INTERNATIONAL STANDARD

ISO/IEC 20944-2

First edition
2013-01-15

Information technology — Metadata Registries Interoperability and Bindings (MDR-IB) —

Part 2: Coding bindings

Technologies de l'information — Interopérabilité et liaisons des registres de métadonnées (MDR-IB) —

Partie 2: Liaisons de codage

Reference number
ISO/IEC 20944-2:2013(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 20944-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

ISO/IEC 20944 consists of the following parts, under the general title *Information technology — Metadata Registries Interoperability and Bindings (MDR-IB)*:

— *Part 1: Framework, common vocabulary, and common provisions for conformance*

— *Part 2: Coding bindings*

— *Part 3: API bindings*

— *Part 4: Protocol bindings*

— *Part 5: Profiles*

Introduction

The ISO/IEC 20944 series of International Standards provides the bindings and their interoperability for metadata registries, such as those specified in the ISO/IEC 11179 series of International Standards.

This part of ISO/IEC 20944 contains provisions that are common to coding bindings (Clauses 4-10) and the coding bindings themselves (Clause 11 onward). The coding bindings have commonality in their conceptualization of data instances and their internal structures. For example, common features include:

— using datatypes to characterize the nature and operations upon data;

— using ISO/IEC 11404 to define and declare datatypes;

— using common aggregate structures, such as array and record, to describe sets of data;

— using common navigation descriptions to reference components within a set of data.

The individual coding bindings each incorporate a mapping of common data semantics to their individual binding requirements.

Information technology — Metadata Registries Interoperability and Bindings (MDR-IB) —

Part 2:
Coding bindings

1 Scope

The ISO/IEC 20944 series of International Standards describes codings, application programming interfaces (APIs), and protocols for interacting with an ISO/IEC 11179 metadata registry (MDR).

This part of ISO/IEC 20944 specifies provisions that are common across coding bindings for the ISO/IEC 20944 series. This part of ISO/IEC 20944 includes the individual coding bindings themselves.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 11404:2007, *Information technology — General-Purpose Datatypes (GPD)*

ISO/IEC 20944-1:2013, *Information technology — Metadata Registries Interoperability and Bindings (MDR-IB) — Framework, common vocabulary, and common provisions for conformance*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 20944-1 apply.

4 Intended use of this part of ISO/IEC 20944

Bindings concern the mapping of one standard (or framework) into another standard (or framework).[1]

Coding bindings concern the mapping of instances of data models to code elements (representations of data).

More than one standard (or framework) may be used to complete the mapping.[2]

The ISO/IEC 20944 series of International Standards uses at least three tiers of mappings. The first tier concerns the main kind of mapping for the binding: coding bindings, API bindings, and protocol bindings. The

[1] For example, an ASN.1 binding of "technical specification XYZ" implies mapping the features and requirements of "technical specification XYZ" to the features and capabilities of the ASN.1 standard.

[2] When viewed as a series of layers (e.g., data model = Standard XYZ, coding binding = ASN.1, encoding = ASN.1 Basic Encoding Rules (BER)), bindings may also be viewed as "layered standards" or a "layering of standards".

second tier of mapping concerns the mapping of data model instances to a coding-independent representation (CIR) of data — ISO/IEC 11404 specifies the syntax and semantics of this CIR. The third tier of mapping concerns the mapping of CIR to a coding-specific representation (CSR) — Clauses 11 and onward describe the coding-specific mappings. Additional tiers of mapping are possible, such as specifications of encoding mappings.[3]

The purpose of a common CIR of data is to support common semantics and interoperability among coding bindings and other bindings, such as API and protocol bindings.

NOTE XML bindings are useful for integration with other XML technologies. The XML binding provides the requirements of XML interchange without specifying the XML technologies (e.g., XML Schema) to implement the features. DIVP bindings are useful for integration with name-value pair technologies, such as scripting systems, E-mail, and web servers.

EXAMPLE A CIR is developed for a data model. Based upon this CIR, it is possible to transform one coding binding (e.g., XML coding binding) to another coding binding (e.g., ASN.1 coding binding) while sharing common semantics (the CIR) among the coding bindings. Likewise, one coding-specific representation (e.g., XML) can be transformed into another coding-specific representation (e.g., ASN.1).

5 Abstract model

The coding bindings have commonality in their conceptualization of data instances and their internal structures. For example, common features include:

— using datatypes to characterize the nature and operations upon data
— using ISO/IEC 11404 to define and declare datatypes
— using common aggregate structures, such as array and record, to describe sets of data
— using common navigation descriptions to reference components within a set of data

The individual coding bindings (Clauses 11 and onward) each incorporate a mapping of common data semantics to their individual binding requirements.

5.1 Overview of data objects

The conceptual model of data objects is divided into two parts: the data structuring model and the navigation model. The data structuring model describes the "logical" structure of data as it is transferred. The navigation model describes the mapping of the logical structure to a navigation hierarchy.

NOTE The use of hierarchical naming does not imply a hierarchical data model or metadata model.

5.2 Referenced data interchange specification

The ISO/IEC 20944 series of International Standards is organized by an implicit data interchange specification. This data interchange specification is external to the coding, API, and protocol bindings. (See footnote 2.)

[3] The XML, ASN.1, and DIVP coding bindings all have additional tiers for encoding. XML has two additional encoding tiers: character encoding (e.g., ASCII vs. UTF-8), and a lower level byte-ordering encoding for multi-octet representations (e.g., little endian and big endian orderings for UTF-16 and UTF-32). ASN.1 has an additional layer of encoding, Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). DIVP may have encoding rules specified by its outer wrapper, but they are not part of the DIVP coding binding.

5.3 Data structuring model

5.3.1 Data objects

5.3.1.1 General

A unit or collection of data is called a "data object".[4] Data objects are structured data characterized by the following.

5.3.1.2 Atomic data objects

An object may be an "atomic data object" if the data object has values which are intrinsically indivisible, e.g., a basic type such as integer, real, character.

EXAMPLE `17`, `-1.7E8`, `0D19980831235959`, `"hello world"`.

5.3.1.3 Aggregates

An aggregate is a collection of data objects (atomic or not). Each member of the aggregate is called a "component". The components may be ordered or unordered, named or unnamed, typed or untyped. An aggregate may be nested, i.e., a component of aggregate may be an aggregate itself.

EXAMPLE The list `{ x: 17 y: { 18 19 20 } z: 0D19980831232359 }` contains three named data elements `x`, `y`, and `z`. The first component (`x`) is an atomic data object of type integer. The second component (`y`) is an aggregate of three unnamed component. The third component (`z`) is an atomic data object of type date-and-time. The difference between an aggregate and a C programming language structure is: a C structure is always ordered, named, and typed; whereas an aggregate may be ordered, named, and/or typed.

5.3.1.4 Hierarchical organization

The nesting implies a hierarchical organization only from the perspective of data access, not from data implementation. For example, a multi-dimensional array may use the same access method as a nested list. As an analogy, when comparing the C programming language two-dimensional array char x[10][20] to the nested list char *y[10], both types are accessed by the same hierarchical method: x[a][b] and y[a][b].

5.3.1.5 Datatypes

Data objects may have a datatype. ISO/IEC 11404 includes basic types (e.g., boolean, integer, real, date-time, string), type parameters (e.g., precision), and type operators (e.g., lists, records, sets).

Implementations shall support the ISO/IEC 11404 datatypes [5]. Additional supported datatypes are implementation-defined.

5.3.2 Properties

5.3.2.1 General

Properties can be associated with data objects. Properties are attributes of the data object.

5.3.2.2 Property lists

Each data object may have an associated property list. A property list is an ordered, named list of data objects.

EXAMPLE The data object `{ 1 2 3 }` might have the property list `{ read_write_access: read_only size: 123 }`.

---

4 Not to be confused with "objects" of "object-oriented" analysis, design, and programming.

5 ISO/IEC 11404 includes a description of datatype conformity.

5.3.2.3 System and user properties

Some properties are created by the underlying system, e.g., `_type`, `_shape`, `_last_modified`. Some properties are created by the user (or application), e.g., `read_write_access`, `size`. Designation conventions (e.g., a leading underscore for system properties) help avoid collisions between user/application properties and system properties.

5.3.2.4 Static and dynamic properties

Static properties exist for the duration of the object (unless explicitly removed) and can be listed, e.g., `_type`, color. Dynamic properties might not be enumerable and might not exist for the duration of the object.

5.3.2.5 Stored and calculated properties

Some properties have explicit storage associated with the object, e.g., color and (possibly) `_type`. Some properties may be implicit or calculated and have no storage, e.g., `_type` and `_shape`.

5.3.2.6 Memory and volatile properties

Some properties behave the like "memory", i.e., "reading" the property more than once always returns the same value, and "writing" the same value more than once has the same affect as writing it once.

EXAMPLE The properties color and price are "memory"-like properties because they remain the same. The property `_last_modified` is "volatile"-like because the value may change; the property `usage_payment` is "volatile"-like because setting its value more than once (e.g., paying a usage fee of 0.50 USD more than once) might have a different effect than setting the value only once.

5.3.2.7 Properties of the Datatype

Some properties are inherent to the datatype itself, such as "ordered" vs. "unordered". GPD specifies some of these properties. Other properties may exist.

5.4 Designations, identifiers, and navigation

5.4.1 General

Non-atomic objects imply more than one component. Designation methods facilitate navigation of structured data in objects.

5.4.2 Identifiers for navigation

An identifier is a designation that is used for referencing.[6] The meaning of the navigation is defined by the URI syntax, as overlaid with indexes and hierarchies.

5.4.2.1 Designations for indexes

Elements in ordered lists may be accessed by number, e.g., the element numbered `2` of the list `{ 15 16 17 }` is `17`. Elements in lists may be accessed by identifier, e.g., the element number `z` of the list `{ x: 15 16 z: 17 }` is `17`. Labeled elements (i.e., elements with an identifier) of ordered lists may be accessed by identifier or number, e.g., the third element of `{ x: 15 16 z: 17 }` may be accessed by the identifier `z` or the number `2`. Index numbers begin at zero.

6 As used in this context, the term "identifier" is a designation used for navigation.

5.4.2.2 Hierarchical identifiers

Identifiers are hierarchical because the objects they navigate have hierarchical naming. These hierarchical naming conventions, as well as all other forms of external data identifiers that permits hierarchical navigation are also called hierarchical navigation paths.

EXAMPLE The names `C/Z`, `C/2`, `2/Z`, and `2/2` all designate the element `17` in the data object `{ A: 10 B: 11 C: { X: 15 Y: 16 Z: 17 } }`.

5.4.2.3 Merged navigation identifiers

The object, property, link, control, etc., navigation identifiers are merged with the data object navigation identifiers to simplify the number of methods of access, e.g., no need for separate **`getvalue`** and **`getpropery`** operations.

EXAMPLE The property `Y` of the object **`X`** is accessed via the name **`X/.Y`**. For the soft link of `L` that points to `D`: (1) getting the value `L` "chases" the link and returns the value of `D`, (2) getting the value `L/` refers to the link itself, i.e., the reference to `D`, not the value of `D`, (3) getting the value `L/.` chases the link and retrieves `D`.

5.4.2.4 Designation conventions

Several designation conventions are used to increase interoperability and reduce integration cost. The user (application) uses external identifiers. The system (implementation) uses internal identifiers. External identifiers are mapped to/from logical identifiers. A logical identifiers is a sequence of pathname separators and pathname segments. Each pathname segment consists of words separated by word separators. Logical identifiers are mapped to/from internal identifiers.

EXAMPLE The external designation conventions use MIME names, while the internal identifiers use C programming language conventions. The external MIME name **`Abc-Def-Ghi/3/Jkl-Mno`** is mapped to the logical pathname: the first path segment is the words **`Abc`**, **`Def`**, and **`Ghi`**; the second path segment is the word `3`; the third path segment is the words **`Jkl`** **`Mno`**. The logical identifier is mapped to the internal C programming language identifier **`Abc_Def_Ghi[3].Jkl_Mno`** or, possibly, **`Abc_Def_Ghi[3]->Jkl_Mno`**.

5.4.2.5 Hard links

A data object may be accessed by more than one identifier. A hard link is a first class reference to an data object. A data object exists until all its hard links have been removed, then the data object is destroyed.

5.4.2.6 Soft links

An alternate identifier may be created for an data object. A soft link is a second class reference: the link may still exist after the target data object is destroyed. Soft links are implicitly "chased" (automatically "dereferenced").

EXAMPLE If the soft link `L` points to `D`, then (1) getting the value `L` "chases" the link and returns the value of `D`, (2) getting the value `L/` refers to the link itself, i.e., the reference to `D`, not the value of `D`, (3) getting the value `L/.` chases the link and retrieves `D`.

5.4.2.7 Reference

An identifier or pointer to a data object. A reference is a second class name: the link may still exists after the target object is destroyed. References must be explicitly "chased" (or "dereferenced").

5.4.2.8 Views

A view represents a subset of structure data. A simple view might be a subtree of structure data. A complex view may be an SQL select query to describe to subset.

Views describe subsets of information. A view may be used to address the need of creating functional subsets.

6 Semantics

6.1 Datatypes

The ISO/IEC 20944 datatypes are based upon ISO/IEC 11404.

6.2 Inherent structure

Objects contain data that is navigated by hierarchical navigation identifiers.[7]

EXAMPLE

```
struct
{
    int id;
    char name[100];
    time_t datetime;
};
```

A C programming language structure is represented conceptually as:

```
   Name         Value
+-----------+-------------+
| id        | 11223344    |
+-----------+-------------+
| name      | John Doe    |
+-----------+-------------+
| datetime  | 19980102    |
+-----------+-------------+
```

The same conceptual structure may be used for XML:

```
<NAMEINFO>
    <ID>11223344</ID>
    <NAME>John Doe</NAME>
    <DATETIME>19980102</DATETIME>
</NAMEINFO>
```

6.3 Hierarchical naming

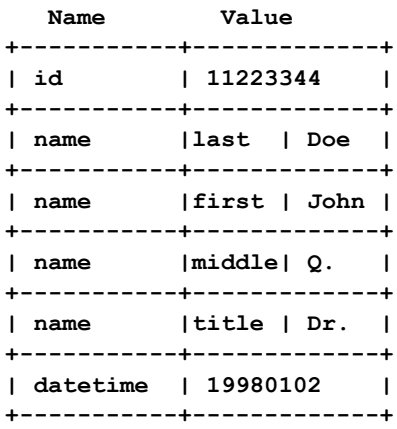
Data is "structured" and accessed via a hierarchical navigation identifier system.

EXAMPLE

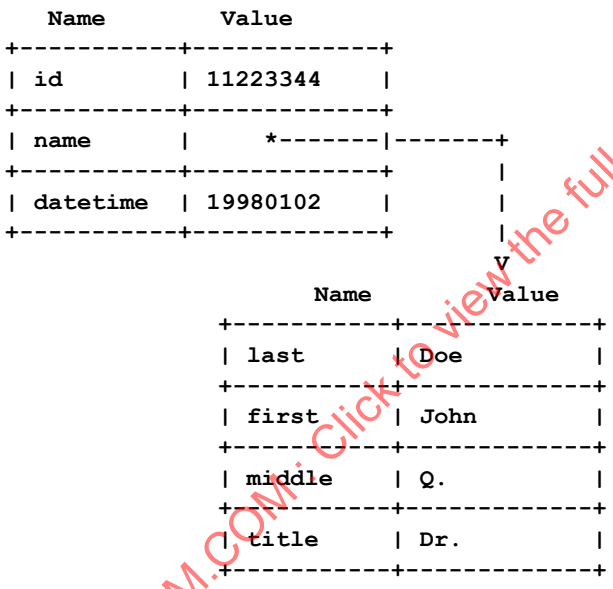
```
struct
{
    int id;
    struct
    {
        char last[40];
        char first[30];
        char middle[30];
        char title[10];
    } name;
    time_t datetime;
};
```

[7] This does not imply hierarchical data.

The nested C structure may be represented conceptually as:



The most important feature is the hierarchical navigation identifier. Using C syntax as an example, **`id`**, **`name.last`**, **`name.first`**, **`name.middle`**, **`name.title`**, **`datetime`**. Note that the naming is the key feature, not the implementation of the structure, so the following is an equivalent conceptual model from a MDRIB perspective:

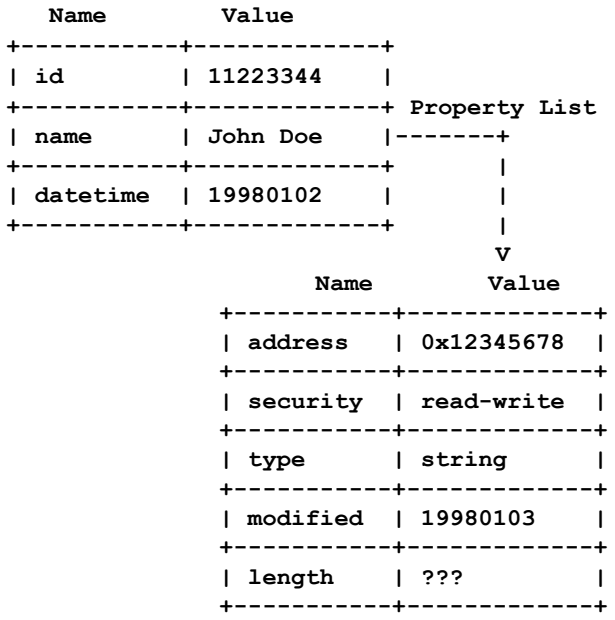


The above conceptual data structures could be represented by UNIX filesystems designation convention (e.g., **`id`**, **`name/last`**, **`name/first`**, **`name/middle`**, **`name/title`**, **`datetime`**) or by Windows Registry designation convention (e.g., **`id`**, **`name\last`**, **`name\first`**, **`name\middle`**, **`name\title`**, **`datetime`**).

These hierarchical naming conventions, as well as all other forms of external data identifiers that permits hierarchical navigation are also called "hierarchical navigation paths"

6.4 Associated properties

Data objects can have properties associated with them. Properties are, conceptually, list of identifier-value pairs. Properties may be system-defined (e.g., pointer address) or user-defined (e.g., security), static (e.g., type) or dynamic (e.g., last modified time), require storage (e.g., security) or calculated on demand (e.g., length).



6.5 Merged navigation identifiers for properties

The navigation identifiers of properties are merged with the navigation identifiers of data objects to simplify access to information, i.e., only a "getvalue" is needed rather than requiring both "get value" and "get property value" operations. Assuming designation conventions of `/` as a pathname separator and `.` as a property introduction, the `_type` property of the data object navigation identifier would be accessed via `name/._type`.

6.6 External, logical, and internal naming conventions

The designation conventions may vary. Conceptually, the "external identifier" is the name used in information interchange, e.g., the navigation identifier used in the "get" service. The "external identifier" is mapped to the "logical identifier". A "logical identifier" is comprised of a list of path name segments, each segment is a list of word segments. The "logical identifier" is mapped to the "internal identifier" within the implementation of the structured data.

Examples of external identifiers with varying designation conventions using path separators (e.g., "/", ".", "\") and word separators (e.g., '-', "_", or case change)::

```
Abc-Def/Ghi-Jkl/Mno-Pqr-Stu/123/456    # MIME or UNIX filesystem
abc_def.ghi_jkl.mno_pqr_stu[123][456]  # C language
AbcDef\GhiJkl\MnoPqrStu\123\456        # Windows Registry
((abc def) (ghi jkl) (mno pqr stu) 123 456)  # LISP
```

The above external names map into the following "logical identifier":

```
path segment #1:
        word #1: abc
        word #2: def
path segment #2:
        word #1: ghi
        word #2: jkl
path segment #3:
        word #1: mno
        word #2: pqr
        word #3: stu
path segment #4:
        word #1: 123
path segment #5:
        word #1: 456
```

The above logical identifier might map into any of the following "internal identifiers"

```
Abc-Def/Ghi-Jkl/Mno-Pqr-Stu/123/456     # MIME or UNIX filesystem
abc_def.ghi_jkl.mno_pgr_stu[123][456]   # C language
AbcDef\GhiJkl\MnoPqrStu\123\456         # Windows Registry
((abc def) (ghi jkl) (mno pqr stu) 123 456)   # LISP
```

The designation conventions for external navigation identifiers are independent of and automatically mapped to the internal designation conventions, e.g., a MIME naming convention might be used externally but a C language convention is used internally.

NOTE The above designation conventions are only examples. The designation conventions are defined via parametric specification.

6.7 The _value property

The **_value** property of a data object is identical to the value of the data object, e.g., reading or writing the value of "object" is identical to reading or writing the value of the property **_value** associated with the object, i.e., getting **object** is the same getting **object/._value**.

6.8 Keywords

Keywords are identifiers indicated by the prefix **"._"**. Keywords have special meaning, as defined below.

- **._typeas**: Returns the value converted to a specific type, e.g., **"zipcode/._typeas/integer"**.
- **._value**: The **"value"** property, which is identical to accessing the data object itself, e.g., **"foo/bar"** is the same as **"foo/bar/._value"**.
- **._prop**: A list of properties, i.e., all the "._" keywords.
- **._type**: The datatype of the object.
- **._label**: The label-only portion of the object, e.g., the value of **"foobar/._label"** is the string **"foobar"** (not the value of **foobar**).

7 Bindings

This part of ISO/IEC 20944 describes the common conceptual model and common semantics across all coding bindings. A conforming coding binding shall conform to the requirements described in Clauses 4, 5, 6, 8, and 9.

8 Administration

8.1 Use of registry-defined datatypes

A conforming implementation that uses datatypes that are defined by a registry shall conform to those requirements of that (dependent) registry. The timing of updates for a dependent registry shall be implementation-defined.

EXAMPLE For an implementation that uses a data element whose value domain is based upon a registry, that implementation shall document how the expected schedule for applying registry changes to the implementation (e.g., if a value domain is based on ISO 3166-1 country codes and a new country code is added, what is the expected schedule for incorporating the country code into the implementation?).

9 Conformance

9.1 Coding conformance paradigm

The conformance paradigm for ISO/IEC 20944 consists of the following conformance roles: data instance, data reader, data writer, data repository.

9.2 Data instance conformance

9.2.1 Data instance

Data instance conformance is independent of binding.

A strictly conforming data instance shall be a set of data that: (1) is structured independent of binding, (2) strictly conforms to the functionality, conceptual model, and semantics of the referenced data interchange specification, (3) shall include all mandatory data elements, (4) may include optional data elements, and (5) shall not include extended data elements.

A conforming data instance shall be a set of data that: (1) is structured independent of binding, (2) conforms to the functionality, conceptual model, and semantics of the referenced data interchange specification (3) shall include all mandatory data elements, (4) may include optional data elements, and (5) may include extended data elements.

Conformity assessment of data instances shall be performed by (1) rendering the data instance in ISO/IEC 11404 notation, and (2) verifying the requirements described by the referenced data interchange specification.

9.2.2 Bound data instance

A strictly conforming bound data instance shall (1) be a strictly conforming data instance, and (2) strictly conform to at least one coding binding.

A conforming bound data instance shall (1) be a conforming data instance, and (2) conform to at least one coding binding.

NOTE The difference between a SC/C data instance, a SC/C coding, and a SC/C bound data instance is: (1) a data instance is an instance of data that is independent of binding, (2) a coding can refer to an instance of data, a set of instances of data, or a syntax of instances of data, and (3) a strictly conforming/conforming bound data instance is associated with a specific binding.

Definitions: support, use

In the context of conformance, the terms "support" and "use" are defined individually in each coding binding.

Definitions: test, access, probe

In the context of conformance, the terms "test", "access", and "probe" are defined as the null operation, i.e., for data instance conformance, the operations "test", "access", and "probe" perform no operations and have no effect.

9.3 Data application conformance

There are two types of data application conformance: strictly conforming and conforming.

A strictly conforming data application is a data application that strictly conforms to the referenced data interchange specification.

NOTE 1 A strictly conforming data application may be minimally conforming but is maximally interoperable with respect to the referenced data interchange specification. Strict conformance concerns (1) the assessment, measurement, and/or availability of a minimal set of features; (2) the data application's non-use of feature-probing; and (3) the data application's non-use of extended feature sets.

A conforming data applications is a data application that conforms to the referenced data interchange specification.

NOTE 2 A conforming data application may be more useful, but may be less interoperable with respect to the referenced data interchange specification. Conformance concerns (1) the assessment, measurement, and/or availability of a minimal set of features; (2) feature-probing for and/or prior agreement to the existence (or availability) of extended features, as permitted by the implementation; and (3) extended features specified external to this Standard.

There are three types of SC/C data applications: data reader, data writer, data repository.

9.3.1 Data reader

A strictly conforming data reader shall interpret data that strictly conforms to (1) the referenced data interchange specification, and (2) at least one binding of the referenced data interchange specification.

NOTE 1 A strictly conforming data reader does not interpret extended data elements.

NOTE 2 Depending upon the binding of the referenced data interchange specification, a strictly conforming data reader may "ignore" data extensions, e.g., a strictly conforming data reader may consume data extensions but the data reader is able to ignore (not interpret) these extensions.

A conforming data reader shall interpret data that conforms to (1) the referenced data interchange specification, and (2) at least one binding of the referenced data interchange specification.

NOTE 3 A conforming data reader may interpret extended data elements.

9.3.2 Data writer

A strictly conforming data writer shall generate data that strictly conforms to (1) the referenced data interchange specification, and (2) at least one binding of the referenced data interchange specification.

NOTE 1 A strictly conforming data writer does not generate extended data elements.

A conforming data writer shall generate data that conforms to (1) the referenced data interchange specification, and (2) at least one binding of the referenced data interchange specification.

NOTE 2 A conforming data writer may generate extended data elements.

9.3.3 Data repository

A strictly conforming data repository shall:

— receive data instances for subsequent retrieval;
— use strictly conforming data interpretation for receiving data instances;
— store data instances in persistent storage so that data extensions may not persist;
— send, on request, previously stored data instances;
— use strictly conforming data generation for sending data instances;
— strictly conform to at least one coding binding; and
— strictly conform to at least one API binding or protocol binding.

NOTE 1 A strictly conforming data repository does not require "preservation" of extended data elements, i.e., data interchange should not be dependent upon expecting extended data elements to persist in a strictly conforming data repository but does not prohibit it either.

A conforming data repository shall:

— receive data objects for subsequent retrieval;
— use conforming data interpretation for receiving data instances;
— store data instances in persistent storage so that data extensions may persist;
— send, on request, previously stored data objects;
— use conforming data generation for sending data instances;
— conform to at least one coding binding; and
— conform to at least one API binding or protocol binding.

NOTE 2 A conforming data repository may, upon storage, add, delete, or change extended data elements for subsequent retrieval.

NOTE 3 A conforming data repository may store some data extensions, but it is not required to store and retrieve all data extensions.

NOTE 4 A conforming data repository may store and retrieve data objects that are not data instances.

9.4 Conformance labels

The following labels indicate certain kinds of conformity to this part:

— Pattern 1: "ISO/IEC 20944-2/B/prefix" where "prefix" represents the kind of binding, as defined in Clauses 11 and onward. For example, "ISO/IEC 20944-2/B/XML" corresponds to the XML coding binding.

— Pattern 2: ISO/IEC 20944-2/B/prefix/role" where "prefix" represents the kind of binding, as defined in Clauses 11 and onward, and "role" is one of "D", "R", "W", or "S" corresponding to data instance, data reader, data writer, and data repository, as defined in this Clause. For example, "ISO/IEC 20944-2/B/XML/S" corresponds to a data repository using the XML coding binding.

The location of the placement of conformance labels is outside the scope of this International Standard.

10 Reserved for future standardization

This Clause is reserved for future standardization.

11 Dotted Identifier Value Pair (DIVP) coding binding

11.1 General

The DIVP coding binding defines the mapping of datatypes to characterstrings.

NOTE The following wording was extracted, excerpted and adapted from, and harmonized with RFC 2068 (HTTP/1.1).

The size of DIVP records produced and consumed is implementation-defined. Records of at least 100,000 octets shall be supported by implementations.

11.1.1 Basic lexical elements

A newline character (CRLF) is defined by its encoding.

NOTE 1 A newline character might be line-feed (e.g., Unix/Linux), carriage-return (e.g., Macintosh), or carriage-return line-feed combination (e.g., Windows).

For maximum interoperability, implementations should use the carriage-return line-feed combination for the newline character when producing data. Implementations that use only one character, either line-feed or carriage-return, for the newline character should ignore the other character, carriage-return or line-feed, respectively, when consuming data.

Leading whitespace (LWS) is defined as at least one or more space or tab characters at the beginning of a line, i.e., the first line or those characters that follow a newline.

NOTE 2 LWS does not include the prior newline. A sequence of space or tab characters not at the beginning of a line is not LWS.

A control character (CTL) is a character in the range 0-31 (decimal) or the octet 127 (decimal) but not the octet 9 (HT).

A text characters is any character except control characters.

The following are token special characters:

```
(   )   <   >   @
,   ;   :   \   "
/   [   ]   ?   =
{   }   SP  HT
```

A token is one or more characters that exclude control characters or token special characters.

A string of text is consumed as a single token (and token special characters are not special) if it is quoted using double-quote marks.

EXAMPLE 1 The characters

```
"abcd -(), :[]"
```

are consumed as a single token.

The backslash character ("\") may be used for single-character quoting, i.e., removing the special nature of a token special character, one character at a time.

EXAMPLE 2 The characters

```
they're
```

are consumed as a single token and the single quote character (') has no special meaning.

EXAMPLE 3 The characters

```
"say \"hello\""
```

are consumed as a single token, the token includes the double quote characters before and after the work hello, and the quote characters that surround the word hello have no special meaning.

11.1.2 Field name and field value

A Name-Value Pair (NVP) shall consist of a field name, followed by a colon (":"), followed by its field value. A field name is a token. Field names shall be case-sensitive.

NOTE 1 The case sensitivity of this DIVP binding differs from RFC 822.

A field value may be preceded by any amount of leading white space (LWS). Conforming implementations should use a single space (SP) as LWS. A field value is zero more characters consisting of combinations of tokens and/or token special characters.

A field value of zero characters represents an empty value being associated with the field name.

NOTE 2 A Name-Value Pair starts at the beginning of a line.

11.1.3 Newline processing

A Name-Value Pair may be extended over multiple lines by preceding each extra line with at least one space (SP) or horizontal tab (HT). All linear white space, including folding, has the same semantics as a single SP character.

A newline (CRLF) shall follow a completed Name-Value Pair.

11.1.4 Syntax summary

Note: The following BNF syntax summary is extracted, excerpted and adapted from, and harmonized with RFC 2616 (HTTP/1.1):

```
CRLF           = <newline character>
CTL            = <any US-ASCII control character
                 (characters 0 - 31) and DEL (127), except HT (9)>
LWS            = [CRLF] 1*( SP | HT )
token          = 1*<any character except CTLs or tspecials>
tspecials      = "(" | ")" | "<" | ">" | "@"
               | "," | ";" | ":" | "\" | <">
               | "/" | "[" | "]" | "?" | "="
               | "{" | "}" | SP | HT
quoted-pair    = "\" character
quoted-string  = <">*<any char except non-quoted
                     double quotes><">
message-header = field-name ":" [ field-value ] CRLF
field-name     = token
field-value    = *( field-content | LWS )
field-content  = <the characters making up the field-value
                 and consisting of combinations of
                 tokens or tspecials>
```

11.2 Generating and producing DIVP

The following rules describe the transformation of data elements, as described by ISO/IEC 20944-2 and by ISO/IEC 11404 notation, to DIVP records.

— Rule 1: For each data element, map all identifiers to fully-qualified field names. A fully-qualified name represents the nested structure of the data element, as described by its "aggregate datatype generator" (ISO/IEC 11404 terminology). A period (".") shall separate each level of nesting in a fully-qualified field name. For array and sequence aggregates, elements are represented by repeated DIVP field names based on the identifier of the aggregate, not the index of the element. A field name is followed by a colon (":"), followed by the value of its associated data element.

— Rule 2: Transform the following DIVP field names (wildcard notation)::

  MDR_*

  to the following DIVP field names (wildcard notation):

  ISO_IEC_11179_MDR_*

All data produced shall be well-formed DIVP.

Rationale

The following is a rationale for these two rules.

The remainder of this subclause is informative and not normative.

Rationale for Rule 1

Rule 1 is the main transformation from ISO/IEC 11404 datatypes to DIVP notation. The following examples use the following definition to illustrate the transformations:

```
A: record
(
    B: integer,
    C: record
    (
        D: integer,
        E: characterstring(iso-10646),
    ),
    F_list: array (0..limit) of (integer),
    G: sample_mlstring_list_type,
)
```

The first three sentences, "For each data element, map all identifiers to fully-qualified field names. A fully-qualified name represents the nested structure of the data element, as described by its "aggregate datatype generator" (ISO/IEC 11404 terminology). A period (".") shall separate each level of nesting in a fully-qualified field name", transform identifiers in to field names, such as:

```
A.B
A.C.D
A.C.E
```

The fourth sentence, "for array and sequence aggregates, elements are represented by repeated DIVP field names based on the identifier of the aggregate, not the index of the element", requires arrays and sequences (lists) to be represented as multiple DIVPs with the same name — permitted in RFC 822-like systems, but with a slightly different meaning (RFC 822 allows these lines to be combed, while this Standard Such-And-Such coding binding does not permit automatic consolidation of DIVPs). For example, the data element F would be represented as:

(correct DIVP binding of F)

```
A.F: xxx
A.F: yyy
A.F: zzz
```

but not as:

(incorrect DIVP binding of F)

```
A.F.0: xxx
A.F.1: yyy
A.F.2: zzz
```

The fifth sentence, "A field name is followed by a colon (":"), followed by the value of its associated data element", associates the field name with its value and separates the two with a colon (":").

Example:

```
A.B: 17
A.C.D: 34
A.C.E: yellow pigs
A.F: 51
A.F: 68
A.F: 85
```

Rationale for Rule 2

This rule is used for rewriting tags to use certain namespace conventions.

11.3 Consuming and interpreting DIVP

The following rules describe the transformation of DIVP records to data elements, as described by ISO/IEC 20944-2 and by ISO/IEC 11404 notation.

— Rule 1: Transform the following DIVP tags (wildcard notation):

ISO_IEC_11179_MDR_*

to the following DIVP tags (wildcard notation):

MDR_*

— Rule 2: For each field name that is associated with an identifier defined by a data element in this part, map each field name to the corresponding data element identifier. The nesting of the field names represents the nesting of the data elements, i.e., the reverse of the operation in Rule #1 of subclause 5.2, Generating and Interpreting Dotted Name-Value Pairs, above. Each field value is converted to the value of the corresponding data element. An empty field value of an aggregate may represent the existence of the aggregate, but not the value of its components.

Rationale

The following is a rationale for these two rules.

The remainder of this subclause is informative and not normative.

Rationale for Rule 1

This rule transforms any namespace prefixes, as necessary. In this illustration, the namespace prefix ("ISO_IEC_11179_MDR") was used to reduce the possibility of namespace collisions.

Rationale for Rule 2

This rule handles the main transformation of DIVPs to data elements.

The first sentence, "for each field name that is associated with an identifier defined by a data element in this part, map each field name to the corresponding data element identifier", (1) ignores all identifiers that are unknown to this Guide, (2) properly pairs the field names and identifiers of data elements, and (3) creates the association with data elements, but does not assign the values of the data elements.

The second sentence, "The nesting of the field names represents the nesting of the data elements, i.e., the reverse of the operation in Rule #1 of subclause 11.2, Generating and Interpreting Dotted Name-Value Pairs, above", assures that the internal structure of the DIVPs, to the extent required by this Guide, agree with the internal structure of the data elements.

The third sentence, "each field value is converted to the value of the corresponding data element", transforms the field values of the data elements, i.e., it "populates" the data elements.

The fourth sentence, "an empty field value of an aggregate may represent the existence of the aggregate, but not the value of its components", merely creates the aggregate.

In the fragment below, A.C, which has an empty value, may be used to indicate the existence of an aggregate:

```
A.B: 17
A.C:
A.C.D: 34
A.C.E: yellow pigs
```

An empty value is a useful technique when aggregates are comprised of optional data elements, i.e., the signaling of the existence of the aggregate, but the lack of aggregate components:

```
A.B: 17
A.C:
```

An empty value is not used to indicate a void type.

11.4 Representation of basic data types

The following subclauses describe the transformation of data element values to/from character representations for information interchange for use within the DIVP binding.

11.4.1 Characters and character strings

Data elements that are of type character shall be represented only as per ISO/IEC 8859-1 when encoded according to the rules of RFC 1522.

11.4.2 Integers

Data elements that are of type integer shall be represented as per ISO/IEC 9899:1999, C Programming Language, subclause 6.4.4.1, Integer Constants; excluding "U", "L", and "LL" suffixes and their lowercase variants; and may include an optional leading sign, either plus ("+") or minus ("-"), but not both.

EXAMPLES:

```
0       // zero
23      // twenty-three
0x17    // same in hexadecimal
027     // same in octal
-34     // negative thirty-four
+34     // positive thirty-four
+34     // same
```

11.4.3 Real numbers

Data elements that are of type real:

— if integral, may be represented integers, as specified above in subclause 19.5, Integers;

— if not integral or not represented as integers, shall be represented as per ISO/IEC 9899:1999, C Programming Language, subclause 6.4.4.2, Floating Constants; excluding "F" and "L" suffixes and their lowercase variants; and may include an optional leading sign, either plus ("+") or minus ("-"), but not both.

EXAMPLES:

```
0       // zero
0.0     // same
130.0   // one hundred thirty
1.3E2   // same
+1.3E2  // same
```

11.4.4 Date and time values

Data elements of type time shall be represented as per ISO 8601, Data elements and interchange formats — Information interchange — Representation of dates and times.

NOTE 1 ISO 8601 is intended to represent dates and times between 1 January 0001 and 31 December 9999 using the Gregorian calendar. It is well understood that there are anomalies with this approach, e.g., the month of September 1752 has less than 30 days, the lack of correlation of timezones prior to the introduction of transcontinental railroads, changing the beginning month of the calendar, the inability to represent dates Before the Common Era, and the inability to represent dates after 31 December 9999.

A date-time is expressed according the following profile of ISO 8601:2000:

— A date-time shall represent a "time-point" and shall not represent a "time-interval" or a "recurring time-internal". Note: See ISO 8601 subclauses 3.8, 3.18, 3.26, and 4.1.

— Truncations shall not be used. Note: See ISO 8601, subclause 4.6.

— Expansions may be used to represent years outside the range of 1 to 9999 C.E. (see ISO 8601, subclause 4.7).

— A date-time shall be a date component (subclause 5.2) or combination of date and time components (subclause 5.4).

— A date-time combination shall be a complete representation (subclause 5.4.1, e.g., "YYYY-MM-DDTHH:MM:SS", "±YYYYYY-MM-DDTHH:MM:SS"), or representation other than complete (subclause 5.4.2, e.g., "±YYYYYY-MM" or "YYYY-MM-DDTHH:MM", but not "MM-DD" or "THH:MM:SS").

— Dates shall be represented as calendar dates and shall not be represented as ordinal dates or week dates. Note: See ISO 8601, subclause 5.2.1.

— Dates shall be represented as one of the following formats:

— complete date in extended format (subclause 5.2.1.1, "YYYY-MM-DD")

— a reduced precision date for a specific month in basic format (subclause 5.2.1.2, "YYYY-MM")

— a reduced precision date for a specific year in basic format (subclause 5.2.1.2, "YYYY")

— an expanded date for a specific day in extended format (subclause 5.2.1.4, e.g., "±YYYYYY-MM-DD")

— an expanded date for a specific month in basic format (subclause 5.2.1.4, e.g., "±YYYYYY-MM")

— an expanded date for a specific year in basic format (subclause 5.2.1.4, e.g., "±YYYYYY").

— Times shall be represented as local time (subclause 5.3.1) or Coordinated Universal Time (subclause 5.3.3).

— Times shall not use format that indicates the difference between local time and Coordinated Universal Time (subclause 5.3.4).

— Times shall use a complete representation ("HH:MM:SS"), a reduced precision for a specific hour and minute ("HH:MM"), or a reduced precision for a specific hour ("HH").

— Times may use decimal fractions (subclause 5.3.1.3, e.g., "HH:MM:SS.SSS").

— The full stop character "." shall be the decimal sign.

— The date-time resolution shall support date-times using an integral number of seconds. Examples of date-time resolution: Conforming: 1 second, 1/2 second, 1/60 second, 1/100 second. Not conforming: 2 seconds, 2/3 second. Note: This means an application must support resolution to an exact second, but may support resolutions that an integral number of parts of a second (i.e., 1/N seconds).

— The time designator "T" shall be used when a time is represented (subclause 5.3.1.5).

— The hour "24" shall not be used to represent midnight (subclause 5.3.2).

11.4.5 Void types

A void type shall have no representation and shall have no encoding.

EXAMPLE The following record

```
A: record
(
    B: integer,
    C: void,
    D: characterstring(iso-10646),
)
```

is represented in an RFC 822-like binding as:

```
(correct DIVP binding of C)
A.B: 17
A.D: hello
but not as:
(incorrect DIVP binding of C)
A.B: 17
A.C:
A.D: hello
```

11.5 Encoding of character representations

A DIVP coding bindings shall encode character representations to character values specified by ISO/IEC 8859-1. Characters outside this repertoire shall be encoded according to the rules of RFC 1522.

11.6 Handling exceptions and extensions

11.6.1 Implementation-defined behavior

The following are implementation-defined behaviors in addition to those described elsewhere in this Guideline.

The following are implementation-defined behaviors in the production and consumption of DIVP codings:

— The encoding, in characters, of the newline character. Note: Implementations can avoid implementation-defined behavior by using the carriage return line feed combination characters for the newline character.

— The maximum size, in characters, of a strictly conforming data instance, coded as a DIVP, that may be processed successfully.

— The time zone information for data elements of time type that have unspecified timezones.

11.6.2 Unspecified behavior

The following are unspecified behaviors in addition to those described elsewhere in this Guideline.

The following is unspecified behaviors in the generation or interpretation of DIVPs:

— The order of processing data elements.

11.6.3 Undefined behavior

The following are undefined behaviors in addition to those described elsewhere in this Guideline.

The following are undefined behaviors in the production or consumption of DIVPs:

— The use of field names that correspond to extended data elements.

— The use of field names that correspond to reserved data elements.

— The use of field names not specified in this DIVP coding binding.

— The use of characters outside the repertoire described in this Guideline.

11.7 Conformance label prefix

The prefix "DIVP" shall be used, in conjunction with 9.4, to indicate conformity to this Clause.

12 XML coding binding

12.1 General

The size of XML records produced and consumed is implementation-defined. Records of at least 100,000 octets shall be supported by implementations.

12.2 Generating and producing XML

The following rules describe the transformation of data elements, as described by ISO/IEC 20944-2 and by ISO/IEC 11404 notation, to XML records.

— Rule 1: For each data element in ISO/IEC 11404 notation, map all identifiers to XML tags, except as noted in Rule 2 below. Balanced XML tags delimit the boundary of the value associated with the data element. The nesting of the XML tags represents the structure of data elements, as described by its "aggregate datatype generator" (ISO/IEC 11404 terminology). For array and sequence aggregates, (1) an XML tag of the same name as the identifier of the aggregate represents the group of aggregates, (2) the individual data elements are represented by repeated XML tags based on the identifier of the aggregate minus the suffix "_list" or "_bucket", not the index of the element.

— Rule 2: Map all mcstring_type datatypes to:

— Rule 2A: The locale element of mcstring_type sets the LANG attribute in parent XML element.

— Rule 2B: The string element sets content of parent tag (i.e., the current target).

— Rule 3: Transform the following XML tags (wildcard notation):

MDR_*

to the following XML tags (wildcard notation):

ISO_IEC_11179_MDR_*

All data produced shall be well-formed XML.

NOTE It is permitted, but not required, to use XML-related technologies such as XML Schema and RELAX-NG.

Rationale

The following is a rationale for these three rules.

The remainder of this subclause is informative and not normative.

Rationale for Rule 1

Rule 1 is the main transformation from ISO/IEC 11404 datatypes to XML tagging conventions. The following examples use the following definition to illustrate the transformations:

```
A: record
(
    B: integer,
    C: record
    (
        D: integer,
        E: characterstring(iso-10646)
    ),
    F_list: array (0..limit) of (integer),
    G: sample_mlstring_list_type,
)
```

The first sentence, "for each data element in ISO/IEC 11404 notation, map all identifiers to XML tags", transforms identifiers, e.g., "X:" -> "<X>".

The second sentence, "balanced XML tags delimit the boundary of the value associated with the data element", requires that (1) the tags are balanced, and (2) the value of the data element is between the tags, e.g., "X: 17" -> "<X>17</X>".

The third sentence, "the nesting of the XML tags represents the structure of data elements, as described by its aggregate datatype generator", requires that the nesting implied in aggregates (records, arrays, sequences/lists) results in similar nesting of the XML tags. Using the definition of A above, the following nesting is implied for elements B, C, D, and E:

```
<A>
    <B>...</B>
    <C>
        <D>...</D>
        <E>...</E>
    </C>
    ...
</A>
```

The fourth sentence, "for array and sequence aggregates, data elements are represented by repeated XML tags based on the identifier of the aggregate, not the index of the element", requires arrays and sequences

(lists) to be represented as multiple tags with the same name — a typical XML style convention. For example, the data element F would be represented as:

```
<!-- correct XML binding of F_list -->
<A>
    ...
    <F_list>
        <F>...</F>
        <F>...</F>
        <F>...</F>
    </F_list>
    ...
</A>
```

but not as:

```
<!-- incorrect XML binding of F_list -->
<A>
    ...
    <F_list>
        <0>...</0>
        <1>...</1>
        <2>...</2>
    </F_list>
    ...
</A>
```

Rationale for Rule 2

Some records use several specialized datatypes, such as multilingual datatypes for describing certain characterstring-type data elements that must be represented in a multilingual and multicultural context — commonly called internationalization (I18N) and localization (L10N) features. Below, is a sample version of a multilingual data type that is not intended to clash with definitions of other multilingual datatypes defined elsewhere in this part. In this illustration, the datatype sample_mcstring_type represents a single pair: a localized string and a locale specification (L10N mapping). The datatype sample_mcstring_array_type represents an array of these string pairs. In this example, the array example_remarks contains three elements, each element is a pair of strings. Presumably, an application would choose the appropriate string from use example_remarks based on the country (locale) that the application was operating in. The following are sample type definitions and value definitions.

```
type sample_mcstring_type =
record
(
    L10N_string: characterstring(iso-10646),
    L10N_locale: string_type,
),

type sample_mcstring_array_type =
    array (0..limit) of (sample_mcstring_type),

value example_remarks:
    sample_mcstring_array_type =
(
    (
        L10N_string: "abc abc abc",
        L10N_locale: "en-US",
    ),
    (
        L10N_string: "def def def",
```