Version 3 Data Types

Chair/Editor: Gunther Schadow
Regenstrief Institute for Health Care, Indianapolis, IN.

Editor: Paul V. Biron
Kaiser Permanente, Southern California.

Contents

1 Introduction *2*
- 1.1 What is a Data Type? 2
- 1.2 Representation of Data Values 3
- 1.3 Properties of Data Values 3
- 1.4 Need for the Abstraction 4
- 1.5 Need for an HL7 Data Type Standard 4
- 1.6 Overview of Contents 5
- 1.7 Acknowledgements 8

2 Formal Data Type Definition Language *10*
- 2.1 Declaration 11
- 2.2 Invariant Statements 12
- 2.3 Type Conversion 14
- 2.4 Literal Form 15
- 2.5 Generic Data Types 18

3 Fundamental Data Types *20*
- 3.1 Data Type 20
- 3.2 Data Value (ANY) 20
- 3.3 Boolean (BL) 23

4 Text *25*
- 4.1 Binary Data (BIN) 25
- 4.2 Encoded Data (ED) 25
- 4.3 Character String (ST) 31

5 Things, Concepts, and Qualities *33*
- 5.1 Concept Descriptor (CD) 33
- 5.2 Instance Identifier (II) 41
- 5.3 Telecommunication Address (TEL) 44
- 5.4 Postal and Residential Address (AD) 47
- 5.5 Person Name (PN) 51
- 5.6 Organization Name (ON) 53

6 Quantities *55*
- 6.1 Quantity (QTY) 55
- 6.2 Integer Number (INT) 57
- 6.3 Real Number (REAL) 58
- 6.4 Ratio (RTO) 62

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

6.5 Physical Quantity (PQ) 63
6.6 Monetary Amount (MO) 66
6.7 Point In Time (TS) 68

7 Generic Collections 75
7.1 Set (SET) 75
7.2 Sequence (LIST) 78
7.3 Bag (BAG) 80
7.4 Interval (IVL) 81

8 Timing Specification 88
8.1 Periodic Interval of Time (PIVL) 88
8.2 Event-Related Periodic Interval of Time (EIVL) 92
8.3 General Timing Specification (GTS) 94

9 Generic Type Extensions 98
9.1 Annotated (ANT) 98
9.2 History Item (HXIT) 99
9.3 History (HIST) 99

10 Uncertainty and Probability 101
10.1 Uncertain Value – Narrative (UVN) 101
10.2 Uncertain Value – Probabilistic (UVP) 101
10.3 Non-Parametric Probability Distribution (NPPD) 102
10.4 Parametric Probability Distribution (PPD) 103

A Object Identifiers (normative) 109

B Summary of Formal Definitions 112

C Detailed Table of Contents 144

1 INTRODUCTION

1.1 What is a Data Type?

Every piece of data has a data type. Data types define the meaning (semantics) of data values that can be assigned to a data field. 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 nor 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. One can think of data values as immutable objects where identity does not matter (identity and equality are the same.)

 Copyright © Health Level Seven, Inc. All rights reserved.

ITS Note: The notion of update modes defined in the HL7 message development framework does not exist for data values because data values have no concept of being updated.

1.2 Representation of Data Values

Data values can be represented through various symbols, or objects 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, strings of true-false values (binary), groups of wooden sticks, bags of glass marbles, small rocks, or scratches on a wall. The representation does not matter as long as it behaves according to the semantic rules.

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, the signs – and +, any two signs that are distinct from each other. The representation of data types does not matter as long as the representations conform to the definition of the data type.

Standards for representing data types using various technological approaches are needed to communicate meaning. Implementations of data types will define computer procedures for the properties of a data type to act on the representation of data values generating new data values. However, the meaning these representations communicated, generated, and processed in computer programs, are defined based on the semantics of the data types. This specification defines the semantics, the meaning of the HL7 data types. This specification is about semantics only, independent from representational and operational concerns of specific implementation technologies.

1.3 Properties of Data Values

Data values have properties defined by their data type. One can think of a data value's property as logical predicates or as mathematical functions; or one can simply state that 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. A property has a domain, which is the set of possible "answer" values. Every property is assigned a data type, and every possible value of the property is a value of that type. 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.

Whether semantic properties have arguments is not a fundamentally relevant distinction.

A data type's semantic property without arguments is not necessarily a "data field" or component of a valid representation of that data type. For example, for integer values, we can defined 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, and return values, throwing exceptions, etc. These are all valid concepts of computer systems implementation of data types but are not relevant for defining 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

1.4 Need for the Abstraction

This specification makes an 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, technology is going to change, and with it representation and implementation technologies. HL7 standards are primarily targeted to health care domain information, independent from the technology supporting this information. The expectation is that the HL7 specifications that are independent from today's technology will be of use after the next technological "paradigm shift".

The issue of data types is closer to implementation technology than most other HL7 information standards and therein lies a certain danger for data types to be specified too implementation technology dependent.

The majority of HL7 standards are 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.

In addition, any concrete implementation of the HL7 standards must ultimately use the built-in data types of their implementation technology. Therefore, the mapping between HL7 abstract data types and implementation technology provided data types must be very flexible. This flexibility can be gained through a semantic specification to which 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.

These ITS-mappings need not abide by any abstract syntax that would be foreign to the Implementation Technology, because this standard does not put forth an abstract syntax. 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 and its properties is mostly descriptive of what implementation technologies typically provide and of what is relevant on the application layer.

For another example, a decimal representation, a floating-point register and a scaled integer are all possible native representations of real numbers on 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 HL7 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.

1.5 Need for an HL7 Data Type Standard

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, 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

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.

All but the health-care standards related types are defined for a certain implementation technology. Nevertheless, among these technology dependent data types the ISO 11404 language independent data types and the Common Lisp and Scheme data type specification have been of particular interest for their approach to specify the semantics of data types independent from abstract syntax.

1.6 Overview of Contents

This specification attempts to define all the data types needed for health care information interchange. To cover the field exhaustively the survey of other data type systems was used as well as an *a-priori* approach to find all data types needed and to avoid duplication. In statistics, data is commonly classified according to Stevens' 4 scale types: nominal, ordinal, interval, and ratio; however, these classic scale types do not seem to explain what we see in the area of data types. This specification used a phenomenological approach to the field. It begins with the Boolean, the most general data type that is capable of encoding all information. From this analysis, we found three major partitions:

1. **Text** – the area of representation of ideas. All information can be expressed as text but text must be interpreted to have meaning. Text is only used for information for which this specification does not define semantics, information that is mainly interpreted by humans or encoded for interpretation by means external to this specification. Written text, spoken text (audio) as well as graphics or images belong in this category.

2. **Concepts and Things** – symbols and identifiers for concepts and things (corresponding to Steven's nominal scale.) Concepts are uniformly referred to with codes. Things can be identified with instance identifiers, or pointers, that have no other meaning than the thing they point to. Many things can be identified through natural identifiers, such as names for people, organizations and places. This specification does not completely cover these natural identifiers (e.g., geographical coordinates may be a candidate data type to identify places.)

3. **Quantities** – comprising Steven's interval and ratio scales, quantities are a class of types whose semantics are generally well understood. Particularly numbers (integer and real) but also physical quantities are well-defined in mathematical and scientific systems of numbers and measurements. Time is another area of quantities that, though it can be easily defined in terms of real numbers, due to the measurement of time in calendars and due to complex timing of repeating events, required much attention.

These three areas overlap each other. Numbers are the textual representations of quantities and thus fall between text and quantities. Textual symbols (words, codes) are text with a meaning given by reference to code sets. Ordinal values, finally, are coded concepts with a semi-quantitative interpretation. (However, ordinals, are not covered by a special data type in this specification.)

In addition to these basic data types, a set of generic data types and data type extensions is defined. This provides for type-safe collections of values in sets, sequences and intervals. It also provides for extensions of basic types expressing uncertainty or time-dependence that comes with many real-world data.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

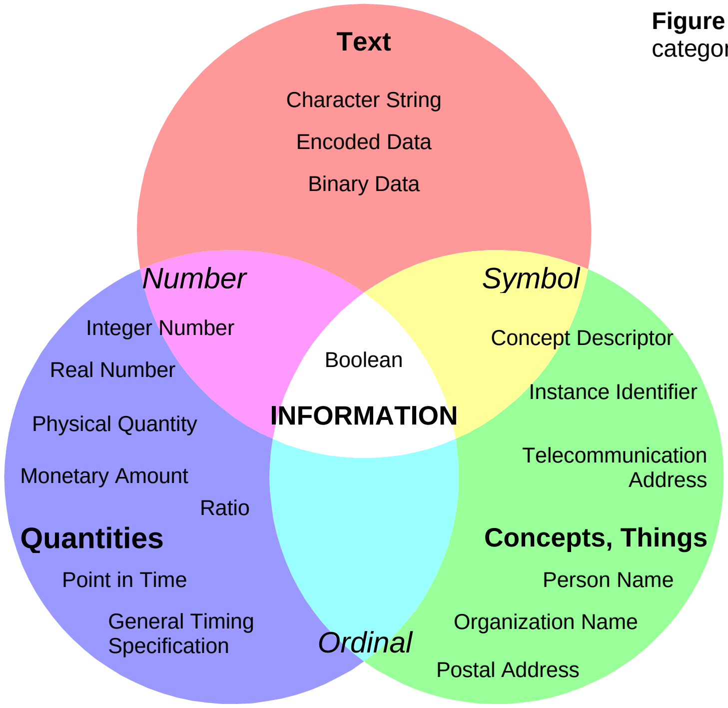


Figure 1: Overview of the three major categories of data types.

While the *a-priori* method to find data types and the strong focus on semantics should ensure a stable set of data types, this specification's coverage is not guaranteed to be complete. Areas known to be incomplete or missing are other natural identifiers (e.g., geographical coordinates) and ordinal values. The following table provides an overview of the data types defined in this specification, and shows (roughly) how they relate to data types of HL7 version 2.

Table 1: Overview of HL7 version 3 data types and mapping to HL7 v2.3

Name	Symbol	Description	v2.3
Boolean	BL	The Boolean type stands for the values of two-valued logic. A Boolean value can be either true or false.	ID (Y/N)
		Text	
Encoded Data	ED	Can convey any data that is primarily shown to human beings for interpretation. ED can be any kind of text, whether unformatted or formatted written language or other multi-media data. The plain character string type ST is equivalent to ED of media type *text/plain*. Instead of the data itself, an ED may contain only a reference (URL.)	TX, FT, ED, RP
Character String	ST	Used when the appearance of text does not bear meaning, this is true for formalized text and all kinds of names. If used as a data type for free text an ST instance is equivalent with an ED of media type *text/plain*.	ST
		Things, Concepts, and Qualities	
Concept Descriptor	CD	A descriptor for a concept, usually through a code from a coding system. For complex domains, such as findings, diagnoses, the concept descriptor may contain translations into other coding systems or free text descriptions. This data type also supports post-coordinated (compositional) coding. Use of this data type is typically constrained, hiding some of the power and complexity of the concept descriptor.	ID CE
Coded Simple Value	CS	A restriction of the concept descriptor (CD). 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. CS is used for coded attributes that has a single HL7-defined value set.	ID
Coded Value	CV	A restriction of the concept descriptor (CD). CV suppresses the CD properties *translation* and *modifier*. CV is used when any reasonable use case will require only a single code value to be sent.	ID,CE

 Copyright © Health Level Seven, Inc. All rights reserved.

Coded With Equivalents	CE	A restriction of the concept descriptor (CD). CE suppresses the CD *modifier* property. The CE also restricts the translation property such that the translation is a set of CV values that may not themselves contain translations. Used when the use alternative codes may exist.	CE
Instance Identifier	II	Used to uniquely identify some individual entity, a piece of data or a real world entity. Examples are medical record number, placer and filler order id, service catalog item number, etc. In HL7 version 2.x no clear distinction between instance identifier and concept code was made, often codes where used to refer to instance entities. In HL7 version 3 identifiers, not codes, are used for instance entities.	ID, IS, CE, HD, EI
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.)	TN, XTN
Postal and Residential Address	AD	The main use of such declared data is to be printed on mailing labels (postal address,) or to allow a person to physically reach the location (residential address.)	AD, XAD
Person Name	PN	Used for one full name of a natural person. Names usually consist of several name parts that can be classified as given, family, nickname etc. The PN of HL7 version 2 has been divided into a PN data type (capturing just one name) and an information model class (capturing name purpose code, change history, etc.) This data type is intended to be used only in the Person_name class. Instead of directly using this data type for an attribute of another class, one should consider drawing an association to the Person_name class.	PN, XPN
Organization Name	ON	Used to name an organization. Similar but simpler than the name of a natural person.	XON
		Quantities	
Integer Number	INT	Integer numbers are the positive and negative whole numbers, typically the results of counting and enumerating. Integer numbers are discrete, the set of integers is infinite but countable. We impose no bounds on the size of integer numbers.	NM
Real Number	REAL	Fractional numbers. Real numbers are needed beyond integers whenever quantities of the real world are *measured*, *estimated*, or computed from other real numbers. The standard representation is decimal, where the number of significant decimal digits is known as the *precision*.	NM
Physical Quantity	PQ	A dimensioned quantity expressing the result of a measurement. Consists of a real number value and a physical unit. Physical Quantities should be preferred instead of two attributes expressing a number and a unit separately. Physical quantities are often constrained to a certain dimension by specifying some unit representing the dimension (e.g. m, kg, s, kcal/d, etc.)	CQ
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.)	MO
Ratio	RTO	A ratio quantity is the pair of a numerator quantity and a denominator quantity both explicitly recorded (e.g. 1:128.) Ratios occur in laboratory medicine as "titers", i.e., the maximal dissolution at which an analyte can still be detected. The Ratio type is used whenever the reduction to a simple real number or physical quantity is to be avoided. In other words when you want the numerator and denominator to stand separate, use the ratio.	SN
Point in Time	TS	A scalar defining a point on axis of natural time.	TS
General Timing Specification	GTS	A data type used to specify the timing of events. Every event spans one time interval (*occurrence interval*), i.e., a continuous range of natural time between a start-point and an end-point in time. 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 (Mo – Fr) between 8:00 and 17:00 for 10 minutes.”	TQ
		Generic Collections	
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.	SN, XNM
		Generic Type Extensions	
Annotated	ANT⟨*t*⟩	A generic data type extension supporting arbitrary free-text annotations for any data value.	

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

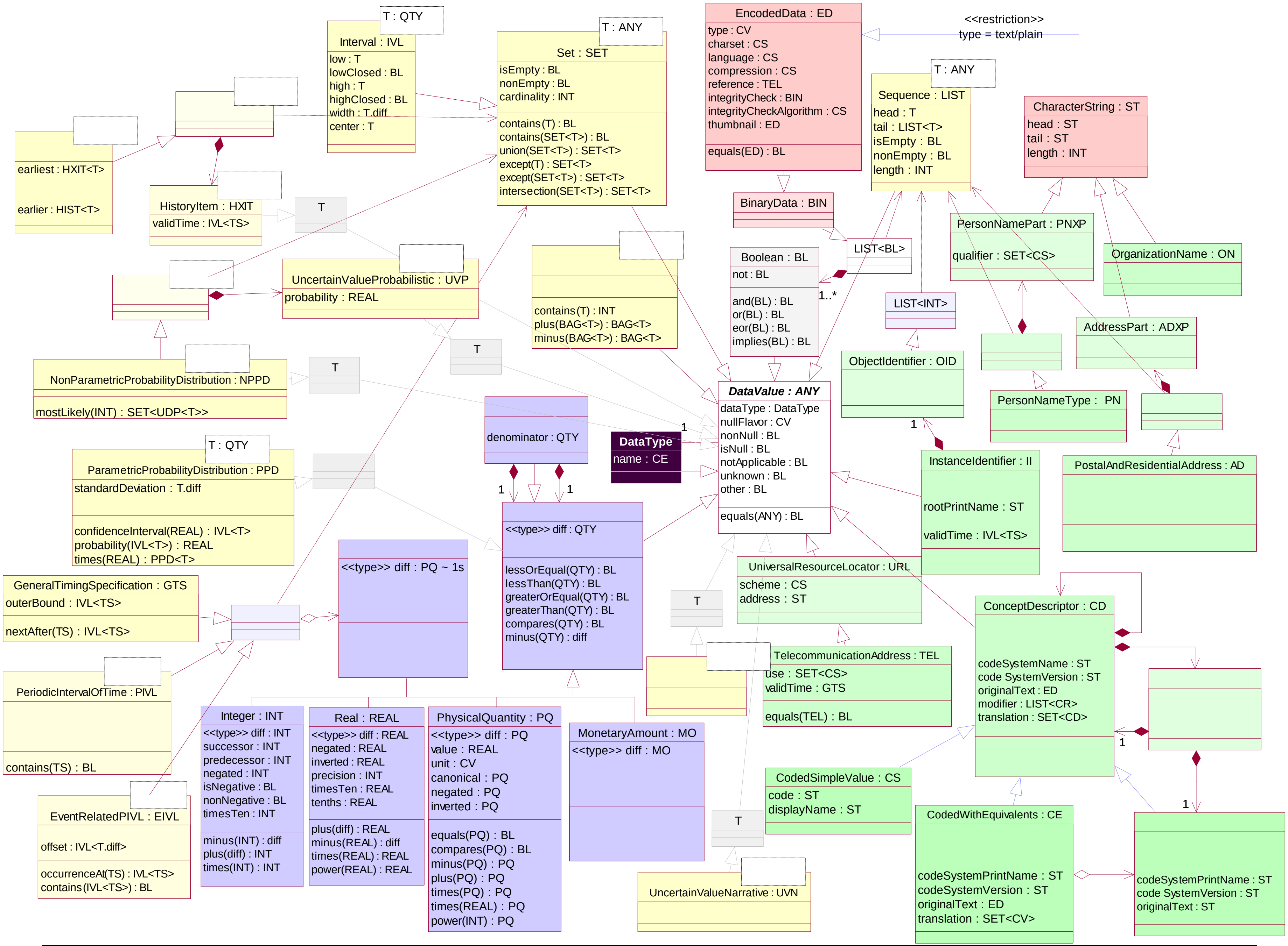
Uncertain value narrative	UVN⟨*t*⟩	A generic data type extension for annotating a value with a verbal statement of uncertainty, such as “estimated”, “probably”, or “unlikely”. Use UVP⟨*t*⟩ instead.
History	HIST⟨*t*⟩	A collection of data where each element is tagged with a valid-time interval.
Uncertain 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.

Note that some data types that existed in HL7 version 2 no longer exist in version 3. Many of the old composite types, such as CN, contain multiple concepts, and are now represented more explicitly in the information model as either attributes or classes. Other types, such as ID, IS, and CE, received a more rigorous definition so that an automatic 1:1 mapping is often not possible.

1.7 Acknowledgements

This standard is the result of over one and a half years of intense work through e-mail, telephone conferences and meeting discussions. Gunther Schadow (Regenstrief Institute for Health Care) chaired this task force, and he 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), and David Webber. Thanks to James Case (University of California, Davis), Norman Daoust (Health Partners), Irma Jongeneel (HL7 The Netherlands), Michio Kimura (HL7 Japan), John Molina (SMS), Richard Ohlmann (HBO & Company), 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.

 Copyright © Health Level Seven, Inc. All rights reserved.



Copyright © 2000, Health Level Seven, Inc. All rights reserved.

2 FORMAL DATA TYPE DEFINITION LANGUAGE

"Why, you don't even know what they're about!" said Alice.
"Read them," said the King.
The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.
"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

A given task is as complex as it is. Postponing formality doesn't eliminate the necessity for it, it simply puts the onus on the programmer rather than the designer – resulting in the ambiguity being resolved many times instead of just once.

Note: All formal definitions shown in this section are examples only and do not contain normative value.

This specification defines data types in both textual description and in a formal definition. The data type definition language used in this specification is tailored to the specific needs of this particular specification and is explained in this section.

A formal definition of data types is used in order to clarify the semantics of the proposed types as unambiguously as possible. 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.

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, the **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 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.
- allowable type conversions;
- syntax of character string value literals (if any;)

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.

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.)

 Copyright © Health Level Seven, Inc. All rights reserved.

2.1 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.

```
type Boolean alias BL extends ANY
```

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.

The header of 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.

```
type Boolean alias BL extends ANY
    values(true, false)
```

The header of the data type declaration is followed by a block declaring the semantic properties of every value of the data type. This block is enclosed in curly braces. Each property declaration is finished with a semicolon and each data type declaration is finished by a semicolon after the closing curly brace.

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

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.

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.

The **`extends`**-clause has the usual meaning of a specialization relationship known from the object oriented method. 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.

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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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.

The **`type`**-declaration may be qualified by the keyword **`abstract`** and **`protected`**. An abstract type is a type generalization where no value is of this type without also being of another type that extends the abstract type. 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. (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.)

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.

2.2 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.

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.nonNull;
  false.nonNull;
  true.not.equals(false);
  false.not.equals(true);
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

2.2.1 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*.) No primitive data types, or operations, pre-exist the definition of any data type. The only preexisting features of the assertion expression language are:

- 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*.

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++.

Since assertions are Boolean expressions, a Boolean data type is defined early on (see Section 3.3).

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.

2.2.2 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:

```
invariant(SET 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*.”

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
invariant(SET x) 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.

2.3 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.

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.

Three types 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 don't; 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.

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.

2.3.1 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). This conversion would only be possible if the IVL's has at least one finite boundary.

In the data type definition language, a demotion is declared using the keyword **`demotion`** and the data type name to which to demote:

 Copyright © Health Level Seven, Inc. All rights reserved.

```
type EncodedData alias ED {
  ...
  demotion  ST;
  ...
};
```

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

2.3.2 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 types. 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);
  ...
};
```

2.4 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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

decimal point, etc. The literal for the Point in Time (TS) type is the well-known HL7 v2.x TS format (e.g., “`200004190035`” for April 19, 2000, 12:35 am.) 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.

The different grammars of literals are not meant to be combined into one overall HL7 value expression grammar. No attempt is made to resolve ambiguities between the literals of different types. For example “1.2" can be a valid literal for both Object Identifier (OID) and a Real Number.

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 are used.

2.4.1 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;
  ...
};
```

2.4.2 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).

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.

For example, consider the following simple definition of a data type for cardinal numbers (positive integers.) This type definition uses 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;
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

2.4.2.1 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 a value of a defined data type. The data type of the symbol’s value is declared where the symbol is defined:

```
  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.

2.4.2.2 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); };
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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`.

Note that the equals property defined below for any data value is a relation, a test for equality, not an assignment statement. One can not assign a value to another value. Unlike YACC and LEX analyzers, 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.

2.4.2.3 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 2.

Table 2: 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. E.g., `/ab?c/` matches “`ac`” and “`abc`”.
…*	The preceding pattern may occur zero or many times. E.g., `/ab*c/` matches “`ac`”, “`abc`”, “`abbc`”, “`abbbc`”, etc.
…+	The preceding pattern may occur one or more times. E.g., `/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 $0 \leq n \leq m$. E.g., `/ab{2,4}c/` matches “`abbc`”, “`abbbc`”, and “`abbbbc`”.
… | …	The pattern on either side of the bar may match. E.g., `/ab|cd/` matches “`abd`” and “`acd`” but not “`abcd`”.
(…)	The pattern in parentheses is used as one pattern for the above operators. E.g., `/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 match. E.g., `/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. E.g, `/a\+b/` matches “`a+b`”.
... \/ ...	Matches the slash as a character. E.g., `/a\/bc/` macthes “`a/bc`”.

2.5 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

Copyright © Health Level Seven, Inc. All rights reserved.

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;
};
```

2.5.1 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 the notion of “duplicate ⟨T⟩ data type requires all elements of a set to be of the same data type.
- **Sequence** (LIST⟨T⟩.) A sequence, or list, of values is a collection of values in an arbitrary but particular order. A list has a head and a tail, where the head is an element and the tail is the list 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 7ff.

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 the respective sections when coming across a generic type being used to define another type.

2.5.2 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⟨type 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. Generic type extensions are also called “mixins”, since their effect is to *mix* certain properties *in*to the preexisting 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

3 FUNDAMENTAL DATA TYPES

This section defines the fundamental properties of all data types and all data values. It also specifies the Boolean data type, which is required to formally state facts about the data types.

3.1 Data Type

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;
};
```

The only property of the type DataType that is needed by this specification is the name property. A data type name is a code with equivalents (CE). 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.

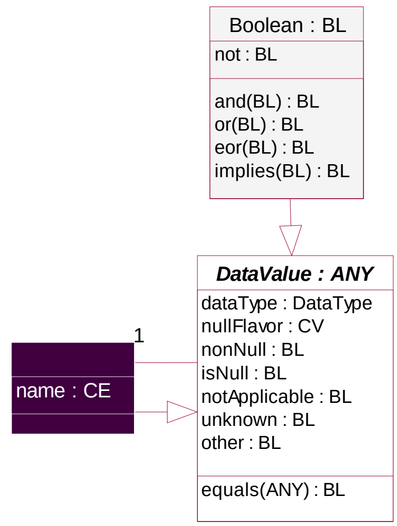


Figure 2: Fundamental data types.

3.2 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);
};
```

3.2.1.1 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;
};
```


Copyright © Health Level Seven, Inc. All rights reserved.

3.2.1.2 The predicates isNull and nonNull

A property can be of an exceptional value. 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 it NULL. If the value is NULL, the nullFlavor property is non-NULL. If the value is not NULL, its null flavor attribute is NULL (not applicable.)

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

Table 3: 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.)
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. This is only defined in messages, never in application data! All values not present in the message must be replaced by the applicable default, or no-information (NI) as the default of all defaults.*

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.)

Note also the fine difference in coded data types between NULL/*other* on the one hand and *coded with extensibility* (CWE) on the other hand. 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.

Some of these null flavors are defined as 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. Remember 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
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 HL7 specifications the term "mandatory" is used while this specification formulates the mandatory constraint explicitly.

Thus says the following invariant 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: while NULL-flavors are applicable to any property of a data value or a higher-level object attribute, ITS are explicitly allowed not to represent NULL-flavors in cases where the difference of null flavors is not significant. If nothing else is noted in this specification, ITS need not represent NULL-flavors for property values.

3.2.1.3 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

3.3 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.

```
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.

3.3.1.1 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);
};
```

3.3.1.2 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;
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

3.3.1.3 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));
};
```

3.3.1.4 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*).

 Copyright © Health Level Seven, Inc. All rights reserved.

4 TEXT

All information can be expressed as text but text must be interpreted to have meaning. Text is only used for information for which this specification does not define semantics, information that is mainly interpreted by humans or encoded for interpretation by means external to this specification. Written text, spoken text (audio) as well as graphics or images belong in this category.

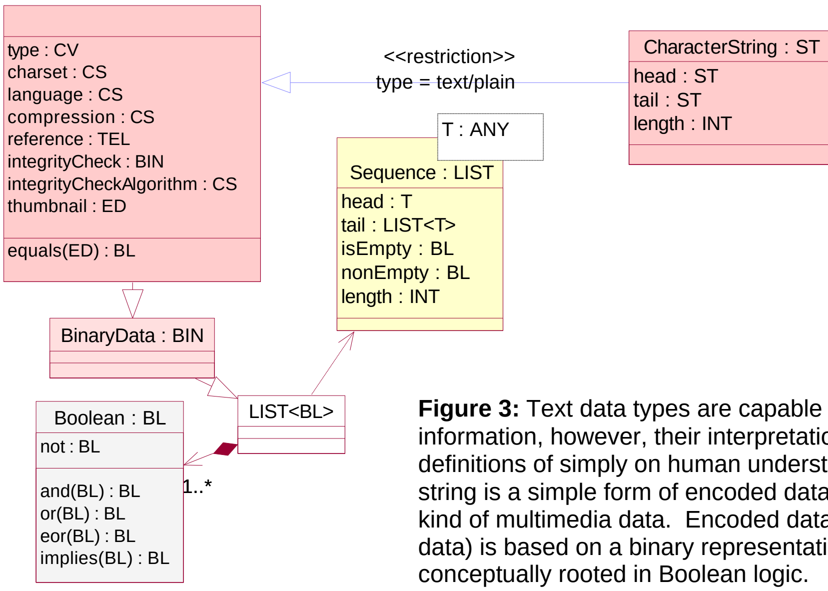


Figure 3: Text data types are capable of expressing any kind of information, however, their interpretation depends on additional definitions of simply on human understanding. The character string is a simple form of encoded data. Encoded data can be any kind of multimedia data. Encoded data (just as all communicated data) is based on a binary representation, which in turn conceptually rooted in Boolean logic.

4.1 Binary Data (BIN)

All communicated information must ultimately be physically encoded as binary data. Binary data is the most primitive yet the omnipotent encoding of all information.

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 encoded data (ED) type described below.

```
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);
};
```

4.2 Encoded Data (ED)

The encoded data type can convey any data. However, in order for that data to convey meaning, encoded data must be decoded and further interpreted. Encoded data may be a plain character string,

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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

1. **type** – specifies the protocol, or application used to decode and interpret the data (also known as the “media type” as referring to multi-media data.)
2. **charset** – identifies the character set and character encoding for character-based “media.”
3. **compression** – data may be given in a compressed form in which case compression identifies the compression algorithm used.

Encoded data can be present in two forms, inline or by reference. Inline data is communicated or moved as part of the encoded 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 EncodedData alias ED extends BIN {
            CS        type;
            CS        charset;
            CS        language;
            CS        compression;

            TEL       reference
            BIN       integrityCheck;
            CS        integrityCheckAlgorithm;

            ED        thumbnail;

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

4.2.1.1 type : CS

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

The encoded data’s type is a mandatory property, i.e., every non-NULL instance of encoded 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:

1. top level media type, and
2. 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

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 4 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.

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.

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.

deprecated

Deprecated media types should not be used.

Some media types are inherently 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.

other

By default, any media type falls into the status *other*. This status means, HL7 does neither forbid nor endorse the use of this media type. 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.

Table 4: 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. Corresponds to HL7 v2’s TX and ST data types.
text/x-hl7-ft	recommended *for compatibility with HL7 v2*	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 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*

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

		Voice Frequencies. Geneva, 1972.
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	recommended	GIF is a popular format that is universally well supported.
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.
image/tiff	other	Although TIFF (Tag Image File Format) is an international standard it has many interoperability problems in practice. Too many different versions that are not handled by all software alike.
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)

4.2.1.2 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 5 lists a few of the IANA defined character sets that are of interest to current HL7 members.

Table 5: Selected 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 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 nserve 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	other	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 7^{th} bit of each transferred byte.
ISO-8859-1	other	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	other	ISO 8859 Latin-2 character set for the Slavic languages of Central Europe.
ISO-8859-5	other	ISO 8859 Cyrillic character set for the languages Bulgarian, Byelorussian, Macedonian, Serbian and Ukrainian.

Open Issue: There are at least 10 different MIME designators for Japanese charsets some being singular character sets (e.g., JIS X208, X212, etc.) or various versions thereof, some being suites of character sets and a switching encoding (e.g., ISO2022, EUC-JP, Shift-JIS, etc.) Allowing that many charsets and versions for

Copyright © Health Level Seven, Inc. All rights reserved.

> Japanese HL7 would be a disservice to the goal for HL7 interoperability. It is unclear what charsets besides JIS X221 (ISO 10646) is needed at all. HL7 Japan is asked to submit two or three (maximum) IETF/MIME registered charsets along with their preferred IETF registered charset code and a description for inclusion into this standard.

4.2.1.3 language : CS

For character based information the language property specifies the language of the text. 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.

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.

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.

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

> 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 Encoded Data value.

4.2.1.4 compression : CS

The compression code indicates whether the raw byte data is compressed, and what compression algorithm was used.

Table 6: 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]
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 encoded data depending on the attribute or component that is declared encoded data. Character strings (see Section 4.2.1.9) may never be compressed.

4.2.1.5 reference : TEL

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

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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

By-reference encoded data may not be allowed depending on the attribute or component that is declared encoded data. Character strings (see Section 4.2.1.9) must always be inline.

4.2.1.6 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 encoded 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.

4.2.1.7 integrityCheckAlgorithm : CS

This property defines the algorithm used to compute the value in integrity check.

Table 7: 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.

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.

4.2.1.8 thumbnail : ED

A thumbnail is an abbreviated rendition of the full data. A thumbnail requires significantly fewer resources than the full data, while still maintaining some distinctive similarity with the full data. A thumbnail is typically used with by-reference encoded data. It allows a user to select data more efficiently before actually downloading through the reference.

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.

Thumbnails may not be allowed depending on the attribute or component that is declared encoded data. Character strings (see Section 4.2.1.9) 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 encoded data value to its thumbnail.

 Copyright © Health Level Seven, Inc. All rights reserved.

4.2.1.9 Equality

Two values of type Encoded 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.

4.3 Character String (ST)

The character string is a restricted encoded 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.

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

ISO/IEC 10646-1: 1993 defines a character as "A member of a set of elements used for the organisation, control, or
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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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.

The properties head, tail, and length now 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.

4.3.1.1 Literal Form

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.

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

 Copyright © Health Level Seven, Inc. All rights reserved.

5 THINGS, CONCEPTS, AND QUALITIES

A major distinction exists between codes and identifiers. Codes refer to concepts, nominal values that stand for classes of things or qualitative properties of things (*universals*.) Identifiers on the other hand refer to individual things, such as computer objects or people (*individuals*.) Diagnosis codes, procedure codes, medication codes, gender, marital status and religion codes are examples of concepts. Medical record numbers, Social Security Numbers, Provider IDs, and Manufacturer IDs are examples of identifiers.

5.1 Concept Descriptor (CD)

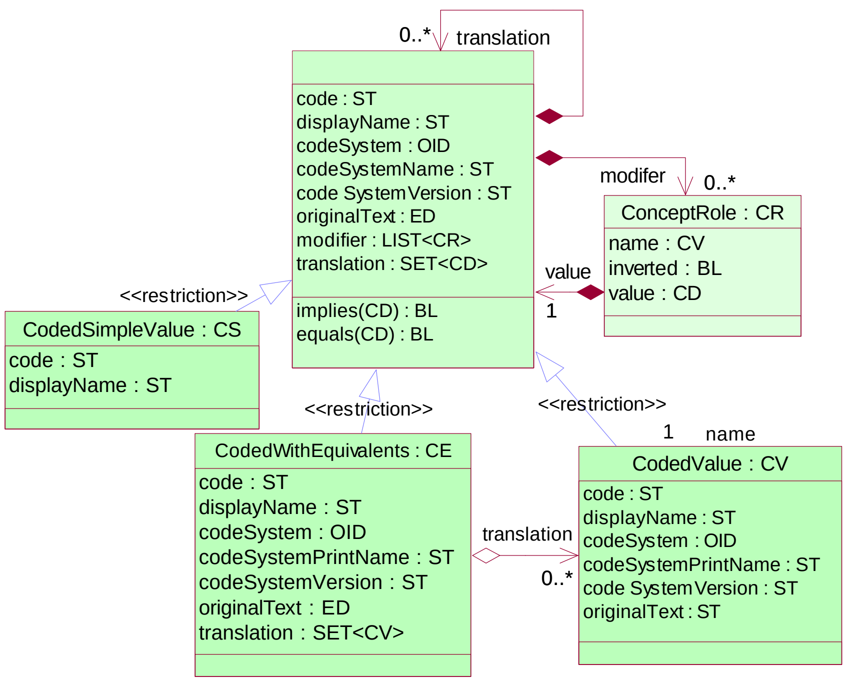


Figure 4: The Concept Descriptor information model. The concept descriptor is the most general means to describe a concept using codes. This includes code modifiers (where defined by the underlying code system) and translations into other code systems.

However, the full- featured form of the concept descriptor is rarely used. Instead, it is recommended to us one of the restricted forms. The restrictions imply that those properties not listed in the restricted type are tightly constrained. Semantically these properties are still valid.

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 (and different from translations) compositional code system 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 5.1.3.

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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

  demotion  ED;
};
```

5.1.1.1 code : ST

This is the plain code symbol, e.g., “784.0” is the code symbol of the ICD-9 code “784.0” for headache. The code must be defined in the coding system.

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;
};
```

5.1.1.2 codeSystem : OID

This property 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.

Appendix A *Object Identifiers (normative)* lists a starter set of object identifiers. Under HL7’s branch, 2.16.840.1.113883, the sub-branches 5 and 6 contain HL7 standard and external code system identifiers respectively. These two branches are maintained by the HL7 Vocabulary Technical Committee.

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.

 Copyright © Health Level Seven, Inc. All rights reserved.

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

5.1.1.3 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);
};
```

5.1.1.4 codeSystemVersion : ST

This is a version descriptor defined specifically for the given code system. The code system version is cited as a plain character string. HL7 shall specify how these version strings are formed. 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.

For the purpose of this specification, the term “version” means the following: Different versions of one code system must be compatible in general. 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);
};
```

5.1.1.5 equals(CD x) : BL

The equality of two concept descriptor values is determined solely based upon the code and coding system. If modifiers are present, the modifiers are included in the equality test. Translations are not included in the equality test. Exceptional concept descriptor values are not equal even if they have the same null flavor or the same original text.

```
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)));
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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 and reused.

The same null value or the same original text does not count as equality since it would be unsafe to equate two concepts on the basis that both were not codeable 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.

Translations are not included in the equality test of concept descriptors. This is so 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.

5.1.1.6 implies(CD x) : BL

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 9 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.

5.1.1.7 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);
};
```

5.1.1.8 translation : SETáCDñ

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 does rarely exist between two structurally different coding systems. For this reason, not all of the translations will be equally exact.

 Copyright © Health Level Seven, Inc. All rights reserved.

5.1.1.9 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 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 converted 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);
};
```

5.1.1.10 producer : II

5.1.1.12 modifier : LIST áCRñ

A concept descriptor may have modifiers if the code system defines such modifiers. Modifiers can only be used with code systems that define rules of postcoordination, where multiple codes together make up one concept. A concept descriptor with modifiers is called a *code phrase*.

For example, SNOMED allows constructing concepts as a combination of multiple codes and HCFA procedure codes come with modifiers. SNOMED RT defines a concept “cellulitis,” a role “has-topology,” and another concept “left foot.” The concept role allows you to add the modifier “has-topology: left foot” to the primary code “cellulitis” to construct the meaning, “cellulitis of

The order of modifiers is preserved, particularly for the case where the coding system allows postcoordination but defines no role names (e.g., some ICD-9 codes, SNOMED, HCFA procedure codes.)

ITS Note: All the modifier names and subordinate codes of a code phrase should come from the same coding system. Thus, the coding system mentioned for the primary CD value should be made the default for all subordinated modifier names and values.

5.1.2 Concept Role (CR)

The concept role is used to send 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”.

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;
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

5.1.2.1 name : CV

This is the role name. The role name 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 CRR.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;
};
```

5.1.2.2 value : CD

The code related to 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 CRR.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;
};
```

5.1.2.3 inverted : BL

This property indicates if the meaning of the role is inverted. This can be used in cases where the underlying code system defines inversion but does not provide reciprocal pairs of role names.

For example, a code system may define the role relation “causes” besides the concepts “Streptococcus pneumoniae” and . 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.

> 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.

5.1.3 Restrictions for the Concept Descriptors

The concept descriptor data type is very expressive, however, if all of its features, such as coding exceptions, text, translations and modifiers are used at all times, implementation and use becomes very difficult and unsafe. Therefore, the CD type is most often used in a restricted form with reduced features.

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 is to be used.

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,

 Copyright © Health Level Seven, Inc. All rights reserved.

conceptually a suppressed property is still semantically applicable, it is safe for an HL7 interface to assume the implicit default value without testing.

The unrestricted concept descriptor is currently only assigned to the following RIM attributes: Service.service_cd, Service.body_site_cd, Material.type_cd, and is allowed for use in Observation.value.

5.1.3.1 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.

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.

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

CS can only be used for a coded attribute which has a single HL7-defined value set, and where code additions to that value set require formal HL7 action (such as harmonization.) For these examples, the designation of the domain qualifier will always be CNE (*coded, non-extensible*) and the context determines unambiguously which HL7 value set applies.

Such coded attributes that are designated "structural" codes must be assigned the CS restriction.

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

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

5.1.3.2 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 encoded data (ED) type.

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

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.

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

5.1.3.3 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.

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

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.

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

 Copyright © Health Level Seven, Inc. All rights reserved.

5.2 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.

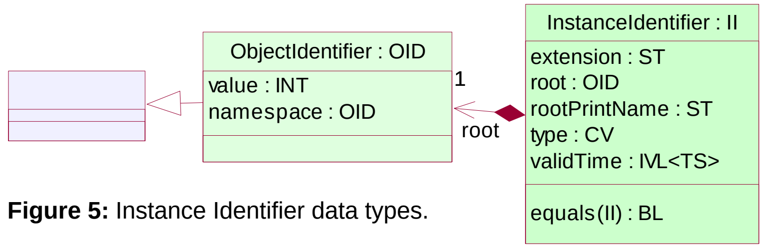


Figure 5: Instance Identifier data types.

5.2.1 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 objdect 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

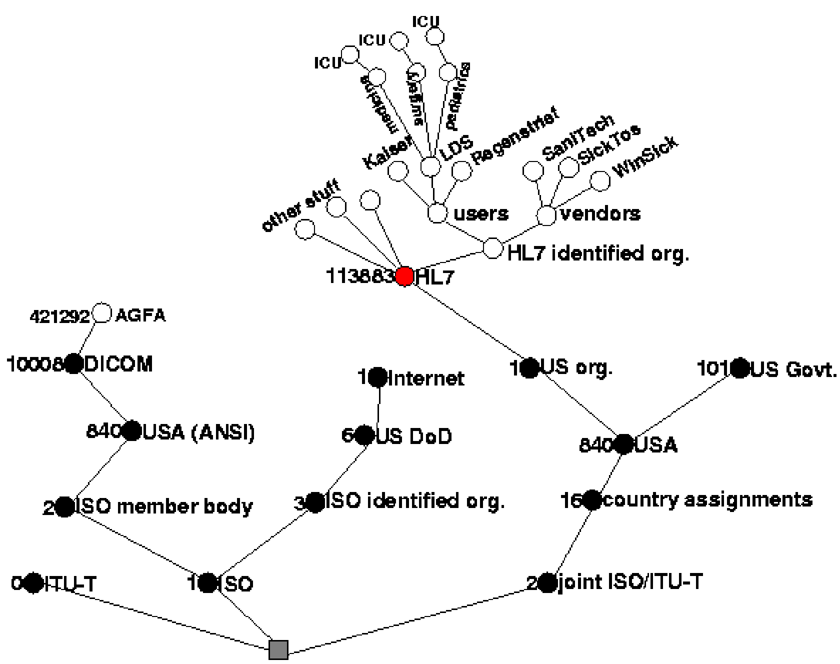


Figure 6: Example for a tree of ISO object identifiers. HL7's OID is 2.16.840.1.113883.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

Figure 6 shows part of the identifier tree that includes the HL7 OID. The OID of the HL7 organization is joint ISO/ITU-T (2) country assignments (16) .840.1.113883.

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.) The HL7 assigned OIDs must be used for these organizations, regardless whether these organizations have other OIDs assigned from other sources. If HL7 can confirm that such organizations have an OID assigned from other sources, HL7 may include such OIDs in its registry. HL7 will not assign another OID in its namespace for any entity for which it can confirm that an OID already exists. However, since OID assignment is highly distributed and no global OID registry exists, HL7 may not be able to detect whether an entity has a previously assigned OID.

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 can not be generalized.

```
type ObjectIdentifier alias OID extends ANY {
            INT       value;
            OID       namespace;
  literal   ST;
  demotion  LIST⟨INT⟩;
};
```

5.2.1.1 Literal Form

```
literal ST {
    OID : OID "." INT  { $.namespace().equals($1);
                         $.value().equals($3); }
        | INT          { $.value().equals($1); }
}
```

ITS Note: 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.

5.2.1.2 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 demotion to LIST⟨INT⟩ represents this path of object identifier component values from the root to the leaf.

The value and namespace properties take the opposite view. The leaf (the last object identifier component value in the list) is considered the value property and the all the preceding object identifier component values except for the leaf are considered the namespace property. The namespace property is an OID by itself.

The value/namespace view on ISO object identifiers has important semantic relevance. It represents the concepts of identifier value versus identifier assigning authority (namespace.)

 Copyright © Health Level Seven, Inc. All rights reserved.

5.2.2 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);
};
```

5.2.2.1 extension : ST

A character string value of the identifier. The extension must be unambiguous (unique) within the domain of the root OID. 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.

Open Issue: the 3-2-4 grouping in U.S. social security numbers is pure decoration and has no meaning. Some systems save space not storing the dashes and may not fill in the dashes when sending social security numbers. This means, that equivalence of two identifiers may be weaker than the equivalence of the extensions as character strings. For example, a system may has to consider the SSNs “123456789” and “123-45-6789” to be equivalent. It is therefore recommended to strip off all decorating meaningless characters when comparing extensions. However, what constitutes a meaningless character is entirely dependent upon the identifier scheme identified in the root property. Since social security numbers are numeric strings, they could also be assigned to the end of an OID. This specification will be more restrictive in the future to reduce the number of different cases.

5.2.2.2 root : OID

The root of an instance identifier guarantees the uniqueness of the identifier. The root alone may be the entire unique identifier, an extension value is not needed.

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.

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 and OID specifically registered for an identifier scheme.

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

5.2.2.3 assigningAuthorityName : ST

This is a human readable name or mnemonic for the assigning authority. 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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.

5.2.2.4 validTime : IVLáTSñ

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 9.2f). This means that the data types HXIT⟨II⟩ and II are the same.

5.2.2.5 type : CV

A code identifying the type of identifier. For example, codes to represent the US National Provider ID, US National Payor ID, US Health Care ID, medical record number, social security number.

> The purpose of these identifier types is to provide some guidance for use to applications. However, no complete terminology of identifier types exists, and it is unlikely that it will ever exist (identifier "types" depend on the information model, and local practices.) Most processing logic should depend on the assigning authority. For example, to find a U.S. Social Security Number, one should look for the OID defined by HL7 for the U.S. Social Security Administration.
>
> **Open Issue: This property needs a vocabulary domain, which is probably very hard to define. If this property does not have a satisfactory reasonable value domain by summer of 2001, it will be deleted.**

5.2.2.6 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)));
}
```

5.3 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.

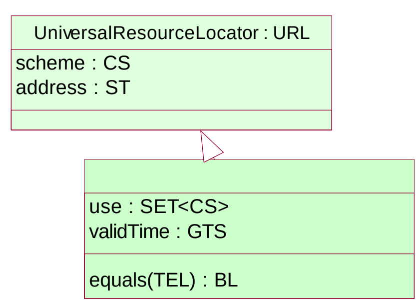


Figure 7: Telecommunication Address and URL data types.

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 encoded data, ED.)

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

 Copyright © Health Level Seven, Inc. All rights reserved.

protocol and the contact point defined by that protocol for the resource. Notable uses of the telecommunication address data type is for telephone and telefax numbers, e-mail addresses, Hypertext references, FTP references, etc.

5.3.1 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;
};
```

5.3.1.1 Literal Form

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

5.3.1.2 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).

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 8: 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	deprecated	Host-specific local file names [RCF 1738]. Note that the file scheme works only for local files. There is little use for exchanging local file names between systems, since the receiving system likely will not be able to access the file.
telnet	other	Reference to interactive sessions [RFC 1738]. Some sites, (e.g., laboratories) have

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

		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.

5.3.1.3 address : ST

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

5.3.1.4 Telephone and FAX addresses.

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); };

    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.

5.3.2 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;
```

 Copyright © Health Level Seven, Inc. All rights reserved.

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

5.3.2.1 validTime : GTS

This General Time Specification (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 (Section 9.2f). Thus, HXIT⟨TEL⟩ is a simple restriction of TEL.

5.3.2.2 use : SET⟨CS⟩

The purpose of the use code is to advise a system or user which telecommunication address in a set of like addresses to select for a given telecommunication need.

Table 9: 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.

Note: 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.

5.3.2.3 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));
}
```

5.4 Postal and Residential Address (AD)

The postal and residential 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 (postal address), or to allow a person to physically visit that address (residential address).

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

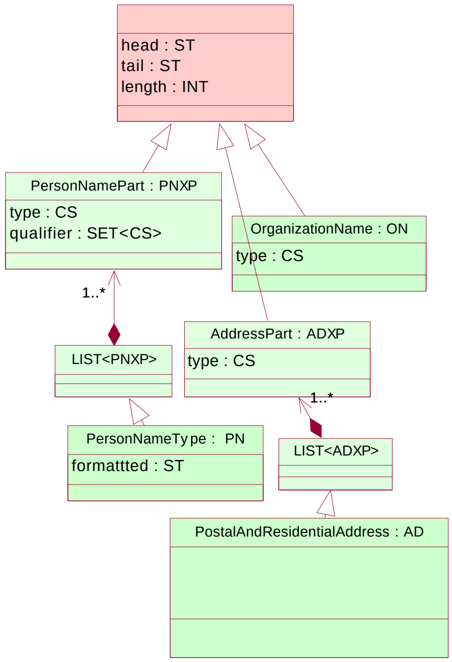


Figure 8: Data types for Person and Organization Names, and Postal and Residential Addresses are all extensions of a character string. These data types are primarily interpreted by humans and humans depend on certain form of representing them. For example, Addresses are broken into lines, where the fine structure of each line is often too complex to track in information systems (e.g., a house number along a street is often not separated from the name of the street.) The structure of these data types allows an "incremental markup" of the elements to a variable degree of precision and conforming to different culturally defined styles of representing names and addresses.

For example, a post box address is a postal address but not a residential address. Most residential addresses are also postal addresses. The residential address 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.

The postal and residential address data type is essentially a sequence of address part values.

5.4.1 Address Part (ADXP)

An address part is essentially a character string that may have a type-tag signifying it's 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, which is indicated by special line-break tokens.

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

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.

5.4.1.1 type : CS

The type of an address part indicates whether an address part is the ZIP code, city, country, post box, etc. If the type is NULL the address part is unclassified and simply appears on the label as is.

Table 10: 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

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.

5.4.2 Properties of Postal and Residential Addresses

Addresses are essentially a sequence of address parts, but adds a “use” code for information about when and if 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.

```
type PostalAndResidentialAddress alias AD extends LIST〈ADXP〉 {
            GTS        validTime;
            SET〈CS〉    use;

            BL         equals(AD x);

            ST         formatted;
};
```

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*.

5.4.2.1 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 (Section 9.2f). Thus, HXIT〈AD〉 is a simple restriction of AD.

5.4.2.2 use : SETáCSñ

The address use code is a set of indicators what a given address is to be used for:

Table 11: Address Use Code

Concept	Code	Implies	Definition
residential address	RES		Used primarily to visit an address.
postal address	PST		Used to send mail.
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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

5.4.2.3 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(TEL x, y) x.nonNull.and(y.nonNull) {
  x.equals(y).equals(((LIST〈ADXP〉)x).equals((LIST〈ADXP〉)y));
}
```

5.4.2.4 Formatting Addresses

This address data type’s main purpose is to capture postal and residential 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.

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 whitespace. The following six rules govern the setting of whitespace.

1. Whitespace 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 whitespace 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 whitespace 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 whitespace 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 two valid encodings

```
〈AD purpose=“RES”〉
    〈LIT〉1050 Wishard Blvd, RG 5th floor〈/LIT〉〈DEL/〉
    〈LIT〉Indianapolis, IN 46240〈LIT〉
〈/AD〉

〈AD purpose=“RES”〉
    〈STR〉1050 Wishard Blvd〈/STR〉〈ADL〉RG 5th floor〈/ADL〉〈DEL/〉
    〈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

 Copyright © Health Level Seven, Inc. All rights reserved.

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.

5.5 Person Name (PN)

A person name data value specifies one full name of a person. A name such as “Jim Bob Walton, Jr.” is one instance of the person name type (PN). The parts of this name “Jim”, “Bob”, “Walton”, and “Jr.” are person name parts (PNXP). A person name is simply a list of person name parts.

5.5.1 Person Name Part (PNXP)

Person names are sequences of character string tokens that may have a tag signifying the role of the token. Typical name parts that exist in about every name are given names, and family names, other part types may be defined culturally.

```
protected type PersonNamePart alias PNXP extends ST {
             CS         type;
             SET⟨CS⟩    qualifier;
};
```

5.5.1.1 type : CS

A person name name part has a type, such as given name vs. family name and name of public records vs. nickname. The type code may not be available for unknown person names.

Example: would you know what is the given name and what is the family name in “Rogan Sulma”?

Table 12: 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 MID	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). Suffices can not 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.

5.5.1.2 qualifier : SET áCSñ

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 13: 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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

		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 es 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.
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: a person may have multiple names as defined through the RIM class Person_name, which is outside the scope of this specification.

5.5.2 Properties of Person Name

Person names have no additional properties that add information to the sequence of person name parts. The property “formatted” has a character string value with the formatted person name.

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*.

```
type PersonNameType alias PN extends LIST⟨PNXP⟩ {
            ST        formatted;
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

5.5.2.1 Formatting Person Names

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

Person names are ordered lists of person name parts. Each person name part is printed in the order of the list from left to right (or in any other language-specific reading direction.) Every person name part (except for those marked "invisible") is printed. Most person 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.
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 person 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 person names in an XML encoded form, where the XML tag is the person name part type and the data content is the person 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:

```
〈PN〉
    〈GIV〉John〈/GIV〉
    〈GIV〉W.〈/GIV〉
    〈FAM〉Doe〈/FAM〉
〈/PN〉
```

none of the special qualifiers need to be mentioned if they are not known. The next example shows extensive use of multiple given names, prefixes, suffixes, for academic degrees, nobility titles, *vorvoegsels* ("van"), and professional designations.

```
〈PN〉
    〈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〉
〈/PN〉
```

5.6 Organization Name (ON)

A name for an organization, such as "Health Level Seven, Inc." An organization name is essentially a character string, extended with a tag that indicates what kind of organization name it is.

```
type OrganizationName alias ON extends ST {
            CS          type;
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

5.6.1.1 type : CS

This is a code identifying the style of the organization name.

Table 14: Organization Name Type

Concept	Code	Definition
Legal	L	The full legal name of the organization as used in public records.
Alias	A	An alias, typically a shorter name than the legal name. This is the default..
Stock exchange	ST	A stock market ticker symbol.

 Copyright © Health Level Seven, Inc. All rights reserved.

6 QUANTITIES

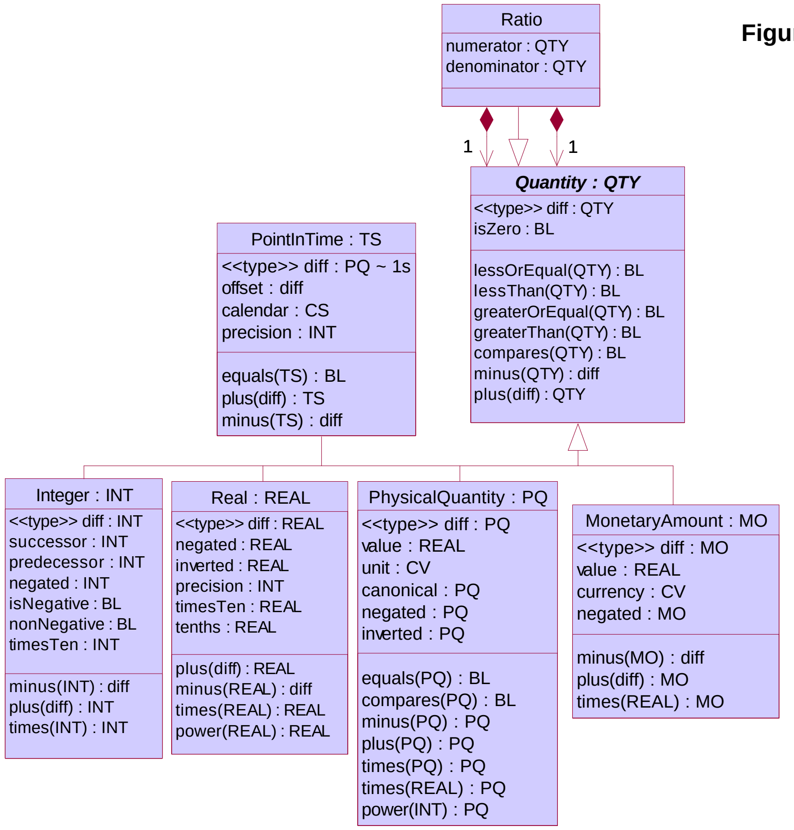


Figure 9: Data types for quantities.

6.1 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);
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

6.1.1.1 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 ($x \leq y$ or $y \leq x$).

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)));
};
```

6.1.1.2 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*. That difference *D* must be meaningful independently from the values *x* and *y*. This independence exists if one can meaningfully derive a value *v* given another value *u* such that *D* would also be the difference between *u* and *v*. The judgement for what is *meaningful* can not be defined formally.

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.

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.

 Copyright © Health Level Seven, Inc. All rights reserved.

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

6.2 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 {
            INT       successor;
            INT       predecessor;

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

            INT       times(INT x);
            INT       timesTen();

  literal   ST;
  promotion REAL;
};
```

6.2.1.1 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.

The semantics of integer numbers can be defined by complete induction using only the value *zero* and the primitive operations successor (successor) and predecessor (predecessor). 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 forms. The operation timesTen is a primitive needed for the definition of the literal form, since the concept of 10 must be defined before the semantics of the literal form “10” can be defined.

```
invariant(INT x, y) where x.nonNull.and(y.nonNull) {
  x.successor.greaterThan(x);
  y.greaterThan(x).and(y.lessThan(x.successor)).not;
  x.successor.predecessor.equals(x);
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
  x.plus(0).equals(x);
  y.greaterThan(0)
     .implies(x.plus(y).equals(x.plus(y.predecessor).successor);
  y.lessThan(0).implies(x.plus(y).equals(x.plus(y.successor).predecessor);

  x.plus(x.negated).isZero;
  x.minus(y).equals(x.plus(y.negated));
  x.nonNegative.equals(0.lessOrEqual(x));
  x.nonNegative.equals(x.isNegative.not);

  x.times(0).equals(0);
  x.times(1).equals(x);
  x.times(-1).equals(x.negated);
  y.greaterThan(1)
     .implies(x.times(y).equals(x.times(y.predecessor).plus(x)));
  y.lessThan(-1).implies(x.times(y).equals(x.times(y.negated).negated));

  x.timesTen.equals(x.times(10));
};
```

6.2.1.2 Literal Form

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

```
INT.literal ST {
  INT : uint                 { $.equals($1); }
      | "+" uint             { $.equals($2); }
      | "-" uint             { $.equals($2.negated); };

  INT uint : digit           { $.equals($1); }
           | number digit    { $.equals($1.timesTen().plus($2)); };

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

6.3 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

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      timesTen;
            REAL      tenths;
            REAL      power(REAL x);

  literal   ST;
            INT       precision;

  demotion  INT;
  promotion PQ;
  promotion RTO;
};
```

6.3.1.1 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 10.4.

The rules for what digits are significant are as follows:

1. All non-zero digits are significant.
2. All zeroes to the right of a significant digit are significant.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

3. 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 10.4 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 –

6.3.1.2 Algebraic Operations

Since the real number data type includes all of the semantics of the mathematical real number concept, the basic operations plus (addition), times (multiplication) and power (exponentiation) 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 forms. The operations timesTen and times oneTenth are primitives needed for the definition of the literal form, since the concept of 10 (0.1) must be defined before the semantics of the literal form “10” (“0.1”) can be defined. Unlike the integer numbers, the real numbers semantics is not inductively defined.

```
invariant(REAL x, y, z)
    where x.nonNull.and(y.nonNumm).andz.nonNull) {
  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 */
```

 Copyright © Health Level Seven, Inc. All rights reserved.

```
  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));

  x.timesTen.equals(x.times(10));
  x.timesTen.tenths.equals(x);

  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);
};
```

6.3.1.3 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)); };
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
  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.timesTen.plus($2));
                                    $.precision.equals(
                                      $1.precision.successor; };

  REAL fractional
      : digit                     { $.equals($1.tenths);
                                    $.precision.equals(1); }
      | digit fractional          { $.equals($1.plus($2.tenths));
                                    $.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.

6.4 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.

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. atios.

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.

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

  demotion  REAL;
  demotion  PQ;
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

6.4.1.1 numerator (QTY)

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

6.4.1.2 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();
};
```

6.4.1.3 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 a denominator INT 1 is assumed.

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

6.5 Physical Quantity (PQ)

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

```
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);
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
  literal   ST;
  demotion  REAL;
};
```

6.5.1.1 value : REAL

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

6.5.1.2 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.

6.5.1.3 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, nor 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));
};
```

6.5.1.4 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

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**) 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 */
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
  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®));
  };
};
```

6.5.1.5 Literal Form

The literal form for a physical quantity is a real number literal followed by a single 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($3); }

  CS unit : ST         { $.value.equals($1);
                         $.codeSystem.equals(2.16.840.1.113883.3.2); };
};
```

6.6 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.

```
MO.interface 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;
};
```

6.6.1.1 value : REAL

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

 Copyright © Health Level Seven, Inc. All rights reserved.

> 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.

6.6.1.2 currency : CS

The currency unit as defined in ISO 4217.

Table 16: Selected ISO 4217 currency codes.

Country	Currency	Code
Argentina	Argentine Peso	ARS
Australia	Australian Dollar	AUD
Brazil	Brazilian Real	BRL
Canada	Candian Dollar	CAD
Chile	Unidades de Formento	CLF
China	Yuan Renminbi	CNY
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
Mexico	Mexican Nuevo Peso	MXN
Netherlands	Netherlands Guilder	NLG
New Zealand	New Zealand Dollar	NZD
Philippines	Philippine Peso	PHP
Russian Federation	Russian Ruble	RUR
South Africa	Rand	ZAR
Spain	Spanish Peseta	ESP
Switzerland	Swiss Franc	CHF
Thailand	Baht	THB
United Kingdom	Pound Sterling	GBP
United States	US Dollar	USD

6.6.1.3 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 **Error! Reference source not found.**.) 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);
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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);
};
```

6.6.1.4 Literal Form

```
MO.literal ST {
  MO : value “ ” currency     { $.value.equals($1);
                                $.currency.equals($3); }

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

  CS currency : ST            { $.currency.value.equals($1);
                                $.currency.codeSystem
                                   .equals(2.16.840.1.113883.3.3); }
};
```

6.7 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, natural time is independent from calendars. The semantic properties of point in time are best described by its 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;
```

 Copyright © Health Level Seven, Inc. All rights reserved.

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

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

6.7.1.1 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.

6.7.1.2 calendar : CS

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

Table 17: 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.

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.

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.

Traditionally mankind has measured the even flow of time in various cycles defined by calendars. Such cycles in the western calendar are years, months, days, hours, minutes, seconds. Some of these cycles are synchronized and some are not. Traditionally calendars have been defined based on astronomical phenomena, however, calendar years, months and days are not attached directly to astronomical phenomena. The closest fit is the calendar day to the solar day, but the calendar month is definitely not the same as a lunar (synodal) month.

Figure 7 below visualizes a calendar as a trajectory summed up from four such cyclical movements, year, month, day and hour. Imagine a clock that measures those cycle, but 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. After rolling the time axes into the traditional cycles, a calendar expresses time as a sequence of integer counts of cycles, e.g., for year, month, day, hour, etc.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

A for the purpose of defining the literal form based on the calendar two private data types, Calendar and CalendarCycle, 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.

A calendar is defined as a set of calendar cycles. A calendar has a name and a code. The head of the calendar is the largest calendar cycle 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 decimal digits occupied in the calendar expression. The property *start* specifies where counting starts

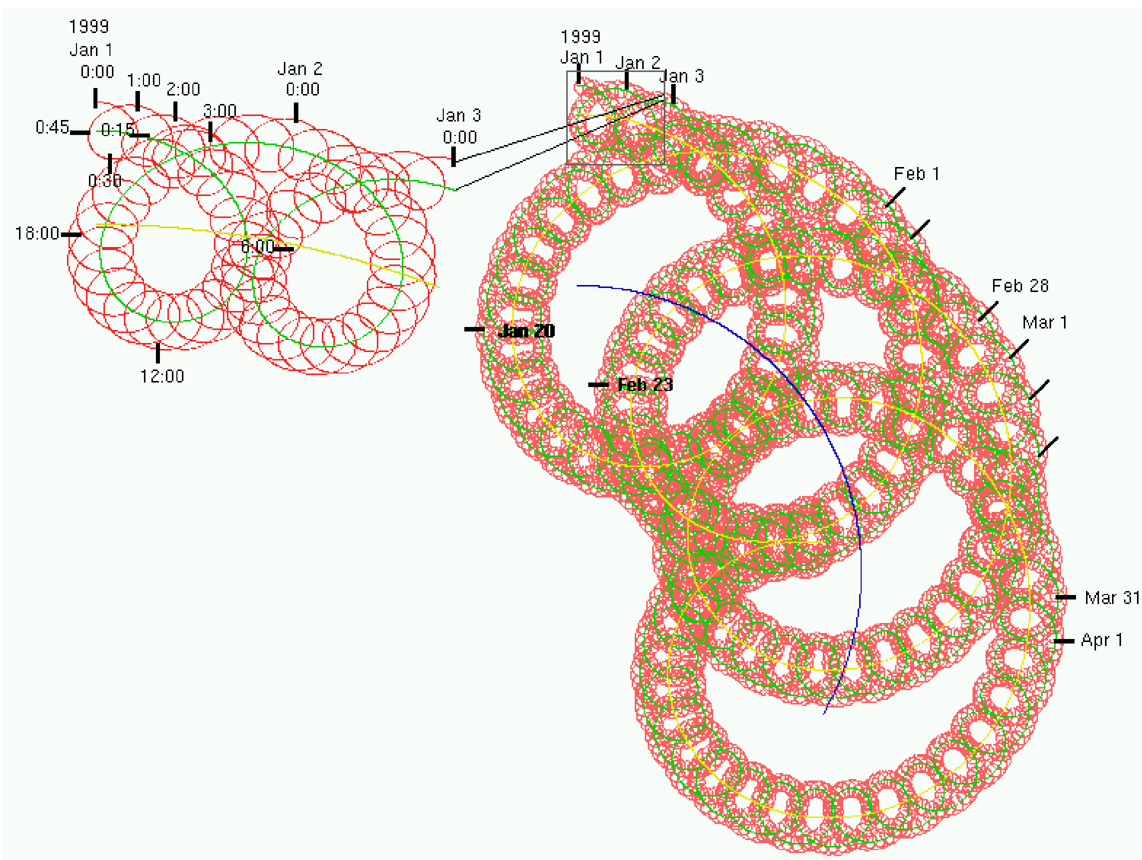


Figure 10: A calendar "rolls" the time axis into a complex convolute according to the calendar periods year (blue), month (yellow), day (green), hour (red), etc. The cycles need not be aligned, for example, the week (not shown) is not aligned to the month.

Copyright © Health Level Seven, Inc. All rights reserved.

(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 18 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.

Table 18: 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
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*		

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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.

6.7.1.3 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 10.4.

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.

6.7.1.4 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.)

6.7.1.5 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

```
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));
};
```

6.7.1.6 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 18).

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 18. Periods with lower order numbers stand to the left of periods with higher order numbers. Periods with no assigned order number can not occur in the calendar expression for points in time.

The “Counter/digits” column of Table 18 specifies the exact number of digits for the counter number for any period.

Thus, Table 18 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, “`2000040103 1520.34`” 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; };
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
  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)); };
}
```


Copyright © Health Level Seven, Inc. All rights reserved.

7 GENERIC COLLECTIONS

This section defines data types that can “collect” other data values, Set, Sequence, Bag and Interval. These collection types are are defined as generic (parameterized) types. The concept of generic types is described in Section 2.5.

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.

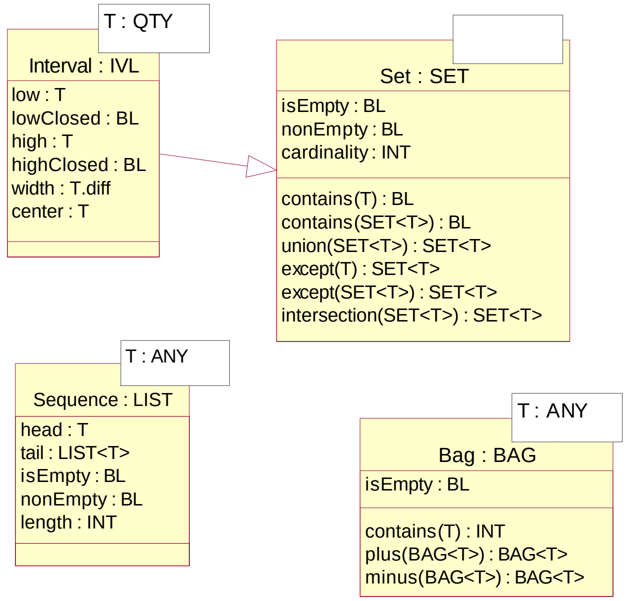


Figure 11: Generic Collection Data Types

7.1 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);

  promotion SET⟨T⟩    (T x);
};
```

7.1.1.1 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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

7.1.1.2 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).

7.1.1.3 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)));
};
```

7.1.1.4 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)));
            });
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

7.1.1.5 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));
            });
};
```

7.1.1.6 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)));
            });
};
```

7.1.1.7 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
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.

7.1.1.8 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;
};
```

7.2 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;

  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.

 Copyright © Health Level Seven, Inc. All rights reserved.

```
invariant(LIST〈T〉 x) 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)));
};
```

7.2.1.1 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); };
  LIST〈T〉 elements
         : T ";" elements             { $.head.equals($1);
                                        $.tail.equals($3); }
         | T                          { $.head.equals($1);
                                        $.tail.isNull; };
};
```

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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

> 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.

7.2.1.2 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.isNull;
};
```

7.3 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.

7.3.1.1 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);
};
```

7.3.1.2 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

```
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);
              };
            });
}
```

7.3.1.3 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)) };
};
```

7.4 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 meters 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
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;
  literal   ST;
  promotion IVL⟨T⟩     (T x);
  demotion  T;
};
```

7.4.1.1 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);
};
```

7.4.1.2 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);
};
```

7.4.1.3 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));
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

7.4.1.4 center : T

The center is defined of finite intervals and is then the arithmetic mean of the interval (low pus 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;
};
```

7.4.1.5 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);
};
```

7.4.1.6 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);
};
```

7.4.1.7 Literal Form

The literal form for the interval data type is defined such that it is as intuitive to humans a possible. Four different forms are defined:

1) the interval form using square brackets, e.g., “[3.5; 5.5[”;
2) the dash-form, e.g., “3.5–5.5”;
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[”;

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

other boundary is infinite or unknown.) The dash form, while being the weakest of all, is the most intuitive form for double bounded intervals.

```
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(false); };
```

 Copyright © Health Level Seven, Inc. All rights reserved.

```
  IVL⟨TS⟩ comparator
  | “<”  T                     { $.high.equals(T);
                                 $.high.closed(false);
                                 $.high.negativelyInfinite; }
  | “>”  T                     { $.low.equals(T);
                                 $.low.closed(false);
                                 $.low.positivelyInfinite; }
  | “<=” T                     { $.high.equals(T);
                                 $.high.closed(true);
                                 $.high.negativelyInfinite; }
  | “>=” T                     { $.low.equals(T);
                                 $.low.closed(true);
                                 $.low.positivelyInfinite; };
};
```

Table 19: Examples of interval literals.

literal	low closed	low	high	high closed	alternate center	alternate width
3.5-5.5	true	3.5	5.5	false	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	false	–3.5	3.5	false	0.0	7.0
-5.5--3.5	false	–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	true	N/A	∞
>-5.5	true	–5.5	∞	false	N/A	∞
[3.5;3.5]	true	3.5	3.5	true	3.5	0

7.4.1.8 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 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;
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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 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);
};
```

7.4.2 Interval of Physical Quantities (IVLáPQñ)

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 `[50 nm; 2 m]`” is also allowed.

 Copyright © Health Level Seven, Inc. All rights reserved.

7.4.3 Interval of Points in Time (IVLáTSñ)

A special literal form for interval of time is defined to allow an abbreviated dash-form notation where the low boundary does not need to repeat digits on the left that are the same as for the high boundary. The low boundary is right-aligned and the digits to the left copied from the high boundary. This is a simple string operation and not shown formally here.

Example: May 12, 2000 from 8 to 9:30 PM is "`200005122000-2130`". However, note that May 12, 2000 9:30 PM to May 13, 2000 8 AM is "`200005122130-230800`", and not "`200005122130-0830`".

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

8 TIMING SPECIFICATION

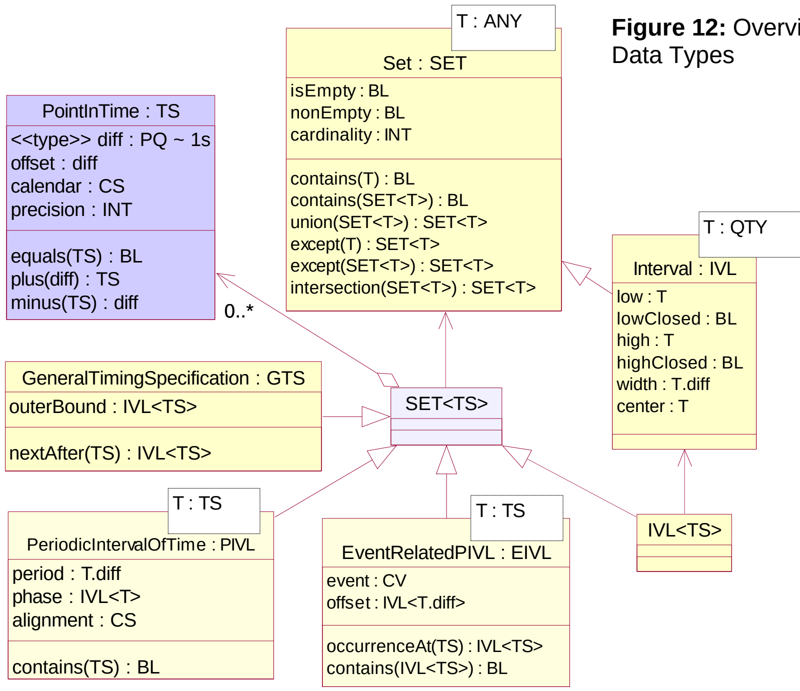


Figure 12: Overview of Timing Specification Data Types

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 adds 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.

8.1 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.

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;
```

 Copyright © Health Level Seven, Inc. All rights reserved.

```
            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.

8.1.1.1 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;
};
```

8.1.1.2 Phase : IVL〈T〉

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);
};
```

8.1.1.3 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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. 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.

8.1.1.4 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));
     });
};
```

8.1.1.5 Literal Form

There are two forms of literals for periodic intervals of time:

1. generic form, continuous: ⟨*phase* : IVL⟨T⟩⟩ / ⟨*period* : T.diff⟩ [**@** ⟨*alignment*⟩] [**IST**].
2. calendar pattern, aligned: ⟨*anchor*⟩ [⟨*calendar digits*⟩] / ⟨*number* : INT⟩ [**IST**]

The generic form is used to reflect all semantic properties directly.

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.

The continuous form is used to specify calendar-aligned timing more intuitively using calendar patterns. 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. The calendar digits may also omit digits on the right.

For example: "M0219" is February 19 the entire day every year. "M02191100-1110" is February 19, 11:00 to 11:10 AM.

When digits are omitted on the right this means the interval from lowest to highest for these digits. Other intervals can be specified using a dash followed by calendar digits matching the preceding calendar expression from the right (see interval of point in time, Section 7.4.3)

 Copyright © Health Level Seven, Inc. All rights reserved.

For example: “`M0219`” is February 19 the entire day from 0:00 to 23:59:59.999... which could have been specified as “`D0219-20`” (the high boundary is assumed to be open.) “`M02191100-1110`” is February 19, 11:00 to 11:10 AM. This is also equal `M02191100-191110`” and “`M02191100-02191110`” as the calendar digits after the dash are right aligned to the calendar digits before the dash.

With the calendar pattern form, the calendar alignment is the right-most calendar period for which digits are provided.

For example: “`M0219`” is February 19 every year. This periodic interval has the February 19 of any year as its phase, a period of one year, and alignment month of the year (MY).

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 19th 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 after the calendar period identifier no calendar digits follow, the pattern matches any date. The integer number following the slash indicates the length of the cycle.

For example: “`CD/2`” is every other day, “`H/8`” is every 8th hour.

`IST`” to indicate that within the larger calendar cycle (e.g., day for hour of the 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.)

The syntax and semantics of these periodic intervals are not yet formally specified in the body of this specification.

Table 20: Examples for literal expressions periodic intervals of time.

Literal Expression		Meaning
`M09`		September
`MY09`		September (using explicit two letter code)
`M0915`		September 15
`M091516`		September 15 at 4 PM
`M09151630`		September 15 at 4:30 PM
`M0915163034.12`		September 15 at 4:30:34.12 PM
`M/2`		every other month of the year (January, March, …)
`M04-09`		April 1 to September 30
`J6`		every Saturday
`DW6`		every Saturday (using explicit two letter code)
`J/2`		every other day of the week (Tuesday, Thursday, Saturday)
`J1-5`		Monday to Friday
`W/2`		every other week (continuous)
`WY/2`		every other week of the year
`WY15`		the 15th calendar week of every year
`WM2`		the second week of the month
`DY128`		the 128th day of the year
`[10 min]/(2 d)`		every other day for 10 minutes.
`H/8`		every eighth hour (each time a 60 minutes interval)
`H/8 IST`		three times a day at institution specified times
Abbreviation	**Normal form**	
`BID`	`H/12 IST`	two times a day at institution specified time
`TID`	`H/8 IST`	three times a day at institution specified time
`QID`	`H/6 IST`	four times a day at institution specified time

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

8.2 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;
};
```

8.2.1.1 Event : CV

The event is 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 21: 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

8.2.1.2 Offset : IVLáT.diffñ

The offset is 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.)

8.2.1.3 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

 Copyright © Health Level Seven, Inc. All rights reserved.

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);
                       });
};
```

8.2.1.4 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]`”.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

8.3 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:

1) 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 timing covered by the GTS value.

2) 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.

3) 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.

4) 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 one. 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 id 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.

```
type GeneralTimingSpecification alias GTS extends SET⟨TS⟩ {
            IVL⟨TS⟩    outerBound;
            IVL⟨TS⟩    nextAfter(TS 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. Periodic intervals and event-related intervals are defined in the next two sub-sections.

8.3.1.1 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 next-after property maps to every point in time the greatest occurrence interval that begins later than the given point in time.

This property is currently not completely formally defined. It is specified here to mark what is probably the GTS' most important use case: The GTS is used to specify and then generate a repeated schedule of activities.

```
invariant(GTS x, TS t, IVL⟨TS⟩ o) where o.equals(x.nextAfter(t)) {
  x.contains(o);
  forall(IVL⟨TS⟩ p) where x.contains(p) {
    p.contains(o).implies(p.equals(o));
  };
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

A GTS value can be converted into a generic Sequence of time intervals (LIST⟨IVL⟨TS⟩⟩) where each occurrence interval is fully specified:

8.3.1.2 Outer Bound Interval

A GTS value has one outer bound interval that is the least common superset of all occurrence intervals.

```
invariant(GTS x, IVL⟨TS⟩ b) where b.equals(x.outerBound) {
  b.contains(x);
  forall(TS e) where x.contains(e) {
    e.lessOrEqual(b.high);
    e.greaterOrEquals(b.low);
  };
};
```

8.3.1.3 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. Unions are speechified by a semicolon-separated list. Intersections are specified by a whitespace-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.

This literal form is specified based on the simpler time set data types interval, periodic interval, and event related periodic interval.

```
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); };

  SET⟨TS⟩ exclusion
  : exclusion “\” intersection { $.equals($1.except($3)); };

  SET⟨TS⟩ intersection
  : factor “ ” intersection   { $.equals($1.intersection($3)); }
  | factor;                   { $.equals($1); }

  SET⟨TS⟩ factor
  : IVL⟨TS⟩                   { $.equals($1); }
  | PIVL⟨TS⟩                  { $.equals($1); }
  | EIVL⟨TS⟩                  { $.equals($1); };
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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 22: 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 time)
`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.

The following Table 23 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.

Table 23: Abbreviations for General Timing Specifications

Code	Definition	Equivalent
AM	Every morning at institution specified times.	`H0600-1200 IST`
PM	Every afternoon at institution specified times.	`H1200-1800 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 the Christian 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`

Copyright © Health Level Seven, Inc. All rights reserved.

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 not 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

9 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`**`〈type 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.

9.1 Annotated (ANT)

Annotated〈T〉 is a generic data type extension supporting arbitrary free-text annotations (note) for any value of type T. The data type of the note property is CE, meaning that the note may be free text (usually) or may alternatively be coded.

```
template〈type T〉
type Annotated alias ANT extends T {
            CE        note;
};
```

Note that annotations are annotations of specific values. It is improper use of annotations to tag information to values which belong to another structure. For example, “specimen hemolyzed” is a property of a specimen, not of an observation value, and yet, in HL7 v2.x it was common to send this as an annotation to the observation segment.

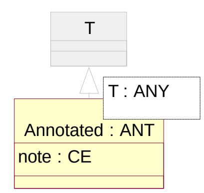


Figure 13: Annotation Extension

Annotations must not be used when there are methods defined that are more robust. In HL7 v2.x installations, there has been a great number of annotations, about reference ranges or interpretation of values, recommendations, and more. In HL7 v3 most of this can – and **must** – be communicated in properly defined data structures. For example, recommendations may be treated as Service recommendation objects. Knowledge about interpretation can be presented in a defined text attribute, or as detailed knowledge structures.

Another problematic example is the utterance “forwarded to reference lab.” This is likely not an annotation of any specific data attribute, but rather a complex statement about the specimen or an order. However, this is an interesting case that may or may not be covered by properly standardized HL7-structures. What this shows is that such cases should be brought to the attention of the standards committee rather than hidden in some code tagged to a data value to which it doesn’t belong. In general, annotations should be used sparingly – almost never on a routine basis –, or else are an indication for a use case that should be given proper attention in the information model.

> The domain of annotation notes can not be defined or circumscribed; every concept could potentially be used as an annotation. Therefore, coded annotations are very likely to be non-standardized and non-interoperable.
>
> HL7 does not define a standard domain of coded annotations. In the HL7 methodology, the use cases and information requirements are explicitly modeled as well defined data elements. Instead of using coded annotations, HL7 users and implementers should reuse this same HL7 methodology to extend the standard, e.g., adding new classes, fields, messages, etc. Such extended messages are not standard HL7 messages, but are still clearer and better than lengthy tables of coded annotations; furthermore HL7 extensions may be fed into the HL7 standardization process.
>
> Coded annotations should therefore be only used casually and temporary, to provide immediate remedy to an urgent business need, never to define long-lasting solutions.

9.1.1.1 note : CE

This is the annotation as free-text or in coded form. Free text notes are conveyed in the *original text* property of the CE value, while the code property is NULL. The annotation is meant for display to

 Copyright © Health Level Seven, Inc. All rights reserved.

users or administrators or for computer-processing of exceptions. The annotation must not be used for routine data exchange that is covered elsewhere by HL7. Example: “original was illegible.”

9.2 History Item (HXIT)

This generic data type extension is 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⟩.

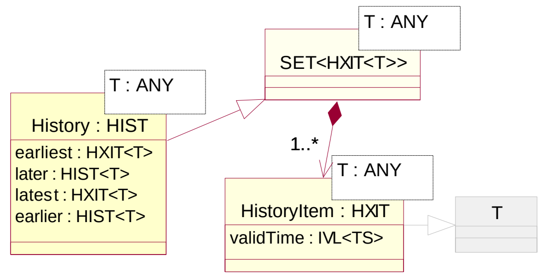


Figure 14: History Item and History

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.

```
template⟨type T⟩
type HistoryItem⟨T⟩ alias HXIT⟨T⟩ extends T {
            IVL⟨TS⟩    validTime;
};
```

9.2.1 validTime : IVL áTSñ

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.

9.3 History (HIST)

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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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;

  forall(HXIT 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

10 UNCERTAINTY AND PROBABILITY

This section defines a suite of generic data type extensions that are used to express uncertainty, probability, and frequency of other data values. This includes a qualitative annotation of data with a "confidence code" (UVN), but concentrates on quantitative specification of uncertainty using probabilities.

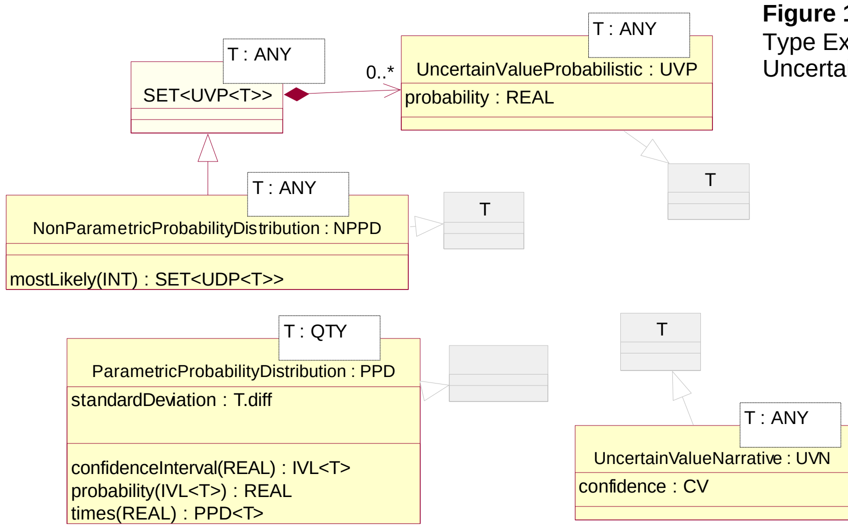


Figure 15: Generic Data Type Extensions for Uncertain Information

10.1 Uncertain Value – Narrative (UVN)

This is a generic data type extension to specify one uncertain value tagged with a coded confidence qualifier. The confidence qualifier is a coded representation of the confidence as used in narrative utterances, such as "probably", "likely", "may be", "would be supported", "consistent with", imately", etc.

```
template ⟨type T⟩
type UncertainValueNarrative⟨T⟩ alias UVN⟨T⟩ extends T {
            CV          confidence;
};
```

10.1.1.1 confidence : CV

The confidence qualifier assigned to the value.

No standard terminology of confidence qualifiers exists, and it is unclear whether it can ever possibly exist. For instance, is "probably" more or less confidant than "could be"? The use of the confidence qualifier is thus only suited to communicate information between humans.

10.2 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.

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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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⟨type 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 $x_i \in X$.

10.2.1.1 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.

10.3 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, veterinary 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

```
template⟨type 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 can (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⟨x⟩ d, e; SET⟨UVP⟨x⟩⟩ m; INT n)
      where x.contains(d)
       .and(m.equals(x.mostLikely(n)))
       .and(m.contains(e)) {
    e.greaterOrEqual(d).or(m.contains(d));
  };
};
```

10.4 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.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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 would be translated to $f'(x')$ that maps a length x' to a probability 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.

10.4.1.1 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 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.”

10.4.1.2 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 24 lists the defined probability distributions. Many distribution types are defined in terms of special parameters (e.g., the parameters a and b for the g-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 24 contains the relationship between the special parameters and the mean m and standard deviation s.

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 24: 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 $2 s \sqrt{3}$. Thus, the uniform distribution assigns the probability densities $f(x) = (2 s \sqrt{3})^{-1}$ to values $m - s \sqrt{3} \geq x \leq m + s \sqrt{3}$ 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 m and standard deviation s. Note however that mean m and standard deviation s 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 m_{log} and s_{log} relate to the mean m and

Copyright © Health Level Seven, Inc. All rights reserved.

		standard deviation s of the data value through s_{log}^2 = log (s^2/m^2 + 1) and m_{log} = log m – $s_{log}^2/2$.
g (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 g-distribution has a two parameters a and b. The relationship to mean m and variance s^2 is $m = a\ b$ and $s^2 = a\ b^2$.
exponential	E	Used for data that describes extinction. The exponential distribution is a special form of g-distribution where a = 1, hence, the relationship to mean m and variance s^2 are $m = b$ and $s^2 = b^2$.
c^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 c^2-distribution is n, so called the *number of degrees of freedom* (which is the number of independent parts in the sum). The c^2-distribution is a special type of g-distribution with parameter $a = n/2$ and b = 2. Hence, $m = n$ and $s^2 = 2\ n$.
t (Student)	T	Used to describe the quotient of a normal random variable and the square root of a c^2 random variable. The *t*-distribution has one parameter n, the number of degrees of freedom. The relationship to mean m and variance s^2 are: m = 0 and $s^2 = n / (n - 2)$
F	F	Used to describe the quotient of two c^2 random variables. The *F*-distribution has two parameters n_1 and n_2, which are the numbers of degrees of freedom of the numerator and denominator variable respectively. The relationship to mean m and variance s^2 are: $m = n_2 / (n_2 - 2)$ and $s^2 = (2\ n_2\ (n_2 + n_1 - 2)) / (n_1\ (n_2 - 2)^2\ (n_2 - 4))$.
b (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 a and b are available to adjust the curve. The mean m and variance s^2 relate as follows: $m = a/\ (a + b)$ and $s^2 = ab/((a + b)^2\ (a + b + 1))$.

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.

10.4.1.3 Literal Form

The parametric probability distribution has a literal form. The general syntax is as follows:

```
PPD.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.

10.4.2 Probability Distribution over Real Numbers (PPDáREALñ)

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 is defined in this section.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

10.4.2.1 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 s is $s = l / 2$.

Table 25: Examples standard deviation computed from precision and order of magnitude.

Representation	*x*	*e*	*p*	*e - p* + 1	*l*	*s*
`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

10.4.2.2 Concise Literal Form for PPD⟨REAL⟩

Besides the generic literal form of the PPD defined in Section 10.4.1.3, 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.

 Copyright © Health Level Seven, Inc. All rights reserved.

10.4.3 Parametric Probability Distributions over Physical Quantities (PPDáPQñ)

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);
};
```

10.4.3.1 Concise Literal Form for PPDáPQñ

A concise literal form for probability distributions of physical quantities is defined based on the concise literal form of PPD⟨REAL⟩ (cf. Secion 10.4.1.3) 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⟩.

10.4.4 Probability Distribution over Time Points (PPDáTSñ)

The parametric probability distribution over time points is fully defined by the generic data type.

```
type ParametricProbabilityDistribution⟨TS⟩ alias PPD⟨TS⟩;
```

Note that the standard deviation is of type TS.diff, which is a duration (a physical quantity in the dimension of time.)

10.4.4.1 Converting a point in time (TS) to an uncertain point in time (PPDáTSñ)

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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 judgement that June 9th spans all day of June 9th with noon as the center, not midnight.

 Copyright © Health Level Seven, Inc. All rights reserved.

A OBJECT IDENTIFIERS (NORMATIVE)

Note: some details of this table, namely the branches 5 and 6 (code systems) are subject to change in the weeks of the ballot.

The HL7 root Object Identifier is

"*Joint ISO/ITU-T* (2) . *Country Assignments* (16) . *United States of America* (840) . *U.S. Organizations* (1) . *Health Level Seven* (113883)"; or "2.16.840.1.113883" for short.

Within the HL7 branch of OIDs the assignments of Table 26 are made. The Control Query Technical Committee (CQ) is charged with the ongoing maintenance of the OID assignments. For this purpose, CQ with the assistance of the HL7 headquarter will put into place a subcommittee or commissioner as well as technical infrastructure so that applications for new OID assignments can be processed in a timely manner.

Some OID branches will be assigned to the stewardship of other technical committees, which means that suggestions for additions on these branches must come from the respective steward committee.

Upon request, the CQ committee shall assign one OID to all HL7 members and users (including non-members) who do not have an OID already.

Table 26: HL7 Assigned Object Identifiers

Object Identifier	Definition
2.16.840.1.113883	Root of Health Level Seven's Object Identifier subtree.
2.16.840.1.113883.3	**Object identifiers assigned to HL7 members, users, and vendors.**
2.16.840.1.113883.4	**HL7 recognized external identifiers**
2.16.840.1.113883.4.1	United States Social Security Number (SSN). Assigned by the U.S. Social Security Administration. Note: IRS assigned ITINs are often used as drop-ins for social security numbers.
2.16.840.1.113883.4.2	United States Individual Taxpayer Identification Number (ITIN). Assigned by the U.S. Internal Revenue Service (IRS) to alien taxpayers not eligible to a social security number. ITIN are used as drop-ins for Social Security Numbers.
2.16.840.1.113883.4.3	United States Driver License Number. These identifiers are assigned by each state. The OID numbers in this branch have been assigned according to FIPS PUB 5-2 numeric state codes. U.S. territories have not been included. This branch is to be maintained such that each driver license assigning authorities has one entry, independent of political organization and independent of FIPS PUB 5-2.
2.16.840.1.113883.4.3.1	Alabama Driver License Bureau
2.16.840.1.113883.4.3.2	Alaska Driver License Bureau
2.16.840.1.113883.4.3.4	Arizona Driver License Bureau
2.16.840.1.113883.4.3.5	Arkansas Driver License Bureau
2.16.840.1.113883.4.3.6	California Driver License Bureau
2.16.840.1.113883.4.3.8	Colorado Driver License Bureau
2.16.840.1.113883.4.3.9	Connecticut Driver License Bureau
2.16.840.1.113883.4.3.10	Delaware Driver License Bureau
2.16.840.1.113883.4.3.11	District of Columbia Driver License Bureau
2.16.840.1.113883.4.3.12	Florida Driver License Bureau
2.16.840.1.113883.4.3.13	Georgia Driver License Bureau
2.16.840.1.113883.4.3.15	Hawaii Driver License Bureau
2.16.840.1.113883.4.3.16	Idaho Driver License Bureau
2.16.840.1.113883.4.3.17	Illinois Driver License Bureau
2.16.840.1.113883.4.3.18	Indiana Bureau for Motor Vehicles (BMV)
2.16.840.1.113883.4.3.19	Iowa Driver License Bureau
2.16.840.1.113883.4.3.20	Kansas Driver License Bureau
2.16.840.1.113883.4.3.21	Kentucky Driver License Bureau
2.16.840.1.113883.4.3.22	Louisiana Driver License Bureau
2.16.840.1.113883.4.3.23	Maine Driver License Bureau
2.16.840.1.113883.4.3.24	Maryland Driver License Bureau
2.16.840.1.113883.4.3.25	Massachusetts Driver License Bureau
2.16.840.1.113883.4.3.26	Michigan Driver License Bureau

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

2.16.840.1.113883.4.3.27	Minnesota Driver License Bureau
2.16.840.1.113883.4.3.28	Mississippi Driver License Bureau
2.16.840.1.113883.4.3.29	Missouri Driver License Bureau
2.16.840.1.113883.4.3.30	Montana Driver License Bureau
2.16.840.1.113883.4.3.31	Nebraska Driver License Bureau
2.16.840.1.113883.4.3.32	Nevada Driver License Bureau
2.16.840.1.113883.4.3.33	New Hampshire Driver License Bureau
2.16.840.1.113883.4.3.34	New Jersey Driver License Bureau
2.16.840.1.113883.4.3.35	New Mexico Driver License Bureau
2.16.840.1.113883.4.3.36	New York Driver License Bureau
2.16.840.1.113883.4.3.37	North Carolina Driver License Bureau
2.16.840.1.113883.4.3.38	North Dakota Driver License Bureau
2.16.840.1.113883.4.3.39	Ohio Driver License Bureau
2.16.840.1.113883.4.3.40	Oklahoma Driver License Bureau
2.16.840.1.113883.4.3.41	Oregon Driver License Bureau
2.16.840.1.113883.4.3.42	Pennsylvania Driver License Bureau
2.16.840.1.113883.4.3.44	Rhode Island Driver License Bureau
2.16.840.1.113883.4.3.45	South Carolina Driver License Bureau
2.16.840.1.113883.4.3.46	South Dakota Driver License Bureau
2.16.840.1.113883.4.3.47	Tennessee Driver License Bureau
2.16.840.1.113883.4.3.48	Texas Driver License Bureau
2.16.840.1.113883.4.3.49	Utah Driver License Bureau
2.16.840.1.113883.4.3.50	Vermont Driver License Bureau
2.16.840.1.113883.4.3.51	Virginia Driver License Bureau
2.16.840.1.113883.4.3.53	Washington Driver License Bureau
2.16.840.1.113883.4.3.54	West Virginia Driver License Bureau
2.16.840.1.113883.4.3.55	Wisconsin Driver License Bureau
2.16.840.1.113883.4.3.56	Wyoming Driver License Bureau
2.16.840.1.113883.4.4	U.S. IRS Assigned Employer Identification Number EIN. An EIN is a nine-digit number (for example, “12-3456789”) assigned to sole proprietors, corporations, partnerships, estates, trusts, withholding agents, and other entities for tax filing and reporting purposes. An EIN can not be used in place of a social security number (SSN).
2.16.840.1.113883.4.5	U.S. IRS Assigned Preparer Tax Identification Number PTIN. Section 3710 of the Internal Revenue Service Restructuring and Reform Act of 1998 defines the PTIN. The PTIN has the form of an SSN. It is used as an alias SSN to identify paid preparers of tax returns.
2.16.840.1.113883.5	**HL7 maintained code systems. (Steward: Vocabulary TC)**
2.16.840.1.113883.5.1	Sex code
2.16.840.1.113883.5.2	Marital status
2.16.840.1.113883.5.*n*	HL7 version 2.x table, where *n* is the table number and *n* < 1000.
2.16.840.1.113883.5.1001	Service mood code.
2.16.840.1.113883.5.1002	Service relationship type code
2.16.840.1.113883.5.1003	Service actor type code
2.16.840.1.113883.5.1004	Service target type code
2.16.840.1.113883.5.1007	Data type code. As specified by this specification.
2.16.840.1.113883.5.1008	Flavors of null. As specified in Table 3 of this specification.
2.16.840.1.113883.5.1009	Compression codes. As specified by Table 6 of this specification.
2.16.840.1.113883.5.1010	Integrity check algorithm. As specified by Table 7 of this specification.
2.16.840.1.113883.5.1011	Telecommunication address use code. As specified in Table 9 of this specification.
2.16.840.1.113883.5.1012	Postal Address use code. As defined by Table 11 of this specification.
2.16.840.1.113883.5.1013	Postal Address part type code. As defined by Table 10 of this specification.
2.16.840.1.113883.5.1014	Person Name part type as defined by Table 12 of this specification
2.16.840.1.113883.5.1015	Person Name part qualifier as defined by Table 13 of this specification.
2.16.840.1.113883.5.1016	Organization name type code as defined by Table 14 of this specification.
2.16.840.1.113883.5.1017	Calendar type. As specified in Section Table 17 of this specification.
2.16.840.1.113883.5.1018.1	Calendar cycle one letter code. As specified in Table 18 of this specification.
2.16.840.1.113883.5.1018.2	Calendar cycle two letter code. As specified in Table 18 of this specification.
2.16.840.1.113883.5.1019	Probability distribution type code. As specified in Table 24 of this specification.
2.16.840.1.113883.5.1020	Periodic Interval of Time literal abbreviations. As specified in Table 20 of this specification.
2.16.840.1.113883.5.1021	GTS literal abbreviation. As specified in Table 23 of this specification.
2.16.840.1.113883.5.1022	Event codes for event-related periodic intervals, as specified by table Table 21.
2.16.840.1.113883.6	**External coding systems known to HL7. (Steward: Vocabulary TC)**
2.16.840.1.113883.6.1	Logical Observation Identifier Names and Codes (LOINC)

Copyright © Health Level Seven, Inc. All rights reserved.

2.16.840.1.113883.6.2	International Classification of Diseases revision 9, with Clinical Modifications (ICD 9 CM)
2.16.840.1.113883.6.3	International Classification of Diseases revision 10 (ICD 10)
2.16.840.1.113883.6.4	ICD Procedure Coding System (ICD 10 PCS)
2.16.840.1.113883.6.5	Systemized Nomenclature in Medicine (SNOMED)
2.16.840.1.113883.6.6	The Read code
2.16.840.1.113883.6.7	NAACCR Cancer Registry
2.16.840.1.113883.6.8	Unified Code for Units of Measure
2.16.840.1.113883.6.9	ISO 4217 currency code
2.16.840.1.113883.6.10	IETF MIME media types
2.16.840.1.113883.6.11	Universal Resource Locator (URL) schemes. Currently there is no single authority for URL schemes. The authority for URL scheme assignments clearly lies within IANA or W3C and it is likely that a formal URL/URI assigning authority will be formed soon.
2.16.840.1.113883.6.12	American Medical Association's Current Procedure Terminology 4 (CPT-4) codes.
2.16.840.1.113883.6.13	American Dental Association's Current Dental Terminology 2 (CDT-2) codes.
2.16.840.1.113883.6.14	Healthcare Financing Administration (HCFA) Common Procedure Coding System (HCPCS). Is composed of three "levels". Level I is CPT-4, level II is CDT-2, and level 3 are the HCPCS modifiers. Only the HCPCS modifiers are maintained by the Alpha-Numeric Editorial Panel, consisting of the Health Insurance Association of America and the Blue Cross and Blue Shield Association.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

B SUMMARY OF FORMAL DEFINITIONS

This section is a summary of the formal definition. It has been automatically extracted from the body of this specification.

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

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

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

            BL        equals(ANY x);
};

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

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

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

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

 Copyright © Health Level Seven, Inc. All rights reserved.

```
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);
}

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);
};

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

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

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

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));
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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

protected type BinaryData alias BIN extends LIST⟨BL⟩;

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

type EncodedData alias ED extends BIN {
            CS        type;
            CS        charset;
            CS        language;
            CS        compression;

            TEL       reference
            BIN       integrityCheck;
            CS        integrityCheckAlgorithm;

            ED        thumbnail;

            BL        equals(ED x);
};

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

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

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


Copyright © Health Level Seven, Inc. All rights reserved.

```
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;
}

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));
};

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

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;
};

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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

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

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

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

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)));
};

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

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

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

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

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

 Copyright © Health Level Seven, Inc. All rights reserved.

```
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);

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

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

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

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

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
type ObjectIdentifier alias OID extends ANY {
            INT        value;
            OID        namespace;
  literal   ST;
  demotion  LIST⟨INT⟩;
};

literal ST {
    OID : OID "." INT  { $.namespace().equals($1);
                         $.value().equals($3); }
        | INT          { $.value().equals($1); }
}

type InstanceIdentifier alias II extends ANY {
            ST         extension;
            OID        root;
            ST         assigningAuthorityName;
            CV         type;
            IVL⟨TS⟩    validTime;

            BL         equals(II x);
};

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

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)));
}

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

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

 Copyright © Health Level Seven, Inc. All rights reserved.

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

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

type TelecommunicationAddress alias TEL extends URL {
            GTS        validTime;
            SET⟨CS⟩    use;

            BL         equals(TEL x);
};

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

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

type PostalAndResidentialAddress alias AD extends LIST⟨ADXP⟩ {
            GTS        validTime;
            SET⟨CS⟩    use;

            BL         equals(AD x);

            ST         formatted;
};

invariant(TEL x, y) x.nonNull.and(y.nonNull) {
  x.equals(y).equals(((LIST⟨ADXP⟩)x).equals((LIST⟨ADXP⟩)y));
}
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
protected type PersonNamePart alias PNXP extends ST {
            CS          type;
            SET⟨CS⟩     qualifier;
};

type PersonNameType alias PN extends LIST⟨PNXP⟩ {
            ST          formatted;
};

type OrganizationName alias ON extends ST {
            CS          type;
};

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);
};

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)));
};
```


Copyright © Health Level Seven, Inc. All rights reserved.

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

type IntegerNumber alias INT extends QTY {
            INT       successor;
            INT       predecessor;

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

            INT       times(INT x);
            INT       timesTen();

  literal   ST;
  promotion REAL;
};

invariant(INT x, y) where x.nonNull.and(y.nonNull) {
  x.successor.greaterThan(x);
  y.greaterThan(x).and(y.lessThan(x.successor)).not;
  x.successor.predecessor.equals(x);

  x.plus(0).equals(x);
  y.greaterThan(0)
     .implies(x.plus(y).equals(x.plus(y.predecessor).successor);
  y.lessThan(0).implies(x.plus(y).equals(x.plus(y.successor).predecessor);

  x.plus(x.negated).isZero;
  x.minus(y).equals(x.plus(y.negated));
  x.nonNegative.equals(0.lessOrEqual(x));
  x.nonNegative.equals(x.isNegative.not);
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
    x.times(0).equals(0);
    x.times(1).equals(x);
    x.times(-1).equals(x.negated);
    y.greaterThan(1)
       .implies(x.times(y).equals(x.times(y.predecessor).plus(x)));
    y.lessThan(-1).implies(x.times(y).equals(x.times(y.negated).negated));

    x.timesTen.equals(x.times(10));
  };

  INT.literal ST {
    INT : uint                  { $.equals($1); }
        | "+" uint              { $.equals($2); }
        | "-" uint              { $.equals($2.negated); };

    INT uint : digit            { $.equals($1); }
             | number digit     { $.equals($1.timesTen().plus($2)); };

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

  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      timesTen;
              REAL      tenths;
              REAL      power(REAL x);

    literal   ST;
              INT       precision;
```

 Copyright © Health Level Seven, Inc. All rights reserved.

```
  demotion  INT;
  promotion PQ;
  promotion RTO;
};

invariant(REAL x, y, z)
    where x.nonNull.and(y.nonNumm).andz.nonNull) {
  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 */

  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));

  x.timesTen.equals(x.times(10));
  x.timesTen.tenths.equals(x);

  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);
};

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

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
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.timesTen.plus($2));
                                        $.precision.equals(
                                          $1.precision.successor; };

REAL fractional
       : digit                        { $.equals($1.tenths);
                                        $.precision.equals(1); }
       | digit fractional             { $.equals($1.plus($2.tenths));
                                        $.precision.equals(
                                          $1.precision.successor); };

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

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

  demotion   REAL;
  demotion   PQ;
};
```


Copyright © Health Level Seven, Inc. All rights reserved.

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

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

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;
};

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));
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
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 */
};

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);
};

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

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);
}

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)));
  };
```


Copyright © Health Level Seven, Inc. All rights reserved.

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

PQ.literal ST {
  PQ : REAL " " unit   { $.value.equals($1);
                         $.unit.equals($3); }

  CS unit : ST         { $.value.equals($1);
                         $.codeSystem.equals(2.16.840.1.113883.3.2); };
};

MO.interface 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;
};

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);
};

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);
};

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);
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
MO.literal ST {
  MO : value " " currency     { $.value.equals($1);
                                $.currency.equals($3); }

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

  CS currency : ST            { $.currency.value.equals($1);
                                $.currency.codeSystem
                                   .equals(2.16.840.1.113883.3.3); }
};

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;
};

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

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);
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

```
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);
};

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

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));
};

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); };
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
  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)); };
}

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);

  promotion SET⟨T⟩    (T x);
};

invariant(SET⟨T⟩ x) where x.nonNull {
  forall(T e) where x.contains(e) { x.nonNull; };
};

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

 Copyright © Health Level Seven, Inc. All rights reserved.

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

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)));
};

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)));
            });
};

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));
            });
};

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));
            });
};

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)));
            });
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

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

invariant(T x) {
  ((SET⟨T⟩)x).contains(x);
  ((SET⟨T⟩)x).except(x).isEmpty;
};

template⟨ANY T⟩
type Sequence⟨T⟩ alias LIST⟨T⟩ extends ANY {
             T          head;
             LIST⟨T⟩    tail;
             BL         isEmpty;
             BL         nonEmpty;
             INT        length;

  promotion  LIST⟨T⟩    (T x);
};

invariant(LIST⟨T⟩ x) x.isEmpty {
  x.head.isNull;
  x.tail.isNull;
  x.length.isZero;
};

invariant(LIST⟨T⟩ x) {
  x.nonEmpty.equals(x.isEmpty.not);
}

invariant(LIST⟨T⟩ x) where x.nonEmpty {
  x.length.equals(x.tail.length.successor);
};

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

 Copyright © Health Level Seven, Inc. All rights reserved.

```
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)));
};

LIST⟨T⟩.literal ST {
  LIST⟨T⟩ : "(" elements ")"          { $.equals($2); };
  LIST⟨T⟩ elements
         : T ";" elements             { $.head.equals($1);
                                        $.tail.equals($3); }
         | T                          { $.head.equals($1);
                                        $.tail.isNull; };
};

invariant(T x) {
  ((LIST⟨T⟩)x).head.equals(x);
  ((LIST⟨T⟩)x).tail.isNull;
};

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);
};

invariant(BAG⟨T⟩ x; T e) where x.nonNull.and(e.nonNull) {
  x.contains(e).nonNegative;
  x.isEmpty.equals(x.contains(e).isZero);
};

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)));
            });
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
    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);
                };
              });
  }

  invariant(T x) {
    ((BAG⟨T⟩)x).contains(x).equals(1);
    forall(T y) { ((BAG⟨T⟩)x).contains(y).implies(x.equals(y)) };
  };

  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;
    literal   ST;
    promotion IVL⟨T⟩     (T x);
    demotion  T;
  };

  invariant(IVL⟨T⟩ x; T e) where x.nonNull.and(x.contains(e)) {
    x.low.lessOrEqual(e);
  };

  invariant(IVL⟨T⟩ x; T e) where x.nonNull.and(x.contains(e)) {
    e.lessOrEqual(x.high);
  };

  invariant(IVL⟨T⟩ x) {
    x.low.lessOrEqual(x.high);
    x.width.equals(x.high.minus(x.low));
  };
```


Copyright © Health Level Seven, Inc. All rights reserved.

```
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;
};

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);
};

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);
};

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); };
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
  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(false); };

  IVL⟨TS⟩ comparator
  | “<”  T                     { $.high.equals(T);
                                 $.high.closed(false);
                                 $.high.negativelyInfinite; }
  | “>”  T                     { $.low.equals(T);
                                 $.low.closed(false);
                                 $.low.positivelyInfinite; }
  | “<=” T                     { $.high.equals(T);
                                 $.high.closed(true);
                                 $.high.negativelyInfinite; }
  | “>=” T                     { $.low.equals(T);
                                 $.low.closed(true);
                                 $.low.positivelyInfinite; };
};

invariant(T x) {
  ((IVL⟨T⟩)x).low.equals(x);
  ((IVL⟨T⟩)x).high.equals(x);
  ((IVL⟨T⟩)x).highClosed;
  ((IVL⟨T⟩)x).lowClosed;
};

invariant(IVL 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);
};
```


Copyright © Health Level Seven, Inc. All rights reserved.

```
type Inteval⟨PQ⟩ alias IVL⟨PQ⟩ {
            IVL⟨REAL⟩  value;
            CS         unit;
};

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);
};

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); };
};

template⟨TS T⟩
protected type PeriodicInterval⟨T⟩ alias PIVL⟨T⟩ extends SET⟨T⟩ {
            T.diff     period;
            IVL⟨T⟩     phase;
            CS         alignment;

            BL         contains(TS);
  literal   ST;
};

invariant(PIVL⟨T⟩ x) where x.nonNull {
  x.period.nonNull;
};

invariant (PIVL⟨T⟩ x) where x.nonNull {
  x.phase.nonNull;
  x.phase.width.lessOrEqual(x.period);
};
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
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)));
     });
};

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));
     });
};

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;
};

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);
};

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);
                       });
};

EIVL⟨TS⟩.literal ST {
  EIVL⟨TS⟩ : event      { $.event.equals($1); }
  | event offset        { $.event.equals($1); $.offset.equals($2); };
```

 Copyright © Health Level Seven, Inc. All rights reserved.

```
  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); };
};

type GeneralTimingSpecification alias GTS extends SET⟨TS⟩ {
            IVL⟨TS⟩    outerBound;
            IVL⟨TS⟩    nextAfter(TS x)
  demotion  LIST⟨IVL⟨TS⟩⟩;
  literal   ST;
};

invariant(GTS x, TS t, IVL⟨TS⟩ o) where o.equals(x.nextAfter(t)) {
  x.contains(o);
  forall(IVL⟨TS⟩ p) where x.contains(p) {
    p.contains(o).implies(p.equals(o));
  };
};

invariant(GTS x, IVL⟨TS⟩ b) where b.equals(x.outerBound) {
  b.contains(x);
  forall(TS e) where x.contains(e) {
    e.lessOrEqual(b.high);
    e.greaterOrEquals(b.low);
  };
};

GTS.literal ST {
  GTS : symbol                { $.equals($1); }
  | union                     { $.equals($1); };
  | exclusion                 { $.equals($1); };

  SET⟨TS⟩ union
  : intersection ";" SET⟨TS⟩  { $.equals($1.union($3)); }
  | intersection              { $.equals($1); };
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
SET⟨TS⟩ exclusion
  : SET⟨TS⟩ "\" intersection   { $.equals($1.except($3)); };

  SET⟨TS⟩ intersection
  : factor " " intersection   { $.equals($1.intersection($3)); }
  | factor;                   { $.equals($1); }

  SET⟨TS⟩ factor
  : IVL⟨TS⟩                   { $.equals($1); }
  | PIVL⟨TS⟩                  { $.equals($1); }
  | EIVL⟨TS⟩                  { $.equals($1); };
};

template⟨type T⟩
type Annotated alias ANT extends T {
            CE        note;
};

template⟨type T⟩
type HistoryItem⟨T⟩ alias HXIT⟨T⟩ extends T {
            IVL⟨TS⟩    validTime;
};

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⟩;
};

invariant ⟨HIST x⟩ where x.nonNull {
  x.nonEmpty;

  forall(HXIT 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));
```

 Copyright © Health Level Seven, Inc. All rights reserved.

```
  ((T)x).equals(x.latest);
};

template ⟨type T⟩
type UncertainValueNarrative⟨T⟩ alias UVN⟨T⟩ extends T {
            CV          confidence;
};

template⟨type T⟩
type UncertainValueProbabilistic⟨T⟩ alias UVP⟨T⟩ extends T {
            REAL        probability;
};

invariant(UVP⟨T⟩ x) where x.nonNull.and(x.probability.nonNull) {
  ((IVL⟨REAL⟩)[0;1]).contains(x.probability);
};

template⟨type T⟩
type NonParametricProbabilityDistribution⟨T⟩
    alias NPPD<T> extends SET⟨UDP⟨T⟩⟩ {
            SET⟨UDP⟨T⟩⟩ mostLikely(INT n);
};

invariant(NPPD⟨T⟩ x) where x.nonNull {
  x.nonEmpty;

x.contains(x.mostLikely(n));
  x.mostLikely(n).
  forall(UVP⟨x⟩ d, e; SET⟨UVP⟨x⟩⟩ m; INT n)
      where x.contains(d)
       .and(m.equals(x.mostLikely(n)))
       .and(m.contains(e)) {
    e.greaterOrEqual(d).or(m.contains(d));
  };
};

template⟨QTY T⟩
type ParametricProbabilityDistribution⟨T⟩ alias PPD⟨T⟩ extends T {
            T.diff      standardDeviation;
            CS          type;
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

```
            IVL⟨T⟩     confidenceInterval(REAL p);
            REAL       probability(IVL⟨T⟩ x);

            PPD⟨T⟩     times(REAL x);
};

PPD.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(); };
};

type ParametricProbabilityDistribution⟨REAL⟩ alias PPD⟨REAL⟩;

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); };
};

type ParametricProbabilityDistribution⟨PQ⟩ alias PPD⟨PQ⟩ {
            PPD⟨REAL⟩  value;
            CS         unit;
};

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);
};
```

 Copyright © Health Level Seven, Inc. All rights reserved.

```
PPD⟨PQ⟩.literal ST {
  PPD⟨PQ⟩ : PPD⟨REAL⟩ " " unit         { $.value.equals($1);
                                         $.unit.equals($3); }
};

type ParametricProbabilityDistribution⟨TS⟩ alias PPD⟨TS⟩;
```

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

C DETAILED TABLE OF CONTENTS

Version 3 Data Types Error! Bookmark not defined.

Contents **Error! Bookmark not defined.**

*1 Introduction **Error! Bookmark not defined.***
1.1 What is a Data Type? **Error! Bookmark not defined.**
1.2 Representation of Data Values 3
1.3 Properties of Data Values 3
1.4 Need for the Abstraction 4
1.5 Need for an HL7 Data Type Standard 4
1.6 Overview of Contents 5
1.7 Acknowledgements 8

2 Formal Data Type Definition Language 10
2.1 Declaration 11
2.2 Invariant Statements 12
2.2.1 Assertion Expressions 13
2.2.2 Nested Quantifier Expressions 13
2.3 Type Conversion 14
2.3.1 Demotion 14
2.3.2 Promotion 15
2.4 Literal Form 15
2.4.1 Declaration 16
2.4.2 Definition 16
2.4.2.1 Syntax Definition 17
2.4.2.2 Semantics Definition 17
2.4.2.3 Terminal Symbols 18
2.5 Generic Data Types 18
2.5.1 Generic Collections 19
2.5.2 Generic Type Extensions 19

3 Fundamental Data Types 20
3.1 Data Type 20
3.2 Data Value (ANY) 20
3.2.1.1 dataType : Data Type 20
3.2.1.2 The predicates isNull and nonNull 21
3.2.1.3 Equality 22
3.3 Boolean (BL) 23
3.3.1.1 Negation 23
3.3.1.2 Conjunction 23
3.3.1.3 Disjunction 24
3.3.1.4 Implication 24

4 Text 25
4.1 Binary Data (BIN) 25
4.2 Encoded Data (ED) 25
4.2.1.1 type : CS 26
4.2.1.2 charset : CS 28
4.2.1.3 language : CS 29
4.2.1.4 compression : CS 29
4.2.1.5 reference : TEL 29
4.2.1.6 integrityCheck : BIN 30
4.2.1.7 integrityCheckAlgorithm : CS 30
4.2.1.8 thumbnail : ED 30
4.2.1.9 Equality 31
4.3 Character String (ST) 31
4.3.1.1 Literal Form 32

5 Things, Concepts, and Qualities 33
5.1 Concept Descriptor (CD) 33
5.1.1.1 code : ST 34
5.1.1.2 codeSystem : OID 34
5.1.1.3 codeSystemName : ST 35
5.1.1.4 codeSystemVersion : ST 35
5.1.1.5 equals(CD x) : BL 35
5.1.1.6 implies(CD x) : BL 36
5.1.1.7 displayName : ST 36
5.1.1.8 translation : SET⟨CD⟩ 36
5.1.1.9 originalText : ED 37
5.1.1.10 producer : II 37
5.1.1.12 modifier : LIST ⟨CR⟩ 37
5.1.2 Concept Role (CR) 37
5.1.2.1 name : CV 38
5.1.2.2 value : CD 38
5.1.2.3 inverted : BL 38
5.1.3 Restrictions for the Concept Descriptors 38
5.1.3.1 Coded Simple Value (CS) restricts CD 39
5.1.3.2 Coded Value (CV) restricts CD 40
5.1.3.3 Coded With Equivalents (CE) 40
5.2 Instance Identifier (II) 41
5.2.1 ISO Object Identifier (OID) 41
5.2.1.1 Literal Form 42
5.2.1.2 Structured Form: Sequence of Integers versus Value and Namespace 42
5.2.2 Properties of the Instance Identifier 43
5.2.2.1 extension : ST 43
5.2.2.2 root : OID 43
5.2.2.3 assigningAuthorityName : ST 43
5.2.2.4 validTime : IVL⟨TS⟩ 44
5.2.2.5 type : CV 44
5.2.2.6 Equality 44
5.3 Telecommunication Address (TEL) 44
5.3.1 Universal Resource Locator (URL) 45

Copyright © Health Level Seven, Inc. All rights reserved.

5.3.1.1 Literal Form 45
5.3.1.2 scheme : CS 45
5.3.1.3 address : ST 46
5.3.1.4 Telephone and FAX addresses. 46
5.3.2 Properties of Telecommunication Address 46
5.3.2.1 validTime : GTS 47
5.3.2.2 use : SET⟨CS⟩ 47
5.3.2.3 Equality 47
5.4 Postal and Residential Address (AD) 47
5.4.1 Address Part (ADXP) 48
5.4.1.1 type : CS 48
5.4.2 Properties of Postal and Residential Addresses 49
5.4.2.1 validTime : GTS 49
5.4.2.2 use : SET⟨CS⟩ 49
5.4.2.3 Equality 50
5.4.2.4 Formatting Addresses 50
5.5 Person Name (PN) 51
5.5.1 Person Name Part (PNXP) 51
5.5.1.1 type : CS 51
5.5.1.2 qualifier : SET ⟨CS⟩ 51
5.5.2 Properties of Person Name 52
5.5.2.1 Formatting Person Names 53
5.6 Organization Name (ON) 53
5.6.1.1 type : CS 54

6 Quantities 55
6.1 Quantity (QTY) 55
6.1.1.1 Ordering 56
6.1.1.2 Difference 56
6.2 Integer Number (INT) 57
6.2.1.1 Algebraic Operations 57
6.2.1.2 Literal Form 58
6.3 Real Number (REAL) 58
6.3.1.1 precision : INT 59
6.3.1.2 Algebraic Operations 60
6.3.1.3 Literal Form 61
6.4 Ratio (RTO) 62
6.4.1.1 numerator (QTY) 63
6.4.1.2 denominator (QTY) 63
6.4.1.3 Literal Form 63
6.5 Physical Quantity (PQ) 63
6.5.1.1 value : REAL 64
6.5.1.2 unit : CS 64
6.5.1.3 Equality, Ordering and the Canonical Form 64
6.5.1.4 Algebraic Operations 64
6.5.1.5 Literal Form 66
6.6 Monetary Amount (MO) 66
6.6.1.1 value : REAL 66
6.6.1.2 currency : CS 67
6.6.1.3 Algebraic Operations 67
6.6.1.4 Literal Form 68
6.7 Point In Time (TS) 68
6.7.1.1 offset : PQ 69
6.7.1.2 calendar : CS 69
6.7.1.3 precision : INT 72
6.7.1.4 time zone : PQ 72
6.7.1.5 Addition and Subtraction 72
6.7.1.6 Literal Form 73

7 Generic Collections 75
7.1 Set (SET) 75
7.1.1.1 Element 75
7.1.1.2 Cardinality and Empty Set 76
7.1.1.3 Subset 76
7.1.1.4 Union 76
7.1.1.5 Difference 77
7.1.1.6 Intersection 77
7.1.1.7 Literal Form 77
7.1.1.8 Promotions of any Values to Sets 78
7.2 Sequence (LIST) 78
7.2.1.1 Literal Form 79
7.2.1.2 Promotions of any Values to Sequences 80
7.3 Bag (BAG) 80
7.3.1.1 Elements 80
7.3.1.2 Addition and Subtraction 80
7.3.1.3 Promotions of any Values to Bags 81
7.4 Interval (IVL) 81
7.4.1.1 low : T 82
7.4.1.2 high : T 82
7.4.1.3 width : T.diff 82
7.4.1.4 center : T 83
7.4.1.5 lowClosed : BL 83
7.4.1.6 highClosed : BL 83
7.4.1.7 Literal Form 83
7.4.1.8 Conversion Between Point Values and Intervals 85
7.4.2 Interval of Physical Quantities (IVL⟨PQ⟩) 86
7.4.3 Interval of Points in Time (IVL⟨TS⟩) 87

8 Timing Specification 88
8.1 Periodic Interval of Time (PIVL) 88
8.1.1.1 Period : T.diff 89
8.1.1.2 Phase : IVL⟨T⟩ 89
8.1.1.3 Alignment : CS 89
8.1.1.4 Periodic Intervals as Sets 90
8.1.1.5 Literal Form 90
8.2 Event-Related Periodic Interval of Time (EIVL) 92
8.2.1.1 Event : CV 92
8.2.1.2 Offset : IVL⟨T.diff⟩ 92
8.2.1.3 Resolving the Event-Relatedness 92
8.2.1.4 Literal Form 93
8.3 General Timing Specification (GTS) 94

Copyright © 2000, Health Level Seven, Inc. All rights reserved.

8.3.1.1 GTS as a Sequence of Occurrence Intervals 94
8.3.1.2 Outer Bound Interval 95
8.3.1.3 GTS Literal Form 95

9 Generic Type Extensions 98
9.1 Annotated (ANT) 98
9.1.1.1 note : CE 98
9.2 History Item (HXIT) 99
9.2.1 validTime : IVL ⟨TS⟩ 99
9.3 History (HIST) 99

10 Uncertainty and Probability 101
10.1 Uncertain Value – Narrative (UVN) 101
10.1.1.1 confidence : CV 101
10.2 Uncertain Value – Probabilistic (UVP) 101
10.2.1.1 probability : REAL 102
10.3 Non-Parametric Probability Distribution (NPPD) 102
10.4 Parametric Probability Distribution (PPD) 103
10.4.1.1 standardDeviation : T.diff 104
10.4.1.2 type : CS 104
10.4.1.3 Literal Form 105
10.4.2 Probability Distribution over Real Numbers (PPD⟨REAL⟩) 105
10.4.2.1 Converting a real number (REAL) to an uncertain real number (PPD⟨REAL⟩) 106
10.4.2.2 Concise Literal Form for PPD⟨REAL⟩ 106
10.4.3 Parametric Probability Distributions over Physical Quantities (PPD⟨PQ⟩) 107
10.4.3.1 Concise Literal Form for PPD⟨PQ⟩ 107
10.4.4 Probability Distribution over Time Points (PPD⟨TS⟩) 107
10.4.4.1 Converting a point in time (TS) to an uncertain point in time (PPD⟨TS⟩) 107

A Object Identifiers (normative) 109

B Summary of Formal Definitions 112

C Detailed Table of Contents 144

 Copyright © Health Level Seven, Inc. All rights reserved.

Copyright © 2000, Health Level Seven, Inc. All rights reserved.