICD	between lunch and dinner
ICV	between dinner and the hour of sleep

5.2.1.2 Offset : IVL<T.diff>

An interval that marks the offsets for the beginning, width and end of the event-related periodic interval measured from the time each such event actually occurred.

For example: if the specification is "one hour before breakfast for 10 minutes" the offset's low boundary is –1 h and the offset's width is 10 min (consequently the offset's high boundary is –50 min.)

5.2.1.3 Resolving the Event-Relatedness

An event-related periodic interval of time is a set of time, that is one can test whether a particular time or time interval is an element of the set. Whether an event-related periodic interval of time contains a given interval of time is decided using a relation *event* × *time* referred to as EVENT(*event*, *time*). The property occurrenceAt(*t*) is the occurrence interval that would exist if the event occurred at time *t*.

```
invariant(EIVL<T> x, T eventTime, IVL<T> v)
    where v.equals(x.occurrenceAt(eventTime)) {
  v.low.equals(eventTime.plus(x.offset.low));
  v.high.equals(eventTime.plus(x.offset.high));
  v.lowClosed.equals(x.offset.lowClosed);
  v.highClosed.equals(x.offset.highClosed);
};
```

Thus, an event related interval of time contains a point in time *t* if there is an event time *e* with an occurrence interval *v* such that *v* contains *t*.

```
invariant(EIVL<T> x, T y) {
  x.contains(y).equals(exists(T e, IVL<T> v)
                         where EVENT(x.event, y)
                          .and(v.resolvedAt(y)) {
                       v.contains(y);
                      });
};
```

5.2.1.4 Literal Form

The literal form for an event related interval begins with the event code followed by an optional interval of the time-difference.

```
EIVL<TS>.literal ST {
  EIVL<TS> : event      { $.event.equals($1); }
  | event offset        { $.event.equals($1); $.offset.equals($2); };

  CV event : ST         { $.code.equals($1);
                          $.codeSystem.equals(2.16.840.1.113883.5.1019); }

  IVL<TS.diff> offset
  : "+" IVL<TS.diff>    { $.equals($2); }
  | "-" IVL<TS.diff>    { $.low.equals($2.high.negate);
                          $.high.equals($2.low.negate);
                          $.width.equals($2.width);
                          $.lowClosed($2.highClosed);
                          $.highClosed($2.lowClosed); };
};
```


For example, one hour after meal would be "`PC+[1h;1h]`". One hour before bedtime for 10 minutes: "`HS-[50min;1h]`".

5.3 General Timing Specification (GTS)

The general timing specification (GTS) semantically is a general set of points in time. The purpose of the GTS is to specify the complex timing of events and actions (mainly in orders and scheduling systems.) The GTS also supports the cyclical validity patterns that may exist for certain kinds of information, such as phone numbers (evening, daytime), addresses (so called "snowbirds," residing in the south during winter and north during summer) and office hours.

The GTS data type has the following aspects:

- GTS as a general set of points in time (SET<TS>). From this aspect GTS answers whether any given point in time falls in the schedule described by the GTS value.
- GTS as the combination of multiple periodic intervals of time. This aspect describes how both simple and complex repeat-patterns are specified with the GTS.
- GTS as a generator of a sequence of intervals of point in time (LIST<IVL<TS>>). From this aspect, GTS can generate all occurrence intervals of an event or action, or all validity periods for a fact.
- GTS as an expression-syntax defined for a calendar. This aspect is the GTS literal form.

In all cases the GTS is defined as a set of point in time (SET<TS>). Using the set operations, union, intersection and difference, more complex sets of time can be constructed from simpler ones. Ultimately the building blocks from which all GTS values are constructed are interval, periodic interval, and event-related periodic interval. The construction of the GTS can be specified in the literal form. No special data type structure is defined that would generate a combination of simpler time-sets from a given GTS value. While any implementation would have to contain such a structured representation, it is not needed in order to exchange GTS values given the literal form.[53]

```
type GeneralTimingSpecification alias GTS extends SET<TS> {
            IVL<TS>    hull;
            IVL<TS>    nextTo(TS x)
            IVL<TS>    nextAfter(TS x)

            GTS        periodicHull(GTS x);
            BL         interleaves(GTS x);

  demotion  LIST<IVL<TS>>;
  literal   ST;
};
```

The GTS data type is defined as using intervals, periodic intervals, and event-related periodic intervals. Intervals of time have been defined above

[53] The GTS is an example of a data type that is only defined algebraically without giving any definition of a data structure that might implement the behavior of such a data type. The algebraic definition looks extremely simple, so that one might assume it is incomplete. Since at this point we are relying entirely on the literal form to represent GTS values, all the definition of data structure is really contained in the grammar for the literal definition.

5.3.1.1 Convex Hull

A convex hull is the least interval that is a superset of all occurrence intervals. As noted in Section 3.1.2, all totally ordered sets have a convex hull. Because a GTS is a SET<TS> and because a SET<TS> is a totally ordered set, all GTS values have a convex hull.

The convex hull of a GTS can less formally be called "outer bound interval". Thus, the convex hull of a GTS describes the absolute beginning and end of the repeating schedule. For infinite repetitions (e.g., a simple periodic interval) the convex hull has infinite bounds.

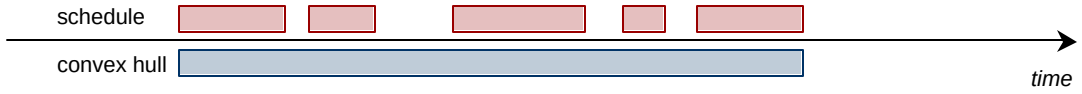


Figure 14: Convex Hull of a Schedule

5.3.1.2 GTS as a Sequence of Occurrence Intervals

A GTS value is a generator of a sequence of time intervals during which an event or activity occurs, or during which a state is effective.

The **nextTo**-property maps to every point in time *t* the greatest continuous subset (an "occurrence interval") *v* of the GTS value *S*; where *v* is the interval closest to *t* that begins later than *t* or that contains *t*.

```
invariant(GTS S, TS t, IVL<TS> v) {
  v.equals(S.nextTo(t)).equals(
         S.contains(o)
    .and(forall(IVL<TS> u) where x.contains(u) {
           u.contains(v).implies(u.equals(v)); })
    .and(    v.contains(t)
         .or(forall(TS i) where t.lessOrEqual(i)
                          .and(i.lessThan(v.low)) {
              S.conatins(i).not; })));
};
```

The **nextAfter**-property maps to every point in time *t* the greatest continuous subset (an "occurrence interval") *v* of the GTS value *S*; where *v* is the interval closest to *t* that begins later than *t*.

```
invariant(GTS S, TS t) where {
  S.contains(t).not
     .implies(S.nextAfter(t).equals(S.nextTo(t)));
  S.contains(t)
     .implies(S.nextAfter(t).equals(S.except(nextTo(t)).nextTo(t)));
};
```

A GTS value can be converted into a generic Sequence of time intervals (LIST<IVL<TS>>) of occurrence intervals.

```
invariant(GTS x) where x.isEmpty { ((LIST<IVL<TS>>)x).isEmpty; };
```



```
invariant(GTS x, IVL<TS> first)
    where x.nonEmpty
     .and(x.hull.low.nonNull)
     .and(first.equals(x.nextTo(x.hull.low))
{
  ((LIST<IVL<TS>>)x).head.equals(first);
  ((LIST<IVL<TS>>)x).tail.equals((LIST<IVL<TS>>)x.except(first));
};
```

5.3.1.3 Interleaving Schedules and Periodic Hull

For two GTS values *A* and *B* we say that A interleaves B if their occurrence intervals interleave on the time line. This concept is visualized in Figure 15.

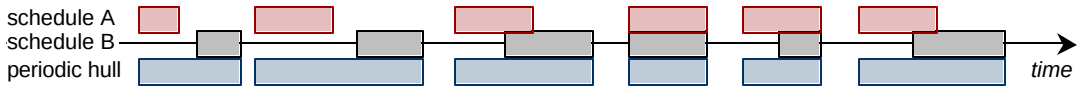


Figure 15: Interleaving Schedules and Periodic Hull

For the GTS values *A* and *B* to interleave the occurrence intervals of both groups can be arranged in pairs of corresponding occurrence intervals. It must further hold that for all corresponding occurrence intervals $a \in A$ and $b \in B$, a starts before b starts (or at the same time) and b ends after a ends (or at the same time).

The interleaves-relation holds when two schedules have the same average frequency, and when the second schedule never "outpaces" the first schedule. That is, no occurrence interval in the second schedule may start before its corresponding occurrence interval in the first schedule.

With two interleaving GTS values one can derive a periodic hull such that the occurrence intervals of the periodic hull is the convex hull of the corresponding occurrence intervals.

The periodic hull is important to construct two schedules by combining GTS expressions. For example, to construct the periodic interval from Memorial Day to Labor Day every year, one first needs to set up the schedules *M* for Memorial Day (the last Monday in May) and *L* for Labor Day (the first Monday in September) and then combine these two schedules using the periodic hull of *M* and *L*.

```
invariant(GTS A, B) where x.nonNull.and(y.nonNull) {
  A.interleaves(B).equals(
    forall(IVL<TS> a, b, c; TS t)
        where a.equals(A.nextTo(t))
         .and(b.equals(B.nextTo(a.low)))
         .and(c.equals(A.nextTo(b.high))) {
      b.equals(B.nextTo(a.high));
      a.low.lessOrEqual(b.low);
      c.equals(A.nextTo(b.high));
      c.equals(a).or(c.equals(A.nextAfter(a.high)));
    });
};
```

For two GTS values *A* and *B* where *A* interleaves *B*, a periodic hull is defined as the pair wise convex hull of the corresponding occurrence intervals of *A* and *B*.

```
invariant(GTS A, B, C) where A.interleaves(B) {
  A.periodicHull(B).equals(C).equals(
    forall(IVL<TS> a, b; TS t)
        where a.equals(A.nextTo(t))
         .and(b.equals(B.nextTo(a.low))) {
      C.contains(c).equals(c.equals(a.hull(b)));
    });
};
```

The interleaves-relation is reflexive, asymmetric, and intransitive. The periodic hull operation is non-commutative and non-associative.[54]

5.3.2 GTS Literal Form

The GTS literal allows specifying combinations of intervals, periodic intervals, and event related periodic intervals of time using the set operations, unions and intersection. This literal form is specified based on the simpler time set data types interval, periodic interval, and event related periodic interval.[55]

Unions are speechified by a semicolon-separated list. Intersections are specified by a white space separated list. Intersection has higher priority than union. Exclusions (set differences) can be specified using a backslash; exclusions have an intermediate priority, i.e. weaker than intersection but stronger than union.

Table 43: GTS Set-Operators

Operation	Operator	Priority
Intersection	" " (white space)	high
Union	";" (semicolon)	low
Exclusion	"\" (back slash)	low
Periodic Hull	".." (two periods)	high

Also parentheses can be used to overcome operator precedence when necessary.

```
GTS.literal ST {
  GTS : symbol                 { $.equals($1); }
  | union                      { $.equals($1); };
  | exclusion                  { $.equals($1); };

  SET<TS> union
  : intersection ";" union     { $.equals($1.union($3)); }
  | intersection               { $.equals($1); };
```

[54] The interleaves property may appear overly constrained. However, these constraints are reasonable for the use case for which the interleaves and periodic hull properties are defined. To safely and predictably combine two schedules one would want to know which of the operands sets the start points and which sets the endpoints of the periodic hull's occurrence intervals.

[55] This literal specification again looks surprisingly simple, so one might assume it is incomplete. However, the GTS literal is based on the TS, IVL, PIVL, and EIVL literals and does also imply the literals for the extensions of TS, notably the PPD<TS>. The GTS literal specification itself only needs to tie the other literal forms together, which is indeed a fairly simple task by itself.


```
  SET<TS> exclusion
  : exclusion "\" intersection { $.equals($1.except($3)); };

  SET<TS> intersection
  : factor intersection       { $.equals($1.intersection($2)); }
  | factor;                   { $.equals($1); }

  SET<TS> hull
  : factor ".." hull          { $.equals($1.periodicHull($3)); }
  | factor;                   { $.equals($1); }

SET<TS> factor
  : IVL<TS>                   { $.equals($1); }
  | PIVL<TS>                  { $.equals($1); }
  | EIVL<TS>                  { $.equals($1); }
  | "(" GTS ")"               { $.equals($1); };
};
```

The following table contains paradigmatic examples for complex GTS literals. For simpler examples confer to the literal forms for interval, periodic interval, and event related interval.

Table 44: Examples for Literal Expressions for Generic Timing Specifications

Literal Expression	Meaning
`M09 D15 H16 N30 S34.12`	September 15 at 4:30:34.12 PM as the intersection of multiple periodic intervals of times (calendar patterns)
`M0915163034.12`	September 15 at 4:30:34.12 PM as one simple periodic interval of time (calendar pattern)
`M01; M03; M07`	January, March, and July (a union of three periodic intervals of time)
`M04..09 M/2`	Every second month from April to September (April, June, August)
`J1; J2; J4`	Monday, Tuesday, Thursday
`W/2 J2`	every other Tuesday (intersection of every other week and every Tuesday)
`1999 WY15`	the 15th calendar week in 1999 (period code is optional for the highest calendar unit)
`WM2 J6`	Saturday of the 2nd week of the month
`M05 WM2 J6`	Saturday of the 2nd week of May
`M05 DM08..14 J7`	Mother's day (second Sunday in May.)
`J1..5 H0800..1600`	Monday to Friday from 8 AM to 4 PM
`J1..4 H0800..1600; J5 H0800..1200`	Monday to Thursday 8 AM to 4 PM and Friday 8 AM to 12 noon.
`[10 d] H/8`	Three times a day over 10 days (each time a 60 minutes interval).
`H0800..1600 \J3`	Every day from 8 AM to 4 PM, except Wednesday.
`(M0825..31 J1)..M0831`	The last calendar week of August.
`JHNUSMEM..JHNUSLBR`	The season from the U.S. holidays Memorial Day to Labor Day

5.3.2.1 Symbolic Abbreviations for GTS expressions.

The following Table 45 defines symbolic abbreviations for GTS values that can be used in GTS literals instead of their equivalent GTS term. Abbreviations are defined for common periods of the day (AM, PM), for periods of the week (business day, weekend), and for holidays. The computation for the dates of some holidays, namely the Easter holiday, involve some sophistication that goes beyond what one would represent in a GTS literal term. It is assumed that the dates of these holidays are drawn from some table or some generator module that is outside the scope of this specification.

These abbreviations are named GTS values and they can in turn be a factor of a GTS term. For example, one can say "JHCHRXME H08..12" to indicate that the office hours on Christmas Eve is from 8 AM to 1PM only. And one can say "JHNUSMEM..JHNUSLBR" for the typical mid-western swimming pool season from Memorial Day to Labor Day.

Table 45: Abbreviations for General Timing Specifications

Code	Definition	Equivalent
AM	Every morning at institution specified times.	H00..11 IST
PM	Every afternoon at institution specified times.	H12..23 IST
BID	two times a day at institution specified time	H/12 IST
TID	three times a day at institution specified time	H/8 IST
QID	four times a day at institution specified time	H/6 IST
JB	Regular business days (Monday to Friday excluding holidays)	J1..5 \JH
JE	Regular weekends (Saturday and Sunday excluding holidays)	J6..7 \JH
JH	Holidays	
	Christian Holidays (Roman/Gregorian "Western" Tradition.)	
JHCHRXME	Christmas Eve (December 24)	M1224
JHCHRXMS	Christmas Day (December 25)	M1225
JHCHRNEW	New Year's Day (January 1)	M0101
JHCHREAS	Easter Sunday. The Easter date is a rather complex calculation based on Astronomical tables describing full moon dates. Details can be found at [http://www.assa.org.au/edm.html, and http://aa.usno.navy.mil/AA/faq/docs/easter.html]. Note that some Eastern Orthodox Holidays are based on the Julian calendar.	
JHCHRGFR	Good Friday, is the Friday right before Easter Sunday.	
JHCHRPEN	Pentecost Sunday, is seven weeks after Easter (the 50th day of Easter.)	
JHNUS	**United States National Holidays** (public holidays for federal employees established by U.S. Federal law 5 U.S.C. 6103.)	
JHNUSMLK	Dr. Martin Luther King, Jr. Day, the third Monday in January.	M0115..21 J1
JHNUSPRE	Washington's Birthday (Presidential Day) the third Monday in February.	M0215..21 J1
JHNUSMEM	Memorial Day, the last Monday in May.	M0525..31 J1
JHNUSMEM5	Friday before Memorial Day Weekend	M0522..28 J5
JHNUSMEM6	Saturday of Memorial Day Weekend	M0523..29 J6
JHNUSMEM7	Sunday of Memorial Day Weekend	M0524..30 J7
JHNUSIND	Independence Day (4th of July)	M0704
JHNUSIND5	Alternative Friday before 4th of July Weekend [5 U.S.C. 6103(b)].	M0703 J5
JHNUSIND1	Alternative Monday after 4th of July Weekend [5 U.S.C. 6103(b)].	M0705 J1
JHNUSLBR	Labor Day, the first Monday in September.	M0901..07 J1
JHNUSCLM	Columbus Day, the second Monday in October.	M1008..14 J1
JHNUSVET	Veteran's Day, November 11.	M1111
JHNUSTKS	Thanksgiving Day, the fourth Thursday in November.	M1122..28 J4
JHNUSTKS5	Friday after Thanksgiving.	M1123..29 J5

Note: this table is not complete. Neither does it include religious holidays other than Christian (of the Gregorian (western) tradition), nor does it contain national holidays on other countries. This is a limitation to be remedied by subsequent additions.

Note: holidays are locale-specific. Exactly which religious holidays are subsumed under JH depends on the locale and other tradition. For global interoperability, using constructed GTS expressions is safer than named holidays. However, some holidays that depend on moon phases (e.g., Easter) or ad-hoc decree can not be easily expressed in a GTS form.

