The *open*EHR Reference Model

Support Information Model

Editors: {T Beale, S Heard}[a], {D Kalra, D Lloyd}[b]		
Revision: 1.6.1	*Pages:* 66	*Date of issue:* 20 Oct 2008
Status: STABLE		

a. Ocean Informatics

b. Centre for Health Informatics and Multi-professional Education, University College London

Keywords: EHR, openehr, reference model

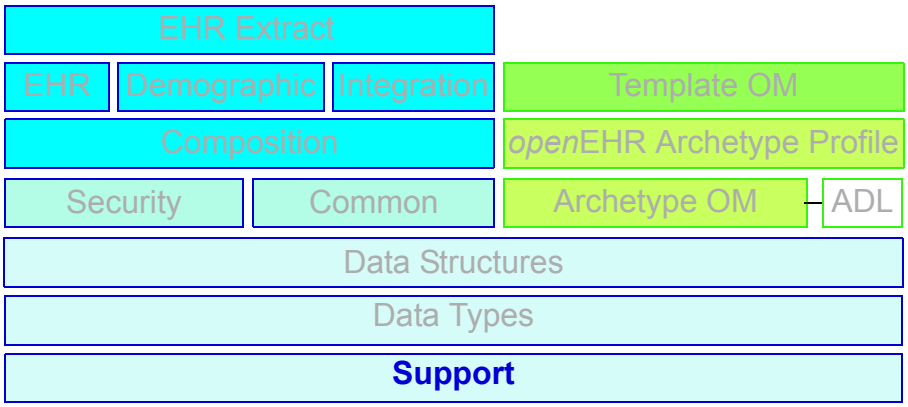

Table of Contents

1 Introduction

1.1 Purpose

This document describes the *open*EHR Support Reference Model, whose semantics are used by all *open*EHR Reference Models. The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development organisations developing EHR systems;
- Academic groups studying the EHR;
- The open source healthcare community.

1.2 Related Documents

Prerequisite documents for reading this document include:

- The *open*EHR Architecture Overview
- The *open*EHR Modelling Guide

1.3 Status

This document is under development, and is published as a proposal for input to standards processes and implementation works.

This document is available at http://svn.openehr.org/specification/TAGS/Release-1.0.1/publishing/architecture/rm/support_im.pdf.

The latest version of this document can be found at http://svn.openehr.org/specification/TRUNK/publishing/architecture/rm/support_im.pdf.

Blue text indicates sections under active development.

1.4 Peer review

Areas where more analysis or explanation is required are indicated with “to be continued” paragraphs like the following:

To Be Continued: more work required

Reviewers are encouraged to comment on and/or advise on these paragraphs as well as the main content. Please send requests for information to info@openEHR.org. Feedback should preferably be provided on the mailing list openehr-technical@openehr.org, or by private email.

1.5 Conformance

Conformance of a data or software artifact to an *open*EHR Reference Model specification is determined by a formal test of that artifact against the relevant *open*EHR Implementation Technology Specification(s) (ITSs), such as an IDL interface or an XML-schema. Since ITSs are formal, automated derivations from the Reference Model, ITS conformance indicates RM conformance.

2 Support Package

2.1 Overview

The Support Reference Model comprises types used throughout the *open*EHR models, including assumed primitive types defined outside of *open*EHR. The package structure is illustrated in FIGURE 1. The `assumed_types` 'pseudo-package' stands for types assumed by the *open*EHR specifcations to exist in an implementation technology, such as a programming language, schema language or database environment. The four Support packages define the semantics respectively for constants, terminology access, access to externally defined scientific units and conversion information. The class `EXTERNAL_ENVIRONMENT_ACCESS` is a mixin class providing access to the service interface classes.

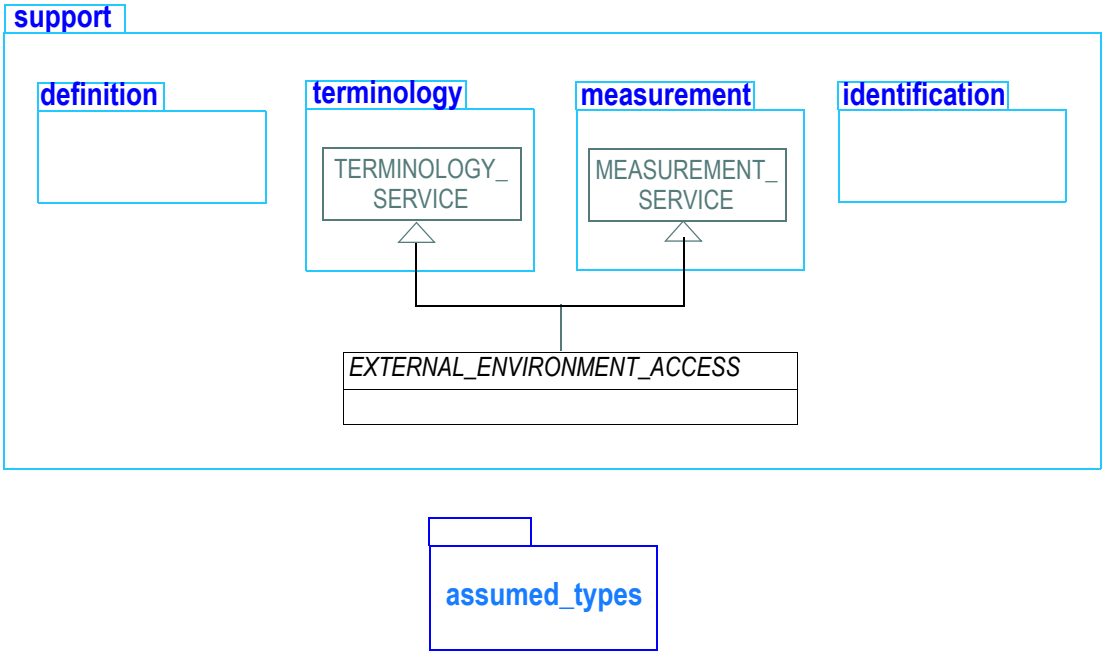

FIGURE 1 rm.support and assumed_types Packages

2.2 Class Definitions

2.2.1 EXTERNAL_ENVIRONMENT_ACCESS Class

CLASS	*EXTERNAL_ENVIRONMENT_ACCESS (abstract)*	
Purpose	A mixin class providing access to services in the external environment.	
Functions	**Signature**	**Meaning**
	eea_terminology_svc: `TERMINOLOGY_SERVICE`	Return an interface to the terminology service
	eea_measurement_svc: `MEASUREMENT_SERVICE`	Return an interface to the measurement service

CLASS	*EXTERNAL_ENVIRONMENT_ACCESS (abstract)*
Invariants	***Terminology_service_exists***: eea_terminology_svc /= Void ***Measurement_service_exists***: eea_measurement_svc /= Void

3 Assumed Types

3.1 Overview

This section describes types assumed by all *open*EHR models. The set of types chosen here is based on a common set from various published sources, including:

- ISO 11404 (2003 revision) general purpose data types;
- ISO 8601 (2004) date/time specification;
- Well-known interoperability formalisms, including OMG IDL, W3C XML-schema;
- Well-known object-oriented programming languages, including Java, C#, C++ and Eiffel.

The intention in *open*EHR is twofold. Firstly, to ensure that *open*EHR software based on the models integrates as easily as possible with existing implementation technologies, and secondly, to make the minimum possible assumptions about types found in implementation formalisms, while making sufficient assumptions to both enable *open*EHR models to be conveniently specified. The ISO 11404 (2003) standard contains basic semantics of "general purpose data types" (GPDs) for information technology, and is used here as a normative basis for describing assumptions about types. The operations and properties described here are compatible with those used in ISO 11404, but not always the same, as 11404 does not use object-oriented functions. For example, the notional function `has(x:T)` (test for presence of a value in a set) defined on the type `Set<T>` below is not defined on the ISO 11404 Set type; instead, the function `IsIn(x: T; s: Set<T>)` is defined. However, in object-oriented formalisms, the function `IsIn` defined on a Set type would usually mean 'subset of'. In the interests of clarity for developers, an object-oriented style of functions and properties has been used here.

ISO8601:2004 is used as the definitional basis for assumed date/time types, since it is commonly used around the world, and is also the basis for the date/time types in W3C XML-schema. See section 3.4 on page 21 below for details of dates and times.

Two groups of assumed types are identified: primitive types, which are those built in to a formalism's type system, and library types, which are assumed to be available in a (class) library defined in the formalism. Thus, the type `Boolean` is always assumed to exist in a formalism, while the type `Array<T>` is assumed to be available in a library. For practical purposes, these two categories do not matter that much - whether `String` is really a library class (the usual case) or an inbuilt type doesn't make much difference to the programmer. They are shown separately here mainly as an explanatory convenience.

The assumptions that *open*EHR makes about existing types are documented below in terms of interface definitions. Each of these definitions contains *only the assumptions required for the given type to be used in the openEHR Reference Model* - **it is not by any means a complete interface definition**. The name and semantics of any function used here for an assumed type **might not be identical** to those found in some implementation technologies. Any mapping required should be stated in the relevant implementation technology specification (ITS). To give a concrete example, where the assumed `Set<T>` type defined below has an operation *has*(item: `T`): `Boolean` which is used throughout the *open*EHR specifications, Java has the method *contains*() on its `Set<T>` class. In a Java implementation, the *contains*() method should then be used throughout the *open*EHR classes as expressed in Java, in place of the *has*() method.

3.2 Inbuilt Primitive Types

The following types consititute the minimum set of primitive types assumed by *open*EHR of an implementation formalism.

Type name in *open*EHR	Description	ISO 11404 Type
`Octet`	represents a type whose value is an 8-bit value.	`Octet`
`Character`	represents a type whose value is a member of an 8-bit character-set (ISO: "repertoire").	`Character`
`Boolean`	represents logical True/False values; usually physically represented as an integer, but need not be	`Boolean`
`Integer`	represents 32-bit integers	`Integer`
`Real`	represents 32-bit real numbers in any interoperable representation, including single-width IEEE floating point	`Real`
`Double`	type which represents 64-bit real numbers, in any interoperable representation including double-precision IEEE floating point.	`Real`

FIGURE 2 illustrates the built-in primitive types. Simple inheritance relationships are shown which facilitate the type descriptions below. A class "Any" is used to stand for the usual top-level class in all object-oriented type systems, typically called something like "Any" or "Object". Inheritance from or subsitutability for an `Any` class is *not* assumed in *open*EHR (hence the dotted lines in the UML). It is used here to enable basic operations like '=' to be described once for the type `Any`, rather than in every subtype. The type `Ordered_numeric` is on the other hand assumed for purposes of specification in the *open*EHR `data_types.quantity` package, and is intended to be mapped to an equivalent type in a real type system (e.g. in Java, `java.lang.Number`). Here it is assumed that the operations defined on `Ordered_numeric` are available on the types `Integer`, `Real` and `Double` in implementation type systems, where relevant. Data-oriented implementation type systems such as XML-schema do not have such operations.

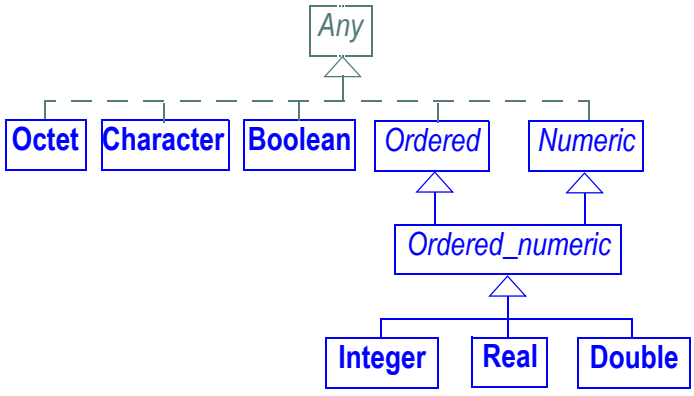

FIGURE 2 Primitive Types Assumed by *open*EHR

3.2.1 Any Type

INTERFACE	*Any (abstract)*	
Description	Abstract supertype. Usually maps to a type like "Any" or "Object" in an object system. Defined here to provide the value and reference equality semantics.	
Abstract	**Signature**	**Meaning**
	is_equal (other: `Any`): `Boolean`	Value equality
Functions	**Signature**	**Meaning**
	infix '=' (other: `Any`): `Boolean`	Reference equality
	instance_of (a_type: `String`)	Dynamic type of object as a String. Used for type name matching.
Invariants		

3.2.2 Ordered Type

INTERFACE	*Ordered (abstract)*	
Purpose	Abstract notional parent class of ordered, types i.e. types on which the '<' operator is defined.	
Abstract	**Signature**	**Meaning**
	infix '<' (other: ***like*** Current): `Boolean`	Arithmetic comparison. In conjunction with '=', enables the definition of the operators '>', '>=', '<=', '<>'. In real type systems, this operator might be defined on another class for comparability.
Invariants		

3.2.3 Numeric Type

INTERFACE	*Numeric (abstract)*	
Purpose	Abstract notional parent class of numeric types, which are types which have various arithmetic and comparison operators defined.	
Abstract	**Signature**	**Meaning**

INTERFACE	*Numeric (abstract)*	
	infix "*" (other: ***like*** Current): ***like*** Current ***require*** *other_exists*: other /= void ***ensure*** *result_exists*: Result /= void	Product by `other'. Actual type of result depends on arithmetic balancing rules.
	infix "+" (other: ***like*** Current): ***like*** Current ***require*** *other_exists*: other /= void ***ensure*** *result_exists*: Result /= void *commutative*: equal (Result, other + Current)	Sum with `other' (commutative). Actual type of result depends on arithmetic balancing rules.
	infix "-" (other: ***like*** Current): ***like*** Current ***require*** *other_exists*: other /= void ***ensure*** *result_exists*: Result /= void	Result of subtracting `other'. Actual type of result depends on arithmetic balancing rules.
Invariants		

3.2.4 Ordered_numeric Type

INTERFACE	*Ordered_numeric (abstract)*	
Purpose	Abstract notional parent class of ordered, numeric types, which are types with '<' and arithmetic operators defined.	
Inherit	`ORDERED, NUMERIC`	
Function	**Signature**	**Meaning**
Invariants		

3.2.5 Boolean Type

INTERFACE	**Boolean**	
Purpose	Boolean type used for two-valued mathematical logic.	
Function	**Signature**	**Meaning**

INTERFACE	Boolean	
	infix "and" (other: `Boolean`): `Boolean` ***require*** *other_exists*: other /= void ***ensure*** *de_morgan*: Result = **not** (**not** Current **or not** other) *commutative*: Result = (other **and** Current)	Logical conjunction
	infix "and then" (other: `Boolean`): `Boolean` ***require*** *other_exists*: other /= void ***ensure*** *de_morgan*: Result = **not** (**not** Current **or else not** other)	Boolean semi-strict conjunction with *other*
	infix "or" (other: `Boolean`): `Boolean` ***require*** *other_exists*: other /= void ***ensure*** *de_morgan*: Result = **not** (**not** Current **and not** other) *commutative*: Result = (other **or** Current) *consistent_with_semi_strict*: Result **implies** (Current **or else** other)	Boolean disjunction with *other*
	infix "or else" (other: `Boolean`): `Boolean` ***require*** *other_exists*: other /= void ***ensure*** *de_morgan*: Result = **not** (**not** Current **and then not** other)	Boolean semi-strict disjunction with `other'
	infix "xor" (other: `Boolean`): `Boolean` ***require*** *other_exists*: other /= void ***ensure*** *definition*: Result = ((Current **or** other) **and not** (Current **and** other))	Boolean exclusive or with `other'

INTERFACE	Boolean	
	infix "implies" (other: `Boolean`): `Boolean` ***require*** *other_exists*: other /= void ***ensure*** *definition*: Result = (**not** Current **or else** other)	Boolean implication of `other' (semi-strict)
Invariants	*involutive_negation*: is_equal (**not** (**not** Current)) *non_contradiction*: **not** (Current **and** (**not** Current)) *completeness*: Current **or else** (**not** Current)	

3.2.6 Real Type

INTERFACE	Real	
Purpose	Type used to represent decimal numbers. Typically corresponds to a single-precision floating point value in most languages.	
Function	**Signature**	**Meaning**
	floor: `Integer`	Return the greatest integer no greater than the value of this object.
Invariants		

3.3 Assumed Library Types

The types described in this section are also assumed to be fairly standard in implementation technologies by *open*EHR, but usually come from type libraries rather than being built into the type system of implementation formalisms.

Type name in *open*EHR	Description	ISO 11404: 2003 Type
`String`	represents unicode-enabled strings	`Character-String/ Sequence`
`Array<T>`	physical container of items indexed by number	`Array`
`List<T>`	container of items, implied order, non-unique membership	`Sequence`
`Set<T>`	container of items, no order, unique membership	`Set`
`Hash<T, U:Comparable>`	a table of values of any type T, keyed by values of any basic comparable type U, typically String or Integer, but may be more complex types, e.g. a coded term type.	`Table`
`Interval<T>`	Intervals with open or closed upper and lower bounds.	–

FIGURE 3 illustrates the assumed library types. As with the assumed primitive types, inheritance and abstract classes are used for convenience of the definitions below, but are not formally assumed in *open*EHR.

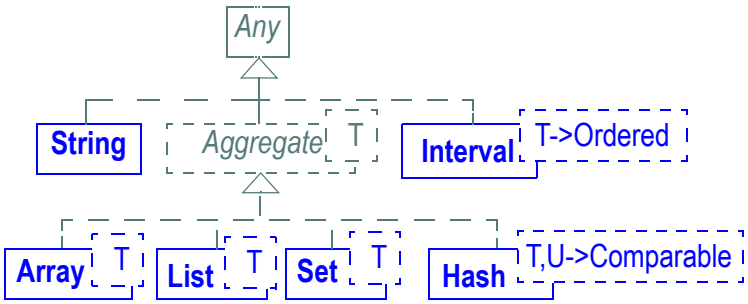

FIGURE 3 Library Types Assumed by *open*EHR

3.3.1 String Type

INTERFACE	String	
Description	Strings of characters, as used to represent textual data in any natural or formal language.	
Functions	**Signature**	**Meaning**
	infix '+' (other: `String`): `String`	Concatenation operator - causes 'other' to be appended to this string
	is_empty: `Boolean`	True if string is empty, i.e. equal to "".
	is_integer: `Boolean`	True if string can be parsed as an integer.
	as_integer: `Integer` ***require*** is_integer	Return the integer corresponding to the integer value represented in this string.
Invariants		

3.3.1.1 UNICODE

It is assumed in the *open*EHR specifications that Unicode is supported by the type `String`. Unicode is needed for all Asian, Arabic and other script languages, for both data values (particularly plain text and coded text) and for many predefined string attributes of the classes in the *open*EHR Reference Model. It encompasses all existing character sets. In *open*EHR, UTF-8 encoding is assumed.

3.3.2 Aggregate Type

INTERFACE	*Aggregate <T> (abstract)*
Description	Abstract parent of of the aggregate types `List<T>`, `Set<T>`, `Array<T>` and `Hash<T,K>`.

INTERFACE	*Aggregate <T> (abstract)*	
Functions	**Signature**	**Meaning**
	has (v: `T`): `Boolean`	Test for membership of a value
	count: `Integer`	Number of items in container
	is_empty: `Boolean`	True if container is empty.
Invariants		

3.3.3 List Type

INTERFACE	*List <T> (abstract)*	
Description	Ordered container that may contain duplicates.	
Functions	**Signature**	**Meaning**
	first: `T`	Return first element.
	last: `T`	Return last element.
Invariants	***First_validity***: **not** is_empty **implies** first /= Void ***Last_validity***: **not** is_empty **implies** last /= Void	

3.3.4 Set Type

INTERFACE	*Set <T> (abstract)*	
Description	Unordered container that may not contain duplicates.	
Functions	**Signature**	**Meaning**
Invariants		

3.3.5 Array Type

INTERFACE	*Array <T> (abstract)*	
Description	Container whose storage is assumed to be contiguous.	
Functions	**Signature**	**Meaning**
Invariants		

3.3.6 Hash Type

INTERFACE	Hash <T, U: Comparable>	
Description	Type representing a keyed table of values. T is the value type, and U the type of the keys.	
Functions	**Signature**	**Meaning**
	has_key (a_key: `U`): `Boolean`	Test for membership of a key
	item (a_key: `U`): `T`	Return item for key 'a_key'. Equivalent to ISO 11404 *fetch* operation.
Invariants		

3.3.7 Interval Type

INTERFACE	Interval <T:Ordered>	
Purpose	Interval of ordered items.	
Attributes	**Signature**	**Meaning**
	lower: `T`	lower bound
	upper: `T`	upper bound
	lower_unbounded: `Boolean`	lower boundary open (i.e. = -infinity)
	upper_unbounded: `Boolean`	upper boundary open (i.e. = +infinity)
	lower_included: `Boolean`	lower boundary value included in range if not *lower_unbounded*
	upper_included: `Boolean`	upper boundary value included in range if not *upper_unbounded*
Functions	**Signature**	**Meaning**
	has(e:`T`**)**: `Boolean`	True if (lower_unbounded **or** ((lower_included **and** v >= lower) **or** v > lower)) **and** (upper_unbounded **or** ((upper_included **and** v <= upper **or** v < upper)))

INTERFACE	Interval <T:Ordered>
Invariants	***Lower_included_valid***: lower_unbounded **implies not** lower_included ***Upper_included_valid***: upper_unbounded **implies not** upper_included ***Limits_consistent***: (**not** upper_unbounded **and not** lower_unbounded) **implies** lower <= upper ***Limits_comparable***: (**not** upper_unbounded **and not** lower_unbounded) **implies** lower.strictly_comparable_to(upper)

3.4 Date/Time Types

Although the ISO 11404 (2003) standard defines a date-and-time type generator (section 8.1.6), and a `timeinterval` type (section 10.1.6), a more widely used specification of date/times is given by ISO 8601:2004, which is used as the normative basis for both string literal representation and properties used within *open*EHR. The types are shown in FIGURE 4.

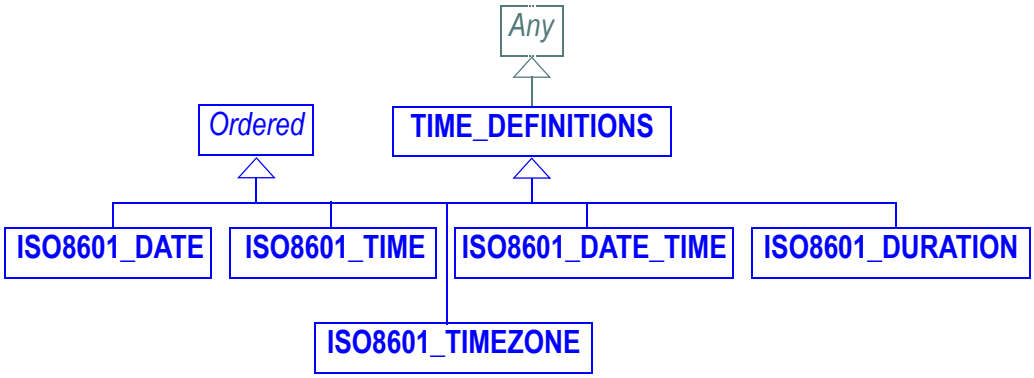

FIGURE 4 Date/Time types assumed by *open*EHR

ISO 8601 semantics not used in *open*EHR include:

- "expanded" dates, which have year numbers of greater than 4 digits, and may be negative; in *open*EHR, only 4-digit year numbers are assumed;
- the YYYY-WW-DD method of expressing dates (since this is imprecise and difficult to compute with due to variable week starting dates, and not required in health);
- partial date/times with fractional minutes or hours, e.g. hh,hhh or mm,mm; in *open*EHR, only fractional seconds are supported;
- the interval syntax. Intervals of date/times are supported in *open*EHR, but their syntax form is defined by ADL, and is standardised across all comparable types, not just dates and times.

Deviations from the published standard include the following:

- durations are supposed to take the form of `PnnW` or `PnnYnnMnnDTnnHnnMnnS`, but in *open*EHR, the W (week) designator can be used in combination with the other designators, since it is very common to state durations of pregnancy as some combination of weeks and days.
- partial variants of `ISO8601_DATE_TIME` can include missing hours, days and months, whereas ISO 8601:2004 (section 4.3.3 c) only allows missing seconds and minutes. The reasons for this deviation are:

- the same deviation is used in HL7v2 and HL7v3 TS (timestamp) type, i.e. there are data in existing clinical systems matching this specification;
- in a typed object model, this deviation is more sensible anyway; the ISO 8601 rule is most likely a limitation of the purely syntactic means of expression. In real systems where a timestamp/date-time is specified in a screen form, it makes sense to allow it to be as partial as possible, rather than artifically restricted to only missing seconds and minutes.

- the time 24:00:00 (or 240000) is not allowed anywhere, whereas in ISO8601:2004 it appears to be legal at least for times. This deviation is also appears to be used in HL7v2 and HL7v3 (where midnight is defined as the time 00:00:00), and is preferable to the documented standard, since a date/time with time of 24:00:00 is really the next day, i.e. the date part is then incorrect.

The following class definitions provide an object-oriented expression of the semantics of the subset of ISO 8601:2004 used by *open*EHR.

See http://www.cl.cam.ac.uk/~mgk25/iso-time.html and the official ISO standard for ISO 8601 details. Note that in the date, time and date_time formats shown below, 'Z' and 'T' are literals. In the duration class shown below, 'P', 'Y', 'M', 'W', 'D', 'H', 'S' and 'T' are literals.

3.4.1 TIME_DEFINITIONS Class

INTERFACE	TIME_DEFINITIONS	
Purpose	Definitions for date/time classes. Note that the timezone limits are set by where the international dateline is. Thus, time in New Zealand is quoted using +12:00, not -12:00.	
Constants	**Signature**	**Meaning**
1..1	**Seconds_in_minute**: `Integer` = 60	
1..1	**Minutes_in_hour**: `Integer` = 60	
1..1	**Hours_in_day**: `Integer` = 24	
1..1	**Nominal_days_in_month**: `Real` = 30.42	Used for conversions of durations containing months to days and / or seconds.
1..1	**Max_days_in_month**: `Integer` = 31	Used for validity checking.
1..1	**Days_in_year**: `Integer` = 365	
1..1	**Days_in_leap_year**: `Integer` = 366	
1..1	**Max_days_in_year**: `Integer` = Days_in_leap_year	Used for validity checking.

INTERFACE	TIME_DEFINITIONS	
1..1	**Nominal_days_in_year**: `Real` = 365.24	Used for conversions of durations containing years to days and / or seconds.
1..1	**Days_in_week**: `Integer` = 7	
1..1	**Months_in_year**: `Integer` = 12	
1..1	**Min_timezone_hour**: `Integer` ***ensure*** Result = 12	Minimum hour value of a timezone (note that the -ve sign is supplied in the `ISO8601_TIMEZONE` class).
1..1	**Max_timezone_hour**: `Integer` ***ensure*** Result = 13	Maximum hour value of a timezone.
Functions	**Signature**	**Meaning**
	valid_year (y: `Integer`): `Boolean` ***ensure*** Result = y >= 0	True if y >= 0
	valid_month (m: `Integer`): `Boolean` ***ensure*** Result = m >= 1 and m <= Months_in_year	True if m >= 1 and m <= Months_in_year
	valid_day (y, m, d: `Integer`): `Boolean` ***ensure*** Result = d >= 1 and d <= days_in_month(m, y)	True if d >= 1 and d <= days_in_month(m, y)
	valid_hour (h, m, s: `Integer`): `Boolean` ***ensure*** Result = (h >= 0 and h < Hours_in_day) or (h = Hours_in_day and m = 0 and s = 0)	True if (h >= 0 and h < Hours_in_day) or (h = Hours_in_day and m = 0 and s = 0)
	valid_minute (m: `Integer`): `Boolean` ***ensure*** Result = m >= 0 and m < Minutes_in_hour	True if m >= 0 and m < Minutes_in_hour

INTERFACE	TIME_DEFINITIONS	
	valid_second (s: `Integer`): `Boolean` ***ensure*** Result = s >= 0 and s < Seconds_in_minute	True if s >= 0 and s < Seconds_in_minute
	valid_fractional_second (fs: `Double`): `Boolean` ***ensure*** Result = fs >= 0.0 and fs < 1.0	True if fs >= 0.0 and fs < 1.0
Invariants		

3.4.2 ISO8601_DATE Class

INTERFACE	ISO8601_DATE	
Purpose	Represents an absolute point in time, as measured on the Gregorian calendar, and specified only to the day.	
Inherit	`ORDERED`, `TIME_DEFINITIONS`	
Function	**Signature**	**Meaning**
	as_string: `String`	ISO8601 string for date, in format **YYYYMMDD** or **YYYY-MM-DD**, or a partial invariant. See *valid_iso8601_date* for validity.
	year: `Integer`	Year.
	month: `Integer` ***require*** **not** month_unknown	Month in year.
	day: `Integer` ***require*** **not** day_unknown	Day in month.
	month_unknown: `Boolean`	Indicates whether month in year is unknown. If so, the date is of the form "YYYY".
	day_unknown: `Boolean`	Indicates whether day in month is unknown. If so, and month is known, the date is of the form "YYYY-MM" or "YYYYMM".
	is_partial: `Boolean`	True if this date is partial, i.e. if day or more is missing.

INTERFACE	ISO8601_DATE	
	is_extended: `Boolean`	True if this date uses '-' separators.
	infix '<' (other: ***like*** Current): `Boolean`	Arithmetic comparison with other date. True if this date is closer to the origin than *other*.
	valid_iso8601_date (s: `String`): `Boolean`	String is a valid ISO 8601 date, i.e. takes the complete form: • **YYYYMMDD** or the extended form: • **YYYY-MM-DD** or one of the partial forms: • **YYYYMM** • **YYYY** or the equivalent extended form: • **YYYY-MM** Where: • **YYYY is the string form of any positive number in the range "0000" - "9999" (zero-filled to four digits)** • **MM is "01" - "12" (zero-filled to two digits)** • **DD is "01" - "31" (zero-filled to two digits)** The combinations of YYYY, MM, DD numbers must be correct with respect to the Gregorian calendar.
Invariants	***Year_valid***: valid_year(year) ***Month_valid***: **not** month_unknown **implies** valid_month(month) ***Day_valid***: **not** day_unknown **implies** valid_day(year, month, day) ***Partial_validity***: month_unknown **implies** day_unknown	

3.4.3 ISO8601_TIME Class

INTERFACE	ISO8601_TIME	
Purpose	Represents an absolute point in time from an origin usually interpreted as meaning the start of the current day, specified to the second. A small deviation to the ISO 8601:2004 standard in this class is that the time 24:00:00 is not allowed, for consistency with `ISO8601_DATE_TIME`.	
Inherit	`ORDERED`, `TIME_DEFINITIONS`	
Function	**Signature**	**Meaning**

INTERFACE	ISO8601_TIME	
	as_string: `String`	ISO8601 string for time, i.e. in form: **hhmmss**[,**sss**][**Z**|±**hh[mm]**] or the extended form: **hh**:**mm**:**ss**[**,sss**][**Z**|±**hh[mm]**], or a partial invariant. See *valid_iso8601_time* for validity.
	hour: `Integer`	Hour in day, in 24-hour time.
	minute: `Integer` ***require*** **not** minute_unknown	Minute in hour.
	second: `Integer` ***require*** **not** second_unknown	Second in minute.
	fractional_second: `Double` ***require*** **not** second_unknown	Fractional seconds.
	has_fractional_second: `Boolean`	True if the *fractional_second* part is signficant (i.e. even if = 0.0).
	timezone: `ISO8601_TIMEZONE`	Time zone; may be Void.
	minute_unknown: `Boolean`	Indicates whether minute is unknown. If so, the time is of the form “hh”.
	second_unknown: `Boolean`	Indicates whether second is unknown. If so and month is known, the time is of the form “hh:mm” or “hhmm”.
	is_partial: `Boolean`	True if this time is partial, i.e. if seconds or more is missing.
	is_extended: `Boolean`	True if this time uses ‘:’ separators.
	is_decimal_sign_comma: `Boolean`	True if this time has a decimal part indicated by ‘,’ (comma) rather than ‘.’ (period).
	infix ‘<’ (other: ***like*** Current): `Boolean`	Arithmetic comparison with other time. True if this date is closer to previous midnight than *other*.

INTERFACE	ISO8601_TIME	
	valid_iso8601_time (s: `String`): `Boolean`	String is a valid ISO 8601 date, i.e. takes the form: • **hhmmss[,sss][Z | ±hh[mm]]** or the extended form: • **hh:mm:ss[,sss][Z | ±hh[mm]]** or one of the partial forms: • **hhmm** or **hh** or the extended form: • **hh:mm** with an additional optional timezone indicator of: • **Z** or **±hh[mm]** Where: • **hh is "00" - "23" (0-filled to two digits)** • **mm is "00" - "59" (0-filled to two digits)** • **ss is "00" - "60" (0-filled to two digits)** • **sss is any numeric string, representing an optional fractional second** • **Z is a literal meaning UTC (modern replacement for GMT), i.e. timezone +0000** • **±hh[mm], i.e. +hhmm, +hh, -hhmm, -hh indicating the timezone.**
Invariants	***Hour_valid***: valid_hour(hour, minute, second) ***Minute_valid***: **not** minute_unknown **implies** valid_minute(minute) ***Second_valid***: **not** second_unknown **implies** valid_second(second) ***Fractional_second_valid***: has_fractional_second **implies (not** second_unknown **and** valid_fractional_second(fractional_second)) ***Partial_validity***: minute_unknown **implies** second_unknown	

3.4.4 ISO8601_DATE_TIME Class

INTERFACE	ISO8601_DATE_TIME	
Purpose	Represents an absolute point in time, specified to the second. Note that this class includes 2 deviations from ISO 8601:2004: • for partial date/times, any part of the date/time up to the month may be missing, not just seconds and minutes as in the standard; • the time 24:00:00 is not allowed, since it would mean the date was really on the next day.	
Inherit	`ORDERED`, `TIME_DEFINITIONS`	
Function	**Signature**	**Meaning**

INTERFACE	ISO8601_DATE_TIME	
	as_string: `String` ***ensure*** valid_iso8601_date_time(Result)	ISO8601 string for date/time, in format **YYYYMMDDThhmmss[,sss][Z | ±hh[mm]]** or in extended format **YYYY-MM-DDThh:mm:ss[,sss][Z | ±hh[mm]]** or a partial variant; see *valid_iso8601_date_time*() below.
	year: `Integer`	year
	month: `Integer` ***require*** **not** month_unknown	month in year
	day: `Integer` ***require*** **not** day_unknown	day in month
	hour: `Integer` ***require*** **not** hour_unknown	hour in day
	minute: `Integer` ***require*** **not** minute_unknown	minute in hour
	second: `Integer` ***require*** **not** second_unknown	second in minute
	fractional_second: `Double` ***require*** has_fractional_second	fractional seconds
	has_fractional_second: `Boolean`	True if the *fractional_second* part is signficant (i.e. even if = 0.0).
	timezone: `ISO8601_TIMEZONE`	Timezone; may be Void.
	month_unknown: `Boolean`	Indicates whether month in year is unknown.
	day_unknown: `Boolean`	Indicates whether day in month is unknown.
	hour_unknown: `Boolean`	Indicates whether hour in day is known.
	minute_unknown: `Boolean`	Indicates whether minute in hour is known.
	second_unknown: `Boolean`	Indicates whether minute in hour is known.
	is_partial: `Boolean`	True if this date is partial, i.e. if seconds or more is missing.

INTERFACE	ISO8601_DATE_TIME	
	is_decimal_sign_comma: `Boolean`	True if this time has a decimal part indicated by ‘,’ (comma) rather than ‘.’ (period).
	infix ‘<’ (other: ***like*** Current): `Boolean`	Arithmetic comparison with other date/time. True if this date/time is closer to origin than *other*.
	is_extended: `Boolean`	True if this date/time uses ‘-’, ‘:’ separators.
	valid_iso8601_date_time (s**:** `String`): `Boolean`	String is a valid ISO 8601 date-time, i.e. takes the form: • **YYYYMMDDThhmmss[,sss] [Z | ±hh[mm]]** or the extended form: • **YYYY-MM-DDThh:mm:ss[,sss] [Z | ±hh[mm]]** or one of the partial forms: • **YYYYMMDDThhmm** • **YYYYMMDDThh** or the equivalent extended forms: • **YYYY-MM-DDThh:mm** • **YYYY-MM-DDThh** (meanings as in `DV_DATE`, `DV_TIME`) and the values in each field are valid.
Invariants	***Year_valid***: valid_year(year) ***Month_valid***: valid_month(month) ***Day_valid***: valid_day(year, month, day) ***Hour_valid***: valid_hour(hour, minute, second) ***Minute_valid***: **not** minute_unknown **implies** valid_minute(minute) ***Second_valid***: **not** second_unknown **implies** valid_second(second) ***Fractional_second_valid***: has_fractional_second **implies (not** second_unknown **and** valid_fractional_second(fractional_second)) ***Partial_validity_year***: **not** month_unknown ***Partial_validity_month***: **not** month_unknown ***Partial_validity_day***: **not** day_unknown ***Partial_validity_hour***: **not** hour_unknown ***Partial_validity_minute***: minute_unknown **implies** second_unknown	

3.4.5 ISO8601_TIMEZONE Class

INTERFACE	ISO8601_TIMEZONE
Purpose	Represents a timezone as used in ISO 8601.

INTERFACE	ISO8601_TIMEZONE	
Inherit	TIME_DEFINITIONS	
Function	**Signature**	**Meaning**
	as_string: String	ISO8601 timezone string, in format • **Z | ±hh[mm]** where: • **hh is "00" - "23" (0-filled to two digits)** • **mm is "00" - "59" (0-filled to two digits)** • **Z is a literal meaning UTC (modern replacement for GMT), i.e. timezone +0000**
	hour: Integer	Hour part of timezone - in the range 00 - 13
	minute: Integer ***require*** **not** minute_unknown	Minute part of timezone. Generally 00 or 30.
	sign: Integer	Direction of timezone expresssed as +1 or -1.
	is_gmt: Boolean	True if timezone is UTC, i.e. +0000
	minute_unknown: Boolean	Indicates whether minute part known.
Invariants	***Min_hour_valid***: sign = -1 **implies** hour > 0 **and** hour <= Min_timezone_hour ***Max_hour_valid***: sign = 1 **implies** hour > 0 **and** hour <= Max_timezone_hour ***Minute_valid***: **not** minute_unknown **implies** valid_minute(minute) ***Sign_valid***: sign = 1 or sign = -1	

3.4.6 ISO8601_DURATION Class

INTERFACE	ISO8601_DURATION	
Purpose	Represents a period of time corresponding to a difference between two time-points.	
Inherit	ORDERED, TIME_DEFINITIONS	
Function	**Signature**	**Meaning**
	as_string: String	ISO8601 string for duration, in format • **P[nnY][nnM][nnW][nnD][T[nnH][nnM][nnS]]**
	years: Integer	number of years of nominal 365-day length
	months: Integer	number of months of nominal 30 day length
	weeks: Integer	number of 7 day weeks

INTERFACE	ISO8601_DURATION	
	days: `Integer`	number of 24 hour days
	hours: `Integer`	number of 60 minute hours
	minutes: `Integer`	number of 60 second minutes
	seconds: `Integer`	number of seconds
	fractional_second: `Double`	fractional seconds
	infix '<' (other: ***like*** Current): `Boolean`	Arithmetic comparison with other duration. True if this duration is smaller than *other*.
	valid_iso8601_duration (s**:** `String`): `Boolean`	String is a valid ISO 8601 duration, i.e. takes the form: • **P[nnY][nnM][nnW][nnD][T[nnH][nnM][nnS]]** Where each nn represents a number of years, months, etc. nnW represents a number of 7-day weeks. **Note**: allowing the W designator in the same expression as other designators is an exception to the published standard, but necessary in clinical information (typically for representing pregnancy duration).
	is_decimal_sign_comma: `Boolean`	True if this time has a decimal part indicated by ',' (comma) rather than '.' (period).
	to_seconds: `Double`	Total number of seconds equivalent (including fractional) of entire duration.
Invariants	***years_valid***: years >= 0 ***months_valid***: months >= 0 ***weeks_valid***: weeks >= 0 ***days_valid***: days >= 0 ***hours_valid***: hours >= 0 ***minutes_valid***: minutes >= 0 ***seconds_valid***: seconds >= 0 ***fractional_second_valid***: fractional_second >= 0.0 and fractional_second < 1.0	

4 Identification Package

4.1 Overview

The `rm.support.identification` package describes a model of references and identifiers for information entities and is illustrated in FIGURE 5.

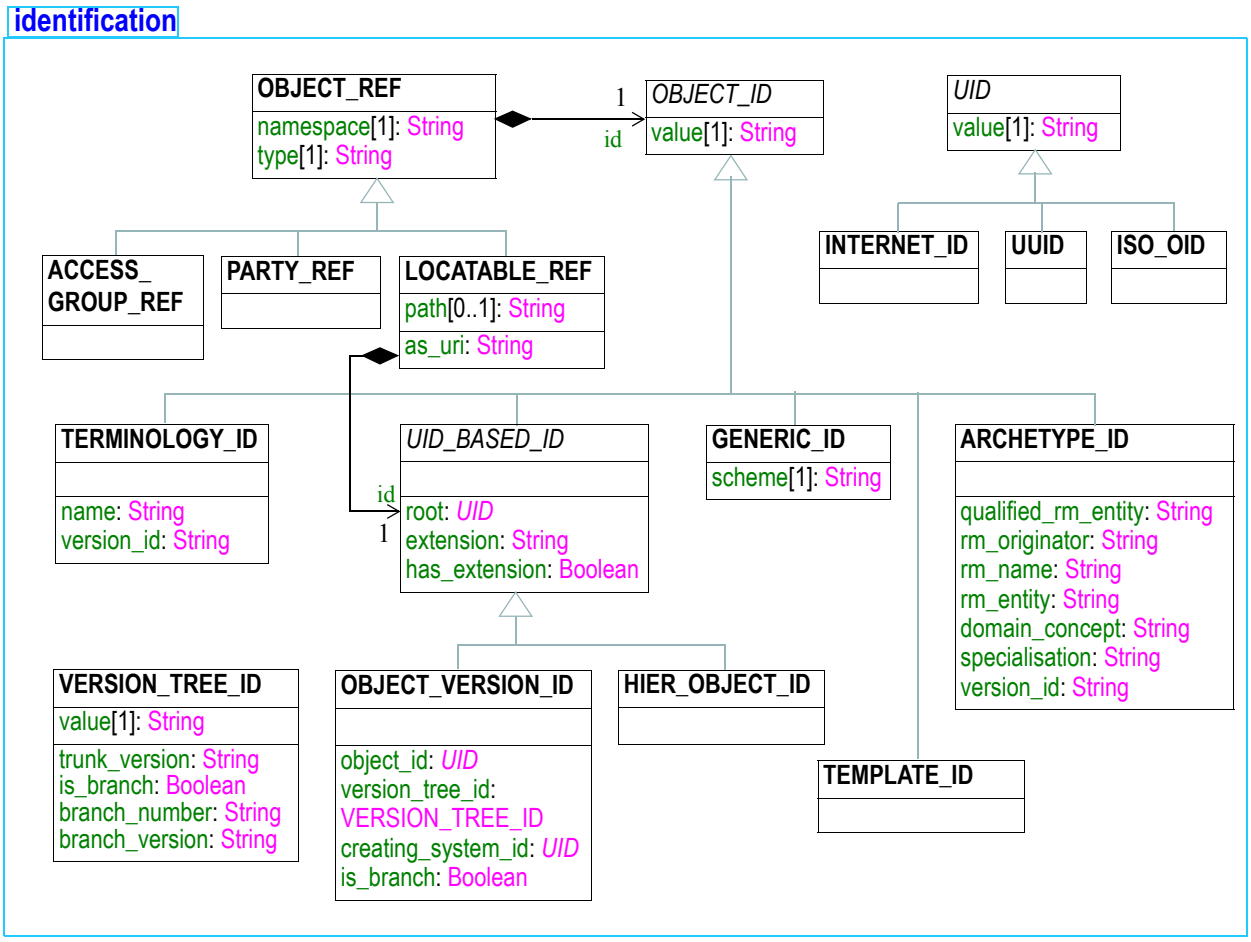

FIGURE 5 rm.support.identification Package

4.1.1 Requirements

Identification of entities both in the real world and in information systems is a non-trivial problem. The needs for identification across systems in a health information environment include the following:

- real world identifiers such as social security numbers, veterans affairs ids etc can be recorded as required by health care facilities, enterprise policies, or legislation;
- identifiers for informational entities which represent real world entities or processes should be unique;
- it should be possible to determine if two identifiers refer to information entities that represent the same real world entity, even if instances of the information entities are maintained in different systems;
- versions or changes to real-world entity-linked informational entities (which may create new information instances) should be accounted for in two ways:

  - it should be possible to tell if two identifiers refer to distinct versions of the same informational entity in the same version tree;
  - it should not be possible to confuse same-named versions of informational entities maintained in multiple systems which purport to represent the same real world entity. E.g. there is no guarantee that two systems' "latest" version of the Person "Dr Jones" is the same.

  Medico-legal use of information relies on previous states of information being distinguishable from other previous states and the current state.
- It should be possible for an entity in one system or service (such as the EHR) to refer to an entity in another system or service in such a way that:
  - the target of the reference is easily finable within the shared environment, and
  - the reference does is valid regardless of the physical architecture of servers and applications.

The following subsections describe some of the features and challenges of identification.

Identification of Real World Entities (RWEs)

Real world entities such as people, car engines, invoices, and appointments can all be assigned identifiers. Although many of these are designed to be unique within a jurisdiction, they are often not, due to data entry errors, bad design (ids that are too small or incorporate some non-unique characteristic of the identified entities), bad process (e.g. non-synchronised id issuing points); identity theft (e.g. via theft of documents of proof or hacking). In general, while some real world identifiers (RWIs) are "nearly unique", none can be guaranteed so. It should also be the case that if two RWE identifiers are equal, they refer to the same RWE, but this is often not the case. For practical purposes, RWIs cannot be regarded as computationally safe for making the inferences described here.

Identification of Informational Entities (IEs)

As soon as information systems are used to record facts about RWEs, the situation becomes more complex because of the intangible nature of information. In particular:

- the same RWE can be represented simultaneously on more than one system ('spatial multiplicity');
- the same RWE may be represented by more than one "version" of the same IE in a system ('temporal multiplicity').

At first sight, it appears that there can also be *purely* informational entities, i.e. IEs which do not refer to any RWE, such as books, online-only documents and software. However, as soon as one considers an example it becomes clear that there is always a notional 'definitive' or 'authoritative' (i.e. trusted) version of every such entity. These entities can better be understood as 'virtual RWEs'. Thus it can still be said that multiple IEs may refer to any given RWE.

The underlying reason for the multiplicity of IEs is that 'reality' - time and space - in computer systems is not continuous but discrete, and each 'entity' is in fact just a snapshot of certain attribute values of a RWE, at a point in time, in a particular system. If identifiers are assigned to IEs without regard to versions or duplicates, then no assertion can be made about the identified RWE when two IE ids are compared.

Identification of Versions

The notion of 'versioning' applies only to informational entities, i.e. distinct instances of content each representing a snapshot of some logical entity. Where such instances are stored and managed in versioned containers within a versioning system of some kind, explicit identification of the versions is

required. The requirements are discussed in detail in the Common IM, change_control package. They can be summarised as follows:

- it must be possible to distinguish two versions of the same logical entity, i.e. know from the identifier if they are the same or different versions of the same thing;
- it must be possible to distinguish two versions of the same logical entity created in two distinct systems;
- it must be possible to tell the relationship between the items in a versioned lineage, from the version identifiers.

Referencing of Informational Entities

Within a distributed information environment, there is a need for entities not connected by direct references in the same memory space to be able to refer to each other. There are two competing requirements:

- that the separation of objects in a distributed computing environment not compromise the semantics of the model;
- that different types of information can be managed relatively independently; for example EHR and demographic information can be managed by different groups in an organisation or community, each with at least some freedom to change implementation and model details.

4.2 Design

This package models only informational identifiers, i.e. transparent identifiers understood by *open*EHR or related computational systems. Real World Entity Identifiers such as driver's license numbers are modelled using the data type DV_IDENTIFIER. This is not to imply that such identifiers are any less systematic or well-managed than the system identifiers defined here, only that from the point of view of *open*EHR, they have the same status as other informational attributes such as name, address etc of a Person.

A key design decision has been to choose a string representation for all identifiers, with subparts being made available by appropriate functions which perform simple parsing on the string. This ensures that the data representation of identifiers (e.g. in XML) is as small as possible, while not losing object-oriented typing.

4.2.1 Primitive Identifiers

Three kinds of types are defined in this package. The abstract UID type and its subtypes correspond to permanent, computationally reliable, primitive identifiers. Such identifiers are regarded as 'primitive' because they are treated as having no further internal structure, in the sense that part of such an identifier is not in general meaningful. The three subtypes UUID, ISO_OID and INTERNET_ID all have these properties, and are commonly accepted ways of uniquely identifying entities in computer systems. In *open*EHR (and generally in health informatics) they are usually used as parts of other identifiers.

A consequence of the string representation approach used in these classes is that to set an attribute of type UID from a string value, as would be done when reading from a database, deserialising from XML or another text form, a piece of code that inspects the string structure has to be used in order to decide which of the subtypes of UID it is. This is a safe thing to do, since all three subtypes have mutually exclusive string patterns, and can easily be distinguished.

4.2.2 Composite Identifiers

The OBJECT_ID type and its hierarchy of subtypes define all of the identifier types used within *open*EHR systems. Most of these have a multi-part structure, and some are 'meaningful' i.e. human readable. The identifier types can be used to represent identifier values that fall into two groups semantically: those defined by *open*EHR (which may incorporate generic standard identifiers, such as ISO Oids etc) and those defined by external organisations. The groups are as shown in the following table. Identifiers whose form is defined by the HIER_OBJECT_ID type are used both by *open*EHR and many other organisations.

*open*EHR-defined identifiers	Externally defined identifiers
OBJECT_VERSION_ID	TERMINOLOGY_ID
ARCHEYTPE_ID	GENERIC_ID
TEMPLATE_ID	HIER_OBJECT_ID
HIER_OBJECT_ID	

4.2.2.1 UID-based Identifiers

The abstract type UID_BASED_ID and its two subtypes HIER_OBJECT_ID and OBJECT_VERSION_ID provide respectively, UID-based identifiers for non-versioned and versioned items. The design of the latter subtype is explained in the *open*EHR Common IM, change_control package.

4.2.2.2 Archetype Identifiers

The ARCHETYPE_ID subtype defines a multi-axial identifier for archetypes, meaning that each identifier instance denotes a single archetype within a multi-dimensional space. The space is can be thought of as 3-dimensional, or as a versioned 2-dimensional space, consisting of the following axes:

- reference model entity, i.e. target of archetype, defined as:
  - name of model issuer;
  - name of model (there may be more than one from the same issuer);
  - name of concept in model, i.e. class name
- domain concept;
- version.

The three outer sections are delimited by '.' characters, while the parts of the first section are delimited by '-' characters. As with any multi-axial identifier, the underlying principle of an archetype identifier is that all parts of the identifier must be able to be considered immutable. This means that no variable characteristic of an archetype (e.g. accrediting authority, which might change due to later accreditation by another authority, or may be multiple) can be included in its identifier. The explicit inclusion of version as part of the identifier means that two 'versions' of an archetype are actually two distinct archetypes. (The rules for archetype versions, revisions and other variants are given in the *open*EHR Archetype Semantics specification[1].)

Examples of archetype identifiers include:

- openEHR-EHR-SECTION.physical_examination.v2
- openEHR-EHR-SECTION.physical_examination-prenatal.v1
- Hl7-RIM-Act.progress_note.v1
- openEHR-EHR-OBSERVATION.progress_note-naturopathy.v2

1. See http://www.openehr.org/releases/1.0.1/architecture/am/archetype_semantics.pdf

The grammar of archetype identifiers is given below in section 4.3.10 on page 44.

WARNING: some archetype authoring tools have historically allowed a non-conforming version part within archetype identifiers which included the lifecycle status. This has led to some archetypes having an identifier whose version part is of the form '.v1draft' or similar. The *open*EHR Foundation will publish guidelines and a timeline on its website for dealing with this problem. New and existing archetype tools may have to support this exception, depending on where they are to be used, and it is recommended that it at least be supported via a command line switch or option. Where such non-conforming archetypes are re-used within a new environment, the identifier should be corrected.

4.2.2.3 Template Identifiers

The template identifier is similar in intention to the achetype identifier - it provides a multi-axial, readable identifier in addition to any UID-style of identifier that may be used. In this release, the exact structure has not been defined, but a current proposal is as follows:

- authoring organisation reverse domain name;
- identifier of reference model class being templated;
- logical name of the template;
- version identifier in the form 'vn' where n is a numerical version identifier.

This would lead to identifiers like the following:

- `uk.nhs.cfh:openehr-EHR-COMPOSITION.admission_ed.v5`

A firm form of template identifiers will be described in a future release.

4.2.2.4 Terminology Identifiers

The `TERMINOLOGY_ID` subtype defines a globally unique single string identifier for terminologies. Terminology identifier values may include a version, either as part of the name, and/or according to the syntax defined in section 4.3.12 below. Examples of terminology identifiers include:

- `"SNOMED-CT"`
- `"ICD9(1999)"`

Currently the best authoritative source for the *name* part of the identifier (i.e. the part excluding the optional version part in parentheses) is the US National Library of Medicine UMLS identifiers for included terminologies - see http://www.nlm.nih.gov/research/umls/metaa1.html.

The scheme defined by the `TERMINOLOGY_ID` class provides for the situation where major 'versions' of a terminology such as the World Health Organisation's 'ICD10' and 'ICD10AM' (AM = 'Australian modifications') can accommodate a finer grain of versioning or revisioning, e.g.:

- `"ICD10AM(3rd_ed)"`
- `"ICD10AM(4th_ed)"`

The *version* part of a terminology identifier is in theory only absolutely necessary for those terminologies which break the rule that the concept being identified with a code loses or changes its meaning over versions of the terminology. This should not be the case for modern terminologies and ontologies, particularly those designed since the publication of Cimino's 'desiderata' [1] of which the principle of 'concept permanance' is applicable here - "A concept's meaning cannot change and it cannot be deleted from the vocabulary". However, there may be older terminologies, or specialised terminologies which may not have obeyed these rules, but which are still used; version ids should always be used for these. At a practical level, versions may be included routinely in some systems to support the potential medico-legal need to prove that a) a given code was in fact defined in the terminology (it

may not have existed in an earlier edition) and b) that the meaning assmued in the system was indeed the one assigned to it in the particular version or edition.

Equivalence

Although there are anomalies in some published terminologies and between some versions or editions of the same terminology, two terminology identifiers that are the same, disregarding the version part, can usually be considered as semantic equivalents in the terminology world. However, depending on which source of strings have been chosen for the name part of the identifier, two different identifiers may also indicate the same terminology, e.g. “ICD10AM_2000” (NLM identifier used in UMLS) and “ICD10AM(2nd_ed)” refer to the same thing.

4.2.2.5 Identifying Versions within *open*EHR Versioned Containers

The `OBJECT_VERSION_ID` defines the semantics of the scheme used in *open*EHR for identifying versions within a versioned container, and uses a three-part identifier, consisting of:

- *object_id*: the identifier of the version container, in the form of an `UID`;
- *version_tree_id*: the location in the version tree, as a 1- or 3-part numeric identifier, where the latter variant expresses branching; this is modelled using the `VERSION_TREE_ID` type;
- *creating_system_id*: the identifier of the system in which this version was created, or type `UID`.

Under this scheme, multiple versions in the same container all have the same value for *object_id*, while their location in the version tree is given by the combination of the version tree identifier and the identifier of the creating system.

The requirements on the third part of the identifier are that it be unique per system, and that it be easy to obtain or generate. It is also helpful if it is a meaningful identifier. The two most practical candidates appear to be GUIDs (which are not meaningful, but are easy to generate) and reverse internet domain identifiers, as recommended in [3] (these are easy to determine if the system has an internet address, and are meaningful and directly processible, however unconnected systems pose a problem). ISO Oids might also be used. All of these identifier types are accommodated via the use of `UID`.

A full explanation of the version identification scheme and its capabilities is given in the `change_control` section of the Common IM.

4.2.2.6 Generic and External Identifiers

The `GENERIC_ID` type provides for identifiers of schemes other than defined concretely in the `rm.support.identification` package. It has a single method *scheme*, which may be used to record the identifier type. The names of schemes are not currently controlled.

4.2.2.7 Hierarchical Identifiers

The `HIER_OBJECT_ID` type is defined to support hierarchical identifiers, often based on ISO Oids or other similar machine-readable and -resolvable schemes.

4.2.2.8 Composite Identifiers and Case

All composite identifiers should follow two rules with regard to case, namely:

- to be *case-preserving* - not change case due to persistence, copying, transfer or other computation processes;
- to be *case-insensitive* - two identifiers identical apart from case are considered to be identical, and therefore to identify the same thing.

The practical consequences of these rules are as follows:

- mixed-case identifiers may be used, such as archetype identifiers, mixed-case reverse domain identifiers (the INTERNET_ID type);
- the original case chosen in the letters of identifiers on creation within an *open*EHR system should be as published by the relevant issuing organisation (e.g. NLM UMLS terminology names are all upper case);
- if identifiers are used as part of filenames within computer file systems, care must be taken to create and preserve filenames correctly. For this reason, software usually has to handle filename creation and modification differently on Unix-style operating systems, which are case-sensitive (and therefore case-preserving), and Windows-style operating systems, which are case-insensitive but usually case-preserving.

These rules do not apply to any identifier constructed in a language in which case does not exist as a concept. For this reason, for identifiers translated in and out of the Turkish language (and possibly in smaller related languages), care must be taken with the 'I/i' characters.

4.2.2.9 Composite Identifiers and Language

In all of the 'meaningful' identifier types above, with the posible exception of GENERIC_ID, the human-readable identifier sections are assumed to use only the basic latin character set, possibly with the addition of other special characters as allowed by the production rules defined below for each identifier. In most cases, the textual parts of these identifiers will be words from the English language, or else they will be recognisable words from other languages, where necessary alliterated into the latin alphabet. Accented and other diacritical letter variants are not allowed. This limitation is made in the interests of practical computability of identifiers, and is in common with class and attribute naming in shared UML models in the standards world, and also with internet domain names and internet URIs.

4.2.3 References

All OBJECT_IDs are used as identifier attributes *within* the thing they identify, in the same way as a database primary key. To *refer* to an identified object from another object, an instance of the class OBJECT_REF should generally be used, in the same way as a database foreign key. The class OBJECT_REF is provided as a means of distributed referencing, and includes the object namespace (typically 1:1 with some service, such as "terminology") and type. The general principle of object references is to be able to refer to an object available in a particular namespace or service. Usually they are used to refer to objects in other services, such as a demographic entity from within an EHR, but they may be used to refer to local objects as well. The type may be the concrete type of the referred-to object (e.g. "GP") or any proper ancestor (e.g. "PARTY").

4.3 Class Descriptions

4.3.1 UID Class

CLASS	*UID (abstract)*
Purpose	Abstract parent of classes representing unique identifiers which identify information entities in a durable way. UIDs only ever identify one IE in time or space and are never re-used.
HL7	The HL7v3 UID Data type.

CLASS	*UID (abstract)*	
Attributes	**Signature**	**Meaning**
	value: `String`	The value of the id.
Invariant	***Value_exists***: value /= Void ***and then not*** value.empty	

4.3.2 ISO_OID Class

CLASS	ISO_OID	
Purpose	Model of ISO's Object Identifier (oid) as defined by the standard ISO/IEC 8824 . Oids are formed from integers separated by dots. Each non-leaf node in an Oid starting from the left corresponds to an assigning authority, and identifies that authority's namespace, inside which the remaining part of the identifier is locally unique.	
HL7	The HL7v3 OID Data type.	
Inherit	`UID`	
Functions	**Signature**	**Meaning**
Invariant		

4.3.3 UUID Class

CLASS	UUID	
Purpose	Model of the DCE Universal Unique Identifier or UUID which takes the form of hexadecimal integers separated by hyphens, following the pattern 8-4-4-4-12 as defined by the Open Group, CDE 1.1 Remote Procedure Call specification, Appendix A. Also known as a GUID.	
HL7	The HL7v3 UUID Data type.	
Inherit	`UID`	
Functions	**Signature**	**Meaning**
Invariant		

4.3.4 INTERNET_ID Class

CLASS	INTERNET_ID	
Purpose	Model of a reverse internet domain, as used to uniquely identify an internet domain. In the form of a dot-separated string in the reverse order of a domain name, specified by IETF RFC 1034 (http://www.ietf.org/rfc/rfc1034.txt).	
Inherit	`UID`	
Functions	**Signature**	**Meaning**
Invariant		

4.3.4.1 Syntax

According to IETF RFC1034, the syntax of a domain name follows the BNF grammar:

```
domain: subdomain | ' '
subdomain: label | subdomain '.' label
label: letter [ [ ldh-str ] let-dig ]
ldh-str: let-dig-hyp | let-dig-hyp ldh-str
let-dig-hyp: let-dig | '-'
let-dig: letter | digit

letter: any one of the 52 alphabetic characters A through Z in
upper case and a through z in lower case

digit: any one of the ten digits 0 through 9
```

It can also be expressed using the regular expression:

```
[a-zA-Z]([a-zA-Z0-9-]*[a-zA-Z0-9])?(\.[a-zA-Z]([a-zA-Z0-9-]*[a-zA-Z0-9]))*
```

4.3.5 OBJECT_ID Class

CLASS	*OBJECT_ID (abstract)*	
Purpose	Ancestor class of identifiers of informational objects. Ids may be completely meaningless, in which case their only job is to refer to something, or may carry some information to do with the identified object.	
Use	Object ids are used inside an object to identify that object. To identify another object in another service, use an `OBJECT_REF`, or else use a UID for local objects identified by UID. If none of the subtypes is suitable, direct instances of this class may be used.	
Attributes	**Signature**	**Meaning**
	value: `String`	The value of the id in the form defined below.
Invariant	***Value_exists***: value /= Void ***and then not*** value.empty	

4.3.6 UID_BASED_ID Class

CLASS	*UID_BASED_ID (abstract)*	
Purpose	Abstract model of UID-based identifiers consisting of a root part and an optional extension; lexical form: `root '::' extension`	
Inherit	*OBJECT_ID*	
Functions	**Signature**	**Meaning**
1..1	**root**: `UID`	The identifier of the conceptual namespace in which the object exists, within the identification scheme. Returns the part to the left of the first '::' separator, if any, or else the whole string.
1..1	**extension**: `String`	Local identifier of the object, if given, within the context of the root identifier. Returns the part to the right of the first '::' separator if any, or else an empty String.
	has_extension: `Boolean` ***ensure*** **not** extension.is_empty **implies** Result	True if extension is not empty.
Invariant	***Root_valid***: root /= Void ***Extension_validity***: extension /= Void ***Has_extension_validity***: extension.is_empty **xor** has_extension	

4.3.6.1 Identifier Syntax

The syntax of the *value* attribute by default follows the following production rules (EBNF):

```
value:      root [ '::' extension ]
root:       uid                                -- see UID above
extension:  string
```

4.3.7 HIER_OBJECT_ID Class

CLASS	HIER_OBJECT_ID	
Purpose	Concrete type corresponding to hierarchical identifiers of the form defined by `UID_BASED_ID`.	
HL7	The HL7v3 II Data type.	
Inherit	*UID_BASED_ID*	
Functions	**Signature**	**Meaning**

CLASS	HIER_OBJECT_ID
Invariant	

4.3.8 OBJECT_VERSION_ID Class

CLASS	OBJECT_VERSION_ID	
Purpose	Globally unique identifier for one version of a versioned object; lexical form: `object_id '::' creating_system_id '::' version_tree_id`	
Inherit	*UID_BASED_ID*	
Functions	**Signature**	**Meaning**
1..1	**object_id**: `UID`	Unique identifier for logical object of which this identifier identifies one version; normally the *object_id* will be the unique identifier of the version container containing the version referred to by this `OBJECT_VERSION_ID` instance.
1..1	**version_tree_id**: `VERSION_TREE_ID`	Tree identifier of this version with respect to other versions in the same version tree, as either 1 or 3 part dot-separated numbers, e.g. "1", "2.1.4".
1..1	**creating_system_id**: `UID`	Identifier of the system that created the Version corresponding to this Object version id.
	is_branch: `Boolean`	True if this version identifier represents a branch.
Invariants	***Object_valid***: object_id /= Void ***Version_tree_id_valid***: version_tree_id /= Void ***creating_system_id_valid***: creating_system_id /= Void	

4.3.8.1 Identifier Syntax

The string form of an `OBJECT_VERSION_ID` stored in its *value* attribute consists of three segments separated by double colons ("::"), i.e. (EBNF):

```
value:              object_id '::' creating_system_id '::' version_tree_id
object_id:          uid          -- see UID below
creating_system_id: uid          -- see UID below
version_tree_id:                 -- see VERSION_TREE_ID below
```

An example is as follows:

```
F7C5C7B7-75DB-4b39-9A1E-C0BA9BFDBDEC::87284370-2D4B-
4e3d-A3F3-F303D2F4F34B::2
```

4.3.9 VERSION_TREE_ID Class

CLASS	VERSION_TREE_ID	
Purpose	Version tree identifier for one version. Lexical form: `trunk_version [ '.' branch_number '.' branch_version ]`	
Attributes	**Signature**	**Meaning**
1..1	**value**: `String`	String form of this identifier.
Functions	**Signature**	**Meaning**
1..1	**trunk_version**: `String`	Trunk version number; numbering starts at 1.
0..1	**branch_number**: `String`	Number of branch from the trunk point; numbering starts at 1.
0..1	**branch_version**: `String`	Version of the branch; numbering starts at 1.
1..1	**is_branch**: `Boolean`	True if this version identifier represents a branch, i.e. has *branch_number* and *branch_version* parts.
1..1	**is_first**: `Boolean`	True if this version identifier corresponds to the first version, i.e. *trunk_version* = "1"
Invariants	***Value_valid***: value /= Void **and then not** value.is_empty ***Trunk_version_valid***: trunk_version /= Void **and then** trunk_version.is_integer **and then** trunk_version.as_integer >= 1 ***Branch_number_valid***: branch_number /= Void **implies** branch_number.is_integer **and then** branch_number.as_integer >= 1 ***Branch_version_valid***: branch_version /= Void **implies** branch_version.is_integer **and then** branch_version.as_integer >= 1 ***Branch_validity***: (branch_number = Void **and** branch_version = Void) **xor** (branch_number /= Void **and** branch_version /= Void) ***Is_branch_validity***: is_branch **xor** branch_number = Void ***Is_first_validity***: **not** is_first **xor** trunk_version.is_equal("1")	

4.3.9.1 Syntax

The format of the value attribute is (EBNF):

```
value:          trunk_version [ '.' branch_number '.' branch_version ]
trunk_version:      { digit }+
branch_number:      { digit }+
branch_version:     { digit }+
digit:              [0-9]
```

4.3.10 ARCHETYPE_ID Class

CLASS	ARCHETYPE_ID	
Purpose	Multi-axial structural identifier type for archetypes. The syntax is defined by a small grammar and also expressed as a regular expression pattern.	
Inherit	*OBJECT_ID*	
Functions	**Signature**	**Meaning**
1..1	**qualified_rm_entity**: `String`	Globally qualified reference model entity, e.g. "openehr-composition-OBSERVATION".
1..1	**domain_concept**: `String`	Name of the concept represented by this archetype, including specialisation, e.g. "biochemistry_result-cholesterol".
1..1	**rm_originator**: `String`	Organisation originating the reference model on which this archetype is based, e.g. "openehr", "cen", "hl7".
1..1	**rm_name**: `String`	Name of the reference model, e.g. "rim", "ehr_rm", "en13606".
1..1	**rm_entity**: `String`	Name of the ontological level within the reference model to which this archetype is targeted, e.g. for openEHR, "folder", "composition", "section", "entry".
1..1	**specialisation**: `String`	Name of specialisation of concept, if this archetype is a specialisation of another archetype, e.g. "cholesterol".
1..1	**version_id**: `String`	Version of this archetype.
Invariant	***Qualified_rm_entity_valid***: qualified_rm_entity /= Void **and then not** qualified_rm_entity.is_empty ***Domain_concept_valid***: domain_concept /= Void **and then not** domain_concept.is_empty ***Rm_originator_valid***: rm_originator /= Void **and then not** rm_originator.is_empty ***Rm_name_valid***: rm_name /= Void **and then not** rm_name.is_empty ***Rm_entity_valid***: rm_entity /= Void **and then not** rm_entity.is_empty ***Specialisation_valid***: specialisation /= Void **implies not** specialisation.is_empty ***Version_id_valid***: version_id /= Void **and then not** version_id.is_empty	

4.3.10.1 Archetype ID Syntax

The syntax of an `ARCHETYPE_ID` is formally defined as follows:

```
# -------- production rules --------
archetype_id: qualified_rm_entity '.' domain_concept '.' version_id

qualified_rm_entity: rm_originator '-' rm_name '-' rm_entity
rm_originator: V_ALPHANUMERIC_NAME
rm_name: V_ALPHANUMERIC_NAME
rm_entity: V_ALPHANUMERIC_NAME

domain_concept: concept_name { '-' specialisation }*
concept_name: V_ALPHANUMERIC_NAME
specialisation: V_ALPHANUMERIC_NAME

version_id: 'v' V_NONZERO_DIGIT [ V_NUMBER ]

# -------- lexical patterns --------
V_ALPHANUMERIC_NAME: [a-zA-Z][a-zA-Z0-9_]+
V_NONZERO_DIGIT: [1-9]
V_NUMBER: [0-9]+
```

The field meanings are as follows:

rm_originator: id of organisation originating the reference model on which this archetype is based;

rm_name: id of the reference model on which the archetype is based;

rm_entity: ontological level in the reference model;

domain_concept: the domain concept name, including any specialisations;

version_id: numeric version identifier.

The PERL regular expression equivalent of the above is as follows:

```
[a-zA-Z]\w+(-[a-zA-Z]\w+){2}\.[a-zA-Z]\w+(-[a-zA-Z]\w+)*\.v[1-9]\d*
```

The classic regular expression equivalent of this is generated from the above with the following substitutions:

```
\w -> [a-zA-Z0-9_]
\d -> [0-9]
```

4.3.11 TEMPLATE_ID Class

CLASS	TEMPLATE_ID	
Purpose	Identifier for templates. Lexical form to be determined.	
Inherit	*OBJECT_ID*	
Functions	**Signature**	**Meaning**
Invariant		

4.3.12 TERMINOLOGY_ID Class

CLASS	TERMINOLOGY_ID	
Purpose	Identifier for terminologies such accessed via a terminology query service. In this class, the value attribute identifies the Terminology in the terminology service, e.g. “SNOMED-CT”. A terminology is assumed to be in a particular language, which must be explicitly specified. The value if the id attribute is the precise terminology id identifier, including actual release (i.e. actual “version”), local modifications etc; e.g. “ICPC2”. Lexical form: `name [ '(' version ')' ]`	
Inherit	*OBJECT_ID*	
Functions	**Signature**	**Meaning**
1..1	**name**: `String`	Return the terminology id (which includes the “version” in some cases). Distinct names correspond to distinct (i.e. non-compatible) terminologies. Thus the names “ICD10AM” and “ICD10” refer to distinct terminologies.
1..1	**version_id**: `String`	Version of this terminology, if versioning supported, else the empty string.
Invariants	***Name_valid***: name /= Void **and then not** name.is_empty ***Version_id_valid***: version_id /= Void	

4.3.12.1 Identifier Syntax

The syntax of the *value* attribute is as follows:

```
# -------- production rules --------
terminology_id: name [ '(' version ')' ]
name: V_NAME
version: V_VERSION

# -------- lexical patterns --------
V_NAME: [a-zA-Z][a-zA-Z0-9_-/+]+
V_VERSION: [a-zA-Z0-9][a-zA-Z0-9_-/.]+
```

4.3.13 GENERIC_ID Class

CLASS	GENERIC_ID
Purpose	Generic identifier type for identifiers whose format is othterwise unknown to *open*EHR. Includes an attribute for naming the identification scheme (which may well be local).
Inherit	*OBJECT_ID*

CLASS	GENERIC_ID	
attributes	**Signature**	**Meaning**
1..1	**scheme**: `String`	Name of the scheme to which this identifier conforms. Ideally this name will be recognisable globally but realistically it may be a local *ad hoc* scheme whose name is not controlled or standardised in any way.
Invariants	***Scheme_valid***: scheme /= Void **and then not** scheme.is_empty	

4.3.14 OBJECT_REF Class

CLASS	OBJECT_REF	
Purpose	Class describing a reference to another object, which may exist locally or be maintained outside the current namespace, e.g. in another service. Services are usually external, e.g. available in a LAN (including on the same host) or the internet via Corba, SOAP, or some other distributed protocol. However, in small systems they may be part of the same executable as the data containing the Id.	
Attributes	**Signature**	**Meaning**
1..1	**id**: *`OBJECT_ID`*	Globally unique id of an object, regardless of where it is stored.
1..1	**namespace**: `String`	Namespace to which this identifier belongs in the local system context (and possibly in any other *open*EHR compliant environment) e.g. “terminology”, “demographic”. These names are not yet standardised. Legal values for the namespace are `“local”	“unknown”	“[a-zA-Z][a-zA-Z0-9_-:/&+?]*”`
1..1	**type**: `String`	Name of the class (concrete or abstract) of object to which this identifier type refers, e.g. “PARTY”, “PERSON”, “GUIDELINE” etc. These class names are from the relevant reference model. The type name “ANY” can be used to indicate that any type is accepted (e.g. if the type is unknown).
Invariant	***Id_exists***: id /= Void ***Namespace_exists***: namespace /= Void ***and then not*** namespace.is_empty ***Type_exists***: type /= Void ***and then not*** type.is_empty	

4.3.15 ACCESS_GROUP_REF Class

CLASS	ACCESS_GROUP_REF	
Purpose	Reference to access group in an access control service.	
Inherit	OBJECT_REF	
Functions	**Signature**	**Meaning**
Invariant	***Type_validity***: type.is_equal("ACCESS_GROUP")	

4.3.16 PARTY_REF Class

CLASS	PARTY_REF	
Purpose	Identifier for parties in a demographic or identity service. There are typically a number of subtypes of the PARTY class, including PERSON, ORGANISATION, etc. Abstract supertypes are allowed if the referenced object is of a type not known by the current implementation of this class (in other words, if the demographic model is changed by the addition of a new PARTY or ACTOR subtypes, valid PARTY_REFs can still be constructed to them).	
Inherit	OBJECT_REF	
Functions	**Signature**	**Meaning**
Invariant	***Type_validity***: type.is_equal("PERSON") **or** type.is_equal("ORGANISATION") **or** type.is_equal("GROUP") **or** type.is_equal("AGENT")**or** type.is_equal("ROLE") **or** type.is_equal("PARTY") **or** type.is_equal("ACTOR")	

4.3.17 LOCATABLE_REF Class

CLASS	LOCATABLE_REF	
Purpose	Reference to a LOCATABLE instance inside the top-level content structure inside a VERSION<T>; the *path* attribute is applied to the object that VERSION.*data* points to.	
Inherit	OBJECT_REF	
Attributes	**Signature**	**Meaning**
1..1 (redefined)	**id**: OBJECT_VERSION_ID	The identifier of the Version.

CLASS	LOCATABLE_REF	
0..1	**path**: `String`	The path to an instance in question, as an absolute path with respect to the object found at `VERSION.`*data*. An empty path means that the object referred to by id being specified.
Functions	**Signature**	**Meaning**
1..1	**as_uri**: `String`	A URI form of the reference, created by concatenating the following: `"ehr://" + id.value + "/" + path`
Invariant	***Path_valid***: path /= Void **implies not** path.is_empty	

5 Terminology Package

5.1 Overview

This section describes the `terminology` package, which contains classes for accessing terminologies and code sets, including the *open*EHR Support Terminology, from within instances of classes defined in the reference model. The classes shown here would normally be inherited via the classes `EXTERNAL_ENVIRONMENT_ACCESS` and `OPENEHR_DEFINITIONS`, although the exact details of how this is done may vary depending on implementation language.

5.2 Service Interface

5.2.1 Code Sets

A simple terminology service interface is defined according to FIGURE 6, enabling *open*EHR code sets and terminology to be referenced formally from within the Reference Model. Two types of coded entities are distinguished in *open*EHR, and are accessible via the service interface. The first is codes from 'code sets', which are the kind of terminology where the code stands for itself, such as the ISO 639-1 language codes. The identifiers themselves of these code sets do not appear to be standardised, but names such as "ISO_639-1" are expected to be used (see below).

In any case, code sets needed within the *open*EHR models themselves (e.g. for attributes whose value is a language code) are not referred to directly by an external name such as "ISO_639-1", but via an internal constant, in this case, the constant Code_set_id_languages, whose value is defined to be "languages". These constants are defined in the class `OPENEHR_CODE_SET_IDENTIFIERS` in FIGURE 6. The mapping between the internal identifiers and external names should be done in configuration files. The service function `TERMINOLOGY_SERVICE`.*code_set_for_id()* is used to retrieve code sets on the basis of a constant. The current mapping and external identifiers assumed in *open*EHR is defined in the *open*EHR Support Terminology document. This use of indirection is employed to ensure that the obsoleting and superseding of code-sets does not directly affect *open*EHR software.

For code sets not mapped to internally used constants, i.e. code sets not required in the *open*EHR model itself, but otherwise known in the terminology service, the function `TERMINOLOGY_SERVICE`.*code_set()* can be used to retrieve these code sets by their external identifier.

5.2.2 Terminologies

Terminologies, including the *open*EHR Support Terminology are accessed via the `TERMINOLOGY_SERVICE` functions *terminology()* and *terminology_identifiers()*, where the argument includes "openehr", "centc251" (for CEN TC/251codes) and names from the US NLM terminologies list (see below). The *open*EHR Terminology supports groups, and the set of groups required by the reference model is defined in the class `OPENEHR_TERMINOLOGY_GROUP_IDENTIFIERS`. These groups correspond to coded attributes found in the *open*EHR Reference Model.

5.2.3 Terms and Codes in the *open*EHR Reference Model

True coded attributes in the Reference Model (i.e. attributes of type `DV_CODED_TEXT`), such as `FEEDER_AUDIT`.*change_type* are defined by an invariant in the enclosing class, such as the following:

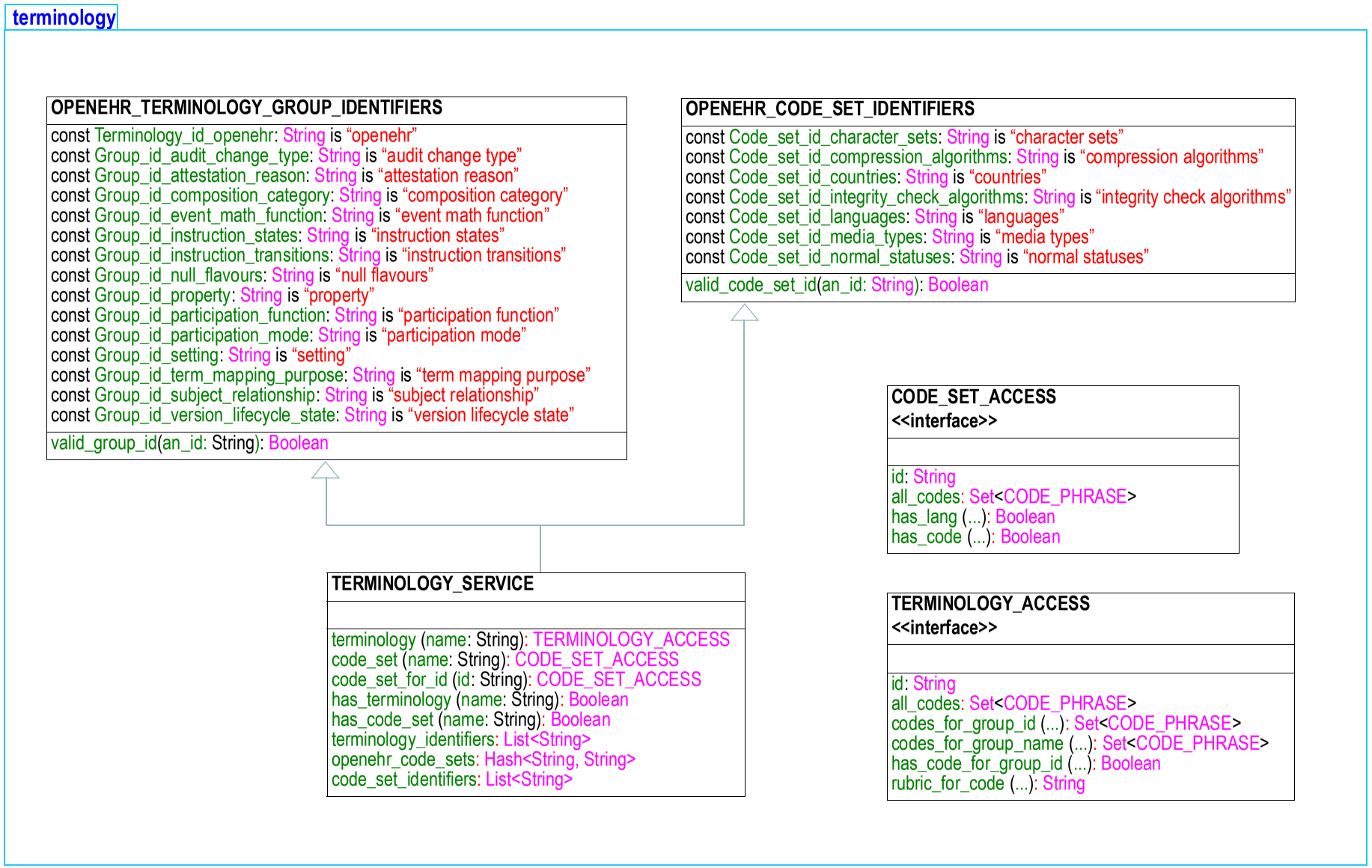

FIGURE 6 rm.support.terminology Package

Change_type_valid:terminology(Terminology_id_openehr).has_code_for_group_id (Group_id_audit_change_type, change_type.defining_code)

This is a formal way of saying that the attribute *change_type* must have a value such that its *defining_code* (its CODE_PHRASE) is in the set of CODE_PHRASEs in the *open*EHR Terminology which are in the group whose indentifier is Group_id_audit_change_type.

A similar invariant is used for attributes of type CODE_PHRASE, which come from a code_set. The following invariant appears in the class ENTRY (rm.composition.content.entry package):

Language_valid: media_type /= Void ***and then***
code_set(Code_set_languages).has_code(language)

5.3 Identifiers

In *open*EHR, the identifier of a terminology or code set is found in the *terminology_id* attribute of the class CODE_PHRASE (Data Types Information Model, text package).

5.3.1 Code Set Identifiers

Internal code set identifiers (such as "languages") used in *open*EHR are defined in the class OPENEHR_CODE_SET_IDENTIFIERS; assumed external identifiers (such as "ISO_639-1") for code sets used by the *open*EHR Reference Model are defined in the *open*EHR Support Terminology document.

5.3.2 Terminology Identifiers

Valid identifiers that can be used for this attribute for terminologies include but are not limited to the following:

- "openehr"
- "centc251"
- an identifier value from the first column of the US National Library or Medicine (NLM) UMLS terminology identifiers table below, in either of two forms:
  - as is, e.g. "ICD10AM_2000", "ICPC93";
  - with any trailing section starting with an underscore removed, e.g. "ICD10AM".

Other identification schemes are used in some standards, such as ISO Oids. These are not specified for direct use in *open*EHR for various reasons:

- they are not currently used by the NLM, and no definitive published list of terminology identifiers is available;
- ISO Oids are long identifiers and may significantly increase the size of persisted information due to the ubiquity of coded terms;
- determing the identity of the terminology in data always requires a request to a service containing the Oid / name mapping;
- there is a safety factor in having human readable terminology identifiers in the data.

The use of Oid-based or other terminology identification schemes is not however incompatible with *open*EHR; all that is required is a terminology identifier / name mapping service or table.

The following table is a snapshot of the US National Library of Medicine UMLS terminology identifiers list. A definitive up-to-date list may be found on the NLM website at http://www.nlm.nih.gov/research/umls/metaa1.html.

UMLS 2003 Terminology Identifiers	
Identifier	**Description**
AIR93	AI/RHEUM,1993
ALT2000	Alternative Billing Concepts, 2000
AOD2000	Alcohol and Other Drug Thesaurus, 2000
BI98	Beth Israel Vocabulary, 1.0
BRMP2002	Portuguese translation of the Medical Subject Headings, 2002
BRMS2002	Spanish translation of the Medical Subject Headings, 2002
CCPSS99	Canonical Clinical Problem Statement System, 1999
CCS99	Clinical Classifications Software, 1999
CDT4	Current Dental Terminology(CDT), 4
COSTAR_89-95	COSTAR, 1989-1995
CPM93	Medical Entities Dictionary, 1993
CPT01SP	Physicians' Current Procedural Terminology, Spanish Translation, 2001
CPT2003	Physicians' Current Procedural Terminology, 2003
CSP2002	CRISP Thesaurus, 2002
CST95	COSTART, 1995
DDB00	Diseases Database, 2000
DMD2003	German translation of the Medical Subject Headings, 2003
DMDICD10_1995	German translation of ICD10, 1995
DMDUMD_1996	German translation of UMDNS, 1996
DSM3R_1987	DSM-III-R, 1987
DSM4_1994	DSM-IV, 1994
DUT2001	Dutch Translation of the Medical Subject Headings, 2001
DXP94	DXplain, 1994
FIN2003	Finnish translations of the Medical Subject Headings, 2003
HCDT4	HCPCS Version of Current Dental Terminology(CDT), 4
HCPCS03	Healthcare Common Procedure Coding System, 2003
HCPT03	HCPCS Version of Current Procedural Terminology(CPT), 2003
HHC96	Home Health Care Classification, 1996
HL7_1998-2002	Health Level Seven Vocabulary, 1998-2002
HLREL_1998	ICPC2E-ICD10 relationships from Dr. Henk Lamberts, 1998
HPC99	Health Product Comparison System, 1999
ICD10AE_1998	ICD10, American English Equivalents, 1998
ICD10AMAE_2000	International Statistical Classification of Diseases and Related Health Problems, Australian Modification, Americanized English Equivalents, 2000
ICD10AM_2000	International Statistical Classification of Diseases and Related Health Problems, 10th Revision, Australian Modification, January 2000 Release
ICD10_1998	ICD10, 1998

UMLS 2003 Terminology Identifiers	
Identifier	**Description**
ICD9CM_2003	ICD-9-CM, 2003
ICPC2AE_1998	International Classification of Primary Care, Americanized English Equivalents, 2E, 1998
ICPC2E_1998	International Classification of Primary Care 2nd Edition, Electronic, 2E, 1998
ICPC2P_2000	International Classification of Primary Care, Version2-Plus, 2000
ICPC93	International Classification of Primary Care, 1993
ICPCBAQ_1993	ICPC, Basque Translation, 1993
ICPCDAN_1993	ICPC, Danish Translation, 1993
ICPCDUT_1993	ICPC, Dutch Translation, 1993
ICPCFIN_1993	ICPC, Finnish Translation, 1993
ICPCFRE_1993	ICPC, French Translation, 1993
ICPCGER_1993	ICPC, German Translation, 1993
ICPCHEB_1993	ICPC, Hebrew Translation, 1993
ICPCHUN_1993	ICPC, Hungarian Translation, 1993
ICPCITA_1993	ICPC, Italian Translation, 1993
ICPCNOR_1993	ICPC, Norwegian Translation, 1993
ICPCPAE_2000	International Classification of Primary Care ,Version2-Plus, Americanized English Equivalents, 2000
ICPCPOR_1993	ICPC, Portuguese Translation, 1993
ICPCSPA_1993	ICPC, Spanish Translation, 1993
ICPCSWE_1993	ICPC, Swedish Translation, 1993
INS2002	French translation of the Medical Subject Headings, 2002
ITA2003	Italian translation of Medical Subject Headings, 2003
JABL99	Online Congenital Multiple Anomaly/ Mental Retardation Syndromes, 1999
LCH90	Library of Congress Subject Headings, 1990
LNC205	LOINC, 2.05
LOINC	LOINC
MCM92	McMaster University Epidemiology Terms, 1992
MDDB99	MasterDrug DataBase, 1999
MDR51	Medical Dictionary for Regulatory Activities Terminology (MedDRA), 5.1
MDRAE51	Medical Dictionary for Regulatory Activities Terminology (MedDRA), American English Equivalents, 5.1
MDREA51	Medical Dictionary for Regulatory Activities Terminology (MedDRA), American English, with expanded abbreviations, 5.1
MDREX51	Medical Dictionary for Regulatory Activities Terminology (MedDRA), with expanded abbreviations, 5.1
MDRPOR51	Medical Dictionary for Regulatory Activities Terminology (MedDRA), 5.1, Portuguese Edition

UMLS 2003 Terminology Identifiers	
Identifier	**Description**
MDRSPA51	Medical Dictionary for Regulatory Activities Terminology (MedDRA), 5.1, Spanish Edition
MIM93	Online Mendelian Inheritance in Man, 1993
MMSL01	Multum MediSource Lexicon, 2001
MMX01	Micromedex DRUGDEX, 2001-08
MSH2003_2002_10_24	Medical Subject Headings, 2002_10_24
MTH	UMLS Metathesaurus
MTHCH03	Metathesaurus CPT Hierarchical Terms, 2003
MTHHH03	Metathesaurus HCPCS Hierarchical Terms, 2003
MTHICD9_2003	Metathesaurus additional entry terms for ICD-9-CM, 2003
MTHMST2001	Metathesaurus Version of Minimal Standard Terminology Digestive Endoscopy, 2001
MTHMSTFRE_2001	Metathesaurus Version of Minimal Standard Terminology Digestive Endoscopy, French Translation, 2001
MTHMSTITA_2001	Metathesaurus Version of Minimal Standard Terminology Digestive Endoscopy, Italian Translation, 2001
NAN99	Classification of Nursing Diagnoses, 1999
NCBI2001	NCBI Taxonomy, 2001
NCI2001a	NCI Thesaurus, 2001a
NCISEER_1999	NCISEER ICD Neoplasm Code Mappings, 1999
NDDF01	FirstDataBank National Drug DataFile, 2001-07
NEU99	Neuronames Brain Hierarchy, 1999
NIC99	Nursing Interventions Classification, 1999
NOC97	Nursing Outcomes Classification, 1997
OMIM97	OMIM, Online Mendelian Inheritance in Man, 1997
OMS94	Omaha System, 1994
PCDS97	Patient Care Data Set, 1997
PDQ2002	Physician Data Query, 2002
PPAC98	Pharmacy Practice Activity Classification , 1998
PSY2001	Thesaurus of Psychological Index Terms, 2001
QMR96	Quick Medical Reference (QMR), 1996
RAM99	QMR clinically related terms from Randolph A. Miller, 1999
RCD99	Clinical Terms Version 3 (CTV3) (Read Codes), 1999
RCDAE_1999	Read thesaurus, American English Equivalents, 1999
RCDSA_1999	Read thesaurus Americanized Synthesized Terms, 1999
RCDSY_1999	Read thesaurus, Synthesized Terms, 1999
RUS2003	Russian Translation of MeSH, 2003
RXNORM_03AA	RXNORM Project, META2003AA
SNM2	SNOMED-2, 2
SNMI98	SNOMED International, 1998

UMLS 2003 Terminology Identifiers	
Identifier	**Description**
SNOMED-CT	SNOMED International Clinical Terms, 2002
SPN02	Standard Product Nomenclature, 2002
SRC	Metathesaurus Source Terminology Names
ULT93	UltraSTAR, 1993
UMD2003	UMDNS: product category thesaurus, 2003
UMLS	UMLS: National Library of Medicine, USA
UWDA155	University of Washington Digital Anatomist, 1.5.5
VANDF01	Veterans Health Administration National Drug File, 2001
WHO97	WHO Adverse Reaction Terminology, 1997
WHOFRE_1997	WHOART, French Translation, 1997
WHOGER_1997	WHOART, German Translation, 1997
WHOPOR_1997	WHOART, Portuguese Translation, 1997
WHOSPA_1997	WHOART, Spanish Translation, 1997

5.4 Class Definitions

5.4.1 TERMINOLOGY_SERVICE Class

CLASS	TERMINOLOGY_SERVICE	
Purpose	Defines an object providing proxy access to a terminology service.	
Inherit	OPENEHR_CODE_SET_IDENTIFIERS, OPENEHR_TERMINOLOGY_GROUP_IDENTIFIERS	
Functions	**Signature**	**Meaning**
	terminology (name: String): TERMINOLOGY_ACCESS ***require*** name /= Void **and then** has_terminology (name) ***ensure*** Result /= Void	Return an interface to the terminology named *name*. Allowable names include • "openehr" • "centc251" • any name from are taken from the US NLM UMLS meta-data list at http://www.nlm.nih.gov/research/umls/metaa1.html
	code_set (name: String): CODE_SET_ACCESS ***require*** name /= Void **and then** has_code_set (name) ***ensure*** Result /= Void	Return an interface to the code_set identified by the external identifier *name* (e.g. "ISO_639-1").
	code_set_for_id(id: String): CODE_SET_ACCESS ***require*** id /= Void **and then** valid_code_set_id (id) ***ensure*** Result /= Void	Return an interface to the code_set identified internally in *open*EHR by *id*.
	has_terminology (name: String): Boolean ***require*** name /= Void **and then** not name.is_empty	True if terminology named *name* known by this service. Allowable names include • "openehr" • "centc251" • any name from are taken from the US NLM UMLS meta-data list at http://www.nlm.nih.gov/research/umls/metaa1.html

CLASS	TERMINOLOGY_SERVICE	
	has_code_set (name: `String`): `Boolean` ***require*** name /= Void **and then** not name.is_empty	True if code_set linked to internal *name* (e.g. "languages") is available.
	terminology_identifiers: `List<String>`	Set of all terminology identifiers known in the terminology service. Values from the US NLM UMLS meta-data list at http://www.nlm.nih.gov/research/umls/metaa1.html
	code_set_identifiers: `List<String>`	Set of all code set identifiers known in the terminology service.
	openehr_code_sets: `Hash<String, String>`	Set of all code sets identifiers for which there is an internal *open*EHR name; returned as a Hash of ids keyed by internal name.
Invariants		

5.4.2 TERMINOLOGY_ACCESS Class

CLASS	TERMINOLOGY_ACCESS	
Purpose	Defines an object providing proxy access to a terminology.	
Functions	**Signature**	**Meaning**
	id: `String`	Identification of this Terminology
	all_codes: `Set<CODE_PHRASE>`	Return all codes known in this terminology
	codes_for_group_id (group_id: `String`): `Set<CODE_PHRASE>`	Return all codes under grouper 'group_id' from this terminology
	has_code_for_group_id (group_id: `String`; a_code: `CODE_PHRASE`): `Boolean`	True if 'a_code' is known in group 'group_id' in the *open*EHR terminology.
	codes_for_group_name (name, lang: **String**): `Set<CODE_PHRASE>`	Return all codes under grouper whose name in 'lang' is 'name' from this terminology
	rubric_for_code (code, lang: `String`): `String`	Return all rubric of code 'code' in language 'lang'.
Invariants	***id_exists***: id /= Void **and then not** id.is_empty	

5.4.3 CODE_SET_ACCESS Class

CLASS	CODE_SET_ACCESS	
Purpose	Defines an object providing proxy access to a code_set.	
Functions	**Signature**	**Meaning**
	id: `String`	External identifier of this code set
	all_codes: `Set<CODE_PHRASE>`	Return all codes known in this code set
	has_lang (a_lang: `CODE_PHRASE`): `Boolean`	True if code set knows about 'a_lang'
	has_code (a_code: `CODE_PHRASE`): `Boolean`	True if code set knows about 'a_code'
Invariants	***Id_valid***: id /= Void **and then not** id.is_empty	

5.4.4 OPENEHR_TERMINOLOGY_GROUP_IDENTIFIERS Class

CLASS	OPENEHR_TERMINOLOGY_GROUP_IDENTIFIERS	
Purpose	List of identifiers for groups in the *open*EHR terminology.	
Constants	**Signature**	**Meaning**
	Terminology_id: `String` is "openehr"	Name of *open*EHR's own terminology
	Group_id_audit_change_type: `String` is "audit change type"	
	Group_id_attestation_reason: `String` is "attestation reason"	
	Group_id_composition_category: `String` is "composition category"	
	Group_id_event_math_function: `String` is "event math function"	
	Group_id_instruction_states: `String` is "instruction states"	
	Group_id_instruction_transitions: `String` is "instruction transitions"	
	Group_id_null_flavours: `String` is "null flavours"	

CLASS	OPENEHR_TERMINOLOGY_GROUP_IDENTIFIERS	
	Group_id_property: `String` is “property”	
	Group_id_participation_function: `String` is “participation function”	
	Group_id_participation_mode: `String` is “participation mode”	
	Group_id_subject_relationship: `String` is “subject relationship”	
	Group_id_setting: `String` is “setting”	
	Group_id_term_mapping_purpose: `String` is “term mapping purpose”	
	Group_id_version_lifecycle_state: `String` is “version lifecycle state”	
Functions	**Signature**	**Meaning**
	valid_terminology_group_id (an_id: `String`): `Boolean`	Validity function to test if an identifier is in the set defined by this class.
Invariants		

5.4.5 OPENEHR_CODE_SET_IDENTIFIERS Class

CLASS	OPENEHR_CODE_SET_IDENTIFIERS	
Purpose	List of identifiers for code sets in the *open*EHR terminology.	
Constants	**Signature**	**Meaning**
	Code_set_id_character_sets: `String` is “character sets”	
	Code_set_id_compression_algorithms: `String` is “compression algorithms”	
	Code_set_id_countries: `String` is “countries”	
	Code_set_id_integrity_check_algorithms: `String` is “integrity check algorithms”	
	Code_set_id_languages: `String` is “languages”	

CLASS	OPENEHR_CODE_SET_IDENTIFIERS	
	Code_set_id_media_types: `String` is "media types"	
	Code_set_id_normal_statuses: `String` is "normal statuses"	
Functions	**Signature**	**Meaning**
	valid_code_set_id (an_id: `String`): `Boolean`	Validity function to test if an identifier is in the set defined by this class.
Invariants		

6 Measurement Package

6.1 Overview

The Measurement package defines a minimum of semantics relating to quantitative measurement, units, and conversion, enabling the Quantity package of the *open*EHR Data Types Information Model to be correctly expressed. As for the Terminology package, a simple service interface is assumed, which provides useful functions to other parts of the reference model. The definitions underlying measurement and units come from a variety of sources, including:

- CEN ENV 12435, Medical Informatics - Expression of results of measurements in health sciences (see http://www.centc251.org);
- the Unified Code for Units of Measure (UCUM), developed by Gunther Schadow and Clement J. McDonald of The Regenstrief Institute (available in HL7v3 ballot materials; http://www.hl7.org).

These of course rest in turn upon a vast amount of literature and standards, mainly from ISO on the subject of scientific measurement.

6.2 Service Interface

A simple measurement data service interface is defined according to FIGURE 7, enabling quantitative semantics to be used formally from within the Reference Model. Note that this service as currently defined in no way seeks to properly model the semantics of units, conversions etc - it provides only the minimum functions required by the *open*EHR Reference Model.

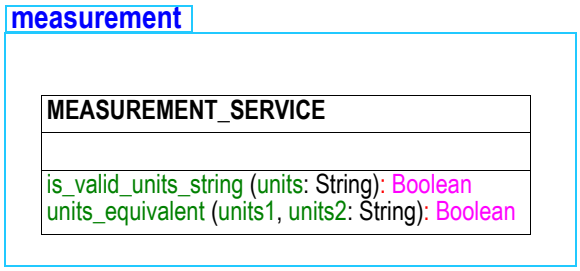

FIGURE 7 rm.support.measurement Package

6.2.1 Class Definitions

6.2.1.1 MEASUREMENT_SERVICE Class

CLASS	MEASUREMENT_SERVICE	
Purpose	Defines an object providing proxy access to a measurement information service.	
Functions	**Signature**	**Meaning**
	is_valid_units_string (units: `String`): `Boolean` ***require*** units /= Void	True if the units string 'units' is a valid string according to the HL7 UCUM specification.

CLASS	MEASUREMENT_SERVICE	
	units_equivalent (units1, units2: `String`): `Boolean` ***require*** units1 /= Void ***and then*** is_valid_units_string(units1) units2 /= Void ***and then*** is_valid_units_string(units2)	True if two units strings correspond to the same measured property.
Invariants		

7 Definition Package

7.1 Overview

The `definition` package, illustrated in FIGURE 8, defines symbolic definitions used by the *open*EHR models. Only a small number are currently defined.

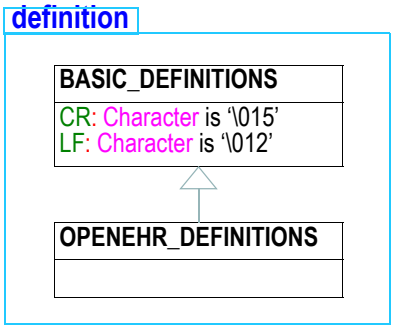

FIGURE 8 rm.support.definition Package

7.2 Class Definitions

7.2.1 OPENEHR_DEFINITIONS Class

CLASS	OPENEHR_DEFINITIONS	
Purpose	Inheritance class to provide access to constants defined in other packages.	
Inherit	`BASIC_DEFINITIONS`	
Attributes	**Signature**	**Meaning**
Invariants		

7.2.2 BASIC_DEFINITIONS Class

CLASS	BASIC_DEFINITIONS	
Purpose	Defines globally used constant values.	
Attributes	**Signature**	**Meaning**
	CR: Character is '\015'	Carriage return character
	LF: Character is '\012'	Linefeed character
Invariants		