The Consultative Committee for Space Data Systems

Draft Recommendation for
Space Data System Standards

# SPACECRAFT ONBOARD INTERFACE SERVICES— XML SPECIFICATION FOR ELECTRONIC DATA SHEETS

## DRAFT RECOMMENDED STANDARD

## CCSDS 876.0-P-1.1

## PINK SHEETS

**July 2021**

The Consultative Committee for Space Data Systems

**Draft Recommendation for**
**Space Data System Standards**

# SPACECRAFT ONBOARD INTERFACE SERVICES— XML SPECIFICATION FOR ELECTRONIC DATA SHEETS

## DRAFT RECOMMENDED STANDARD

## CCSDS 876.0-P-1.1

## PINK SHEETS

**July 2021**

[16] *Spacecraft Onboard Interface Services—Subnetwork Memory Access Service*. Issue 1. Recommendation for Space Data System Practices (Magenta Book), CCSDS 852.0-M-1. Washington, D.C.: CCSDS, December 2009.

[17] *Spacecraft Onboard Interface Services—Subnetwork Synchronisation Service*. Issue 1. Recommendation for Space Data System Practices (Magenta Book), CCSDS 853.0-M-1. Washington, D.C.: CCSDS, December 2009.

[18] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049. Reston, Virginia: ISOC, October 2013.

NOTE – Informative references are contained in annex D.

## 2.3 PURPOSE AND OPERATION OF SOIS ELECTRONIC DATA SHEETS

A SEDS is intended to be a machine-understandable mechanism for describing onboard components, as more fully described in the SOIS Green Book (reference [D2]).

The SEDS is intended to replace the traditional interface control documents and proprietary data sheets which accompany a device and are necessary to determine the operation of the device and how to communicate with it. The SEDS could then be used for a wide variety of purposes, whilst ensuring consistency and completeness of information:

a) generating human-readable documentation;

b) specifying interfaces to the device;

c) automatically generating software implementing the relevant parts of the onboard software for the device;

d) automatically generating device-interface-simulation software for use in test or device-simulation software;

e) transforming the device functional interface into telecommands and telemetry suitable for processing by a command and data handling system onboard and on the ground;

f) capturing interface information for the spacecraft database;

g) providing a common definition to human-friendly authoring mechanisms for SEDS instances;

h) providing the common exchange medium between organizations for the above use cases.

Further information on the potential uses of SEDS can be found in the SOIS Green Book (reference [D2]).

In order to be able to relate the elements of the data sheet to physical (and nonphysical) concepts, and to promote standardization and interoperability, a SANA DoT (reference [1]) provides a core ontology for data sheet authors and users. These core semantic terms effectively form part of the language that is used to write SEDS. Whenever the semantics provided by the SANA DoT are insufficient, a data sheet author may utilize an additional user-defined DoT, which must then be supplied with the data sheet itself. This provides a standard, flexible, and extensible mechanism for capturing the semantics of device operation in a machine-understandable form.

The SEDS schema enumerates some external standards and conventions so that these standards and conventions may be associated with data in a SEDS instance. Examples are error control check words, math operations, and encoding schemes. These enumerations will never cover the variety of such standards and conventions, so the SEDS schema allows for extension of these enumerations. The extensions needed within a project to support a local toolchain can be written into an auxiliary schema, which the SEDS schema includes. The

## 3.2 ELECTRONIC DATA SHEETS AND THE ASSOCIATED SCHEMA

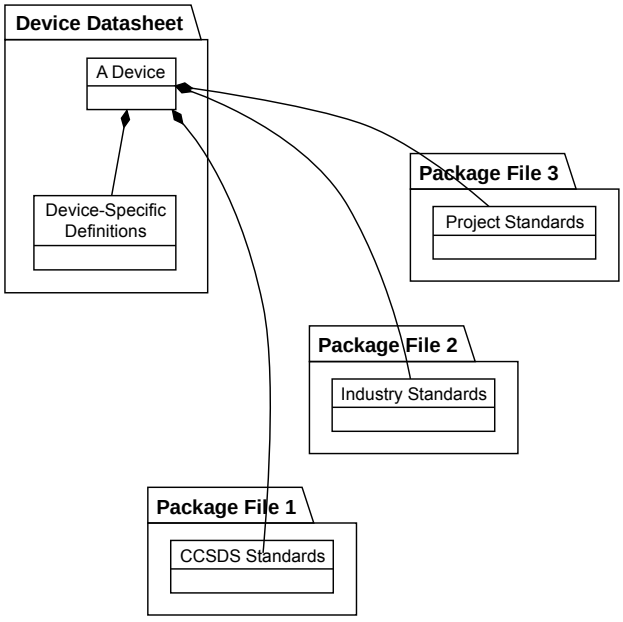



**Figure 3-3: Device Datasheets and Package Files**

Figure 3-3 shows the relationship among SEDS files. (It is not a syntax diagram.) A `Device Datasheet` contains all known information about a particular device or class of devices. It may reference one or more `Package Files` that capture an independently managed subset of that data, such as

- a standard or specification it supports;
- a product line it belongs to;
- a compatibility mode it is capable of; or
- a replaceable hardware or software part it contains or manages.

A `Package File` may describe a composable unit of software or hardware, in the manner of a `Device Datasheet`, but without the `Device` element, and without XInclude (reference [5]). A `Package File` may contain only metadata or data types for a platform, for a project, or for universal reference.

**3.2.1** The basic unit of data exchange of SOIS device information is an XML document known as a device datasheet ~~or package file~~.

**3.2.2** A device datasheet ~~or package file~~ shall be de defined by a single top-level XML file.

**3.2.3** Any files referenced by a device datasheet shall be XML package files compliant to the `PackageFile` element of the SEDS schema.

**3.4.4** The `Category` element shall contain one or more child elements, each of which is either a `Category` element or `MetadataValueSet` element.

**3.4.5** A `MetadataValueSet` element shall contain one or more child elements, each of which is either a `DateValue` element, a `FloatValue` element, an `IntegerValue` element, or a `StringValue` element.

**3.4.6** The `DateValue`, `FloatValue`, `IntegerValue`, and `StringValue` elements are all based on `FieldType`.

**3.4.7** `DateValue` and `StringValue` elements shall contain a `value` attribute specifying the value of the metadata as a literal, per table 3-1.

**3.4.8** `FloatValue` and `IntegerValue` elements may contain a `value` attribute specifying the value of the metadata as a literal, per table 3-1.

~~**3.4.9** If a `FloatValue` or `IntegerValue` element does not contain a value attribute, the body of the element shall specify a `MathOperation` element, as described in 3.15.32 below, or a `Conditional` element, as described in 3.15.37 below, to describe how the value should be calculated.~~

**Table 3-1: Data Types, Encodings, Ranges, and Literals**

| Data Type | Encoding Type | Range Types | Literal Syntax |
|---|---|---|---|
| BinaryDataType | | | xs:hexBinary |
| BooleanDataType | BooleanDataEncoding | | xs:boolean |
| EnumeratedDataType | IntegerDataEncoding | EnumeratedRange | xs:string, matching enumeration label. |
| FloatDataType | FloatDataEncoding | PrecisionRange<br>MinMaxRange | xs:float |
| IntegerDataType | IntegerDataEncoding | MinMaxRange | xs:integer |
| StringDataType | StringDataEncoding | EnumeratedRange | xs:string |
| SubRangeDataType | | as base type | as base type, within range |

**3.7.2** A FloatDataEncoding or IntegerDataEncoding element may carry a byteOrder attribute specifying a value of

a) bigEndian, the default, for values which are to be encoded most significant byte first; or

b) littleEndian for values which are to be encoded least significant byte first.

NOTE – The littleEndian specification applies only to data types whose size is a multiple of 8 bits.

**3.7.3** A BooleanDataEncoding element shall carry a sizeInBits attribute which specifies the size, in bits, of the encoded data as a positive integer.

**3.7.4** A BooleanDataEncoding element may carry a falseValue attribute which specifies the value that corresponds to logical falsehood, with options

a) zeroIsFalse (the default); and

b) nonZeroIsFalse.

**3.7.5** An IntegerDataEncoding element shall carry an encoding attribute which has a value of

a) unsigned, for an unsigned value;

b) signMagnitude, for an encoding with a separate sign bit (the most significant bit is the sign bit, with 1 indicating negative);

c) twosComplement, for twos complement;

d) onesComplement, for ones complement;

e) BCD, for a natural unsigned binary coded decimal, where each byte is a decimal digit encoded as binary; or

f) `packedBCD`, where each byte contains two decimal digits encoded as binary, followed by an optional sign nibble. A negative sign is 1011 or 1101; a positive sign is 1010, 1100, 1110, 1111, or omitted.

**3.7.6** An `IntegerDataEncoding` element shall carry a `sizeInBits` attribute which specifies the size, in bits, of the encoded data as a positive integer.

**3.7.7** The size in bits of a BCD encoding shall be a multiple of 8. The size in bits of a packedBCD shall be a multiple of 4. The size in bits of both forms of binary coded decimals is a fixed value, so all high-order digits that are zero shall be present to fill the fixed size in bits.

**3.7.8** A `FloatDataEncoding` element shall carry an `encodingAndPrecision` attribute which has a value of either

a) `IEEE754_2008_single`;

b) `IEEE754_2008_double`;

c) `IEEE754_2008_quad`;

d) `MILSTD_1750A_simple`; or

e) `MILSTD_1750A_extended`.

NOTE – These represent the supported sizes of IEEE (reference [6]) and MIL-STD-1750A (reference [7]).

**3.7.9** A `FloatDataEncoding` element shall carry a `sizeInBits` attribute which specifies the size, in bits, of the encoded data as a positive integer.

**3.7.10** A `StringDataType` shall carry a `length` attribute which defines the maximum possible length of the string, in bytes.

**3.7.11** A `StringDataType` may carry a `fixedLength` attribute which, if 'false', indicates that the string can be shorter than the value specified by the length attribute.

NOTE – Specification of `fixedLength="false"` indicates a data type that occupies a variable amount of memory. When such a data type is an entry in a container, then the container is of variable length. (See 4.8 for details about string lengths.)

**3.7.12** A `StringDataEncoding` element may carry an `encoding` attribute which has a value of either

a) `UTF-8`, specifying Unicode UTF-8 encoding (reference [8]); or

b) `ASCII`, the default, specifying US ASCII encoding (reference [9]).

**3.7.13** The optional ~~`terminationCharacter`~~`terminationByte` attribute of a `StringDataEncoding` element shall specify the termination ~~`character`~~`byte` for the string.

NOTES

1 For example, a termination character of zero (null) is used by C-language strings.

2 Bytes used by a termination character are not included in the count of bytes that constitute the length of the string.

3 UTF-8 characters use variable-length encoding that can contain as much as 4 bytes per character. Consequently, not all UTF-8 strings of a given character length are representable by a data type with that byte length.

**3.7.14** An `EnumeratedDataType` shall contain an `EnumerationList` element, consisting of a list of one or more `Enumeration` elements.

**3.7.15** Each `Enumeration` element shall have required `label` and `value` attributes, indicating the integer value corresponding to a given label string.

**3.7.16** An `Enumeration` element may carry attributes provided by the standard DoT (reference [1]).

**3.7.17** A `BinaryDataType` shall have a required `sizeInBits` attribute.

**3.7.18** A `BinaryDataType` may have an optional `fixedSize` attribute:

a) If this attribute is `true`, then the `sizeInBits` shall indicate the actual size of data.

b) If this attribute is `false`, then the `sizeInBits` shall be the maximum size of data.

c) The default value shall be `false`.

**3.7.19** A `SplineCalibrator` or `PolynomialCalibrator` child element may be placed in a numeric data type element to specify any calibration that would be required to take the raw value represented by the data type and convert it into the units and other semantic terms associated with the data type (see 3.15).

## 3.8 RANGES

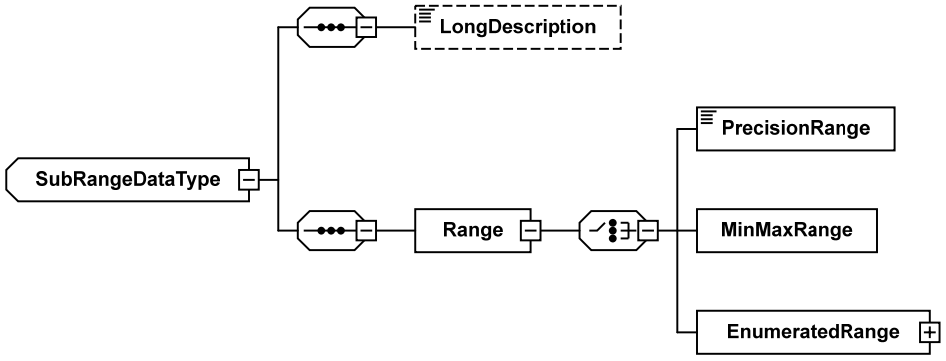



**Figure 3-9: Ranges within a `SubRangeDataType` Element**

NOTE – A range constrains the values of a data type for the purposes of validation and recognition.

**3.8.1** When a range and a calibrator apply to the same data type, the range shall constrain the raw values (before calibration) of the data type.

**3.8.2** Each ~~`EnumeratedDataType`,~~ `FloatDataType`, `IntegerDataType`, ~~or~~and `SubRangeDataType` element shall contain a single `Range` element of a type corresponding to table 3-1.

**3.8.3** A `SubRangeDataType` element shall contain a `baseType` attribute, referring to the numeric or enumerated scalar type which defines all properties other than range.

**3.8.4** The `baseType` attribute of a `SubRangeDataType` should not refer to another `SubRangeDataType`.

**3.8.5** A `PrecisionRange` element shall be either ~~`SINGLE`, `DOUBLE`, or `QUAD`~~`single`, `double`, or `quad`, representing the full supported representation range of the corresponding IEEE754 floating point data encodings.

**3.8.6** A `MinMaxRange` element shall have an attribute `rangeType`, one of the options listed in table 3-2.

**Table 3-2: `MinMaxRange` Options**

| Interval Notation | Relational Notation | XML Notation `rangeType` | `min` | `max` |
|---|---|---|---|---|
| `(a..b)` | {x \| a < x < b} | `exclusiveMinExclusiveMax` | yes | yes |
| `[a..b]` | {x \| a <= x <= b} | `inclusiveMinInclusiveMax` | yes | yes |
| `[a..b)` | {x \| a <= x < b} | `inclusiveMinExclusiveMax` | yes | yes |
| `(a..b]` | {x \| a < x <= b} | `exclusiveMinInclusiveMax` | yes | yes |
| `(a..+∞)` | {x \| a < x} | `greaterThan` | yes | |
| `[a..+∞)` | {x \| a <= x} | `atLeast` | yes | |
| `(-∞..b)` | {x \| x < b} | `lessThan` | | yes |
| `(-∞..b]` | {x \| x <= b} | `atMost` | | yes |

**3.8.7** A `MinMaxRange` element may have attributes `min` and `max`, whose presence and values shall be consistent with table 3-2.

**3.8.8** An `EnumeratedRange` element shall have a list of `Label` child elements, with values that shall be enumeration labels of the corresponding `EnumeratedDataType`.

**3.8.9** Each `StringDataType` element may contain a single `Range` element of a type corresponding to table 3-1.

## 3.9 ARRAYS

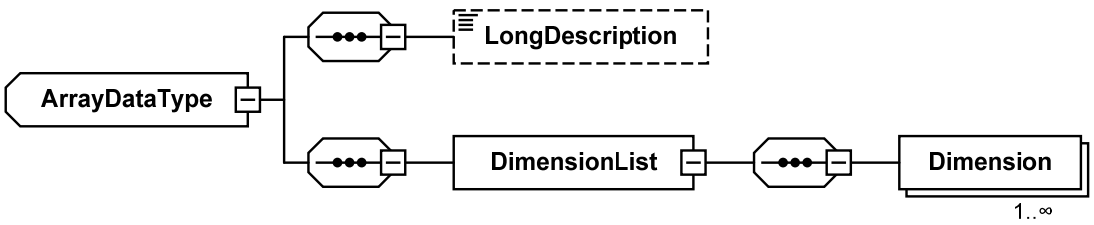



**Figure 3-10: `ArrayDataType` and `Dimension` Elements**

NOTE – Arrays provide the means to specify adjacent repetitions of the same type of data, the elements of which can be accessed by index.

**3.9.1** An `ArrayDataType` element shall contain a `dataTypeRef` attribute, referring to the type of the elements within the array.

NOTE – Subsection 4.3 provides further details.

**3.9.2** An `ArrayDataType` element shall contain a `DimensionList` element with one or more `Dimension` child elements.

**3.9.3** A `Dimension` element determines the length of the array dimension, in elements, and shall either have attribute `size`, directly indicating the maximum length, or attribute `indexTypeRef`, indicating the integer or enumerated data type to be used to index the array. The type referenced by an `indexTypeRef` attribute has maximum and minimum legal values, from which the size of the array can be inferred. When the `size` attribute is used, the index is

**3.10.20** A `ListEntry` element within a container shall specify an attribute `listLengthField` which contains the name of another element of the same container whose value will be used to determine the number of times this entry should be repeated.

**3.10.21** A `LengthEntry` element within a container shall specify an entry whose value is constrained, or derived, based on the length of the container in which it is present.

**3.10.22** If a `LengthEntry` element has a calibration (see 3.11.7), that calibration shall be used to map between the length in bytes of the container and the value of the entry, according to the formula:

`container length in bytes = calibration(entry raw value).`

NOTE – A constraint that refers to a `LengthEntry` compares to the entry raw value.

**3.10.23** Any calibration specified for a `LengthEntry` shall be *reversible*, that is, a linear polynomial, or spline, with all points of degree 1.

**3.10.24** An `ErrorControlEntry` element within a container shall specify an entry whose value is constrained, or derived, based on the contents of the container in which it is present. In addition to a subset of the attributes and elements supported for a regular container entry, it has the mandatory attribute type, which is one of the values specified in the DoT for `errorControlType` as illustrated in table 3-3.

**Table 3-3: Error Control Types**

| Value | Description | Reference |
|---|---|---|
| CRC16_CCITT | G(X) = X^16 + X^12 + X^5 + 1 | [10], subsection 4.1.4.2 |
| CRC8 | G(x) = x^8 + x^2 + x^1 + x^0 | [11], clause 5.2 |
| CHECKSUM | modulo 2^32 addition of all 4-byte | [12], subsection 4.1.2 |
| CHECKSUM_LONGITUDINAL | Longitudinal redundancy check, bitwise XOR of all bytes | [13] |

**3.10.25** A `ContainerDataType` element may carry an optional `encodingRules` attribute, which, if present, indicates that the container should be encoded according to `CBOR` or `CBORindefinite` encoding.

NOTE – Concise Binary Object Representation (CBOR) is defined in IETF RFC 7049 (reference [18]).

**3.10.26** A `ContainerDataType` element may carry an optional `encodedAs` attribute, which, if present, indicates that the container should be encoded as if the totality of its contents were treated as a single instance of the data type referenced by the attribute.

NOTE – For example, a container encoded as an integer data type and used as an entry in a container with CBOR encoding would be packed in an integer and then encoded as the appropriate CBOR type.

**3.10.27** Each `Entry`, `FixedValueEntry`, `ListEntry`, `LengthEntry`, `ErrorControlEntry`, and `PaddingEntry` element within a container may contain a `PresentWhen` element, which, if present, contains constraints to specify when that entry is present in an instance of that container type. The types of constraints in a `PresentWhen` element shall be the same types that may appear in a `ConstraintSet` element of a container.

**3.10.28** Each `RangeConstraint`, `TypeConstraint`, and `ValueConstraint` element within a `ConstraintSet` or `PresentWhen` element may have an optional `negate` attribute, which, if present, indicates when '`true`' that the negation of the constraint applies. The default value of the `negate` attribute shall be '`false`'.

## 3.11 FIELDS

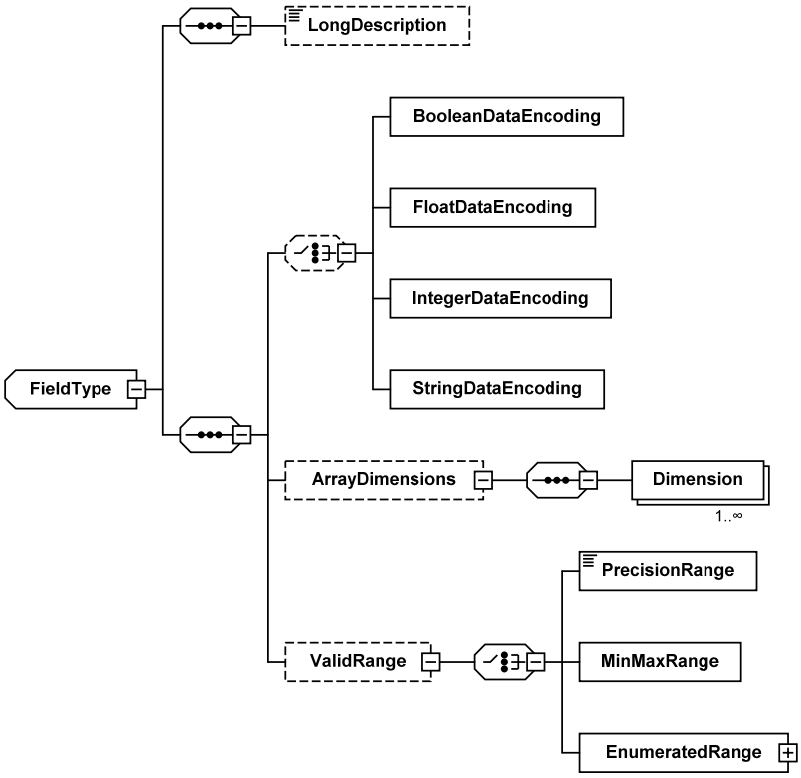



**Figure 3-12: Field Schema Type**

NOTES

1 Data types are instantiated in many different circumstances; however, whenever a data type is instantiated there is a set of common valid attributes and elements. This is referred to as a 'field'. This subsection describes these attributes and elements such that they may be referenced whenever a data type instantiation is described elsewhere in this document.

**3.11.7** A `SplineCalibrator` or `PolynomialCalibrator` child element of an external field specifies any calibration that would be required to take the raw value represented by the data type and convert it into the units and other semantic terms associated with the field (see 3.15).

## 3.12 INTERFACES

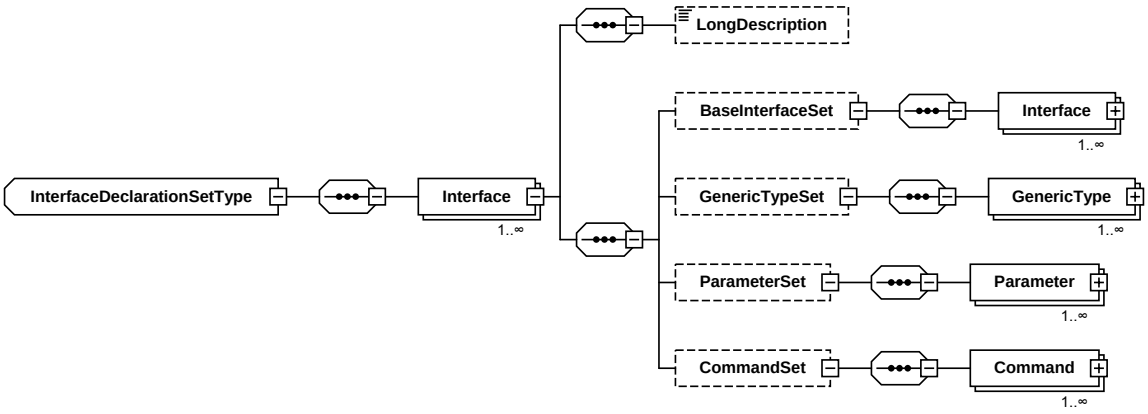



**Figure 3-14: Interfaces within a ~~`InterfaceDeclarationSet`~~`DeclaredInterfaceSet` Element (Which Has Type Name InterfaceDeclarationSetType)**

NOTES

1 Standardized interfaces, including those to the subnetwork, are defined with this interface construct to allow them to be treated symmetrically with user-defined interfaces.

2 Any interface declared within a data sheet is implicitly scoped to the target device, bypassing any complexities associated with having multiple devices.

3 An interface definition can be split into multiple parts, and therefore placed in multiple package files, which can be joined together by specifying the members of the BaseInterfaceSet. This allows separation of aspects of an interface which address different concerns or have different authors.

4 No guarantee of run-time compatibility is implied by two different interfaces sharing a common base interface.

5 All commands and parameters in such a set of aggregated interfaces must have unique names, so no rules are needed to resolve conflicts between them.

6 An interface can be defined in terms of generic types to avoid placing undue restrictions on its implementation or use. Such interfaces need to have those generic types mapped to fully specified types before use (see 3.12.7).

**3.12.1** A~~n `InterfaceDeclarationSet`~~`DeclaredInterfaceSet` element shall contain one or more `Interface` elements.

**3.12.2** Each `Interface` child element of a~~n `InterfaceDeclarationSet`~~`DeclaredInterfaceSet` is based on the `NamedEntityType` (see 3.3.7).

**3.12.3** The name of each `Interface` child element of a~~n `InterfaceDeclarationSet`~~`DeclaredInterfaceSet` element shall be unique.

**3.12.4** An `Interface` element of a `DeclaredInterfaceSet` element shall contain zero or one `BaseInterfaceSet` element referencing one or more `Interface` elements, zero or one `GenericTypeSet` element containing one or more `GenericType` elements, zero or one `ParameterSet` element containing one or more `Parameter` elements, and zero or one `CommandSet` element containing one or more `Command` elements. The total number of commands and parameters in a declared interface shall be greater than zero, summing over inheritance through `BaseInterfaceSet` and over declaration in `ParameterSet` and `CommandSet`.

**3.12.5** An `Interface` element of a `DeclaredInterfaceSet` element may have an optional attribute `abstract`, which, if true, indicates the interface may not be used directly by a component.

**3.12.6** An `Interface` element of a `DeclaredInterfaceSet` element ~~shall~~may have an attribute `level`, ~~with value taken from table 3-4,~~ which, if present, indicates the system level at which it operates. Values of `level` shall be taken from table 3-4; the default value shall be '`application`'.

NOTE – The level of an interface corresponds mostly to the layers in a protocol stack for onboard communications but includes two special layers, `application` and `environment`, which would not be considered to be part of a protocol stack. The application level represents interfaces between applications that serve a purpose other than communication protocol. The environment level represents the interfaces provided by the real world for functions of physical hardware sensors and actuators. The environment level may describe interfaces for simulation, for hardware-in-the-loop, and for specification of environmental constraints on hardware. Hardware constraints in the environmental level may express manufacturer's recommended thermal operating ranges, maximum radio frequency energy, maximum photon count, for example.

**Table 3-4: Interface Levels**

| Name | Description |
|---|---|
| ~~A~~application | Not directly related to device data. |
| ~~F~~functional | Higher-level virtual abstraction of device data. |
| ~~A~~access | Lower-level specification of device data. |
| ~~S~~subnetwork | Raw uninterpreted communication channel to a device. |
| physical | OSI layer 1, pins, voltages, etc. |
| environment | For simulation or hardware-in-the-loop, between component and environment. |

**3.12.12** The name of each `Parameter` child element of a `ParameterSet` element shall be unique within the set of interfaces reachable by `BaseType` references from the containing interface.[1]

**3.12.13** Valid values for the `mode` attribute of a `Parameter` element shall be '`sync`' (the default, indicating a two-way message exchange) or '`async`'.

NOTE – Subsection 4.5 provides further details.

**3.12.14** Valid values for the `readOnly` attribute shall be '`false`' (the default) or '`true`'.

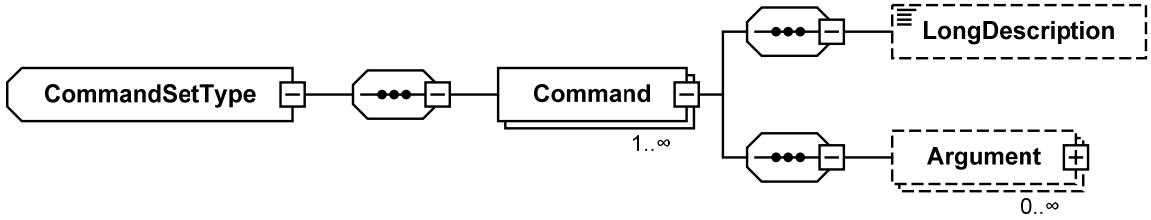



**Figure 3-16: Commands in a `CommandSet` Element**

**3.12.15** Each `Command` child element of a `CommandSet` element is based on the `NamedEntityType` (see 3.3.6), plus ~~an~~ optional `mode` and `pattern` attributes, identifying the command ~~mode~~message pattern.

NOTE – The command message pattern is explained in 4.5.

**3.12.16** The name of each `Command` child element of a `CommandSet` element shall be unique within the set of interfaces reachable by `BaseType` references from the containing interface.

NOTE – The reason for this restriction is to eliminate the possibility of a derived interface overriding a command of a base interface. Instead, the commands of each base type interface are included unchanged in the derived interface. This is a form of aggregation of interface commands.

**3.12.17** Each `Command` child element of a `CommandSet` element identifies a command on an interface and shall contain zero or more `Argument` elements, each of which identifies an argument to the command.

**3.12.18** Each `Argument` child element of a `Command` element shall have the attributes and child elements associated with an external field (see 3.10.25) with the addition of an optional `mode` attribute, identifying the argument mode.

**3.12.19** The name of each `Argument` child element of a `Command` element shall be unique within the command.

[1] See the note attached to 3.12.16; a similar explanation applies to parameters.

**3.13.6** Each `Interface` child element of a `ProvidedInterfaceSet` or `RequiredInterfaceSet` element is based on the `NamedEntityType` (see 3.3.6).

**3.13.7** The name of each `Interface` child element of a `ProvidedInterfaceSet` or `RequiredInterfaceSet` element shall be unique within the containing component.

**3.13.8** Each `Interface` element of a `ProvidedInterfaceSet` or `RequiredInterfaceSet` element shall carry a `type` attribute which identifies the type of the interface by referencing an element of the `DeclaredInterfaceSet` entry of a `Package` or `Component`.

NOTE – Subsection 4.3 provides further details.

**3.13.9** Each `Interface` element may have a `GenericTypeMapSet` element which maps the generic types used to define the interface to the concrete types used in the current component.

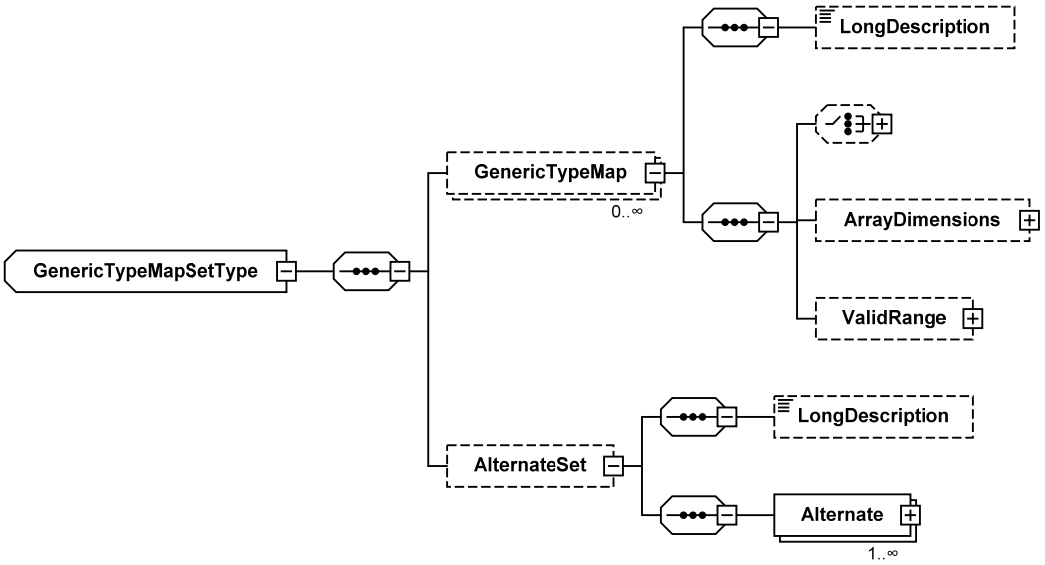



**Figure 3-18: Generic Type Mapping**

**3.13.10** A `GenericTypeMap` element specifies a mapping of a generic type to a concrete type and shall have the attributes and child elements associated with a field (see 3.10.25), with the optional addition of a `fixedValue` attribute.

**3.13.11** The optional `fixedValue` attribute of a `GenericTypeMap` element shall specify a fixed value for the generic type.

NOTE – This is equivalent to specifying a data type with a valid range which contains only the value specified by the `fixedValue` attribute.

**3.13.12** An `AlternateSet` child element of a `GenericTypeMapSet` element, if present, specifies a set of alternative mappings of generic types to a concrete type and shall contain

## 3.14 COMPONENT IMPLEMENTATIONS

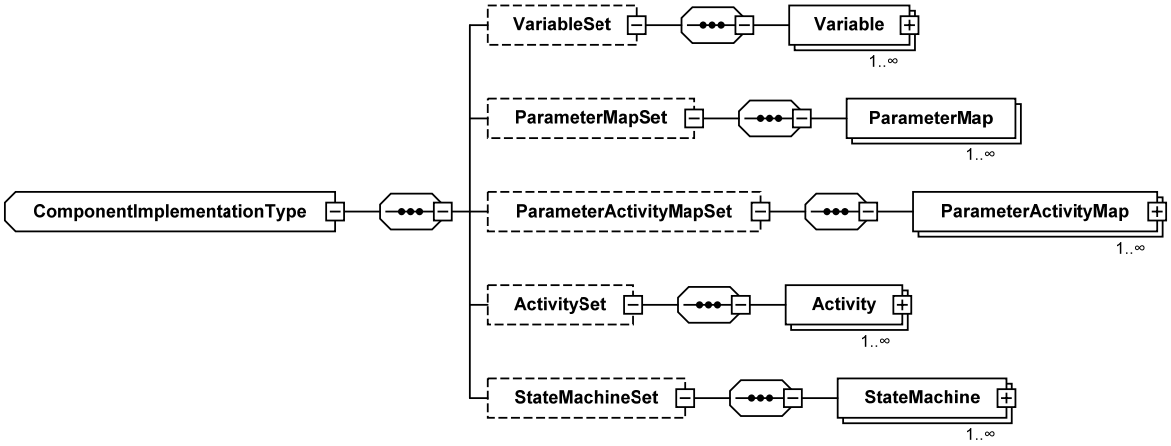



**Figure 3-19: `Implementation` Element of a Component**

NOTES

1 ~~The implementation of a component specifies its behaviour, that is, the way in which data arriving on *required* interfaces gets transformed into data on *provided* interfaces~~The implementation of a component specifies its behaviour, that is, the way in which the component offers data services through *provided* interfaces, and the way in which the component uses data services through *required* interfaces.

2 A set of variables organizes working memory for computation of behaviour.

3 The parameter map set and parameter activity map set provide a terse specification of the transfer of data between required and provided interfaces.

4 For cases in which this is insufficient, the state machine set contains UML state machine graphs which express time-driven and event-driven behaviour.

5 The activity set contains snippets of procedures in a form that is consistent with many structured procedural programming languages. These can be referenced from state machines and parameter activity maps.

**3.14.1** The `Implementation` child element of a `Component` element shall contain zero or one of each of the following elements, in order:

a) `VariableSet`;

b) `ParameterMapSet`;

c) `ParameterActivityMapSet`;

d) `ActivitySet`;

## 3.15 ACTIVITIES

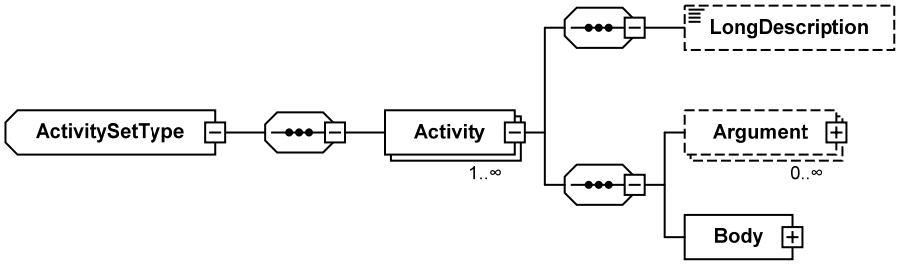



**Figure 3-23: Activities within an ActivitySet Element**

NOTE – An activity is a block of executable statements whose invocation is controlled by one or more state machines.

**3.15.1** The `ActivitySet` element shall contain one or more `Activity` elements.

**3.15.2** Each `Activity` element is based on the `NamedEntityType` (see 3.3.6).

**3.15.3** The name of each `Activity` child element of an `ActivitySet` element shall be unique.

**3.15.4** Each `Activity` element shall contain zero or more `Argument` elements and one `Body` element.

NOTE – `Argument` elements permit the operation of the activity, specified by the `Body` element, to be parameterized. Parameterization means that invocation of the activity must be accompanied by arguments that provide data values to be used by the activity, and the body of the activity contains statements that refer to those arguments. In this document, references to arguments are indicated by the term 'activity argument'.

**3.15.5** Each `Argument` child element of an `Activity` element shall have the attributes and child elements associated with a field (see 3.10.25).

**3.15.6** The name of each `Argument` child element of an `Activity` element shall be unique.

**3.15.7** The `Body` child element of an `Activity` element shall contain one or more of the following elements: `SendParameterPrimitive`, `SendCommandPrimitive`, `Calibration`, `MathOperation`, `Assignment`, `Conditional`, `Iteration`, or `Call`.

**3.15.8** The sequence of elements specified in the `Body` element shall define the sequence of operations of the activity.

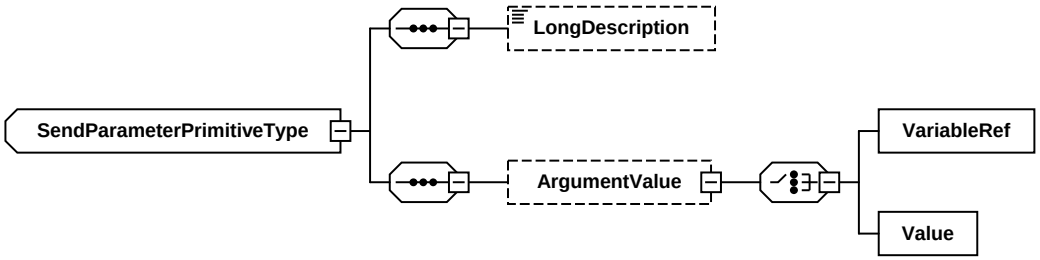



**Figure 3-24: `SendParameterPrimitive` Element**

**3.15.9** A `SendParameterPrimitive` element shall specify the transmission of a parameter request or indication primitive to an interface provided or required by the component type.

**3.15.10** A `SendParameterPrimitive` element shall carry

a) an `interface` attribute, identifying the component interface to which the primitive relates;

b) a `parameter` attribute, identifying the interface parameter to which the primitive relates;

c) an `operation` attribute, identifying whether the primitive is for a `get` or `set` operation;

d) an optional `transaction` attribute which permits this primitive to be related to the opposing primitive of the request/indication pair; and

e) an optional `failed` attribute, defaulting to false, used in an indication to explicitly report failure of the corresponding request.

**3.15.11** The `transaction` attribute of the `SendParameterPrimitive` element shall be present or absent, depending on the conditions given in table 3-5.

**3.15.12** A `SendParameterPrimitive` element may include an `ArgumentValue` element according to the conditions described in table 3-7.

**3.15.13** An `ArgumentValue` element shall include either a `Value` element, specifying a literal value to be associated with the primitive, or a `VariableRef` element, specifying a component variable or activity argument be associated with the primitive (see table 3-7).

**Table 3-7: Arguments to a Primitive**

| Element | Interface Direction | Parameter Operation | Number of Arguments |
|---|---|---|---|
| `SendCommandPrimitive` | any | | 0 or more |
| `OnCommandPrimitive` | any | | 0 or more |
| `SendParameterPrimitive` | required | `Get` | 0 |
| `SendParameterPrimitive` | provided | `Get` | 1 |
| `SendParameterPrimitive` | required | `Set` | 1 |
| `SendParameterPrimitive` | provided | `Set` | 1 |
| `OnParameterPrimitive` | required | `Get` | 1 |
| `OnParameterPrimitive` | provided | `Get` | 0 |
| `OnParameterPrimitive` | required | `Set` | 1 |
| `OnParameterPrimitive` | provided | `Set` | ~~1~~0 |

**3.15.14**The type of the value specified by either the `VariableRef` or `Value` child element of an `ArgumentValue` element shall match the type of the parameter or command argument to which the primitive relates.

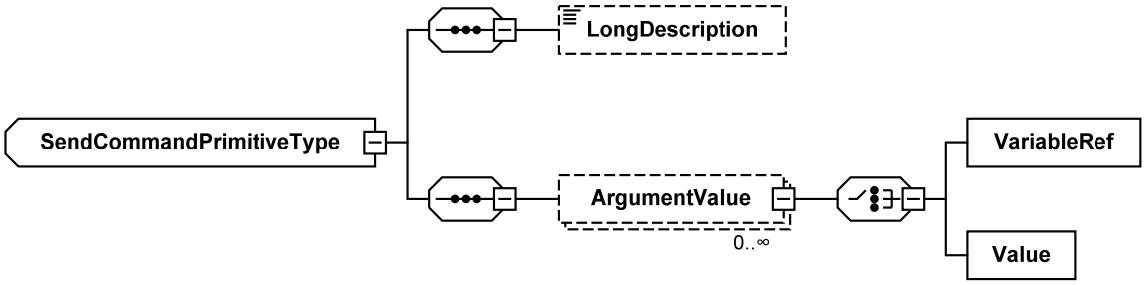



**Figure 3-25: `SendCommandPrimitive` Element**

**3.15.15** A `SendCommandPrimitive` element shall specify the transmission of a command request or indication primitive to an interface provided or required by the component type.

**3.15.16** A `SendCommandPrimitive` element shall carry

a) an `interface` attribute, identifying the component interface to which the primitive relates;

b) a `command` attribute, identifying the interface command to which the primitive relates;

c) an optional `transaction` attribute which permits this primitive to be related to the opposing primitive of the request/indication pair; and

d) an optional `failed` attribute, defaulting to false, used in an indication to explicitly report failure of the corresponding request.

**3.15.17** The `transaction` attribute of the `SendCommandPrimitive` element shall be present or absent according to the conditions expressed in table 3-5.

**3.15.18** A `SendCommandPrimitive` element may include a number of `ArgumentValue` elements according to the conditions described in table 3-7.

**3.15.19** Each `ArgumentValue` child element of a `SendCommandPrimitive` element shall carry a `name` attribute identifying the command argument with which this value is associated.

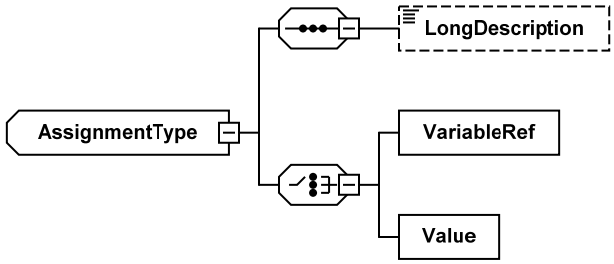



**Figure 3-26: `Assignment` Element**

**3.15.20** An `Assignment` element shall specify the assignment of a value, either by specifying as a literal or by referencing a component variable or activity argument, to a component variable.

**3.15.21** An `Assignment` element shall carry an `outputVariableRef` attribute identifying the component variable to which the value should be assigned.

**3.15.22** An `Assignment` element shall include either a `Value` element, specifying a literal value to be assigned to the output parameter, or a `VariableRef` element which specifies a component variable or activity argument to use as the source of the value to assign to the output parameter.

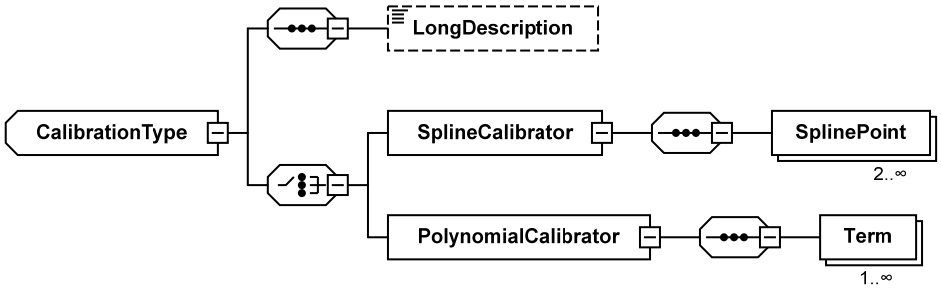



**Figure 3-27: Polynomial and Spline Calibrators within a `Calibration` Element**

**3.15.23** A `Calibration` element shall specify the assignment of a value, either by specifying as a literal or by referencing a component variable or activity argument, to a component variable, translating the value according to a specified calibration operation.

**3.15.24** A `Calibration` element shall carry an `outputVariableRef` attribute identifying the component variable to which the calibrated value should be assigned.

**3.15.25** A `Calibration` element shall include ~~either a `Value` element, specifying a literal value to calibrate before assignment to the output variable, or~~ an `inputVariableRef` element, specifying a component variable or activity argument to use as the source of the value to calibrate before assignment to the output parameter.

**3.15.26** A `Calibration` element shall include either a `SplineCalibrator` or `PolynomialCalibrator` element.

**3.15.27** A `SplineCalibrator` element shall have an attribute `extrapolate`, indicating whether to extrapolate values outside the range of points.

**3.15.28** A `SplineCalibrator` element shall have two or more `SplinePoint` child elements.

**3.15.29** ~~The attributes of a~~A `SplinePoint` child element of a `SplineCalibrator` shall have attributes `raw` and `calibrated`, which together represent a point on the spline curve used to convert from raw to calibrated values, and order, which represents the ~~algorithm used to interpolate values between this point and the next~~maximum degree of the polynomial segment between that point and the point with the next higher raw value. The order attribute value shall be in {1, 2, 3}. Specification of a value *n* for `order` shall also imply that the zeroth through (*n*−1)th derivatives of the polynomial segments ending and beginning at the spline point shall be the same. The value of the polynomial segments at a spline point shall equal the calibrated value of the spline point. If `order` is not specified, it shall be assumed to be 1. The second derivatives of the polynomial segments at the spline points with greatest and least raw values shall be implicitly zero.

NOTES

1 A spline of order 1 is linear (i.e., a traditional point calibration), a spline of order 2 is quadratic, and a spline of order 3 is cubic. There must be the mathematically necessary number of consecutive points of a given order to support higher-order spline curves.

2 The zeroth derivative of a function is the value of the function.

3 The continuity constraints above force equality of polynomial segments at each interior spline point. The spline function passes through each spline point.

**3.15.30** A `PolynomialCalibrator` element shall have one or more `Term` child elements.

**3.15.31** A `Term` child element of a `PolynomialCalibrator` shall have attributes `coefficient` and `exponent`, which together define one term of the polynomial expression used to convert from raw to calibrated values.

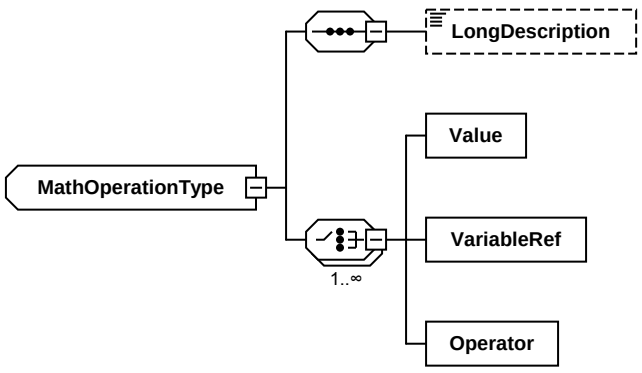



**Figure 3-28: `MathOperation` Element**

**3.15.32** A `MathOperation` element shall specify a mathematical operation in postfix (Reverse Polish) notation.

NOTE – This means that the sequence of values and operators must be valid when taking into account the 'arity' column of table 3-8.

**3.15.33** A `MathOperation` element shall carry an `outputVariableRef` attribute identifying the component variable to which the calculated value should be assigned.

NOTE – The encoding and size of the variable referenced by the `outputVariableRef` attribute determines the precision of the math operation.

**3.15.34** The implementation of math operations should operate internally on floating point numbers. When the output variable reference specifies an integer type, and the math operation computes a non-integer result, the implementation should throw an error condition.

**3.15.35** In order to avoid an error, the math operation must specify how to achieve an integer result using a function such as ceiling, floor, or round.

**3.15.36** A `MathOperation` element shall include a sequence of the following child elements:

a) `Value`;

b) `VariableRef`;

c) `Operator`.

**3.15.37** The `Value` and `VariableRef` child elements of a `MathOperation` element shall have the same contents and meanings as the elements of the same name of an `Assignment element`.

NOTE – Math operations apply to integer and floating point values and variables.

**3.15.38** An `Operator` child element of a `MathOperation` element shall have a single attribute `operator`, which shall be one of the values from table 3-8.

**3.15.43** An `OnConditionTrue` element shall contain one or more of the elements allowed in an activity body specifying the operations to perform if the outcome of the condition expression is 'true'.

**3.15.44** An `OnConditionFalse` element shall contain one or more of the elements allowed in an activity body specifying the operations to perform if the outcome of the condition expression is 'false'.

**3.15.45** An `Iteration` element shall specify the repeated execution of elements of the activity.

**3.15.46** An `Iteration` element shall carry an `iteratorVariableRef` attribute identifying the component variable to use to hold the iteration value.

**3.15.47** An `Iteration` element shall either contain either an `OverArray` element or a `StartAt` element, ~~zero or one~~a `Step` element, and an `EndAt` element, in that order.

**3.15.48** An `Iteration` element shall contain a `Do` element after all other elements.

**3.15.49** The `OverArray` element of an `Iteration` element shall specify an array over which to iterate, assigning the value of each array element, in turn, to the iteration parameter.

**3.15.50** The `StartAt` element shall include either a `Value` element, specifying a literal value to be assigned as the initial value of the iteration parameter, or a `VariableRef` element, specifying a component variable or activity argument to use as the source of the value to use as an initial value of the iteration parameter.

**3.15.51** The `EndAt` element shall include either a `Value` element, specifying a literal value to be used as the final value of the iteration parameter (inclusive), or a `VariableRef` element, specifying a component variable or activity argument to use as the source of the value to use as the final value of the iteration parameter (inclusive).

**3.15.52** A `Step` element shall include either a `Value` element, specifying a literal value to be used as the difference in value of the iteration parameter between iterations, or a `VariableRef` element, specifying a component variable or activity argument to use as the source of the value to be used as the difference in value of the iteration parameter between iterations.

**3.15.53** The `Do` element shall contain one or more of any of the elements allowed in an activity body.

**3.15.54** A `Call` element shall identify a nested activity to be called at this point in the activity execution.

**3.15.55** A `Call` element shall include zero or more `ArgumentValue` elements, each of which, in turn, shall carry a `name` attribute, identifying the name of an activity argument and including either a `Value` element, specifying a literal value to be associated with the named activity argument, or a `variableRef` element, specifying a component variable or activity argument to associate with the named activity argument.

## 3.16 STATE MACHINES

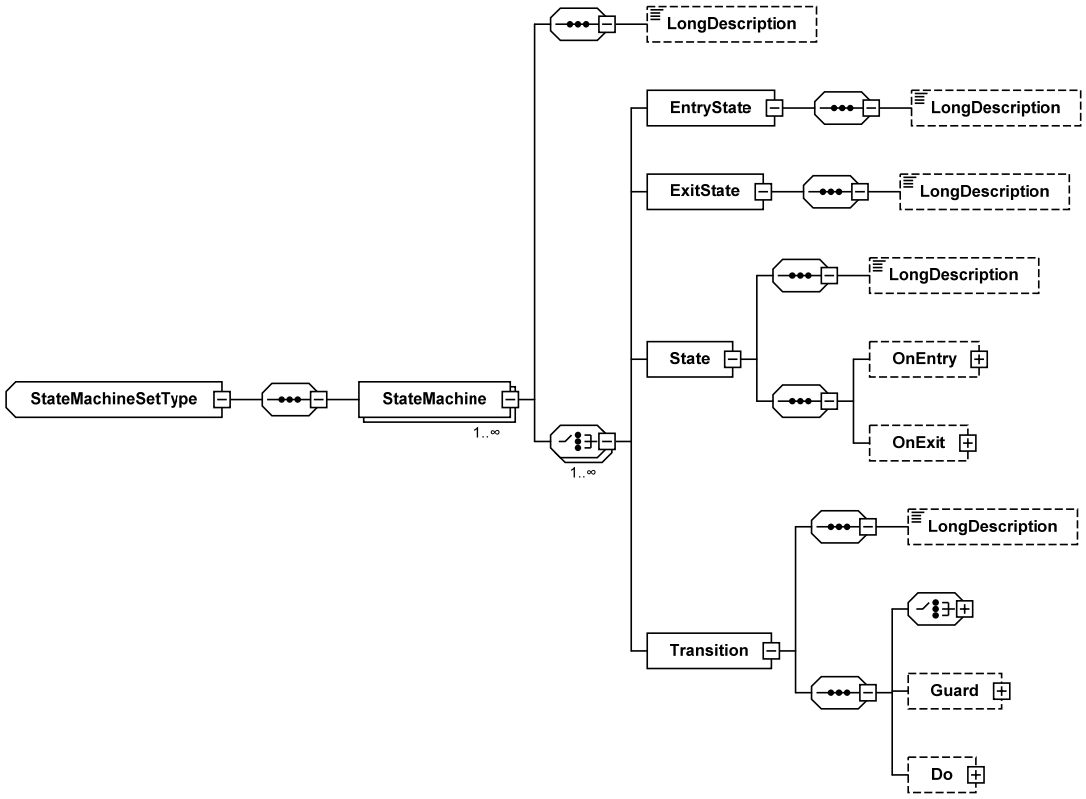



**Figure 3-31: State Machines within a `StateMachineSet` Element**

NOTE – A state machine responds to events and schedules the execution of activities.

~~**3.16.1** Each `StateMachine` element may carry a `defaultEntryState` attribute identifying the name of the state to transition to with no action immediately on initialization.~~

**3.16.1** Each `StateMachine` element shall include one or more of the following elements: `EntryState`, `ExitState`, `State`, and `Transition`.

**3.16.2** Each child element of a `StateMachine` element shall carry a `name` attribute identifying the name of that element.

**3.16.3** The name of each child element of a `StateMachine` element shall be unique within the state machine.

**3.16.4** Each `State` element shall include zero or one of the following elements: `OnEntry`, `OnExit`.

**3.16.5** The `OnEntry`, `OnExit`, and `Do` elements shall each specify the name of an activity, using the `activity` attribute, to be invoked on entry to the state, immediately before exit from the state, and when performing a transition between states, respectively.

**3.16.9** Each `Transition` element shall include one of the following elements:

a) `OnCommandPrimitive`;

b) `OnParameterPrimitive`;

c) `OnTimer`.

**3.16.10** Each `Transition` element shall include zero or one of each of the following elements:

a) `Guard`;

b) `Do`.

**3.16.11** An `OnTimer` element shall contain a `nanosecondsAfterEntry` attribute which indicates the number of nanoseconds that shall elapse between state entry and triggering the transition, providing that the guard condition is met.

**3.16.12** An `OnCommandPrimitive` or `OnParameterPrimitive` element shall identify the primitive that shall be received to trigger the transition, providing that the guard condition is met.

**3.16.13** An `OnParameterPrimitive` element shall carry

a) an `interface` attribute, identifying the component interface to which the primitive relates;

b) a `parameter` attribute, identifying the parameter to which the primitive relates;

c) an `operation` attribute, identifying whether the primitive is for a `get` or `set` operation;

d) an optional `transaction` attribute, permitting the primitive reception to be matched to the corresponding primitive transmission using a string identifier; and

e) an optional `failed` attribute, defaulting to false, identifying whether the transition should be triggered on successful or failed indications.

NOTE – A failed indication can be sent using a primitive with the failed attribute set to true.

**3.16.14** The `transaction` attribute of an `OnCommandPrimitive` or `OnParameterPrimitive` element shall be present according to the conditions defined in table 3-5.

**3.16.15** An `OnParameterPrimitive` element may, according to the conditions defined in table 3-7, include a `VariableRef` element specifying a component variable to receive the value associated with the primitive.

**3.16.16** An `OnCommandPrimitive` element shall carry

a) an `interface` attribute, identifying the component interface to which the primitive relates;

b) a `command` attribute, identifying the command to which the primitive relates;

c) an optional `transaction` attribute, permitting the primitive reception to be matched to the corresponding primitive transmission using a string identifier; and

d) an optional `failed` attribute, defaulting to false, identifying whether the transition should be triggered on successful or failed indications.

NOTE – A failed indication can be sent using a primitive with the failed attribute set to true.

**3.16.17** An `OnCommandPrimitive` element shall include zero or more `ArgumentValue` elements, each of which, in turn, includes ~~a `VariableRef` element~~an `outputVariableRef` attribute which specifies a component variable to associate with a command argument to the primitive.

**3.16.18** A `Guard` child element of a transition shall identify the guard condition that shall be met to trigger the transition, providing that the trigger event has been received.

NOTE – If no `Guard` element is present, no condition need be met to trigger the transition.

**3.16.19** A `Guard` element shall specify a Boolean expression as shown in figure ~~3-30~~3-32.

NOTE – Command arguments are accessible to the expression as described in 4.6.2.17.2.

**4.3.2.11** If a component variable, interface parameter, or argument is used in relation to a destination component variable, interface parameter, or argument, ~~the types of the source and destination shall match~~type conversion shall occur (see 4.8).

**4.3.2.12** If a source literal is used in relation to a destination component variable, interface parameter, or argument, the value of the literal shall be valid according to table 3-1.

**4.3.2.13** Activity or state machine operations which reference nonliteral values shall reference component variables and activity arguments only, not interface parameters.

NOTES

1 Interface parameters can only be accessed using the dedicated operations described in 4.5.

2 Command arguments are accessible to `Guard` expressions as described in 4.6.2.17.2.

**4.3.2.14** Activity or state machine operations which reference a parameter, variable, or argument which is an instance of a container parameter type may select a single entry from the container using the following syntax:

```
{parameter name}.{entry name}
```

**4.3.2.15** Activity or state machine operations which reference a parameter, variable, or argument which is an array may select a single element from the array using the following syntax:

```
{parameter name}[{0-based element index}]
```

NOTES

1 The element index may be specified as an enumeration literal, or the current value of a variable or argument of integer or enumerated type. When an enumerated type indexes an array, the enumeration shall have consecutive values, the lowest of which is zero.

2 Array indexing with out of range (including negative) values is a Failure Detection, Isolation, and Recovery (FDIR) trigger, that is, 'an indication that the device has left the scope of the nominal behaviour documented by the datasheet'. Some cases can be detected statically by datasheet tooling; ideally this would include as a warning the use of an index type that allowed negative values, and as an error in the datasheet the use of an index type with a completely negative range. The implementation of FDIR handling is not specified in this standard.

**4.3.2.16** The above two rules may be chained together to access nested array and container entries. Interpretation shall be left-associative with equal precedence.

NOTE – For chained access, the '`.`' and '`[]`' syntax would be used as appropriate for the type of the object on their left. For example, if '`name`' refered to an instance whose type is an array of quaternions, the array type would be defined with elements that are quaternion type, and the quaternion type could be defined as a container with elements '`x`', '`y`', '`z`', and '`r`'. To refer to the real part of the first quaternion in the array, the syntax is '`name[0].r`'. The '`[0]`' applies to the array type, and yields a quaternion type, to which the '`.r`' applies.

**4.3.2.17** References to data types, by means of `baseType` attributes or by means of `BaseType` elements, collectively form a graph of references. That graph shall contain no circuits.

NOTE – The schema does not enforce this statement. A tool chain is expected to enforce this statement, in order to protect itself from endless referential loops, by not supporting forward references.

**4.3.2.18** The same activity argument name may be used in more than one activity. No activity argument name shall be the same as a component variable name.

**4.3.2.19** State-machine-transition guard expressions may reference arguments of the primitive that triggers the transition.

## 4.5 PRIMITIVE ASSOCIATIONS

### 4.5.1 OVERVIEW

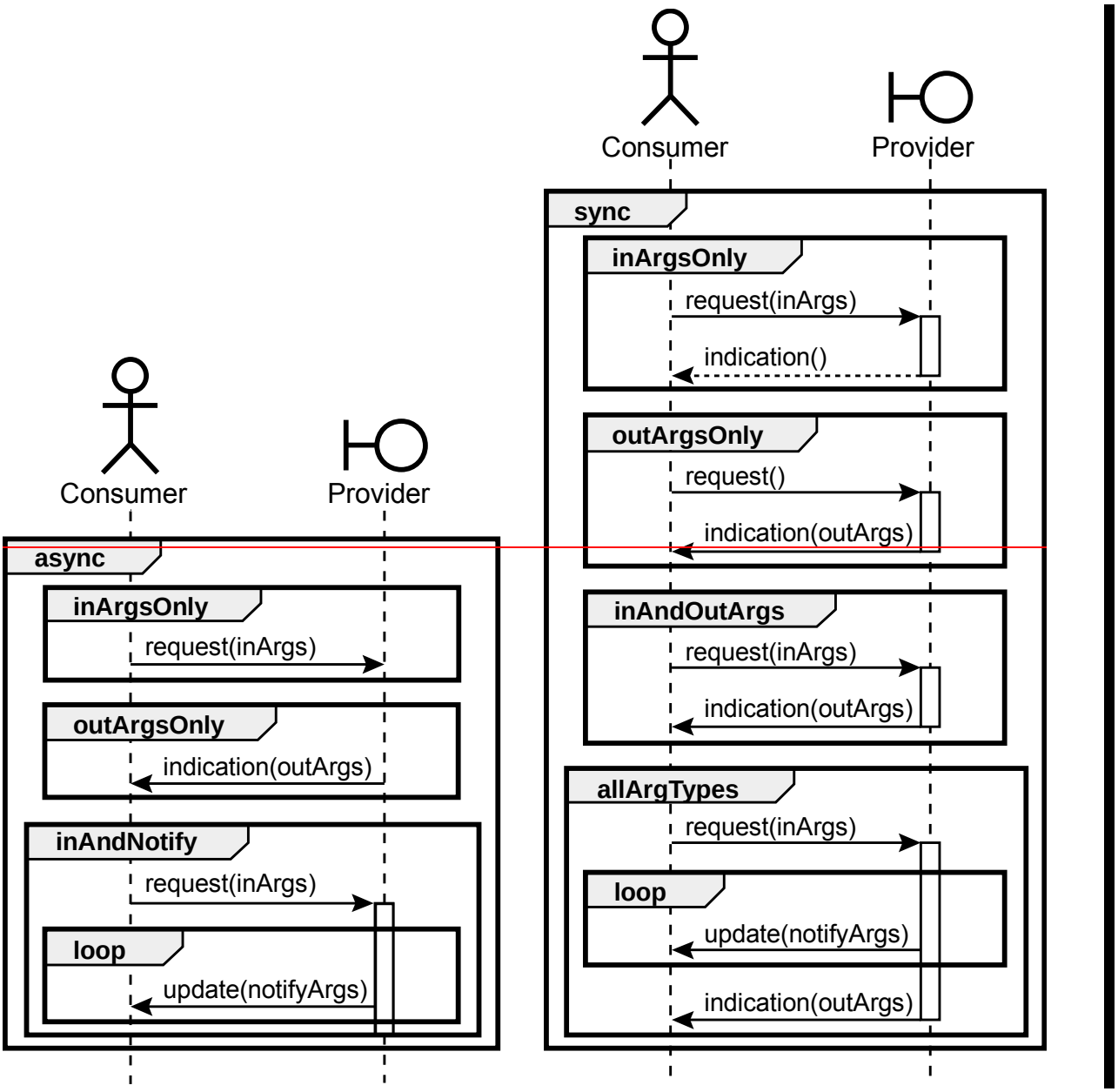
The state machines making up a component are driven by messages arriving on the interfaces to and from that component. These messages, called *primitives*, are defined by the set of parameters and commands present on the interface definitions.

Interfaces are two-way and asymmetric; the component requiring the interface is referred to as the consumer, and the component providing it as the provider.

~~If the parameter or command mode is sync, then messages in both directions must be present even if they have no content. In other cases, empty messages are omitted.~~

If the parameter mode is sync, then messages in both directions will be present even if they have no content. For a command, the `pattern` describes the set of messages that are present (see figure 4-3). If not specified, the pattern of a command is calculated according to table 4-1. In either case, a message that exists but has no content is a pulse; it carries no information, but can still trigger a state transition.

Parameters that are read-only may not be set by a client of that interface.


Consumer
Provider
sync
inArgsOnly
request(inArgs)
indication()
outArgsOnly
request()
indication(outArgs)
inAndOutArgs
request(inArgs)
indication(outArgs)
allArgTypes
request(inArgs)
loop
update(notifyArgs)
indication(outArgs)
Consumer
Provider
async
inArgsOnly
request(inArgs)
outArgsOnly
indication(outArgs)
inAndNotify
request(inArgs)
loop
update(notifyArgs)

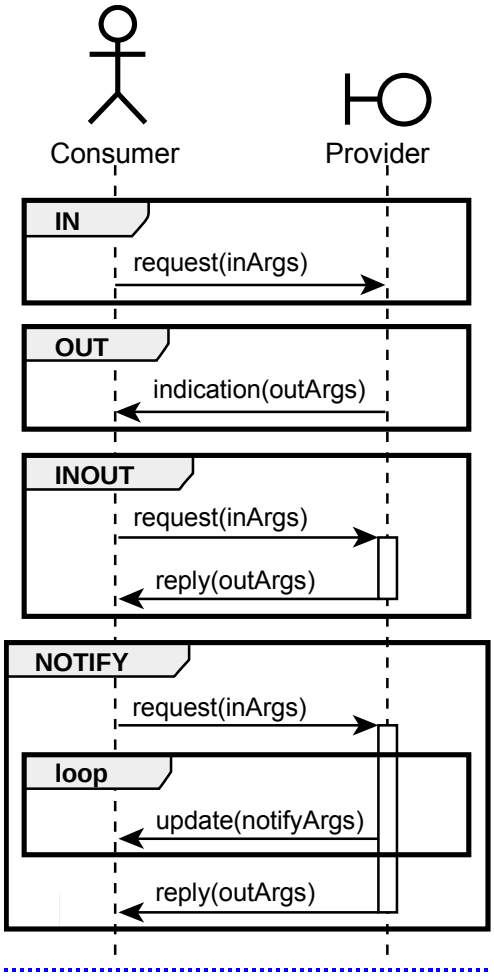



**Figure 4-3: Command Definitions and Primitives**

~~EDS has seven distinct interaction patterns for commands, based on whether the command mode is async or sync, and on the set of input, output, and notify arguments it has.~~

EDS has four distinct interaction patterns for commands. If `pattern` is specified, then it is one of the patterns in figure 4-3; that is `in`, `out`, `inout`, or `notify`. If `pattern` is not specified, then the pattern is inferred as in table 4-1, based on whether the command mode is async or sync, and on the set of input, output, and notify arguments it has.

**Table 4-1: Command Message Pattern Inference**

| Command Argument Modes Present | Command mode="async" | Command mode="sync" |
|---|---|---|
| in | in | in |
| out | out | out |
| notify | error | notify |
| in and out, or inout | error | inout |
| in and notify | error | notify |
| out and notify | error | notify |
| (in and out, or inout) and notify | error | notify |

NOTE – These patterns are similar to those in the Mission Operations Message Abstraction Layer (MAL) (reference [D5]). As such, a SEDS could be used to describe the on-the-wire encoding of a MAL message.

### 4.5.2 SPECIFICATION

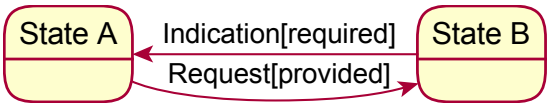



**Figure 4-4: Primitives That Trigger State Transition**

**4.5.2.1** If a parameter primitive is to be received (to trigger a state machine transition), the primitive shall be

a) a get operation primitive from an interface provided by the component identifying a parameter value read request;

b) a set operation primitive from an interface provided by the component identifying a parameter value write request;

c) a get operation primitive from an interface required by the component identifying a parameter value read indication; or

d) a set operation primitive from an interface required by the component identifying a parameter value write indication.

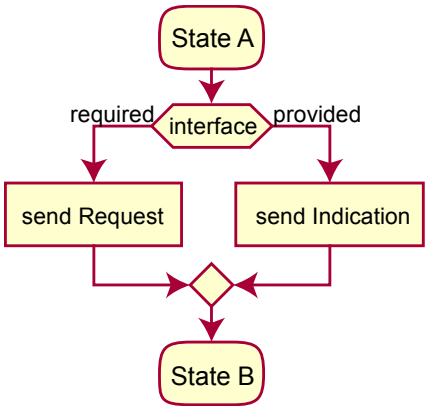



**Figure 4-5: Primitives Sent During Activity Execution**

**4.5.2.2** If a parameter primitive is to be transmitted (by an activity), the primitive shall be

a) a get operation primitive to an interface provided by the component identifying a parameter value read indication;

b) a set operation primitive to an interface provided by the component identifying a parameter value write indication;

**4.5.2.11** The reception of a command request primitive may specify the component variable into which the value of any arguments of modes `in` or `inout` can be stored.

**4.5.2.12** The reception of a command indication primitive may specify the component variable into which the value of any arguments of modes `out` or `inout` can be stored.

**4.5.2.13** The reception of a command update primitive may specify the component variable into which the value of any arguments of modes `notify` can be stored.

**4.5.2.14** The transmission of a command request primitive shall specify a value for all arguments of modes `in` or `inout`.

**4.5.2.15** The transmission of a command indication primitive shall specify a value for all arguments of modes `out` or `inout`.

**4.5.2.16** The transmission of a command update primitive shall specify a value for all arguments of modes `notify`.

**4.5.2.17** A contradiction between specified `pattern` and pattern inferred from `mode` shall be invalid.

NOTE – Otherwise, the data sheet is invalid, although this cannot necessarily be statically detected. When developing a SEDS for a device, a mapping from primitives to state machine transitions is implied by the guard conditions. The mapping could contain complex expressions that depend upon data appearing in the interface, so testing all combinations could be impractical.

**4.6.2.12** Only one `EntryState` element shall be present in a given state machine~~, and if it is present, the `defaultEntryState` attribute shall not be set~~.

**4.6.2.13** If an explicit `EntryState` element is present, it shall be used as the starting state.

NOTE – This allows explicit specification of initialization actions.

~~**4.6.2.14** If the `defaultEntryState` attribute is present, a default starting state shall be used, causing an immediate and unconditional transition, with no action, into the specified state.~~

**4.6.2.14** If a state machine transitions to an exit state, the device should be considered to have left the scope of the nominal behaviour documented by the datasheet.

**4.6.2.15** If an error occurs in an activity, such as an arithmetic error, or such as violation of the range of a range-constrained type, the device should be considered to have left the scope of the nominal behavior documented by the datasheet. Detection of violation of a range constraint should be done by software generated by a tool chain and should not necessarily be explicit in the activity body.

NOTE – Detection of arithmetic errors is typically done by computer hardware.

**4.6.2.16** If an activity contains an assignment, calibration, or math operation in which the semantic tags of the result of computation are incompatible with the semantic tags of the output variable reference, the tool chain may report an error in the SEDS instance.

**4.6.2.17** An implementation of a state machine should satisfy the following sequence of operations.

**4.6.2.17.1** A primitive command or parameter event occurs.

**4.6.2.17.2** The state machine evaluates on-conditions and `Guards` of all transitions whose fromState is the current state.

NOTES

1 The Boolean expression of a `Guard` in a transition with an OnCommandPrimitive element with ArgumentValue elements may refer to the values of those arguments of the command by the name attribute in the ArgumentValue element.

2 The Boolean expression of a `Guard` in a transition with an OnParameterPrimitive element cannot refer to the parameter value, but the definition of the parameter in a

Parameter element in a DeclaredInterfaceSet implies that the type of the parameter must be checked in order to satisfy the OnParameterPrimitive condition.

**4.6.2.17.3** One of three mutually exclusive conditions occurs here.

**4.6.2.17.3.1** If more than one transition passes its on-condition and `Guard`, then that is an error condition in flight, the possibility of which should have been detected by static analysis or by testing before flight. (See 4.6.2.11, which implies that the state machine leaves the scope of nominal behavior defined by the data sheet.)

**4.6.2.17.3.2** If exactly one transition passes its `Guard`, then the next step is 4.6.2.17.4.

**4.6.2.17.3.3** If no transition passes, then the state machine does not change state and the procedure exits.

NOTE – Only step 4.6.2.17.3.2 can continue here.

**4.6.2.17.4** If the transition has an OnCommandPrimitive element with ArgumentValue elements, then the values are stored in the associated component variables. If the transition has an OnParameterPrimitive element with a VariableRef element, then the value of the parameter is stored in a component variable identified in the VariableRef element.

**4.6.2.17.5** The state machine executes the OnExit activity of the current state.

**4.6.2.17.6** The state machine executes the Do activity of the transition.

**4.6.2.17.7** The state machine changes to the state identified by the toState of the transition.

**4.6.2.17.8** The state machine executes the OnEntry activity of the state to which it transitions.

## 4.7 ENCODING AND DECODING

### 4.7.1 OVERVIEW

Any interface definition with `level` set to `subnetwork` corresponds to a subnetwork Service Access Point (SAP). When a component uses one of these SAPs as a required interface and refers to that required interface in a primitive association, the tool chain should interpret the reference as an interaction through a subnetwork SAP.

When primitives are sent on such an interface, they must be translated into Protocol Data Units (PDUs) by a process known as *encoding*. Parts of this encoding (such adding spacewire routing headers) are specified by the definition of the subnetwork protocol in use, and therefore do not need to be specified in the datasheet using such an interface.

Of the actual arguments required by the underlying subnetwork implementation,

## 4.8 TYPE CONVERSION

### 4.8.1 OVERVIEW

In SEDS, a type defines two things:

a) an abstract, implementation-independent set of possible values, optionally with usage constrained by semantic attributes;

b) optionally, a mechanism to translate a value from that set to and from a binary representation.

The latter is known as encoding and decoding (see 4.6.2.17 for details).

Type conversion is the process for taking an abstract value belonging to one type and finding the equivalent abstract value of another type. This process can fail if there are values in the source value set that are not in that of the target type, for example, when the target type has a smaller numeric range. If a device behaves in such a way that this failure actually occurs, the device should be considered to have left the scope of the nominal behavior documented by the datasheet.

Conversion is always used in assignment between, and operations on, component variables, and activity arguments. It is also used for interface command arguments that are not data units.

In contrast to encoding and decoding, type conversion may never change the value converted. A concrete implementation that did so, for a device that had not itself exhibited a prior error, would itself be in error.

Conversion to and from the same type, or identical types, is a null operation that can never fail.

### 4.8.2 SPECIFICATION

**4.8.2.1** A value shall be converted when

a) it is assigned to a component variable via the `outputVariableRef` attribute of an `Assignment`, `Calibration`, or `MathOperation`;

b) it is passed as an argument of an activity;

c) it is passed or received as a non-data-unit argument of a command.

**4.8.2.2** The type to be converted to shall be that of the target field.

NOTE – The named type referenced by the field can be modified by inline type descriptors such as `ValidRange` or `ArrayDimensions` (see 3.11).

**4.8.2.3** Four fundamental categories of types shall be subject to conversion:

- set-based values (`BooleanDataType`, `EnumeratedDataType`, enumerated `SubRangeDataType`, enumerated `StringDataType`, and corresponding fields);
- range-based values (`IntegerDataType`, `FloatDataType`, numeric `SubRangeDataType`, and corresponding fields);
- sequences (non-enumerated `StringDataType`, `BinaryDataType`, `ArrayDataType`, and corresponding fields); and
- structured values (`ContainerDataType`).

**4.8.2.4** Conversion shall fail

a) always between types of different categories;

b) for set-based values outside the enumerated set of the target type;

c) for range-based values outside the valid range of the target type;

d) for sequences with a length outside the valid length of the target sequence;

e) for sequences if it fails for any value within the source sequence;

f) for structured values if the target type has a field not present in the source value;

g) for structured values if it fails for any value within the structure.

**4.8.2.5** For numeric data types that include a calibrator, the range constraints shall refer to the uncalibrated value, and the `unit` attribute shall refer to the calibrated value. If both the source and the target have calibrators, then both calibrators must be identical, or conversion shall fail.

**4.8.2.6** If a conversion in a data sheet will always fail, that data sheet shall be invalid

NOTE – For example, a conversion between different `EnumeratedDataTypes` or from a literal value outside the target range will always fail.

**4.8.2.7** If any conversion fails, the device shall be considered to have left the scope of the nominal behavior documented by the datasheet.

NOTE – If a conversion in a datasheet will possibly fail, datasheet tooling could issue a diagnostic warning. For example, an assignment between numeric values with overlapping ranges will fail if the assigned value is not within the intersection of the ranges. This will be the case when assigning between types representing signed and unsigned integers of the same size. Another example: If the usage constraints of explicit semantic tags of the source differ from those of the target, a diagnostic warning could be issued.

## B1.5 RELIABILITY

While it is assumed that the underlying mechanisms used to implement the devices operate correctly, the SEDS make no assumptions as to their reliability.

## B2 SANA CONSIDERATIONS

The recommendations in this document have created the following SANA registry, named 'Spacecraft Onboard Interface Services Electronic Data Sheets and Dictionary of Terms' and located at http://sanaregistry.org/r/sois/. The registry consists of a set of files that constitute an XML schema, an ontology, and related files.

The registration rule for change to this registry requires an engineering review by a designated managing authority. The managing authority shall be assigned by the SOIS-APP working group Chair, or in absence, Area Director. The managing authority may request assistance from subject matter experts provided or recommended by participating agencies.

At time of publication, the registry contains the items in table B-1:

**Table B-1: SANA Registry Content (Normative Unless Noted Otherwise)**

| File | Description |
|---|---|
| seds.xsd | The schema for SOIS Electronic Data Sheets. |
| seds-core-semantics.xsd | The SOIS Dictionary of Terms in the form of a schema to be included by seds.xsd. |
| seds-extension-semantics.xsd | A non-normative schema to be included by seds.xsd for non-interoperable terms that are needed by a project sooner than the terms can be incorporated in the DoT. |
| ccsds.sois.modops.xml | A non-normative collection of definitions that may be referenced by SEDS instances for models of operations. |
| ccsds.sois.seds.xml | A non-normative collection of definitions that can reduce the number of definitions in an electronic data sheet. |
| ccsds.sois.subnetwork.xml | A non-normative collection of definitions that can facilitate integration with SOIS subnetwork services. |
| dod.milbus.milstd1553.xml | A non-normative collection of definitions that can facilitate integration with DOD Milbus 1553 subnetworks. |
| esa.ecss.milstd1553.xml | A non-normative collection of definitions that can facilitate integration with ESA ECSS 1553 subnetworks. |
| sois.0.owl | The ontology for SOIS Dictionary of Terms. This ontology imports sysml-qudv-si-sois.owl. |
| sysml-qudv.owl | The original proof-of-concept definition of quantities, units, dimensions, and values. |

| | |
|---|---|
| sysml-qudv-si.owl | The original proof-of-concept extension of QUDV to the International System of Units. This ontology imports SysML-QUDV.owl. |
| sysml-qudv-si-sois.owl | An extension of the original QUDV ontologies to support units used in SOIS EDS. This ontology imports SysML-QUDV-SI.owl. |
| soisOwlTools.zip | A non-normative compressed project that contains open-source utilities that are intended to perform the following functions:<br>– Converts a conformant ontology into a seds-core-semantics.xsd<br>~~– Extracts a SEDS instance that contains definitions of standard types~~<br>– Extracts an HTML file for human-readable listing of Dictionary of Terms.<br>– Extracts an XML file of units and quantity kinds for use by tool chain functions for analysis of units |
| seds.xsd | The schema for SOIS Electronic Data Sheets. |

## B3 PATENT CONSIDERATIONS

The technology used in managing SEDS (~~xml~~XML and ~~xsd~~XSD) is in the public domain.

# ANNEX C

# ABBREVIATIONS AND ACRONYMS

# (INFORMATIVE)

| | |
|---|---|
| CBOR | Concise Binary Object Representation |
| CCSDS | Consultative Committee for Space Data Standards |
| DoT | Dictionary of Terms |
| FDIR | Failure Detection, Isolation, and Recovery |
| MAL | Message Abstraction Layer |
| OSI | Open Systems Interconnection |
| OWL | Web Ontology Language |
| PDU | Protocol Data Unit |
| PICS | Protocol Implementation Conformance Statement |
| QUDV | Quantities, Units, Dimensions, Values |
| SANA | Space Assigned Numbers Authority |
| SAP | Service Access Point |
| SEDS | SOIS Electronic Data Sheet |
| SOIS | Spacecraft Onboard Interface Services |
| XML | eXtensible Markup Language |
| XSD | XML Schema Definition |

# ANNEX D

# INFORMATIVE REFERENCES (INFORMATIVE)

[D1] *Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*. 2nd ed. International Standard, ISO/IEC 7498-1:1994. Geneva: ISO, 1994.

[D2] *Spacecraft Onboard Interface Services*. Issue 2. Report Concerning Space Data System Standards (Green Book), CCSDS 850.0-G-2. Washington, D.C.: CCSDS, December 2013.

[D3] *The Application of Security to CCSDS Protocols*. Issue 3. Report Concerning Space Data System Standards (Green Book), CCSDS 350.0-G-3. Washington, D.C.: CCSDS, March 2019.

[D4] *Electronic Data Sheets and Dictionary of Terms for Onboard Devices and Components*. Report Concerning Space Data System Standards. Forthcoming.

[D5] *Mission Operations Message Abstraction Layer*. Issue 2. Recommendation for Space Data System Standards (Blue Book), CCSDS 521.0-B-2. Washington, D.C.: CCSDS, March 2013.

# ANNEX E

# EXAMPLE SEDS/XML SCHEMA INSTANTIATIONS

# (INFORMATIVE)

```
<?xml version="1.0" encoding="UTF-8"?>
<DataSheet
   xmlns="http://www.ccsds.org/schema/sois/seds"
   xmlns:xi="http://www.w3.org/2001/XInclude"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.ccsds.org/schema/sois/seds seds.xsd">
  <Device name="SimpleDevice" shortDescription="Simple arbitrary example of SEDS XML usage">
  </Device>
  <xi:include href="ccsds.sois.subnetwork.xml" xpointer="element(/1/1)"/>
  <Package name="SimpleDemo">
    <DataTypeSet>
      <IntegerDataType name="MyInteger">
        <Range>
            <MinMaxRange min="0" max="4294967296" rangeType="inclusiveMinInclusiveMax" />
        </Range>
    </IntegerDataType>
    </DataTypeSet>
    <DeclaredInterfaceSet>
      <Interface name="DeviceAccessInterface">
        <ParameterSet>
          <Parameter name="DeviceMode" type="MyInteger"/>
        </ParameterSet>
        <CommandSet>
          <Command name="DoSomething">
            <Argument name="WithANumber" type="MyInteger" mode="in"/>
          </Command>
        </CommandSet>
      </Interface>
    </DeclaredInterfaceSet>
    <ComponentSet>
      <Component name="DeviceDACP">
        <ProvidedInterfaceSet>
          <Interface name="VendorSpecificInterface" type="DeviceAccessInterface"/>
        </ProvidedInterfaceSet>
        <RequiredInterfaceSet>
          <Interface name="Subnetwork" type="CCSDS/SOIS/Subnetwork/MASInterfaceType"/>
        </RequiredInterfaceSet>
      </Component>
    </ComponentSet>
  </ Package >
</DataSheet>
```

The above example shows a datasheet defining a device `SimpleDevice` with a single component `DeviceDACP` that in turn provides a single interface, `VendorSpecificInterface`. The interface type `DeviceAccessInterface` has one command `DoSomething` and one parameter `DeviceMode`. Both of those definitions share a single data type, `MyInteger`.

The definition of the subnetwork interface used (*`MASInterfaceType`*) is provided in an external file. It should be noted that in this example, no implementation of the component is defined; a fully specified device datasheet would include the logical transformations needed to map between the required and provided interfaces as state machines.