# 

Report Concerning Space Data System Standards

|  |
| --- |
| Electronic Data Sheets and Common Dictionary of Terms for Onboard Devices and Components |

Draft Informational Report

CCSDS 870.0-G-0

Draft Green Book

June 2017

AUTHORITY

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
|  | Issue: | Draft Green Book, Issue 0 |  |
|  | Date: | June 2017 |  |
|  | Location: | Not Applicable |  |
|  | | | |

**(WHEN THIS INFORMATIONAL REPORT IS FINALIZED, IT WILL CONTAIN THE FOLLOWING STATEMENT OF AUTHORITY:)**

This document has been approved for publication by the Management Council of the Consultative Committee for Space Data Systems (CCSDS) and reflects the consensus of technical experts from CCSDS Member Agencies. The procedure for review and authorization of CCSDS documents is detailed in *Organization and Processes for the Consultative Committee for Space Data Systems* (CCSDS A02.1-Y-4).

This document is published and maintained by:

CCSDS Secretariat

Space Communications and Navigation Office, 7L70

Space Operations Mission Directorate

NASA Headquarters

Washington, DC 20546-0001, USA

FOREWORD

This document is a CCSDS Informational Report to assist readers in understanding the Spacecraft Onboard Interface Services (SOIS) documentation on Electronic Data Sheets (SEDS) and the Common Dictionary of Terms (DoT) for Onboard Devices and Components. It has been prepared by the Consultative Committee for Space Data Systems (CCSDS). The concepts described herein are the baseline concepts for the CCSDS standardisation activities in respect of communication services and generic support services to be used in the flight segment of spacecraft systems.

This Report describes the requirements and use cases for electronic data sheets (SEDS) providing an interchangeable description of the data interface of devices and software onboard spacecraft, including the provision of semantic information through a Common Dictionary of Terms (DoT). It also provides worked examples of SEDS. It is intended to serve as a reference for both SEDS users and SEDS tool chain implementers.

Through the process of normal evolution, it is expected that expansion, deletion, or modification of this document may occur. This document is therefore subject to CCSDS document management and change control procedures which are defined in *Organization and Processes for the Consultative Committee for Space Data Systems* (CCSDS A02.1-Y-4). Current versions of CCSDS documents are maintained at the CCSDS Web site:

http://www.ccsds.org/

Questions relating to the contents or status of this document should be addressed to the CCSDS Secretariat at the address indicated on page i.

At time of publication, the active Member and Observer Agencies of the CCSDS were:

Member Agencies

* Agenzia Spaziale Italiana (ASI)/Italy.
* Canadian Space Agency (CSA)/Canada.
* Centre National d’Etudes Spatiales (CNES)/France.
* China National Space Administration (CNSA)/People’s Republic of China.
* Deutsches Zentrum für Luft- und Raumfahrt (DLR)/Germany.
* European Space Agency (ESA)/Europe.
* Federal Space Agency (FSA)/Russian Federation.
* Instituto Nacional de Pesquisas Espaciais (INPE)/Brazil.
* Japan Aerospace Exploration Agency (JAXA)/Japan.
* National Aeronautics and Space Administration (NASA)/USA.
* UK Space Agency/United Kingdom.

Observer Agencies

* Austrian Space Agency (ASA)/Austria.
* Belgian Federal Science Policy Office (BFSPO)/Belgium.
* Central Research Institute of Machine Building (TsNIIMash)/Russian Federation.
* China Satellite Launch and Tracking Control General, Beijing Institute of Tracking and Telecommunications Technology (CLTC/BITTT)/China.
* Chinese Academy of Sciences (CAS)/China.
* Chinese Academy of Space Technology (CAST)/China.
* Commonwealth Scientific and Industrial Research Organization (CSIRO)/Australia.
* Danish National Space Center (DNSC)/Denmark.
* Departamento de Ciência e Tecnologia Aeroespacial (DCTA)/Brazil.
* European Organization for the Exploitation of Meteorological Satellites (EUMETSAT)/Europe.
* European Telecommunications Satellite Organization (EUTELSAT)/Europe.
* Geo-Informatics and Space Technology Development Agency (GISTDA)/Thailand.
* Hellenic National Space Committee (HNSC)/Greece.
* Indian Space Research Organization (ISRO)/India.
* Institute of Space Research (IKI)/Russian Federation.
* KFKI Research Institute for Particle & Nuclear Physics (KFKI)/Hungary.
* Korea Aerospace Research Institute (KARI)/Korea.
* Ministry of Communications (MOC)/Israel.
* National Institute of Information and Communications Technology (NICT)/Japan.
* National Oceanic and Atmospheric Administration (NOAA)/USA.
* National Space Agency of the Republic of Kazakhstan (NSARK)/Kazakhstan.
* National Space Organization (NSPO)/Chinese Taipei.
* Naval Center for Space Technology (NCST)/USA.
* Scientific and Technological Research Council of Turkey (TUBITAK)/Turkey.
* South African National Space Agency (SANSA)/Republic of South Africa.
* Space and Upper Atmosphere Research Commission (SUPARCO)/Pakistan.
* Swedish Space Corporation (SSC)/Sweden.
* Swiss Space Office (SSO)/Switzerland.
* United States Geological Survey (USGS)/USA.

DOCUMENT CONTROL

|  |  |  |  |
| --- | --- | --- | --- |
| **Document** | **Title and Issue** | **Date** | **Status** |
| CCSDS 870.0-G-0 | Electronic Data Sheets and Common Dictionary of Terms for Onboard Devices and Components, Draft Informational Report, Issue 0 | June 2017 | Current draft |
|  |  |  |  |
|  |  |  |  |

CONTENTS

Section Page

 1

Executive Summary ix

1 Introduction 1-1

1.1 Purpose 1-1

1.2 Scope 1-1

1.3 Applicability 1-1

1.4 Rationale 1-1

1.5 Approach 1-2

1.6 Terms and Definitions 1-2

1.7 References 1-3

2 OVERVIEW 2-5

3 SEDS: Interchangeable Model of Data Interfaces 3-7

3.1 SEDS as a Model 3-7

3.1.1 Spacecraft Onboard Interface Services 3-8

3.1.2 Datasheets in a Vehicle Network 3-10

3.1.3 The State of SEDS Technology 3-12

3.2 Usage of SEDS Inside a Project 3-12

3.2.1 Authoring an Instance of SEDS 3-13

3.2.2 Validating an Instance of SEDS 3-13

3.2.3 Configuration Management of SEDS Library 3-15

4 Devices 4-16

4.1 Scenarios for Devices 4-16

4.1.1 Device Interoperability Across Projects 4-16

4.1.2 Interoperability for Docking Vehicles 4-17

4.2 Device Data Sheet Concepts 4-18

4.2.1 Device Components and Interfaces 4-18

4.2.2 Interfaces and PrimItives 4-19

4.2.3 Data Types 4-21

4.2.4 Generic Types and Interface Usages 4-22

4.2.5 Component Implementation 4-23

4.3 Device Data Sheet Examples 4-24

4.3.1 Summary of Parts of a Device Data Sheet 4-25

4.3.2 Distinguishing Packet Formats by their Headers 4-26

4.3.3 Specifying Device-Specific Behavior 4-30

4.3.4 Mapping from Device-Specific to Functional Interface 4-33

4.3.5 Calibrating Raw Parameters 4-36

4.3.6 Failure Detection Behavior 4-39

4.3.7 The Subnetwork Interface 4-40

4.3.7.1 Simple Generic Types 4-40

4.3.7.2 Alternate Generic Types 4-42

4.3.7.3 Handling Errors Detected in the Subnetwork 4-46

4.3.8 List or Array 4-46

4.3.9 Containers that Contain their Length 4-47

4.3.10 Array Length by Index Type 4-49

4.3.11 Overriding Types in Declared Interfaces 4-49

4.3.12 Acquiring Telemetered Data 4-50

4.3.12.1 Describing the Interfaces 4-51

4.3.12.2 Explicit Movement of Data from Required to Provided Interfaces 4-55

4.3.12.2.1 Crossing the Device-Specific Access Protocol Component 4-56

4.3.12.2.2 Crossing the Device Abstraction Control Procedure Component 4-58

4.3.12.3 Implicit Movement of Data from Required to Provided Interface 4-59

4.3.12.3.1 Implicitly Crossing the Device-Specific Access Protocol Component 4-59

4.3.12.3.2 Implicitly Crossing the Device Abstraction Control Procedure Component 4-61

4.3.13 Commanding a Device 4-62

4.3.13.1 Describing the Interfaces 4-62

4.3.13.2 Explicit Movement of Data from Provided to Required Interfaces 4-65

5 Flight Software 5-68

5.1 Service Interoperability across Projects 5-68

5.2 Service Data Sheet Example 5-69

6 System Design 6-72

6.1 Overall Data-Flow/Process 6-72

6.2 Mission Development Phases 6-74

6.2.1 Selecting Parts 6-75

6.2.2 Validating a Mission Configuration 6-77

6.2.3 Generating Flight Software 6-79

6.2.4 Mission Control Software 6-80

6.2.5 Simulation 6-81

6.2.6 Reality In the Loop 6-83

6.2.7 Device Development 6-84

6.2.8 Maintenance of Terminology 6-84

6.3 Project Parameters 6-85

6.3.1 Parametric Data Sheets 6-85

6.3.1.1 Parametric Data Sheet Concepts 6-86

6.3.1.2 Parametric Data Sheet Example 6-87

6.3.2 Context of Usage 6-87

6.3.2.1 Model of Production 6-88

6.3.2.2 Configuration Management 6-88

6.3.2.3 Model of Operation 6-89

6.3.2.4 Parametric Data 6-89

6.3.3 Deployment Description Data Sheet Place-Holder 6-90

7 Software Verification Facility: Simulation 7-92

7.1 Simulation Testing 7-92

8 Mission Control 8-94

8.1 Interoperability with Mission Operations 8-94

8.2 Interoperability with Spacecraft Monitoring and Control 8-95

9 Documentation 9-96

10 Available Reference Material 10-97

11 Security 11-98

ANNEX A ACRONYMS and Abbreviations A-1

ANNEX B SOURCE of SEDS Software B-2

# Executive Summary

SOIS electronic data sheets (SEDS) facilitate the integration of components to compose a vehicle that can perform a mission. By describing the interfaces of components in algorithmically accessible files, SEDS eliminate the tedium of composition that presently occupies much of the effort of engineers. Instead, that non-recurring engineering effort can be expedited by software tools.

* This document describes many of the functions that SEDS make possible, simply by describing the interfaces of components in a managed language. Anywhere a person would need to make a connection between components, SEDS can help. During the development of a device, a SEDS may be used as a repository for initial requirements, interface control specifications, and validation data. Human-readable documentation can be generated from this information at any time.
* A SEDS can describe the design model of a device, so during manufacture of a device it can provide parameters for verification of function of the device. An instance-specific SEDS can be constructed by copying the design SEDS, adding the serial number to the metadata in the copy, and updating the calibration constants in the copy to agree with measured performance of the instance.
* During design, engineers must select and adapt devices that function together in an assembly to accomplish a mission. SEDS enable construction and search of a database of candidate devices. The collection of SEDS of devices selected for a mission can be transformed to construct a system data base and system data repository. The SEDS of selected devices can be transformed and loaded into standard engineering tools in which engineers add configuration information to produce a system model.
* SEDS provide the device parameters for simulators that can detect some emergent aggregate phenomena before physical integration. Device parameters include nominal operating conditions and performance, which can be checked during simulation.
* During physical integration, engineers validate the design by assembling devices and testing the actual result of composition, which can affect data rates, power levels, thermal conditions, and other conditions that emerge from the aggregate, and are latent in the separate descriptions of devices. SEDS support automation of test procedures by providing nominal bounds for operating conditions and performance.
* SEDS enable algorithmic generation of the protocol stack for data communications between devices and applications. The effect is to generate device drivers for on-board software.
* During operation, it is common in current space flight technology for there to be a virtual channel through telemetry and telecommands between devices of a vehicle and a mission control system. The collection of SEDS for devices selected for a mission can be transformed to construct a mission control system database. SEDS can facilitate automatic generation of graphical user interface layouts to assure the appropriate presentation of data for human operators.
* As space vehicles become more autonomous, the interface descriptions in SEDS can be used by software on board vehicles in detecting and adapting to faults.

The background for the SEDS technology described here has developed from two communities of interest. One source was the United States Air Force Research Laboratory, where the focus of the community was to assemble space vehicles rapidly in response to tactically conceived missions. The current source is the Consultative Committee for Space Data Systems Spacecraft Onboard Interface Services working group, which produced this document, and in which the focus of the community is to enable algorithmic generation of the protocol stack for data communications between components. We expect additional contributions and improvements to come in the future from other communities.

# Introduction

## Purpose

The purpose of this document is to describe the concept and supporting rationale for the SOIS Electronic Data Sheets (SEDS) and Common Dictionary of Terms (DoT) to describe in an interchangeable format the data interfaces of Onboard Devices and Components, complying to the Spacecraft Onboard Interface Services (SOIS), developed by the Consultative Committee for Space Data Systems (CCSDS). This document acts as a handbook for the SEDS and Common DoT, providing a rationale, overview and worked examples for them.

This document is a CCSDS Informational Report and is therefore not to be taken as a CCSDS Recommended Standard.

## Scope

This document:

* provides an introduction and overview of the SEDS and DoT concepts upon which the detailed CCSDS SOIS recommendations are based;
* provides an overview of the business case for the use of SEDS and Common DoT;
* provides an end-to-end mission walkthrough illustrating the expected lifecycle of SEDS and where they may be used;
* provides worked examples;

## Applicability

This document is applicable for any user of SOIS EDS.

## Rationale

The Spacecraft Onboard Interface Services (SOIS) working group is looking at the On-board communications architectures and how would it be possible to maximize reuse and minimize component integration efforts. A key part of this activity is the definition of the Command and Data Acquisition Services (CDAS), which correspond to level 7 in the OSI communication protocol model. As that is above the layers where data can be simply treated as an abstract stream of bytes, implementing those services for a particular device requires knowledge of the layout, interpretation and sequencing of the underlying protocol data units to be transmitted to and from that device.

SOIS Electronic Data Sheets (SEDS) is the proposed standard by which to record such knowledge in a standardized form.

## Approach

A SOIS Electronic Data Sheet is a set of related XML files corresponding to the schema defined in references [3] and [4].

Such a datasheet can contain definitions of:

* The **interfaces** that allow two-way data interchange between layers of the OSI protocol stack.
* The **commands** and **parameters** that make up such interfaces.
* The **components** that make up the services that implement a mapping between two sets of interfaces.
* The **state machines**, **variables** and **activities** that make up such components.
* The **types**, **ranges**, **encodings** and **semantic terms** referenced by any of the above.

The intent of the schema is to be usable as an *interchange format* between a variety of tools that need access to the information it contains. Consequently, the design of the schema has been performed according to the following principles:

* Avoid ambiguity or open areas in the specification.
* Limit supported features to those known to be used by one or more real devices.
* Limit the number of different ways it is possible to represent any given device feature. (In addition to the stated goal of limiting the number of different ways a thing can be expressed there is sometimes an unstated value in having both a simple form of expression and a complex form of expression such that the coverage of the simple form is a subset of the coverage of the complex form.)
* Minimise the amount of external validation needed to detect logically inconsistent specifications.

## Terms and Definitions

With respect to service and protocol definition, SOIS, in general, uses terms and definitions defined within the ISO Open Systems Interconnect model defined in reference [9]. The following definitions are provided:

**component**: A logical element of a system accessed through defined communication interfaces. May be purely conceptual or realized in software or hardware (e.g., as a field-programmable gate array).

**Device Access Service, DAS** – The aggregation of the Device-Specific Access Protocols (DSAP) for each onboard device.

**device**:A physical element of a system accessed through subnetwork-layer interfaces. [D1]

**device abstraction control procedure, DACP**: The control procedure that provides the abstraction of a device-specific access protocol to a functional interface. This may involve e.g., the application of calibrations to raw values provided by the device or combination of multiple raw values to determine a derived value in SI units.

**device service:** A software artifact that can be generated from a device data sheet for execution in a project platform. A device service provides an application programming interface for application software to access an on-board device.

**device-specific access protocol, DSAP**: The protocol that makes use of a subnetwork service to command and/or acquire data from a device. This is specific to each device as no standardisation of access protocols at the device level exists.

**Device Virtualization Service, DVS**: The aggregation of Device Abstraction Control Procedures (DACP) for onboard devices.

**dictionary of terms, DoT**: Ontology of terms used to describe data in interfaces in electronic data sheets.

**electronic data sheet, EDS**: Electronic description of some details of a device, software component or standard. Unless qualified with the acronym “SOIS”, this term is general, referring to any machine-readable data sheet. See “SOIS Electronic Data Sheet, SEDS”.

**interface**: A facility provided or supplied by a component that allows exchange of data.

**packet**: Delimited octet-aligned data unit.

**protocol data unit; PDU**: A unit of data specified in a protocol and consisting of protocol control information and possibly user data.

**semantic attribute:** A semantic attribute is an attribute that constrains the usage of data. For example, the unit of measure of a number limits that number to usage that is cognizant of the unit of measure.

**Software bus:** An inter-Application message-based communications service.

**SOIS Electronic Data Sheet, SEDS**: Electronic description of a device’s metadata, device-specific functional and access interfaces, device-specific access protocol, and, optionally, device abstraction control procedure [3] compliant with SOIS standards. See “electronic data sheet, EDS”.

**subnetwork**: An abstraction of a collection of equipment and physical media, such as a local area network or a data bus, which forms an autonomous whole and can be used to interconnect real systems for the purpose of communication.

**virtual device**: A virtual version of a single physical device, exposing an idealised interface with a structured syntax and a simplified semantics and thus hiding the operation of the real device.

## References

The following documents are referenced in this Report. At the time of publication, the editions indicated were valid. All documents are subject to revision, and users of this Report are encouraged to investigate the possibility of applying the most recent editions of the documents indicated below. The CCSDS Secretariat maintains a register of currently valid CCSDS documents.

1. *Organization and Processes for the Consultative Committee for Space Data Systems*. CCSDS A02.1-Y-4. Yellow Book. Issue 4. Washington, D.C.: CCSDS, April 2014.
2. *Spacecraft Onboard Interface Services*. Informational Report, CCSDS 850.0-G-2. Green Book. Issue 2. Washington, D.C.: CCSDS, December 2013.
3. *Spacecraft Onboard Interface Services -- XML Specification for Electronic Data Sheets*. Recommended Standard, CCSDS 876.0-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, April 2019.
4. *Spacecraft Onboard Interface Services – Specification for Dictionary of Terms for Electronic Data Sheets*. Proposed Draft Recommended Practice (Magenta Book), CCSDS 876.1-R-0. Proposed Red Book. Issue 0. Washington, D.C.: CCSDS, March 2015.
5. *The Application of Security to CCSDS Protocols*. Informational Report, CCSDS 350.0-G-3. Green Book. Issue 3. Washington, D.C.: CCSDS, March 2019.
6. The Extensible Stylesheet Language Family (*XSL)*. https://www.w3.org/Style/XSL/.
7. *XML Essentials*. http://www.w3.org/standards/xml/core.
8. *W3C XML Schema Part 1: Structures Second Edition*. <http://www.w3.org/TR/xmlschema-1>.
9. *Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO/IEC 7498-1. 1994.
10. *Space Packet Protocol*. Recommended Standard, CCSDS 133.0-B2. Blue Book. Issue 2. June 2020.

# OVERVIEW

SOIS Electronic Data Sheets (SEDS) is a concept that has been proposed to allow the capture of the relevant information about a piece of equipment. This should capture the relevant aspects not just to enable an efficient exchange of information (easing its maintainability, enforcing consistency, etc.) but should also enable the development process to be partially automated.

Such automation can be expected to reduce not only development costs, but also revalidation costs associated with manual coding errors. Note that it is not expected or required to *eliminate* validation costs; the goal is a test campaign that passes on the first attempt, not a level of confidence that would justify skipping testing completely. In addition to direct cost savings, this approach should also naturally lead to reduced risk of schedule slips.

These savings and efficiencies can only be realised by the development of tooling that works across a range of devices; otherwise you have merely replaced the need to write code to support a specific device with a need to write a tool to support a specific device. Such device-independent tools have to be based on a standard for the representation of the information about the device that forms the input to the tool.

The SOIS EDS is such a *standardised interchange format*, developed for the specific needs of the space industry.

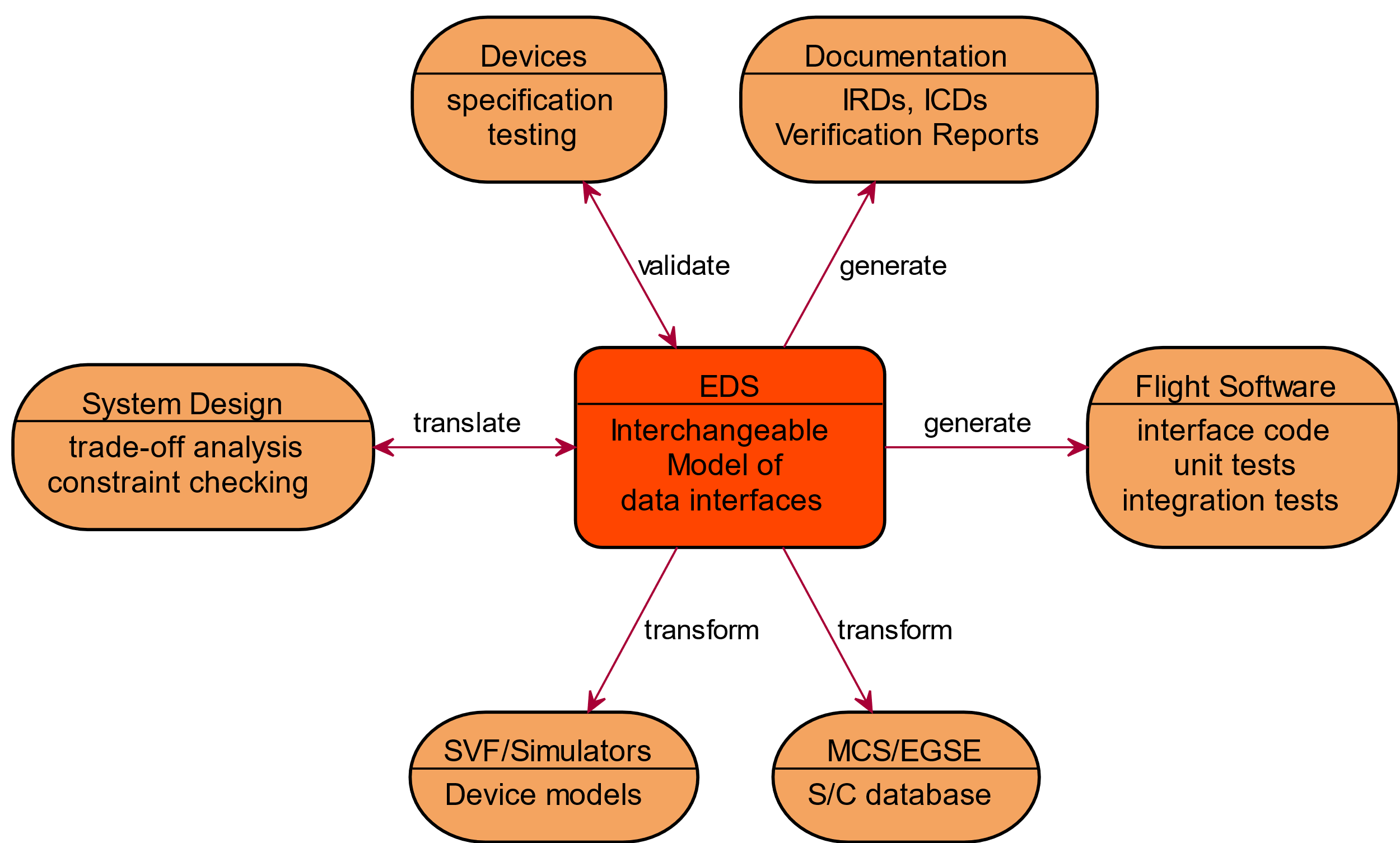


Figure 2‑1 SOIS EDS Scope

Due to the ubiquity of device data across the various activities of the mission lifecycle, the set of such possible tools is wide, as indicated in the diagram above. Consequently, if successful and widely adopted, the benefits will be seen across the mission lifecycle.

The sections that follow correspond to the parts of Figure 2‑1. The central rectangle represents an agency’s internal usage of SEDS. The surrounding rectangles represent the functional interoperability that can be achieved by using a software tool chain to generate, transform, translate, and validate artefacts that are described by SEDS. See ANNEX A for explanation of the acronyms in Figure 2‑1.

* Section 3 describes the internal mechanics of an agency using SEDS, which corresponds to the central rectangle in Figure 2‑1.
* Section 4 explains how SEDS can be used to describe devices.
* Section 5 explains how SEDS can be used to describe flight software components.
* Section 6 is about automation of some tasks in the system design process by using the machine-readable information in SEDS, including mission operations database (Section 6.2.4), simulation for software verification facility (Section 6.2.5), and generation of documentation (Section 6.2.8). Consistency of the mission configuration can be checked, for instance, verification that all finite state machine states are reachable; automatic review of the mission configuration to be sure that every part has the resources that it needs (Section 6.2.2). This includes integration testing, such as generation of tools for automatic recording and playback of data exchange and generation of tools for data sniffing (i.e. on Ethernet or serial lines) and automatic decoding.
* Section 7 briefly mentions some considerations for security when using SEDS.

# SEDS: Interchangeable Model of Data Interfaces

The first part of this section is a high-level summary of the manner in which SEDS models the data interfaces in a spacecraft. The second part describes how the model can be used inside a project.

## SEDS as a Model

This section is a summary of SEDS as a model of the data interfaces that compose a spacecraft communication architecture. Two fundamental concepts in the SEDS model are interfaces and components. A brief description of these concepts is provided here, and elaborated later in this document (Section 4.2).

* An interface describes the communication of data between parts of a spacecraft.
* A component describes a part of a spacecraft. A component may describe hardware (a device) or software (a service executing in an onboard computer). A component may provide interfaces, and it may require interfaces. A component may include a description of behaviour during communication. See Figure 3‑1.

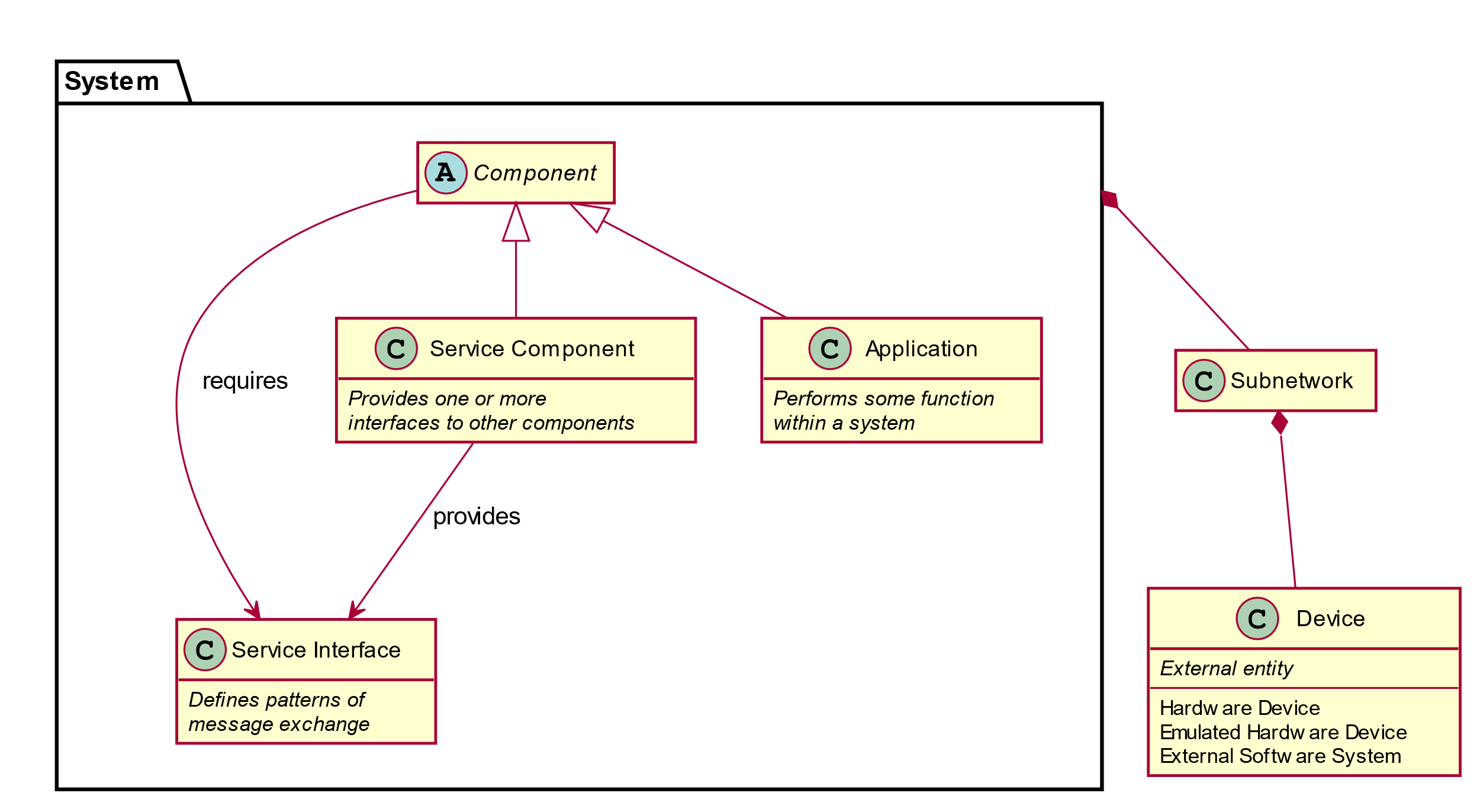


Figure 3‑1 A Concise Summary of Concepts for SEDS

It is important to note that SEDS does not impose a flight software architecture, but it facilitates interoperability of devices and services for any software architecture. This description uses concepts from the abstract architecture of SOIS, which map to the specific architecture used in a vehicle. SEDS also provides independence from platform architectures through the convention that unspecified encoding of data must be specified in the tool chain that generates software, while encoding of data can be optionally specified where it is needed, such as in communication.

### Spacecraft Onboard Interface Services

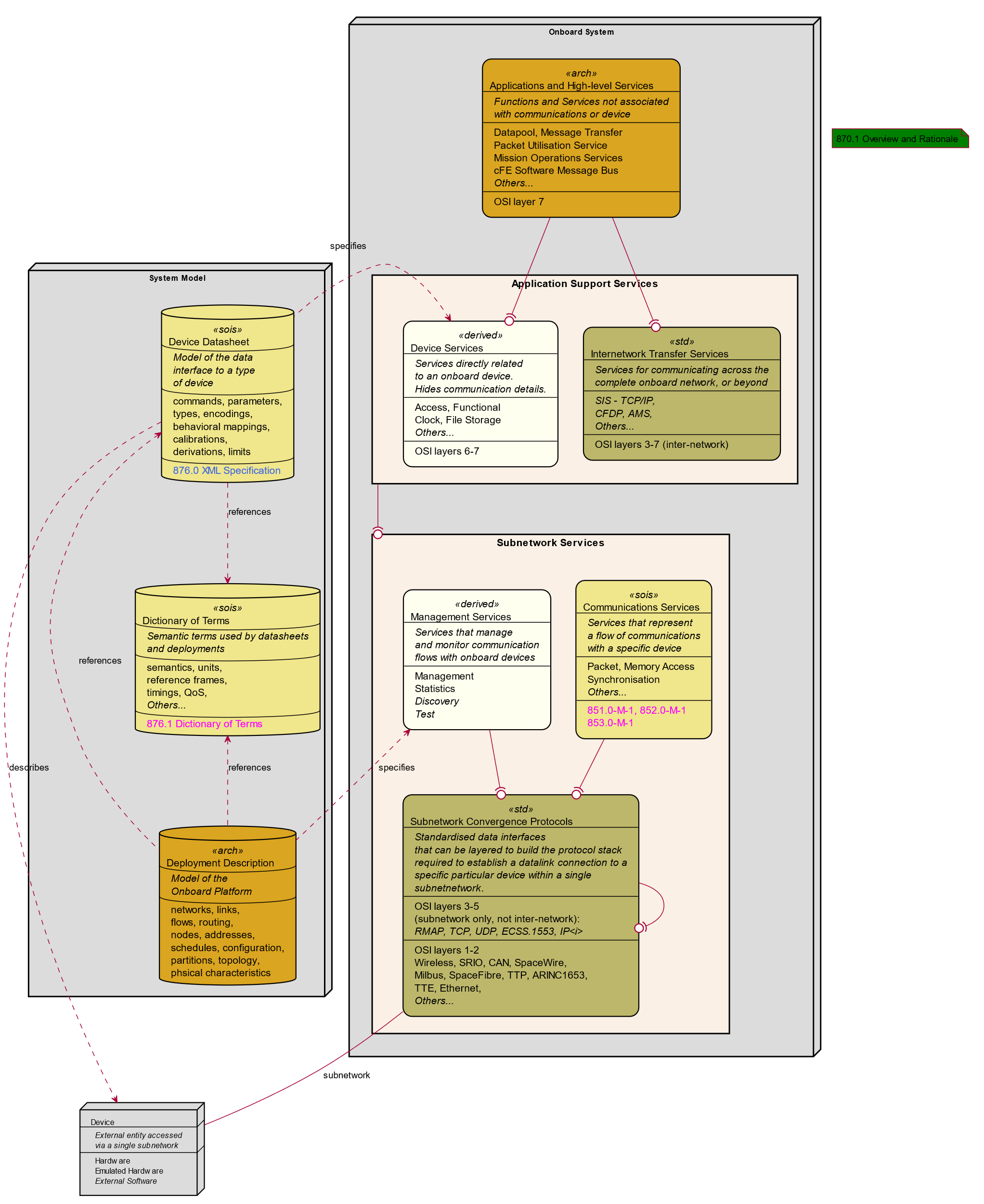


Figure 3‑2: SOIS Reference Communications Architecture

The right side of Figure 3‑2 illustrates the SOIS reference communication architecture, including the application support and subnetwork layer services. This is an architectural representation that must be mapped to physical communications. In essence, SOIS attempts to decompose the communications infrastructure inherent in all spacecraft into a defined set of services typically required by all missions. In general, SOIS does not define protocols; rather these are developed or adopted from commercial, CCSDS, or ECSS documents.

The left side of Figure 3‑2, and its relationship with the right side, is the subject of this book. The system model consists of device datasheets at the time of this writing, and can expand to include deployment descriptions in a future revision. A common dictionary of terms provides consistent terminology across the set of model artifacts and across projects. The device datasheets describe devices onboard a vehicle, and specify device services that present the data interfaces of the device for use by applications.

This architecture conforms to the layering principles of the OSI protocol model, in that each service is at a specific layer and communicates only with those immediately above or below it. The right side of Figure 3‑2, in the bottom compartment of relevant boxes, contains a mapping to the OSI protocol model; it is clear that SOIS factors the model differently than does OSI. The reason for the difference is adaptation of SOIS services to accommodate the variety of ways in which manufacturers of subnetworks package protocols in their products.

The role of the datasheet within SOIS is to provide those device-specific details, allowing a complete and machine-readable specification of a service.

The stereotypes in Figure 3‑2 classify the content of the boxes. Here is an explanation:

* <<sois>> indicates a SOIS specification.
* <<std>> indicates a standard from another source, which SOIS accommodates.
* <<arch>> represents parts specified by the design of a vehicle for a mission.
* <<derived>> indicates an artefact generated by a tool chain using the System Model.

### Datasheets in a Vehicle Network

This section describes the role of electronic data sheets in describing a network (Figure 3‑3).

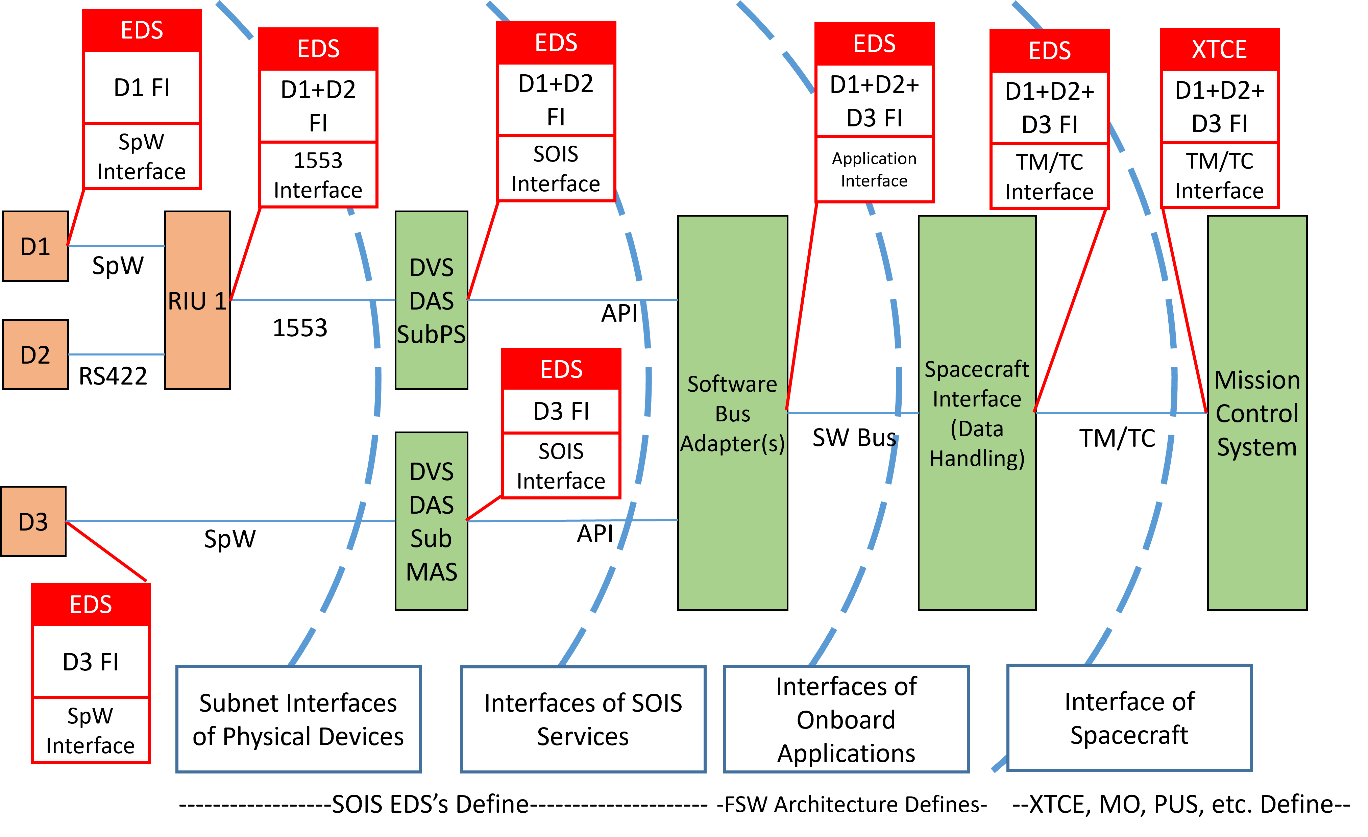


Figure 3‑3 Electronic Data Sheets in a Network

The left side of Figure 3‑3 shows devices and the electronic data sheets that describe them.

A remote interface unit (RIU) also appears near the left side of the diagram, where it acts as a concentrator for the messages of some devices. The SEDS for the RIU is minimally a composite of the SEDS instances for the devices whose messages it concentrates. The SEDS for the RIU may be more than the composite, because it may also describes how the RIU communicates on the bus (1553 in the figure) to the onboard computer, and how the RIU transforms or passes the messages of devices.

To the right of the devices depicted on the left side of the diagram is a layer for the SOIS protocol stack. It consists of subnet, DAS and DVS. DAS and DVS correspond to layers 6 (presentation) and 7 (application). The subnet maps to lower ISO layers. There is a set of SEDS instances that describe the protocol stack, one for each device on the left side of the diagram, including one for each RIU. These SEDS instances are the same as the device and RIU SEDS instances in the leftmost layer. The bottom partition of each SEDS box describes a DSAP, and the top describes an optional DACP; the “FI” stands for “functional interface” of the DACP.

Another feature in Figure 3‑3 is an optional software bus, near the center of the diagram. The SEDS instances labelled “SOIS App Interface” describe the adaptation of a device to communicate on the software bus. These optional adaptation SEDS describe a protocol which encapsulates packets on the software bus. There may be an adaptation SEDS for each device that communicates on the software bus. One side of an adaptation SEDS requires the DSAP and/or DACP interface of the device SEDS; the other side provides an interface to the software bus. There may also be a SEDS for each software application that communicates on the software bus. The adaptation SEDS for components may be implicit when no transformation of the component SEDS interface data is needed. For example, if PUS messages should populate the software bus, then adaptation SEDS are needed.

On the right of the software bus is a command and data handling function for the spacecraft. This function presents the set of SEDS instances of the devices in the network, including any applications that communicate on the optional software bus. Spacecraft command and data handling subscribes (TM) and publishes (TC) to onboard components and sends/receives to the link between the spacecraft and mission control.

A mission control center on the right side of the diagram can monitor and control a vehicle. The set of SEDS instances presented by the spacecraft at the link to mission control should be transformed by the tool chain into a standard language for description of communications between spacecraft and mission control centers, such as XTCE.

A variety of applications of SEDS have been described in this section. For example, there are SEDS for RIU’s, adaptation SEDS, and software SEDS, in addition to device SEDS. The SEDS can be used in these various applications because they can describe data interfaces, and they can describe abstract protocol behavior. In software SEDS, the description of behavior is typically omitted, but the description of interfaces is useful.

### The State of SEDS Technology

The working group that wrote this document has tested many of the concepts described in this document. Figure 3‑4 is a summary of the testing coverage at the time of writing.

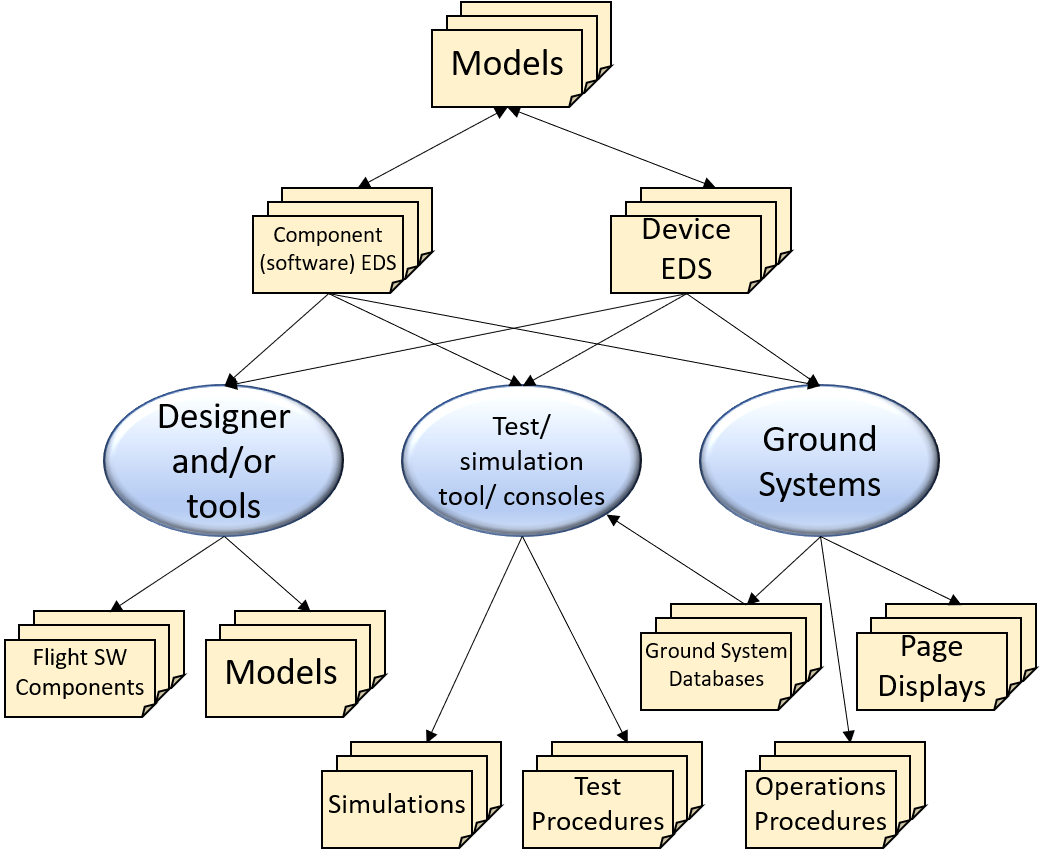


Figure 3‑4 Scope of SEDS Testing

Some of these scenarios have been tested, and are so indicated in this document where they are described in detail. Subsequent scenarios in this document indicate untested scenarios witb the word “proposed”. Testing all scenarios is out of scope for this document, so the untested scenarios appear in this document only to suggest the broad scope outlined in Figure 2‑1.

## Usage of SEDS Inside a Project

This section explains how to use SEDS within a project. A project accumulates a collection of SEDS instances that models the data interfaces inhe project. It is necessary to manage the collection of SEDS to maintain its truthful representation of the project. That management may include authoring instances of SEDS (Section 3.2.1), validating instances of SEDS (Section 3.2.2), and tracking versions of SEDS (Section 3.2.3).

### Authoring an Instance of SEDS

This section provides links to guidance for writing an instance of SEDS; the links refer to specific tasks described elsewhere in appropriate context in this document. This book provides examples for the following tasks of an author.

* The description of a device appears in an instance of SEDS. A “device” in this document is a part of a vehicle that communicates with applications in an onboard computer through a subnetwork that is onboard the vehicle. Some instances of SEDS may originate in a project, and others may come from an external manufacturer. When a project obtains a device built to its specifications, SEDS may serve as the interface control document. When designers seek ready-made instruments for a project, a collection of SEDS produced by manufacturers may serve as an index to the market, and the SEDS facilitates integration of instruments into the project. (Section 4.3)
* The description of a service appears in an instance of SEDS. A “service” in this document is a software component that executes in an onboard computer. (Section 5.2)
* The description of the parameters and constraints that constitute the architectural details of a vehicle appear in SEDS provided by the designers of the mission. This information tailors the behaviour of the tool chain that adapts the devices and services to communicate with one another in the vehicle. (Section 6.3.1)
* The description of the context of usage of a device or service is called metadata. The metadata can appear in any SEDS. (Section 6.3.2)
* The description of the topology, schedule, and other features of a subnetwork onboard a vehicle appear in a “deployment description” provided by the designers of the vehicle. Integration of this information with SEDS is a future objective. (Section 6.3.3)

### Validating an Instance of SEDS

This section describes the tasks that enable SEDS to be used by other projects and by other agencies. That capability results from adherence to a standard syntax and usage of standard terms for well-defined semantic concepts.

If a project has produced an instance of SEDS for use as an interface control document for a device to be provided by an external manufacturer, then validation of SEDS is important to assure clear communication of requirements. If a manufacturer has produced an instance of SEDS for a ready-made device, then validation of SEDS is important to assure correct usage of the device in customer projects. The following mechanisms facilitate validation of SEDS instances.

* A schema described in the SEDS blue book [3] makes it possible to check the syntax of a SEDS instance algorithmically. There are a number of validating XML readers in the software marketplace that can read the schema and validate an instance of SEDS, according to the standards for XML [7] and XML schemas [8].
* A dictionary of terms described in the DoT magenta book [4] provides a mapping between semantic concepts and standard terms for those concepts. The semantic concepts act as constraints on the usage of data. The standard mapping appears as an inclusion in the standard SEDS schema. The DoT magenta book includes a mechanism for extending the dictionary of terms to serve the immediate needs of a project. Over a longer period of time, the extensions used by various projects can be assimilated into the standard DoT part of the SEDS schema.
* The preceding items can be automated, but careful involvement of engineers is necessary to assure that the intended information is recorded in a SEDS instance, both by the authors and by subsequent reviews in the formal engineering process. See Section 6.2.8 for generation of human-readable documents for human reviewers.
* The preceding items provide a static validation of a SEDS instance; a dynamic validation is also possible to assure that a physical device operates as intended with the software that communicates with the device in an onboard computer. This dynamic validation is described in Section 6.2.5.

A SEDS instance may contain parameters that must be tailored for a specific project. Examples of these parameters appear in Section 6.3. Figure 3‑5 shows how a valid SEDS object may be obtained from a parameterized SEDS instance. The tailoring is done by a pre-processor that replaces named sites in the SEDS instance with text strings defined in metadata. Once all the parameters have been replaced, the instance can be read by a validating XML reader, which uses the standard SEDS schema, the standard DoT schema, and (optionally) a schema of local project terms.

Parameterized SEDS instance

Validity Report

SEDS schema

DoT schema

SEDS instance

SEDS object in tool chain

Project Terms schema

Project Paramaters in SEDS instance

Project Parameters in SEDS instance

Figure 3‑5 Static Validation of a SEDS Instance

Reading a collection of valid SEDS instances builds a set of objects in memory that can be used by tool chain software to implement the functions in Figure 2‑1.

Validation of SEDS may include issues that are internal to a SEDS instance (as described above) and issues that are mutual between provided and required interfaces of two SEDS instances (described below).

The relationship between a provided and required interface may be exact. In this case, the SEDS with the required interface can refer to the declaration of the provided interface in the SEDS where the interface is provided.

The relationship between a provided and required interface may be less exact in some use cases.

* For example, when searching a catalog of SEDS to select commodity components during design of a vehicle, an abstract declaration of a required interface could be used as a search argument. When a selection has been made, human engineers or an automated tool chain can generate any shim that may be needed to adapt units of measure.
* Another use case is where the worst latency in communication is expected to be less than any deadlines specified in the component that provides or requires the interface. This use case can be used to eliminate device candidates that cannot support a control loop.

### References within a SEDS Library

This section describes a way to manage a library of SEDS. Two issues are important here: One is resolution of references from one SEDS instance to another. The other issue is applying version information to SEDS instances as the product that they represent evolves.

Two kinds of references are possible between SEDS instances. One uses xInclude to identify the SEDS instance to be used to resolve references. The other implicitly resolves references by using files associated by proximity in a file system.

A SEDS instance produced by a manufacturer may contain xInclude references to any standard SEDS instances that are available in SANA. Within the tool chain for a project, a collection of SEDS instances may be placed together in a file system directory, and the tool chain may resolve implicit references to definitions in other SEDS instances in the same directory. An example of an xInclude reference appears in Section 4.3.1.

References do not have a concept of versions. Versions of referenced files should be managed externally.

# Devices

The description of a device in a SEDS enables the device to be used in a variety of software architectures. An architecture-specific tool chain transforms the SEDS content into device services that are appropriate to the software architecture used in a project. This section describes how those transformations can be performed.

Subsection 4.1 reviews some basic scenarios for devices.

Subsection 4.2 describes concepts for description of devices in SEDS.

Subsection 4.3 explains some examples of SEDS data sheets.

## Scenarios for Devices

Two scenarios have been identified for the use of SEDS that describe devices. The first scenario enables the usage of a device in multiple software architectures. The second scenario enables docking of vehicles into a single assembly that shares some of the devices of each of the vehicles.

### Device Interoperability Across Projects

Status: Tested for one project; proposed for a second project using the same SEDS.

See Figure 4‑1. One of the first use cases of SEDS was to generate a service to represent a device in an onboard computer. If one agency can generate a service to represent the device in their onboard computing architecture, then it is expected that another agency can generate a similar service for an entirely different onboard computing architecture. For example, in the architecture of one agency, applications might communicate with the device service by means of application programming interface (API) calls. In the architecture of the other agency, applications might communicate with the device service by means of a software message bus.

Device x

EDS for Device x

Device Service for x on A

Device Service for x on Z

Vehicle A

FSW for Vehicle A

Vehicle Z

FSW for Vehicle Z

Device x

generation flow

communication

Figure 4‑1 Device Interoperability across Projects

### Interoperability for Docking Vehicles

Status: Proposed.

See Figure 4‑2. This scenario combines the device and service interoperability in the preceding examples in order to facilitate docking vehicles. When docked, the two vehicles may need to share some functions, such as attitude control actuators or a service that provides the network address and orientation of attitude control actuators.

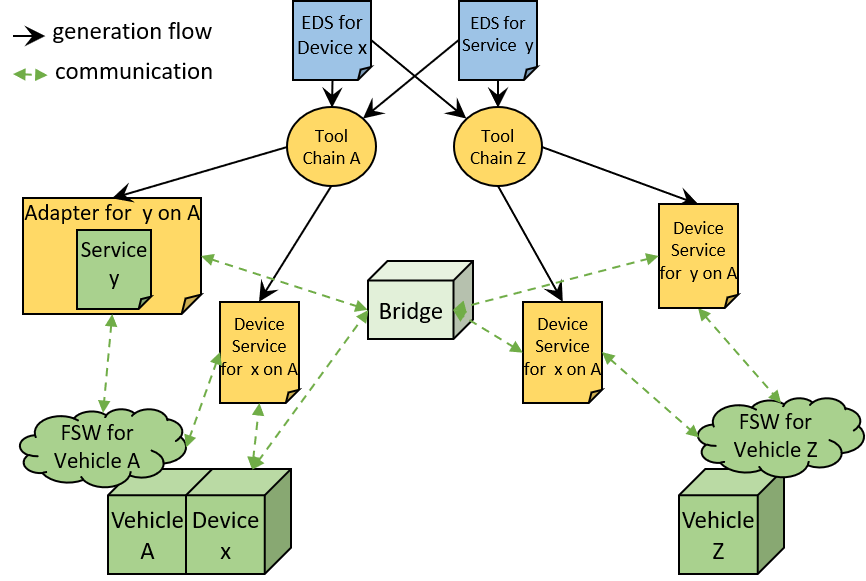


Figure 4‑2 Interoperability for Docking Vehicles

## Device Data Sheet Concepts

In order to describe a device in SEDS, it is first necessary to understand some of concepts that organize the description.

Subsection 4.2.1 defines the structural parts of a device description.

Subsection 4.2.2 decomposes interfaces into their primitive parts.

Subsection 4.2.3 describes types of data that can flow across an interface.

Subsection 4.2.4 describes an abstraction for the usage of different formats of messages.

Subsection 4.2.5 describes abstractions for the behavior of a device service.

### Device Components and Interfaces

A SOIS EDS Datasheet is a package that can contain definitions of:

* **Interfaces** that allow two-way data interchange within the scope of a single device.
* **Components** that map a set of *provided* interfaces to a set of *required* interfaces.

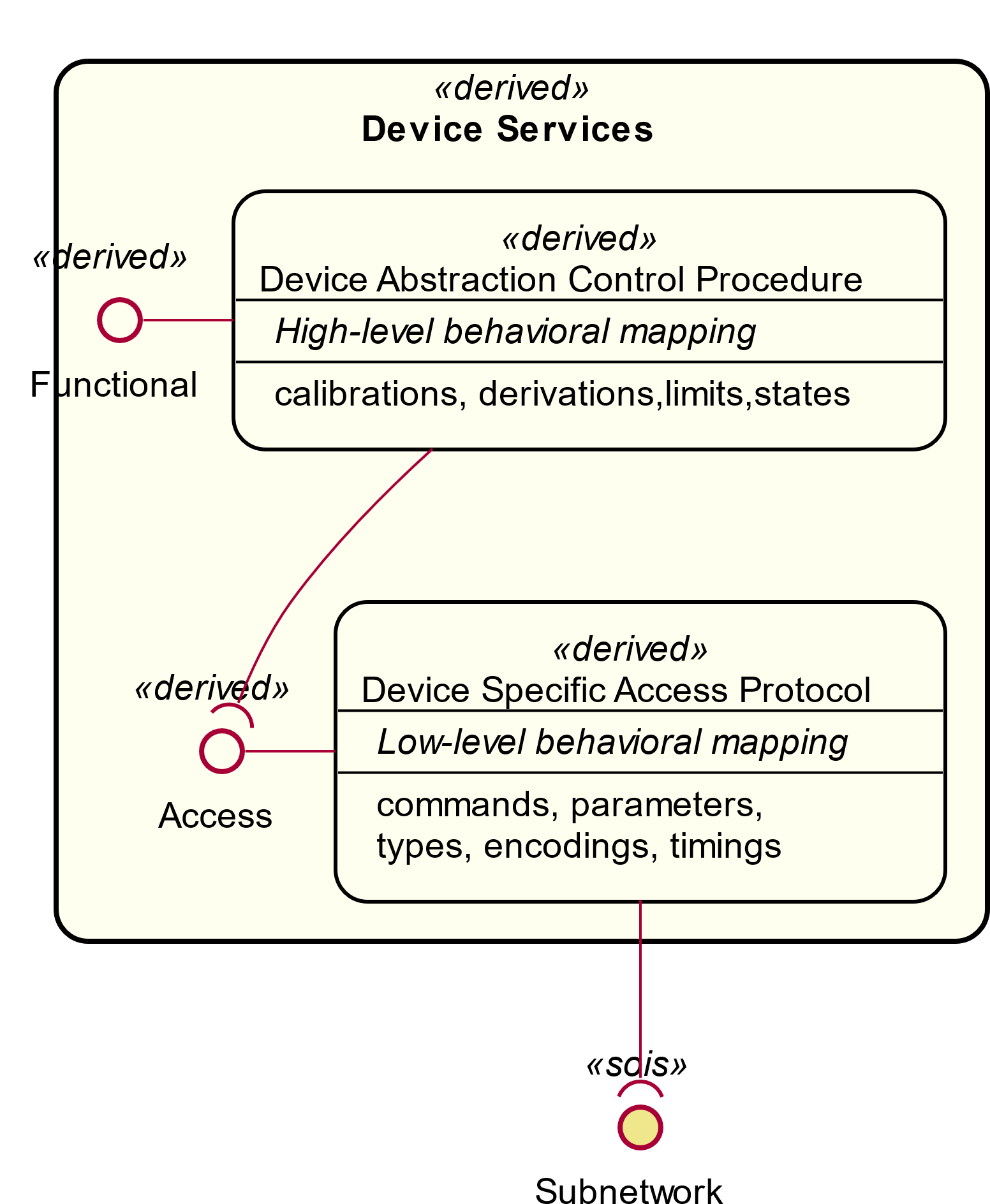


Figure 4‑3 SOIS EDS Device Datasheet Contents

A datasheet for a device can specify two interfaces:

* The **Device-Specific Access Interface**; the lowest-level access to all raw decoded data transmitted to and from the device.
* The **Device-Specific Functional Interface**; higher-level access to calibrated or derived data.

Both of these interfaces are device-specific because different devices support different sets of data. These are split to allow missions the option of supporting only one or both. The access interface will always be needed onboard the vehicle, while the functional interface may be used by other onboard applications or mission control. In the case of a smart device the functional interface may be the only interface provided, in which case the functional interface is needed onboard.

The access interface presents and accepts data in a format suitable for an application programming interface in the flight computer. The Device Specific Access Protocol transforms data between the access interface and the encoded bit stream on the subnetwork.

In the typical case, there will be a single component providing each interface, and the component implementing the higher-level interface will be defined in terms of the lower-level one. The lowest-level component will require one or more subnetwork-level interfaces.

A multi-functional device may have multiple interfaces for those functions. For example a GPS receiver may often provide a clock function in addition to its position function.

In some cases it may be convenient to describe a device in terms of subcomponents and the interfaces between those subcomponents.

In general, redundant hardware would be described by more than one SEDS instance and managed by a separate FDIR agent or service.

### Interfaces and PrimItives

Rather than supporting a fixed set of known interfaces, the SOIS EDS schema has an explicit construct for specifying arbitrary interfaces which can be used for both standard and device-specific interfaces.

Within a SOIS EDS datasheet, an **interface** is defined in terms of the **commands** and **parameters** that make it up. Commands have **arguments**, which can be either input, output, inout or notify. A parameter is a kind of message that represents an item of telemetered data. A read-only parameter can be considered to be a shorthand for a command with a single output argument, avoiding the need to provide two names (e.g. Value and getValue). A non-read-only parameter is equivalent to a pair of single-argument get and set commands.

The data transfer associated with an interface parameter, or the invocation of a command, is modeled using the basic OSI service **primitives** which are used through the SOIS standards. For example, synchronous acquisition (polling) of the value of a parameter on an interface required by a component is modeled as the transmission of a parameter get operation **request primitive** to that interface. At some point later the component requiring the interface will receive a get operation **indication primitive** which delivers the parameter value.

Table 1 Mapping between parameters, commands and primitives

|  |  |  |  |
| --- | --- | --- | --- |
| **Interface Element** | **Options** | **Argument Modes** | **Primitive** |
| Parameter | **sync** |  | request  indication |
| **async** |  | indication |
| Command | **sync** | No **notify** arguments | request  indication |
| At least one **notify** argument | request notify  indication |
| **async** | No **notify**, **out,** or **inout** arguments | request |
| At least one **notify** argument, no **out** or **inout** arguments | request  notify |
| **inout**, or both **in** and **out** | illegal |
| No **notify**, **in,** or **inout** arguments | indication |

The above table shows the mapping between the interface definition and the request and indication primitives that pass across the interface.

Where both a request and indication exist, that parameter or command may be treated as transactional, in that the underlying service layers are responsible for associating the two together, or a state machine must be defined to detect the association, as shown in Figure 4‑4. Often, this treatment as a transaction is unnecessary, when any instance of the indication is sufficient for proper interpretation of the state of a device.

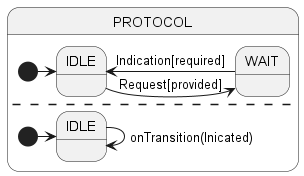


Figure 4‑4 Detecting a transaction indication

The patterns for messages across an interface appear in Figure 4‑4 and Figure 4‑5.



Figure 4‑5 Parameter Interaction Patterns Described by SEDS

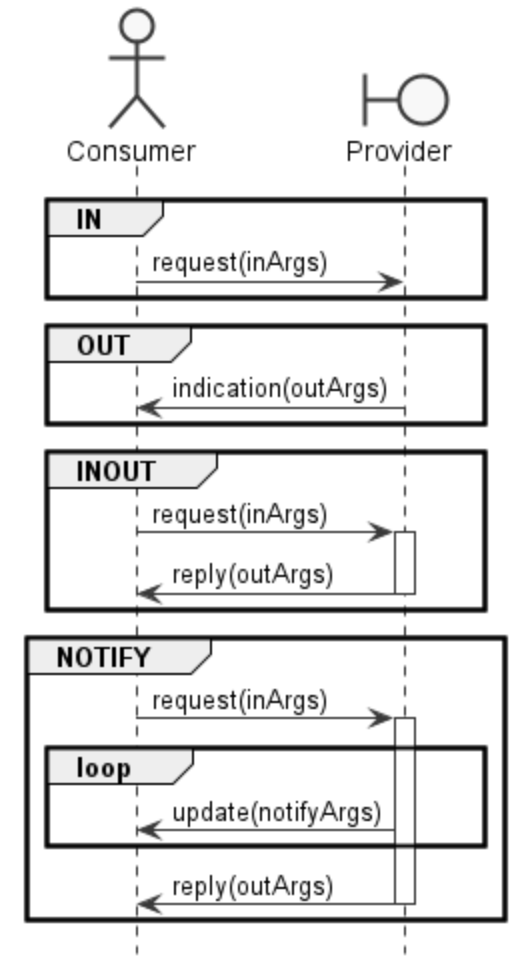


Figure 4‑6 Command Interaction Patterns Described by SEDS

An interface may transfer data at different levels in the architectural layers defined by SOIS. The following list is an informal summary of these layers; see the SEDS Blue Book [3] for a formal definition.

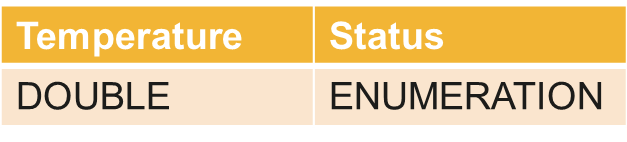
* **Application**: An interface exposed by application software
* **Functional**: An interface containing refined data according to the SOIS device virtualization concept. An example of refinement is calibration to engineering units. Not all vehicles will implement this kind of interface in flight, but at least the mission control center must be able to interpret data in this way.
* **Access**: An interface containing raw data of a device, provided by a device service.
* **Subnetwork**: An interface such as the SOIS Subnet Packet Service or the SOIS Subnet Memory Access Service.
* **Physical**: A physical interface, characterized by pins, voltages, etc.
* **Environment**: An interface between a device and its environment may be used to describe simulation or hardware-in-the-loop testing of a device. It may also describe static properties of a device, such as mass, storage temperature ranges, etc.

### Data Types

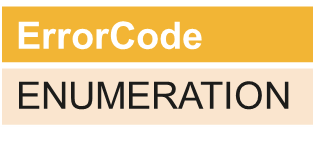
The parameters and command arguments of an interface are defined in terms of named **types**, which can have a **range**, **encoding** and **semantics** (see [3]).

* Range is the basic property of numeric data types.
* Encoding specifies the format of data that is transmitted across a data link.
* Semantics describe the meaning or usage of data.

These types can be simple (Boolean, integer, float, enumerated, etc.) or aggregate (fixed and variable-length arrays, containers). Containers can be **abstract**, in which case they are placeholders for any of a known set of concrete types that share a common header that can be used to classify them.



Packet ID = 3



Packet ID = 4

Figure 4‑7 Using values specified by an abstract container to determine the concrete type

Numeric types can have numeric ranges; enumeration types can have ranges that are arbitrary subsets of the full set of values.

Specification of details of encoding, such as byteOrder, is only necessary for values that will be transmitted as data units on an interface. This is modelled by specifying conversion to and from a sequence of bytes. These byte sequences are the serial forms used in transmitting data.

Arrays have one or more dimensions. The following example shows an array with two dimensions. The elements of the array will appear in the following order in memory and in transmission: (0,0), (0,1), (0,2), (1,0), (1,1), (1,2). The last dimension is least significant.

<ArrayDimensions>

<Dimension size="2"/>

<Dimension size="3"/>

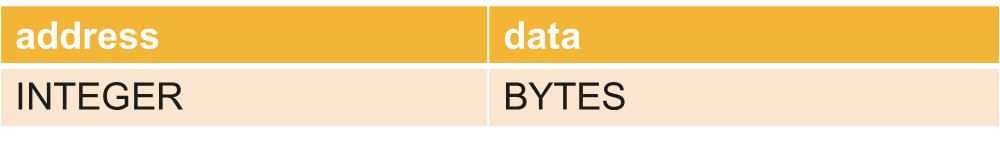
</ArrayDimensions>

Data types are also used to define entries within a container and arguments to activities; see later sections.

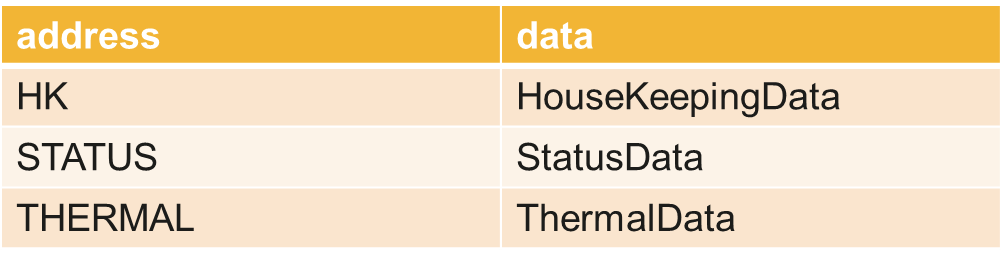
### Generic Types and Interface Usages

When a datasheet references a standardised or otherwise externally-specified interface definition, it is common for the types used to *define* that interface to not be the most appropriate ones to *describe its usage*. For example, a generic interface may allow arbitrary integers, whereas within the context of a specific device, only a fixed set of enumeration values are used. See Section 4.3.2, for example. Similarly, a generic interface such as the Subnetwork Packet Service may be defined in terms of arbitrary byte arrays, but a particular device will only support a fixed set of packets, i.e. an abstract container as above. See Section 4.3.7.

In such cases, the **usage** of the interface specification by the datasheet component can be accompanied by a **generic type mapping**.



Example command on generic interface



Same command on interface usage within datasheet

Figure 4‑8 Using a type mapping to capture the relation between command arguments

In the above diagram, the first command expresses the idea that ‘if you read from an address, you will get some data back’. The second specifies the same abstract interface in terms of its support by a specific device, i.e. ‘if you read from one of these 3 supported addresses, you will get the corresponding known set of data back’. See Section 4.3.2 for a detailed example.

This is possible using generic type mappings to replace the arbitrary address with a specific enumeration type, and then setting up a set of **alternate mappings** for the data type corresponding to each legal value for the address. See Section 4.3.7.2 for an example.

Note that the data type mapped to can itself be abstract, representing variably-structured data that will still need to be classified in order to be fully processed.

### Component Implementation

The implementation of a component can be specified within the datasheet in terms of:

* A set of **state machines** where the triggering events are incoming primitives on the component’s provided and required interfaces.
* A set of **variables** that can hold data values over time.
* A set of **activities** that can be referenced by state machine transitions, perform basic calculations (including **calibration**) and initiate outgoing primitives on the component’s provided and required interfaces



Figure 4‑9 Sample component implemented by state machines

The state machine concept used is taken directly from UML: a [directed](http://en.wikipedia.org/wiki/State_diagram#Directed_graph) graph in which nodes denote **states** and connectors denote state **transitions** protected by **guards**. Guards define the conditions for transitions to occur. See Section 4.3.3 for an example of the use of guards.

### Protocol Specification

Figure 4‑4 and Figure 4‑5 describe some simple protocols for parameters and commands that are defined by <Parameter> and <Command> elements in an interface. A state machine may be used to describe more complicated protocols. This section provides some examples for commands.

A state machine is associated with a command x by having a transition OnCommandPrimitive or by having a reference to an activity with SendCommandPrimitive that refers to the command. The examples that follow are for device driver implementations between two interfaces. Interface A is an access level interface, and interface B is a subnetwork level interface that communicates with a device.

The declaration of the subnetwork interface type appears below. This declaration is in file ccsds.sois.subnetwork.xml on SANA.

<Interface name="PSInterfaceType" level="subnetwork"

shortDescription="SOIS SUBNETWORK PACKET SERVICE from 851.0-M-1">

<GenericTypeSet>

<GenericType name="SendDataType" />

<GenericType name="ReceiveDataType" />

<GenericType name="FailureMetadataType" />

</GenericTypeSet>

<CommandSet>

<Command name="send" mode="async"

shortDescription="PACKET\_SEND.request primitive only">

<Argument name="data" type="SendDataType" mode="in"

dataUnit="true" />

</Command>

<Command name="receive" mode="async"

shortDescription="PACKET\_RECEIVE.indication primitive only">

<Argument name="data" type="ReceiveDataType" mode="out"

dataUnit="true" />

</Command>

<Command name="failure" mode="async"

shortDescription="PACKET\_FAILURE.indication primitive only">

<Argument name="failureMetadata" type="FailureMetadataType" mode="out"/>

</Command>

</CommandSet>

</Interface>

The required subnetwork interface in the component is defined as having name “B” and using the PDU base type of the examples.

<RequiredInterfaceSet>

<Interface name="B" type="CCSDS/SOIS/Subnetwork/PSInterfaceType">

<GenericTypeMapSet>

<GenericTypeMap name="ReceiveDataType" type="TC"/>

<GenericTypeMap name="SendDataType" type="TC"/>

</GenericTypeMapSet>

</Interface>

</RequiredInterfaceSet>

The protocol data units are simplified to consist of just a command code, inheriting from the following base type.

<ContainerDataType name="TC">

<EntryList>

<Entry name="cc" type="CCSDS/SOIS/SEDS/UINT8"/>

</EntryList>

</ContainerDataType>

A few variables are defined in the device driver component in order to pass the PDU’s through. In a practical implementation the component might alter the PDU’s in passing.

<VariableSet>

<Variable name="x" type="TC"/>

<Variable name="timeoutPDU" type="cmdSRt"/>

</VariableSet>

#### Send and Forget

This example defines the command labelled “IN” in Figure 4‑5. The PDU for this command is the following.

<ContainerDataType name="cmdSF" baseType="TC">

<ConstraintSet>

<ValueConstraint entry="cc" value="1"/>

</ConstraintSet>

</ContainerDataType>

The command definition in the declared interface A is as follows.

<Command name="aSF" shortDescription="send and forget">

<Argument name="aIn" type="cmdSF" mode="in" dataUnit="true"/>

</Command>

The activity “send” is defined in the component to deliver a PDU to the subnetwork interface from component variable named “x”. This activity is explicit here to explain the convention in Figure 4‑5, but can be implicit (omitted) in an actual SEDS instance.

<Activity name="send">

<Body>

<SendCommandPrimitive interface="B" command="send">

<ArgumentValue name="data">

<VariableRef variableRef="x"/>

</ArgumentValue>

</SendCommandPrimitive>

</Body>

</Activity>

The state machine “SF” is defined in the component to respond to the arrival of command “aSF” from interface “A”. This state machine is explicit here to explain the convention in Figure 4‑5, but can be implicit (omitted) in an actual SEDS instance.

<StateMachine name="SF">

<State name="Idle"/>

<Transition name="Send" fromState="Idle" toState="Idle">

<OnCommandPrimitive interface="A" command="aSF">

<ArgumentValue name="aIn" outputVariableRef="x"/>

</OnCommandPrimitive>

<Do activity="send" />

</Transition>

</StateMachine>

The state machine consists of a single state and a single transition from and to that state. The transition stores the PDU in variable “x” and then invokes the activity “send”. A diagram for this state machine appears in Figure 4‑10.

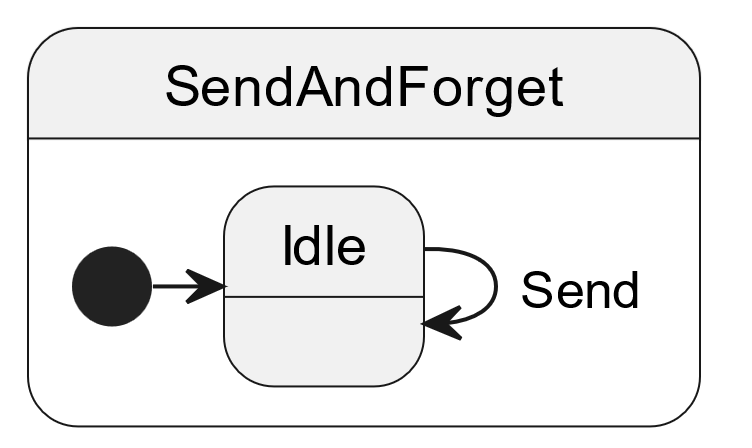


Figure 4‑10 State Machine for Send-and Forget Protocol

#### Receive Indication

This example defines the command labelled “OUT” in Figure 4‑5. The PDU for this command is the following.

<ContainerDataType name="cmdRI" baseType="TC">

<ConstraintSet>

<ValueConstraint entry="cc" value="2"/>

</ConstraintSet>

</ContainerDataType>

The command definition in the declared interface A is as follows.

<Command name="aRI" shortDescription="receive indication">

<Argument name="aOut" type="cmdRI" mode="out" dataUnit="true"/>

</Command>

The activity “receive” is defined in the component to deliver a PDU to the interface “A” command variable “aOut” from component variable named “x”. This activity is explicit here to explain the convention in Figure 4‑5, but can be implicit (omitted) in an actual SEDS instance.

<Activity name="receive">

<Body>

<SendCommandPrimitive interface="A" command="aRI">

<ArgumentValue name="aOut">

<VariableRef variableRef="x"/>

</ArgumentValue>

</SendCommandPrimitive>

</Body>

</Activity>

The state machine “RI” is defined in the component to respond to the arrival of indication “data” from interface “B”. This state machine is explicit here to explain the convention in Figure 4‑5, but can be implicit (omitted) in an actual SEDS instance.

<StateMachine name="RI">

<State name="Idle"/>

<Transition name="Receive" fromState="Idle" toState="Idle">

<OnCommandPrimitive interface="B" command="receive">

<ArgumentValue name="data" outputVariableRef="x"/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef="data"/>

<TypeOperand>cmdRI</TypeOperand>

</TypeCondition>

</Guard>

<Do activity="receive" />

</Transition>

</StateMachine>

The state machine consists of a single state and a single transition from and to that state. The transition stores the PDU in variable “x” and then invokes the activity “receive”. The guard in the transition causes it to ignore PDU’s that have the wrong type. ”. A diagram for this state machine appears in Figure 4‑11.

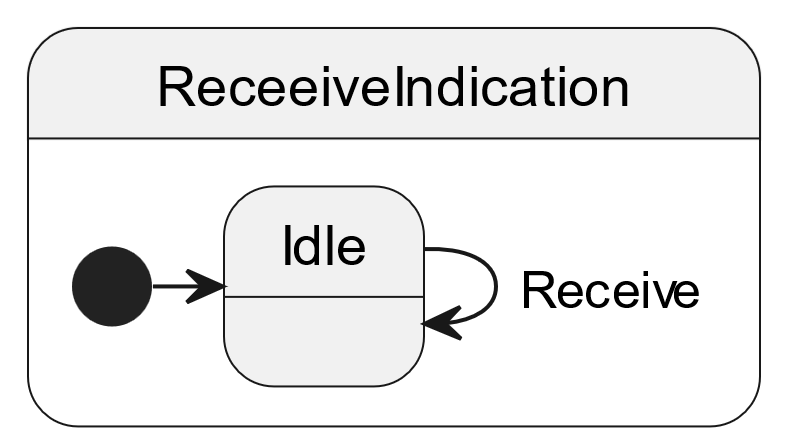


Figure 4‑11 State Machine for Receive-Indication Protocol

#### Send with Reply

This example defines the command labelled “INOUT” in Figure 4‑5. The PDU’s for this command are the following.

<ContainerDataType name="cmdSRs" baseType="TC">

<ConstraintSet>

<ValueConstraint entry="cc" value="3"/>

</ConstraintSet>

</ContainerDataType>

<ContainerDataType name="cmdSRr" baseType="TC">

<ConstraintSet>

<ValueConstraint entry="cc" value="4"/>

</ConstraintSet>

</ContainerDataType>

The command definition in the declared interface A is as follows.

<Command name="aSR" shortDescription="send with reply">

<Argument name="aIn" type="cmdSRs" mode="in" dataUnit="true"/>

<Argument name="aOut" type="cmdSRr" mode="out" dataUnit="true"/>

</Command>

The activity “receiveReply” is defined in the component to deliver a PDU to the interface “A” command variable “aOut” from component variable named “x”. This activity is explicit here to explain the convention in Figure 4‑5, but can be implicit (omitted) in an actual SEDS instance. This example also uses activity “send” in a previous example.

<Activity name="receiveReply">

<Body>

<SendCommandPrimitive interface="A" command="aSR">

<ArgumentValue name="aOut">

<VariableRef variableRef="x"/>

</ArgumentValue>

</SendCommandPrimitive>

</Body>

</Activity>

The state machine “SR” is defined in the component to respond to the arrival of command “aSR” from interface “A” and to respond to the guarded arrival of indication “data” from interface “B”. This state machine is explicit here to explain the convention in Figure 4‑5, but can be implicit (omitted) in an actual SEDS instance.

<StateMachine name="SR">

<State name="Idle"/>

<State name="Communicating"/>

<Transition name="Send" fromState="Idle" toState="Communicating">

<OnCommandPrimitive interface="A" command="aSR">

<ArgumentValue name="aIn" outputVariableRef="x"/>

</OnCommandPrimitive>

<Do activity="send" />

</Transition>

<Transition name="Receive" fromState="Communicating" toState="Idle">

<OnCommandPrimitive interface="B" command="receive">

<ArgumentValue name="data" outputVariableRef="x"/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef="data"/>

<TypeOperand>cmdSRr</TypeOperand>

</TypeCondition>

</Guard>

<Do activity="receiveReply" />

</Transition>

</StateMachine>

The state machine consists of two states and two transitions between those states. The transition “send” stores the PDU in variable “x” and then invokes the activity “receive” from a previous example. The transition “Receive” stores the PDU in variable “x” and then invokes the activity “receiveReply”. The guard in the transition causes it to ignore PDU’s that have the wrong type. A diagram for this state machine appears in Figure 4‑12.

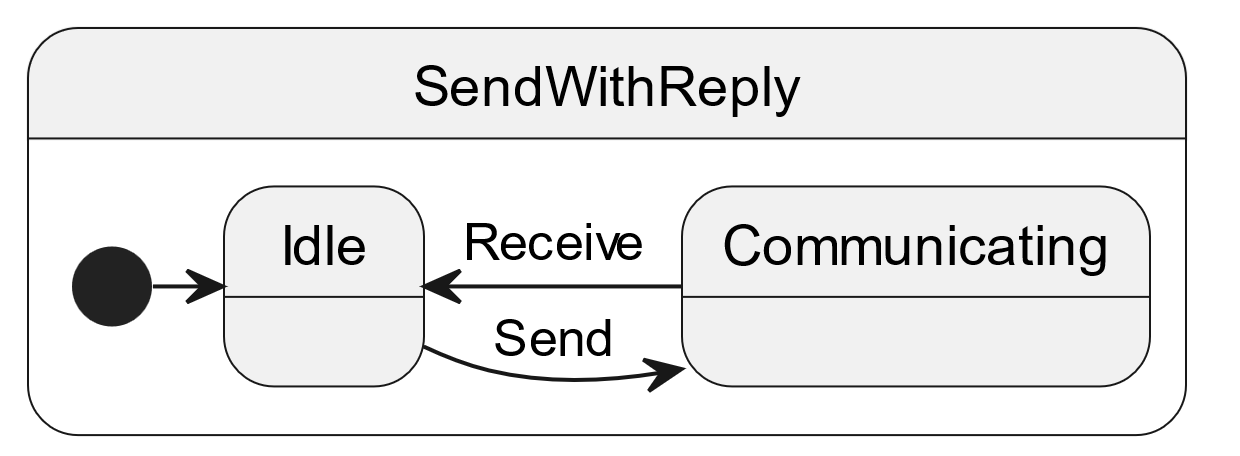


Figure 4‑12 State Machine for Send With Reply Protocol

#### Send with Notifications and Final Reply

This example defines the command labelled “NOTIFY” in Figure 4‑5. The PDU’s for this command are the following.

<ContainerDataType name="cmdSNRs" baseType="TC">

<ConstraintSet>

<ValueConstraint entry="cc" value="5"/>

</ConstraintSet>

</ContainerDataType>

<ContainerDataType name="cmdSNRn" baseType="TC">

<ConstraintSet>

<RangeConstraint entry="cc">

<MinMaxRange rangeType="inclusiveMinInclusiveMax" min="6" max="7"/>

</RangeConstraint>

</ConstraintSet>

</ContainerDataType>

<ContainerDataType name="cmdSNRr" baseType="TC">

<ConstraintSet>

<ValueConstraint entry="cc" value="8"/>

</ConstraintSet>

</ContainerDataType>

The command definition in the declared interface A is as follows.

<Command name="aSNR" shortDescription="send with notifications and reply">

<Argument name="aIn" type="cmdSNRs" mode="in" dataUnit="true"/>

<Argument name="aNotify" type="cmdSNRn" mode="notify" dataUnit="true"/>

<Argument name="aOut" type="cmdSNRr" mode="out" dataUnit="true"/>

</Command>

The activity “receiveNotify” is defined in the component to deliver a notification PDU to the interface “A” command variable “aNotify” from component variable named “x”. The activity “receiveNotifyReply” is defined in the component to deliver a final reply PDU to the interface “A” command variable “aOut” from component variable named “x”. These activities are explicit here to explain the convention in Figure 4‑5, but can be implicit (omitted) in an actual SEDS instance. This example also uses activity “send” in a previous example.

<Activity name="receiveNotify">

<Body>

<SendCommandPrimitive interface="A" command="aSNR">

<ArgumentValue name="aNotify">

<VariableRef variableRef="x"/>

</ArgumentValue>

</SendCommandPrimitive>

</Body>

</Activity>

<Activity name="receiveNotifyReply">

<Body>

<SendCommandPrimitive interface="A" command="aSNR">

<ArgumentValue name="aOut">

<VariableRef variableRef="x"/>

</ArgumentValue>

</SendCommandPrimitive>

</Body>

</Activity>

The state machine “SNR” is defined in the component to respond to the arrival of command “aSNR” from interface “A” and to respond to the guarded arrival of indication “data” from interface “B”. This state machine is explicit here to explain the convention in Figure 4‑5, but can be implicit (omitted) in an actual SEDS instance.

<StateMachine name="SNR">

<State name="Idle"/>

<State name="Communicating"/>

<Transition name="Send" fromState="Idle" toState="Communicating">

<OnCommandPrimitive interface="A" command="aSNR">

<ArgumentValue name="aIn" outputVariableRef="x"/>

</OnCommandPrimitive>

<Do activity="send" />

</Transition>

<Transition name="ReceiveNotify" fromState="Communicating" toState="Communicating">

<OnCommandPrimitive interface="B" command="receive">

<ArgumentValue name="data" outputVariableRef="x"/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef="data"/>

<TypeOperand>cmdSNRn</TypeOperand>

</TypeCondition>

</Guard>

<Do activity="receiveNotify" />

</Transition>

<Transition name="Receive" fromState="Communicating" toState="Idle">

<OnCommandPrimitive interface="B" command="receive">

<ArgumentValue name="data" outputVariableRef="x"/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef="data"/>

<TypeOperand>cmdSNRr</TypeOperand>

</TypeCondition>

</Guard>

<Do activity="receiveNotifyReply" />

</Transition>

</StateMachine>

The state machine consists of two states and three transitions between those states. The transition “Send” stores the PDU in variable “x” and then invokes the activity “send” from a previous example. The transition “Receive” stores the PDU in variable “x” and then invokes the activity “receiveNotifyReply”. The guard in the transition causes it to ignore PDU’s that have the wrong type. The transition “ReceiveNotify” stores the PDU in variable “x” and then invokes the activity “receiveNotify”. The “ReceiveNotify” transition has a guard for the type of the notification PDU’s. A diagram for this state machine appears in Figure 4‑13.

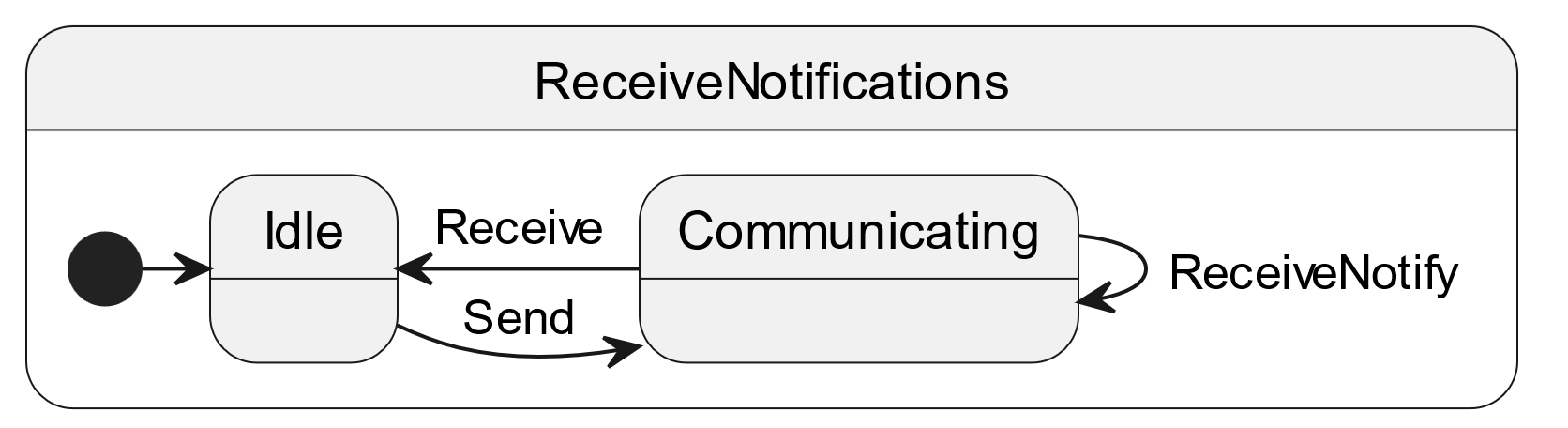


Figure 4‑13 State Machine for Notifications Protocol

#### Send with Reply Ignoring Timeout

This example defines an explicit extension to the command labelled “INOUT” in Figure 4‑5. The PDU’s for this command are the same as in Section 4.2.6.3.

The command definition in the declared interface A is as follows.

<Command name="aSRIT" shortDescription="send with reply, ignoring timeout">

<Argument name="aIn" type="cmdSRs" mode="in" dataUnit="true"/>

<Argument name="aOut" type="cmdSRr" mode="out" dataUnit="true"/>

</Command>

This example uses activities “send” and “receiveReply” in a previous example. These activities must be explicitly encoded in a SEDS instance because this example deviates from the simple protocol in Figure 4‑5.

The state machine “SRIT” is defined in the component to respond to the arrival of command “aSR” from interface “A” and to respond to the guarded arrival of indication “data” from interface “B”. This state machine must be explicitly encoded in a SEDS instance because this example deviates from the simple protocol in Figure 4‑5.

<StateMachine name="SRIT">

<State name="Idle"/>

<State name="Communicating"/>

<Transition name="Send" fromState="Idle" toState="Communicating">

<OnCommandPrimitive interface="A" command="aSR">

<ArgumentValue name="aIn" outputVariableRef="x"/>

</OnCommandPrimitive>

<Do activity="send" />

</Transition>

<Transition name="timeout" fromState="Communicating" toState="Idle">

<OnTimer nanosecondsAfterEntry="1000000000"/>

</Transition>

<Transition name="Receive" fromState="Communicating" toState="Idle">

<OnCommandPrimitive interface="B" command="receive">

<ArgumentValue name="data" outputVariableRef="x"/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef="data"/>

<TypeOperand>cmdSRr</TypeOperand>

</TypeCondition>

</Guard>

<Do activity="receiveReply" />

</Transition>

</StateMachine>

The state machine consists of two states and three transitions between those states. The transition “send” stores the PDU in variable “x” and then invokes the activity “receive” from a previous example. The transition “Receive” stores the PDU in variable “x” and then invokes the activity “receiveReply”. The guard in the transition causes it to ignore PDU’s that have the wrong type. The transition “timeout” returns to the “Idle” state when a timeout occurs, so the state machine can continue to handle commands from interface A. A diagram for this state machine appears in Figure 4‑14.

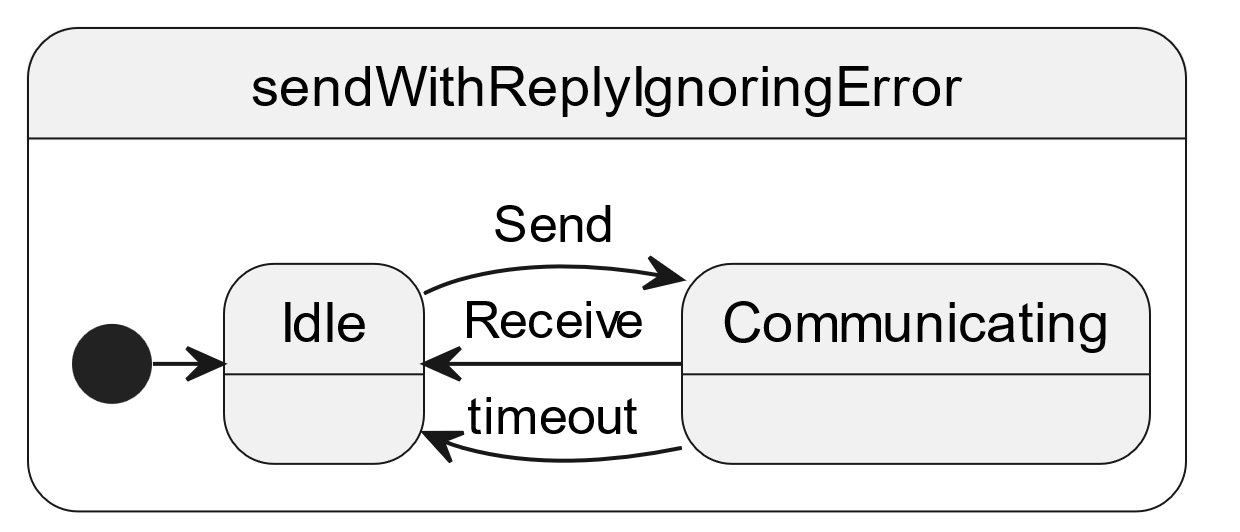


Figure 4‑14 State Machine for Send with Reply Ignoring Error Protocol

#### Send with Reply Reporting Timeout

This example defines an explicit extension to the command labelled “INOUT” in Figure 4‑5. The PDU’s for this command are the same as in Section 4.2.6.3, plus the following.

<ContainerDataType name="cmdSRt" baseType="TC" shortDescription="a timeout has occurred">

<ConstraintSet>

<ValueConstraint entry="cc" value="9"/>

</ConstraintSet>

</ContainerDataType>

The command definition in the declared interface A is as follows.

<Command name="aSRRT" shortDescription="send with reply, reporting timeout">

<Argument name="aIn" type="cmdSRs" mode="in" dataUnit="true"/>

<Argument name="aOut" type="cmdSRr" mode="out" dataUnit="true"/>

<Argument name="aTimeout" type="cmdSRt" mode="out" dataUnit="true"/>

</Command>

This example uses activities “send” and “receiveReply” in a previous example. These activities must be explicitly encoded in a SEDS instance because this example deviates from the simple protocol in Figure 4‑5.

The activity “reportErrord” is defined in the component to deliver a PDU to the interface “A” command variable “aTimeout” from component variable named “x”. The “failed=true” provides information that the send-with-reply command failed. This activity must be explicitly encoded, because it is not implied by the simple protocol in Figure 4‑5. This activity may be implicit (omitted) if the mode attribute of argument aTimeout were specified as mode=”error”.

<Activity name="reportError">

<Body>

<SendCommandPrimitive interface="A" command="aSNR" failed="true">

<ArgumentValue name="aTimeout">

<VariableRef variableRef="timeoutPDU"/>

</ArgumentValue>

</SendCommandPrimitive>

</Body>

</Activity>

The state machine “SRRT” is defined in the component to respond to the arrival of command “aSR” from interface “A” and to respond to the guarded arrival of indication “data” from interface “B”. This state machine must be explicitly encoded in a SEDS instance because this example deviates from the simple protocol in Figure 4‑5. This state machine may be implicit (omitted) if the mode attribute of argument aTimeout were specified as mode=”error”.

<StateMachine name="SRRT">

<State name="Idle"/>

<State name="Communicating"/>

<Transition name="Send" fromState="Idle" toState="Communicating">

<OnCommandPrimitive interface="A" command="aSR">

<ArgumentValue name="aIn" outputVariableRef="x"/>

</OnCommandPrimitive>

<Do activity="send" />

</Transition>

<Transition name="timeout" fromState="Communicating" toState="Idle">

<OnTimer nanosecondsAfterEntry="1000000000"/>

<Do activity="reportError"/>

</Transition>

<Transition name="Receive" fromState="Communicating" toState="Idle">

<OnCommandPrimitive interface="B" command="receive">

<ArgumentValue name="data" outputVariableRef="x"/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef="data"/>

<TypeOperand>cmdSRr</TypeOperand>

</TypeCondition>

</Guard>

<Do activity="receiveReply" />

</Transition>

</StateMachine>

The state machine consists of two states and three transitions between those states. The transition “send” stores the PDU in variable “x” and then invokes the activity “receive” from a previous example. The transition “Receive” stores the PDU in variable “x” and then invokes the activity “receiveReply”. The guard in the transition causes it to ignore PDU’s that have the wrong type. The transition “timeout” invokes activity “reportTimeout” and returns to the “Idle” state when a timeout occurs, so the state machine can continue to handle commands from interface A. A diagram for this state machine appears in Figure 4‑15.

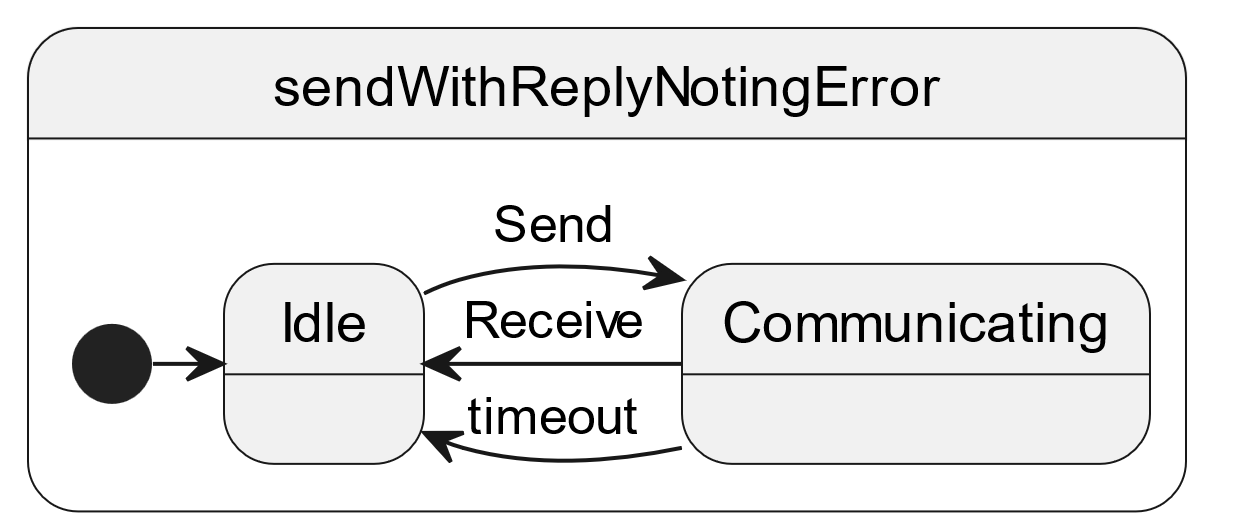


Figure 4‑15 State Machine for Send with Reply Noting Error Protocol

Note that an alternative specification of this command can have implicit (omitted) activity and state machine, if mode=”error” is specified on the aTimeout message in the command definition. The use of mode=”error” in other command arguments implies a similar implicit activity and state machine to report errors in a consistent manner. This example shows timeout as an error condition; however mode=”error” may be used for any error indication; it is not limited to timeouts.

#### Send with Reply Reporting Timeout and Failing State Machine

This example defines an explicit extension to the command labelled “INOUT” in Figure 4‑5. This example is the same as in Section 4.2.6.6 except that the state machine is disabled on timeout, requiring FDIR.

The command definition in the declared interface A is as follows.

<Command name="aSRFT" shortDescription="send with reply, failing state machine on timeout">

<Argument name="aIn" type="cmdSRs" mode="in" dataUnit="true"/>

<Argument name="aOut" type="cmdSRr" mode="out" dataUnit="true"/>

<Argument name="aTimeout" type="cmdSRt" mode="out" dataUnit="true"/>

</Command>

The state machine “SRFT” is defined in the component to respond to the arrival of command “aSR” from interface “A” and to respond to the guarded arrival of indication “data” from interface “B”. This state machine must be explicitly encoded in a SEDS instance because this example deviates from the simple protocol in Figure 4‑5.

<StateMachine name="SRFT">

<State name="Idle"/>

<State name="Communicating"/>

<State name="Error"/>

<Transition name="Send" fromState="Idle" toState="Communicating">

<OnCommandPrimitive interface="A" command="aSR">

<ArgumentValue name="aIn" outputVariableRef="x"/>

</OnCommandPrimitive>

<Do activity="send" />

</Transition>

<Transition name="timeout" fromState="Communicating" toState="Error">

<OnTimer nanosecondsAfterEntry="1000000000"/>

<Do activity="reportTimeout"/>

</Transition>

<Transition name="Receive" fromState="Communicating" toState="Idle">

<OnCommandPrimitive interface="B" command="receive">

<ArgumentValue name="data" outputVariableRef="x"/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef="data"/>

<TypeOperand>cmdSRr</TypeOperand>

</TypeCondition>

</Guard>

<Do activity="receiveReply" />

</Transition>

</StateMachine>

The state machine consists of three states and three transitions among those states. The transition “send” stores the PDU in variable “x” and then invokes the activity “receive” from a previous example. The transition “Receive” stores the PDU in variable “x” and then invokes the activity “receiveReply”. The guard in the transition causes it to ignore PDU’s that have the wrong type. The transition “timeout” invokes activity “reportTimeout” and goes to the “Error” state when a timeout occurs, so the state machine cannot continue to handle commands from interface A until FDIR has restarted it. A diagram for this state machine appears in Figure 4‑16.

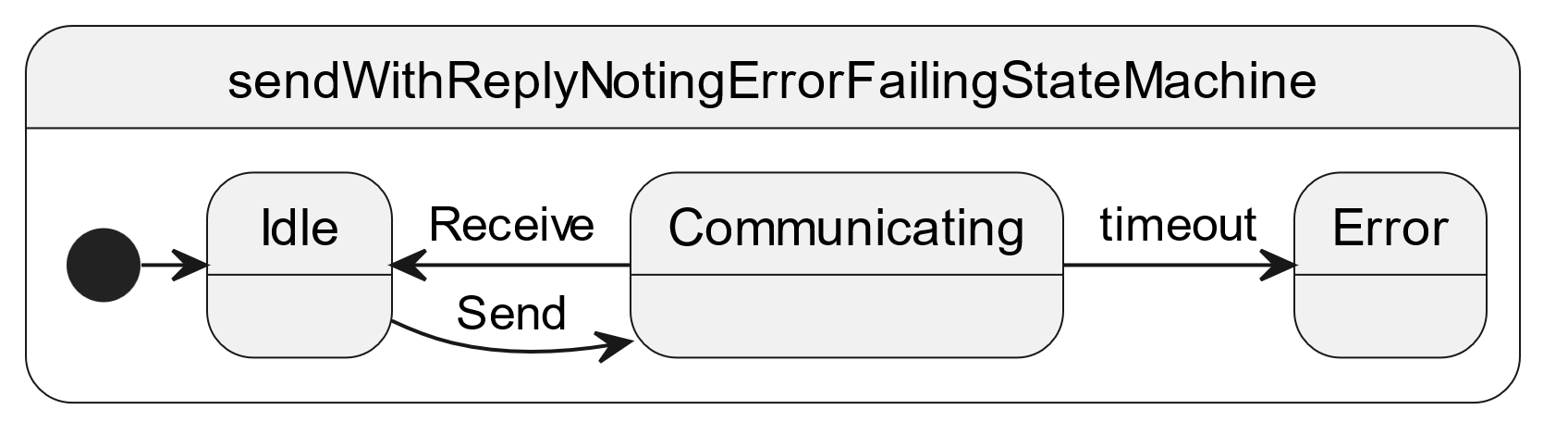


Figure 4‑16 State Machine for Send with Reply Reporting Error and Failing on Error

## Device Data Sheet Examples

This section provides some topical illustrations how different SEDS elements and attributes are used together to define a feature of a data interface of a device or software component. A systematic walk through each SEDS element and attribute, providing a description of each can be obtained in the SEDS blue book [3]. This book provides motivation for when and how to use the schema elements in the SEDS blue book. This section should be read in conjunction with the SEDS and Common DoT standards [3][4].

Section 4.3.1 is an example of the major parts of a device data sheet.

Section 4.3.2 is an example that distinguishes packet formats.

Section 4.3.3 is an example of the behaviour of a device service.

Section 4.3.4 is an example of a mapping of data between a device specific interface and a functional interface.

Section 4.3.5 is an example of the use of calibrators to convert between raw and engineering units.

Section 4.3.6 explains how to detect error conditions and to generate software that responds to those conditions.

Section 4.3.7 explains the relationship between SOIS subnetwork standards and the device services generated by an agency’s tool chain. An example of the use of generic type mappings appears here.

Section 4.3.8 helps to decide whether to define a series of adjacent similar data items as a list or as an array.

Section 4.3.9 tells how to describe the length of a container that contains an entry that is a 1-1 function of the length of the container.

Section 4.3.10 describes how to use an indexing type to specify the length of an array.

Section 4.3.11 explains how to override types in a declared interface.

Section 4.3.12 walks through the description of acquiring telemetered data from a device.

Section 4.3.13 walks through the description of commanding a device.

Section 4.3.14 explains how to describe integers that represent fractions.

Section 4.3.15 explains how to describe strings.

### Summary of Parts of a Device Data Sheet

The file named “jena\_star\_tracker.xml” contains the following high-level features that are common to device data sheets.

<?xml version="1.0" encoding="UTF-8"?>

<DataSheet xmlns:xs="http://www.w3.org/2001/XMLSchema" 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:noNamespaceSchemaLocation="http://www.ccsds.org/schema/sois/seds">

<Device name="JENA\_AS400"/>

<xi:include href="ccsds.sois.seds.xml" xpointer="element(/1/1)"/>

<xi:include href="ccsds.spacelink.packet.xml" xpointer="element(/1/1)"/>

<xi:include href="dod.milbus.milstd1553.xml" xpointer="element(/1/1)"/>

<xi:include href="jena.packet.utilisation.xml" xpointer="element(/1/1)"/>

<xi:include href="ccsds.sois.subnetwork.xml" xpointer="element(/1/1)"/>

<Package name="JENA/AS400">

<DataTypeSet>…</DataTypeSet>

<DeclaredInterfaceSet>…</DeclaredInterfaceSet>

<ComponentSet>…</ComponentSet>

</Package>

</DataSheet>

The base of the document tree for a device is a DataSheet element.

The DataSheet element contains a Device element and one or more Package elements. The DataSheet element may also have a number “include” elements in the “xi” namespace, which point to specific external package files that provide definitions used in the device data sheet.

Each package file contains a single Package element, which can be used by reference from any number of data sheet files.

The Package element may contain a DataTypeSet element to define data types used in the package. It may contain a DeclaredInterfaceSet element, which identifies the external interfaces of the device. The Package element may also contain a ComponentSet element, which defines the behaviour of the device service.

### Distinguishing Packet Formats by their Headers

The packets may contain a variety of formats of data, which is indicated by information in their headers. The example below shows how a SEDS expresses this information.

The primary header of a CCSDS Spacelink packet has the following definition in the SEDS file named “ccsds.spacelink.packet.xml”. See CCSDS reference [10].

<PackageFile …>

<Package name=”CCSDS/SPACELINK/PACKET” …>

<DataTypeSet>

<ContainerDataType abstract=”true” name=”PRIMARY\_HEADER”>

<EntryList>

…

<Entry type=”CCSDS/SOIS/SEDS/UINT8” name=”packetType” …>

…

<IntegerDataEncoding encoding=”unsigned” sizeInBits=”1”/>

</Entry>

<Entry type=”CCSDS/SOIS/SEDS/BOOLEAN” name=”secondaryHeaderFlag”…>

…

<Entry type=”CCSDS/SOIS/SEDS/UINT8” name=”sequenceFlags” …>

…

<Entry type=”CCSDS/SOIS/SEDS/UINT16” name=”packetDataLength” …>

…

</Entry>

</EntryList>

</ContainerDataType>

</DataTypeSet>

</Package>

</PackageFile>

The following features are of interest in the excerpt above.

* The package name is a sequence of progressively more specific names, delimited by solidus characters.
* The data sheet defines the packet as a “ContainerDataType”; in other words, the packet is a collection of other data types, each called an “Entry” within the container. The container data type is marked abstract, to provide a basis for more concrete definitions.
* The CCSDS/SOIS/SEDS/… types are defined in the file “ccsds.sois.seds.xml”, not shown here.
* The CCSDS/SOIS/SEDS/UINT8 types are overridden to fit the number of bits in the CCSDS spacelink packet primary header specification.

The file named “jena.packet.utilization.xml” defines a telemetry packet header as follows.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<ContainerDataType abstract=”true”

baseType=”CCSDS/SPACELINK/PACKET/PRIMARY\_HEADER” name=”HDR\_TM\_PKT”>

…

<ConstraintSet>

<ValueConstraint value=”0” entry=”packetType”/>

<ValueConstraint value=”true” entry=”secondaryHeaderFlag”/>

<ValueConstraint value=”3” entry=”sequenceFlags”/>

</ConstraintSet>

<EntryList>

…

</EntryList>

…

</ContainerDataType>

…

</Package>

The following features are of interest in the excerpt above.

* The container data type refers to the abstract PRIMARY\_HEADER type by concatenating the name of the package in which the PRIMARY\_HEADER type is defined. The toolchain interprets this expression by keeping a collection of SEDS files in its memory, indexed by package name.
* This container data type is marked abstract.
* The constraint set specifies the values of three of the entries in the PRIMARY\_HEADER type. When a CCSDS space link packet has these values in its primary header, this container data type definition applies. There was no constraint set in the PRIMARY\_HEADER type definition, so how do we know that a packet is a CCSDS space link packet? The answer to that question appears later in this example.
* This container data type contains an entry list, whose details are not shown. The entries in this list are appended to the entries in the PRIMARY\_HEADER definition in packets where this container data type applies.

The file named “jena.packet.utilization.xml” contains additional abstract container types derived from the HDR\_TM\_PKT above, such as the following housekeeping packet. The derivation in this example works by adding entries for housekeeping after the header, and by specifying constraints on the content of the header. The abstract header definition provides the common part of a number of different types of concrete packets. The constraints tell the values that appear in the header to identify the specific concrete packet type, and the added entries in the concrete packet describe the data that such a packet contains.

<Package name=”JENA/AS400” …>

…

<ContainerDataType abstract=”true” baseType=”HDR\_TM\_PKT”

name=”PUS\_HK\_REPORT”>

<ConstraintSet>…</ConstraintSet>

<EntryList>

<Entry type=”CCSDS/SOIS/SEDS/UINT32” name=”sid” …/>

</EntryList>

</ContainerDataType>

…

</Package>

The following features are of interest in the excerpt above.

* The base type reference in this case lacks a package name prefix because this type is defined in the same package as the base type.
* This type definition has constraints on the entries in HDR\_TM\_PKT to limit its applicability.
* This type definition has an integer entry that classifies more concrete container types.

The file named “jena\_star\_tracker.xml” contains concrete packet definitions that derive from the PUS\_HK\_REPORT container defined above.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<ContainerDataType baseType=”JENA/PACKET\_UTILIZATION/PUS\_HK\_REPORT”

name=”TM\_ADB”>

<ConstraintSet>

<ValueConstraint value=”55” entry=”packetDataLength”/>

<ValueConstraint value=”11” entry=”sid”/>

</ConstraintSet>

<EntryList>

…

<Entry type=”CCSDS/SOIS/SEDS/INT32” name=”qs”>

…

</Entry>

…

</EntryList>

</ContainerDataType>

…

</Package>

The following features are of interest in the excerpt above.

* This container type is not marked abstract, so it is concrete, which means that it can apply to real packets.
* The author of this SEDS named this packet using a concise summary of its content: “TM” says that this is a telemetry packet. “ADB” says that the packet contains attitude data. Each author is free to formulate their own naming convention within their SEDS The names make it easier for a human to read a SEDS, but the tool chain relies on the attributes and elements in a SEDS to interpret a SEDS.
* The constraint set for this type contains a reference to an entry in the PRIMARY\_HEADER type. No special package name prefix is needed here because the entries in each container must have names that do not appear in any ancestral containers.
* The usage of “sid” in one of the constraints describes how to recognize a container type, and it relies upon there being a data field present in the base container which can be compared to the value in the constraint.
* The constraint on packet length provides a sanity check for how much data is expected to be in packets with this “sid” value.
* This container type has an entry named “qs”, which will be used in Section 4.3.4.

The file named “jena\_star\_tracker.xml” contains a component that will read telemetry packets from the subnetwork where the physical device is connected. The definition of that component appears below.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<ComponentSet>

<Component name=”DSAP”>

…

<RequiredInterfaceSet>

<Interface type=”CCSDS/SOIS/Subnetwork/PSInterfaceType” …>

<GenericTypeMapSet>

<GenericTypeMap type=”JENA/PACKET\_UTIIZATION/HDR\_TM\_PKT” …/>

…

</GenericTypeMapSet>

</Interface>

…

</RequiredInterfaceSet>

…

</Component>

</ComponentSet>

…

</Package>

The following features are of interest in the excerpt above.

* Because this component will read telemetry packets from the device across a subnetwork, it uses an interface through the SOIS Subnet Packet Service, which is defined in the package “CCSDS/SOIS/Subnetwork”, which is defined in the SEDS file named “ccsds.sois.subnetwork.xml”. A spacecraft software architecture that does not support the SOIS subnetwork architecture would have a different required interface in the example.
* The required interface definition consists of a set of generic type maps, one of which refers to the HDR\_TM\_PKT type. This means that any of the packets defined as derivatives of the abstract HDR\_TM\_PKT type could appear at the subnetwork interface, and they will be recognized by the values specified in their constraint sets and the values specified in ancestral constraint sets. This answers the question posed earlier in this example, how do we know that the packets are CCSDS space link packets with PRIMARY\_HEADER as header, which is the most distant ancestral type. The answer is that only the HDR\_TM\_PKT derivatives from PRIMARY\_HEADER will appear in this required interface.

### Specifying Device-Specific Behavior

The preceding section tells how to describe a housekeeping packet named TM\_ADB in the context of the interface where it appears. This section describes how a tool chain can generate software to read that type of packet. That information appears in the implementation element of a component that can read the TM\_ADB packet, as shown below.

The reason for reading the packet is to present the contents of the packet in an interface, so the following declares the interface in which the contents of the packet will be presented. When a tool chain builds a software interface from the description below, it may generate a “struct” data type for the “Parameter” identified below, or it may generate a class with properties for accessing the entries in the container. If the tool chain generates hardware description language for an FPGA, it may reserve a particular space in which other parts of the FPGA firmware can find the packet content when such a packet arrives.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<DeclaredInterfaceSet>

<Interface abstract=”false” level=”access” name=”DSAI” …>

<ParameterSet>

…

<Parameter mode=”async” readOnly=”true” type=”TM\_ADB” name=”TM\_ADB”/>

…

</ParameterSet>

<CommandSet>…</CommandSet>

</Interface>

…

</DeclaredInterfaceSet>

…

</Package>

The following features are of interest in the excerpt above.

* The declared interface set describes the external interfaces of the device or software component defined by this SEDS.
* The interface is named “DSAI”, which happens to be an acronym for “device specific access interface”, but any name could be used.
* The “level” attribute of the interface places it in the SOIS architecture as a device-specific access procedure. See Section 4.2.2.
* The interface presents a parameter named “TM\_ADB”, among other parameters and commands not shown here. The term “parameter” here represents a kind of message that flows across the interface, as described in Section 4.2.2.
* The type of the interface parameter is “TM\_ADB”, which was defined in Section 4.3.2, and is a type of message that crosses the interface.

Having described the “DSAI” interface above, the component below transforms the packets read from the SOIS Packet Service into internal objects as described below. The implementation of the internal objects varies with the target technology that the tool chain generates. For software, the tool chain would generate “struct” or class definitions for the packets. For FPGA, the tool chain would generate hardware description language for interpreting packets in buffers or in other reserved locations.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<ComponentSet>

<Component name=”DSAP”>

<ProvidedInterfaceSet>

<Interface type=”JENA/AS400/DSAI” name=”dsai”/>

</ProvidedInterfaceSet>

<RequiredInterfaceSet>

<Interface type=”CCSDS/SOIS/Subnetwork/PSInterfaceType” …>

…

</Interface>

</RequiredInterfaceSet>

…

</Component>

…

</ComponentSet>

…

</Package>

The following features are of interest in the excerpt above.

* The name of the component is “DSAP”, which happens to be an acronym for “device-specific access procedure”, but any name could be used.
* The component provides an interface named “dsai” of type “DSAI”, which was defined earlier in the declared interface set.
* The component requires the SOIS Packet Service interface, which is defined in a separate package in a separate file named “ccsds.sois.subnetwork.xml”.

Continuing to examine other details of the component named “DSAP”, we examine the internal implementation.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<ComponentSet>

<Component name=”DSAP”>

…

<Implementation>

<VariableSet>

…

<Variable type=”TM\_ADB” name=”working\_tm\_adb”/>

…

</VariableSet>

<ActivitySet>

…

<Activity name=”update\_TM\_ADB”>

<Body>

<SendParameterPrimitive operation=”get” interface=”dsai”

parameter=”TM\_ADB”/>

<ArgumentValue>

<VariableRef variableRef=”working\_tm\_adb”/>

</ArgumentValue>

</Body>

</Activity>

…

</ActivitySet>

<StateMachineSet>

<StateMachine defaultEntryState=”Idle” name=”ASYNC” …>

<State name=”Idle”/>

…

<Transition fromState=”Idle” toState=”Idle” name=”RECV\_TM\_ADB”>

<OnCommandPrimitive interface=”subnetworkPS” command=”receive”>

<ArgumentValue name=”data” outputVariableRef=”working\_tm\_adb”/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef=”data”/>

<TypeOperand>TM\_ADB</TypeOperand>

</TypeCondition>

</Guard>

<Do activity=”update\_TM\_ADB”/>

</Transition>

…

</StateMachine>

…

</StateMachineSet>

</Implementation>

</Component>

…

</ComponentSet>

…

</Package>

The following features are of interest in the excerpt above.

* The implementation has an internal variable of type “TM\_ADB”, named “working\_tm\_adb”, where it stores a copy of a TM\_ADB packet that it reads from the packet service interface.
* The implementation has an activity named “update\_TM\_ADB” which transfers a TM\_ADB packet from the internal variable named “working\_tm\_adb” to the parameter named “TM\_ADB” in the interface named “dsai”.
* The implementation has a state machine named “ASYNC”, which handles the asynchronous protocol.
* The state machine “ASYNC” has a single state named “Idle”, and numerous transitions that exit that state and then go right back to that state. The result of this structure is that the state machine stays in its “Idle” state, and responds to events in its numerous transitions.
* One of the transitions of the state machine is named “RECV\_TM\_ADB”, which is stimulated by the event of receiving a packet in the subnetwork packet service interface.
* The guard condition says to execute this transition when the packet is of type TM\_ADB. This form of expression is usually simpler than restating the constraints on the container definition of the packet type.
* The command primitive says to put the packet into the variable named “working\_tm\_adb”. Note that the guard is able to refer to an argument of the command that triggers the transition.
* When the transition occurs, it executes the activity named “update\_TM\_ADB”.

### Mapping from Device-Specific to Functional Interface

The preceding section tells how to read a housekeeping packet named TM\_ADB and to place it into the device-specific access interface. This section describes how to transfer data from the packet into the functional interface, as shown below.

This example transfers data between the public interfaces of the SEDS declared below.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<DeclaredInterfaceSet>

<Interface abstract=”false” level=”access” name=”DSAI” …>

<ParameterSet>

…

<Parameter mode=”async” readOnly=”true” type=”TM\_ADB” name=”TM\_ADB”/>

…

</ParameterSet>

<CommandSet>…</CommandSet>

</Interface>

<Interface abstract=”false” level=”functional” name=”DSFI” …>

<ParameterSet>

<Parameter mode=”async” readOnly=”true” type=”CCSDS/SOIS/SEDS/FLOAT64”

name=”TM\_ADB\_qs”/>

…

</ParameterSet>

</Interface>

</DeclaredInterfaceSet>

…

</Package>

The following features are of interest in the excerpt above.

* The “DSAI” interface is the same as in Section 4.3.3, but we show it again here because it is used in this example.
* The new interface is named “DSFI”. The “level” attribute of the new interface indicates that this is a functional interface.
* The DSFI interface exposes a number of parameters, one of which we display in the excerpt. The parameter is a float named “TM\_ADB\_qs”.
* There is no semantic information for the parameter TM\_ADB\_qs in the declaration of the DSFI interface, so the tool chain will not be able to check that the data in the interface matches data in an interface in another SEDS.

Having declared the type of the “DSFI” interface above, the component below transforms the packets exposed in the DSAI interface into objects more appropriate for a functional interface exposed in the DSFI interface.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<ComponentSet>

…

<Component name=”DACP”>

<ProvidedInterfaceSet>

<Interface type=”JENA/AS400/DSFI” name=”dsfi”/>

</ProvidedInterfaceSet>

<RequiredInterfaceSet>

<Interface type=”JENA/AS400/DSAI” name=”dsai”/>

</RequiredInterfaceSet>

</Component>

</ComponentSet>

…

</Package>

The following features are of interest in the excerpt above.

* The name of the component is “DACP”, which happens to be an acronym for “device abstraction control procedure”, but any name could be used.
* The component provides an interface named “dsfi” of type “DSFI”, which was defined earlier in the declared interface set.
* The component requires the DSAI interface, which is defined in the same SEDS.

Continuing to examine other details of the component named “DACP”, we examine the internal implementation.

<Package name=”JENA/PACKET\_UTILIZATION” …>

…

<ComponentSet>

…

<Component name=”DACP”>

…

<Implementation>

<ParameterActivityMapSet>

…

<ParameterActivityMap>

<Provided name=”local\_tm\_adb\_qs” interface=”dsfi”

parameter=”TM\_ADB\_qs”/>

<Required name=”local\_tm\_adb” interface=”dsai”

Parameter=”TM\_ADB”/>

<GetActivity>

<Calibration outputVariableRef=”local\_tm\_adb\_qs”

inputVariableRef=”local\_tm\_adb.qs”>

<PolynomialCalibrator>

<Term coefficient=”9.313225746154785E-10” exponent=”1”/>

</PolynomialCalibrator>

</Calibration>

</GetActivity>

</ParameterActivityMap>

…

</ParameterActivityMapSet>

</Implementation>

</Component>

</ComponentSet>

…

</Package>

The following features are of interest in the excerpt above.

* The implementation consists of a set of parameter activity maps.
* Each parameter activity map extracts an entry from the TM\_ADB packet, converts it to engineering units, and places it into the interface named “dsfi”.
* One of the parameter activity maps is shown above.
* The “Provided” element says that the map puts its result into the parameter named “TM\_ADB\_qs” in the interface named “dsfi”. It takes that result from an internal variable named “local\_tm\_adb\_qs”.
* The “Required” element says that the map obtains a TM\_ADB packet from the interface named “dsai” and puts it into an internal variable named “local\_tm\_adb”.
* The “GetActivity” element executes a polynomial calibrator from the “qs” entry of the TM\_ADB packet in internal variable named “local\_tm\_adb” to the internal variable named “local\_tm\_adb\_qs”.

### Calibrating Raw Parameters

One of the typical tasks in mapping a native interface to a functional interface is conversion from counts provided by an analog-to-digital converter to engineering units. A calibration function maps values from the native measurement of a device to engineering units, or from engineering units to the native command arguments of a device. The SEDS schema provides semantic tags to specify units of measure and other terms describing the meaning of data. These semantic tags make a SEDS useful in the context of other SEDS in an assembly of devices and software components.

It is not always necessary or desirable to apply these calibrations in flight. In some flight software architectures, all onboard processing uses the raw counts directly. The calibration information is only used on the ground, typically for human graphical user interfaces.

A calibration element can be referenced from the following areas of a datasheet.

* As part of an activity, either directly or inside an iteration or conditional statement.
* Equivalently, as the content of a mapping element (Section 4.3.4)
* As a specification of the interpretation (i.e. semantics) of a value on an interface.

The example in Section 4.3.4 showed one way to invoke a calibration function; by using the implementation section of a component to specify an algorithmic translation between the raw and calibrated data. This approach is highly general, accounting for cases where the correct calibration is dependant on some mode or parameter of the device or system. However, it naturally results in the raw and engineering values being logically separate parameters, rather than two representations of the same data.

An alternative approach, which preserves that identity, is to associate a calibration with a parameter or command argument *on an interface*. This means that the specified calibration is the one that *would be* needed in order to translate the actual parameter/argument value into the value described by the unit and other semantics.

Tool chains can straightforwardly translate such an interface-based representation of a calibration into the implementation of a component that does that transformation. However, complex state-dependant calibrations specified by a component implementations cannot, in general, be translated in the other direction without loss of information.

Two types of calibrations are supported; polynomials and splines. Polynomial calibrators directly define the coefficients of the mapping function, whereas spline calibrations define a set of fixed points on the map, and how to interpolate between them.

#### Polynomial Calibration

The following example converts a rate measurement of an inertial rate unit to engineering units.

<DataSheet xmlns=<http://www.ccsds.org/schema/sois/seds>>

The Device element that follows is not expanded for this example.

<Device>…</Device>

The Package element that follows provides an interface in which the data is calibrated to engineering units.

<Package name=”inertialRateUnit”>

The angularRate container data type that follows defines the general form of an angular rate vector, binding it to a NumberType that is defined elsewhere, but not binding it to a particular unit of measure. The quantity kind says that it represents an angular rate. The interpretation and transformationType attributes tell the tool chain that this container is a vector.

<DataTypeSet>

<ContainerDataType name=”angularRate”

interpretation=”transformation”

quantityKind=”angularRateQK”

transformationType=”rotationRate”

chirality=”rightHanded”>

<EntryList>

<Entry name=”rx” type=”NumberType”/>

<Entry name=”ry” type=”NumberType”/>

<Entry name=”rz” type=”NumberType”/>

</EntryList>

</ContainerDataType>

…

</DataTypeSet>

<DeclaredInterfaceSet>

The interface that follows provides raw data.

<Interface name=”rawRateIType”>

<ParameterSet>

The rate parameter that follows is an angular rate, as defined above. This parameter has a unit of measure that is the counts from an analog-to-digital converter.

<Parameter name=”rate” type=”angularRate”

purpose=”measurement”

coordinateType=”J2000”

referenceFrame=”device”

unit=”aToDCount”>

</Parameter>

</ParameterSet>

</Interface>

The interface that follows provides data calibrated to engineering units.

<Interface>

<ParameterSet>

The rate parameter that follows is another angular rate, as defined above. This parameter has a unit of measure that is radians per second.

<Parameter name=”rate” type=”angularRate”

purpose=”measurement”

coordinateType=”J2000”

referenceFrame=”device”

unit=”radianPerSecond”>

</Parameter>

</ParameterSet>

</Interface>

</DeclaredInterfaceSet>

<ComponentSet>

The component that follows provides the native data interface of the inertial rate unit.

<Component name=”IRUNative”>

<ProvidedInterfaceSet>

<Interface name=”rawRate” type=”rawRateIType”/>

…

</ProvidedInterfaceSet>

…

</Component>

The component that follows provides the functional interface of the inertial rate unit. The functional interface provides calibrated engineering units.

<Component name=”inertialRateUnit”>

<ProvidedInterfaceSet>

<Interface name=”rate” type=”rateIType”/>

…

</ProvidedInterfaceSet>

<RequiredInterfaceSet>

<Interface name="rawRate" type="rawRateType"/>

…

</RequiredInterfaceSet>

…

<Implementation>

…

<ParameterActivityMapSet>

<ParameterActivityMap>

<Provided interface="rate" parameter="rate" name="rate"/>

<Required interface="rawRate" parameter="rate" name="rawRate"/>

<GetActivity>

The calibrator that follows defines how to convert numbers in the native data interface into numbers in the functional interface, as specified by the input variable reference and the output variable reference. In this example, there is a different analog-to-digital converter for each rate axis.

<Calibration inputVariableRef="rawRate.rx" outputVariableRef="rate.rx">

<PolynomialCalibrator>

<Term coefficient="2.71828" exponent="0"/>

<Term coefficient="3.14159" exponent="1"/>

</PolynomialCalibrator>

</Calibration>

<Calibration inputVariableRef="rawRate.ry" outputVariableRef="rate.ry">

<PolynomialCalibrator>

<Term coefficient="2.719" exponent="0"/>

<Term coefficient="3.142" exponent="1"/>

</PolynomialCalibrator>

</Calibration>

<Calibration inputVariableRef="rawRate.rz" outputVariableRef="rate.rz">

<PolynomialCalibrator>

<Term coefficient="7.71828" exponent="0"/>

<Term coefficient="3.123" exponent="1"/>

</PolynomialCalibrator>

</Calibration>

</GetActivity>

</ParameterActivityMap>

</ParameterActivityMapSet>

</Implementation>

</Component>

…

</ComponentSet>

</Package>

</DataSheet>

The calibrator in the example above may be invoked implicitly when the input variable changes value, or when an observer (such as a GUI in an operations control center) reads the value of the variable, because there is no state machine to determine when to invoke the calibrator. The tool chain implements this convention.

#### Spline Calibration

The following example uses a first-order spline calibrator to convert codes for the length of a frame from a milbus bus controller. The spline is piecewise linear, and the domain of the function is integers, so the meaning is a table that says to interpret the raw value “0” as “34”, and to interpret raw values in {1, …, 31} as 2 higher.

<ContainerDataType abstract="true" name="BC\_TO\_RT\_TRANSFER" …>

<EntryList>

<Entry name="rt" … type="ADDRESS"/>

<FixedValueEntry fixedValue="false" name="txFlag" … type="CCSDS/SOIS/SEDS/BOOLEAN"/>

<Entry name="sa" … type="ADDRESS"/>

<LengthEntry name="numberOfDataWords" type="CCSDS/SOIS/SEDS/UINT8">

<IntegerDataEncoding encoding="unsigned" sizeInBits="5"/>

<ValidRange>

<MinMaxRange max="31" min="0" rangeType="inclusiveMinInclusiveMax"/>

</ValidRange>

<SplineCalibrator>

<SplinePoint calibrated="34.0" order="1" raw="0.0"/>

<SplinePoint calibrated="3.0" order="1" raw="1.0"/>

<SplinePoint calibrated="33.0" order="1" raw="31.0"/>

</SplineCalibrator>

</LengthEntry>

</EntryList>

</ContainerDataType>

### Failure Detection Behavior

The state machine in Section 4.3.3 describes the nominal behavior of the device service interacting with a device. If, during operation, the device is observed to behave differently from that specification, this should normally be treated as a unrecoverable failure of the device. This includes sending an unrecognized message, or a message not expected in the current state. This section describes how to extend the nominal device specification to allow identification of anticipated fault conditions, allowing them to be autonomously recovered from.  
Some error conditions are reported explicitly by a device, and some can be detected implicitly by a state machine specification. In either case, the corresponding state machine can:

* Explicitly ignore the fault by transitioning back to the current state.
* Identify the request that caused the fault, and report it as failed by sending a reply primitive with the attribute ‘failed’ set to ‘true’.
* Transition the state machine to an exit state.  
  The excerpt below, from the example file jena\_star\_tracker.xml, describes a state machine that responds to explicit error indications received. The explicit error conditions are detected when the state machine transitions to an exit state. In the example below the exit state is named “ERROR”. Note that the state name is not semantically significant, the fact that it is an exit state is what matters.

TM\_EV\_CYCLE\_OVERRUN … other similar transitions track other indications … A state machine may explicitly detect a time-out by including an OnTimer transition to an error state. Receipt of a message from the device for which there is no matching transition from the current state, can be treated as implicit detection of an error condition. When decoding data from a device using a container definition in SEDS, it is possible that the device data does not fit the constraints of the container definition; that situation can be treated as implicit detection of an error condition. State machines may access and update component variables. This can be used to specify and check simple device-specific protocols, such as a failure report that is supposed to echo back the transaction id of the command that failed. Explicit transition to an exit state, or detection of device behaviour that disagrees with a state machine, are indications that the flight software should respond to a failure condition. The tool chain must generate the linkage to the software to implement this response. Each agency provides appropriate FDIR software for their mission, optionally using SEDS to describe the interface of that software as in Section 5.

### The Subnetwork Interface

The SOIS subnetwork standards describe abstract interfaces for an on-board subnetwork. If an agency implements these interfaces within the constraints of the SOIS standards, then the agency’s tool chain can use the abstract definition of the interfaces that appear in the file ccsds.sois.subnetwork.xml to generate device services that use the implemented interface.

In the case the SOIS subnetwork standards are not used, a similar package of interface definitions can be created for the available access points to whatever subnetwork standards are in use.

This section describes how a tool chain can generate that linkage between a manufacturer’s device and an agency’s subnetwork service access point.

#### General Mechanism

One of the forms of abstraction that is possible in SEDS is the use of generic types to describe an interface without specifying the concrete data types that flow through the interface. To use that abstract interface, it is necessary to have a mapping from abstract types to concrete types. The mapping is static, in the sense that, at run time, the mapping from abstract types to concrete types does not change from what is defined in a SEDS instance.

A service access point in this context uses a particular protocol to access devices in on-board networks. The protocols are defined outside of SOIS standards, and this section describes how SOIS enables usage of those protocols. For example, the SOIS Packet Service describes service access points for protocols that send and receive packets to and from devices. There are other protocols, such as remote memory access protocol or streaming, which have alternative formulations of service access points.

For the SOIS subnetwork service, the access point corresponds to a particular 'flow id'. This represents the combination of a network address and quality of service parameters that identifies a coherent stream of data flowing to a particular device, with a particular set of timing and reliability guarantees. This may be implemented as is an index to a table of timing, routing, and priority that has been engineered considering the traffic requirements of an entire on-board configuration.

If operation of a device requires multiple differing sets of such parameters, that would be multiple flows, and so multiple access points. In datasheet terms, this would be multiple interface definitions, for example for auxiliary telemetry from a different address, or high priority commands which should be delivered out of sequence.

The following list is a summary of the procedure by which a tool chain can generate a concrete implementation to access a device across on-board subnetworks.

* A SEDS SAP package file instance for a service access point to an on-board subnetwork describes the abstract interface. The description in this package file consists of the parameters and commands of the interface, e.g. sendCommand or receiveData. Each parameter and command argument in this description may be:
  + an abstract placeholder for an item of data that varies according to the device model, or
  + a concrete data type that is specific to the protocol of the service access point, such as a memory address.
* A SEDS device data sheet instance for a particular device model maps concrete types of data to the placeholders in the SAP package file. The data sheet includes parameter and command primitives that apply the concrete data types of the device to the data that will flowing through the service access point when the device is in operation.
* The tool chain author maps the agency’s concrete implementation of the service access point to the corresponding parameters and commands in the SEDS package file instance that describes the abstract service access point.
* The tool chain generates the implementation of the device data sheet in the form of a device service, which exposes the access and/or application interfaces provided in the data sheet for use by applications in a form appropriate to the architecture of the system.

All decisions as to how the device should fit within the software architecture of the system should be made by these generation tools, or their configuration. They should not be treated as specified by the data sheet, which will commonly be prepared without knowledge of the system architecture. For example, the exposed interfaces:

* + may allow applications to specify flow-id’s, or their equivalent, explicitly, or they may control the use of flow-id’s by a project-specific algorithm or table.
  + may support polling and/or update notification.
  + may be based on tables and/or functions. If based on functions,they may declare one function for each command/parameter, or a single function for all.
  + may replace each parameter by corresponding get/set commands
  + may be represented in a specific programming language, or in some locally-chosen inter-language communications mechanism.

The same considerations apply if the device service is coded manually, as opposed to being generated by a tool.

#### Simple Generic Types

Below is an excerpt from the example file ccsds.sois.subnetwork.xml, which shows the abstraction of the packet service interface provided by subnetwork by means of generic types.

<Package name="CCSDS/SOIS/Subnetwork">

<DataTypeSet>…</DataTypeSet>

<DeclaredInterfaceSet>

<Interface name="PSInterfaceType" level="subnetwork" shortDescription="SOIS SUBNETWORK PACKET SERVICE from 851.0-M-1">

<GenericTypeSet>

<GenericType name="SendDataType" />

<GenericType name="ReceiveDataType" />

<GenericType name="FailureMetadataType" />

</GenericTypeSet>

<CommandSet>

<Command name="send" mode="async" …>

<Argument name="data" type="SendDataType" mode="in" dataUnit="true" />

</Command>

<Command name="receive" mode="async" …>

<Argument name="data" type="ReceiveDataType" mode="out" dataUnit="true" />

</Command>

<Command name="failure" mode="async" …>

<Argument name="failureMetadata" type="FailureMetadataType" mode="out"/>

</Command>

</CommandSet>

</Interface>

…

</DeclaredInterfaceSet>

</Package>

In the example above, three generic types are defined simply by naming them. Each of the names of generic data types appears later as the type attributes in command arguments for packets. These names are the abstractions in the interface definitions, which make it possible to reuse the same interface definition for different vehicles that use different packet formats. The package name concatenated to the interface name, “CCSDS/SOIS/Subnetwork/PSInterfaceType”, is the fully qualified name by which other instances of SEDS can refer to the packet service interface defined in this instance of SEDS.

Below is an excerpt from the example file jena\_star\_tracker.xml, which defines a concrete realization of the abstract interface above. The realization occurs by mapping the generic types above to concrete types below.

<Component name="DSAP">

<ProvidedInterfaceSet>…</ProvidedInterfaceSet>

<RequiredInterfaceSet>

<Interface type="CCSDS/SOIS/Subnetwork/PSInterfaceType" name="subnetworkPS">

<GenericTypeMapSet>

<GenericTypeMap type="JENA/PACKET\_UTILISATION/HDR\_TM\_PKT" name="ReceiveDataType"/>

<GenericTypeMap type="JENA/PACKET\_UTILISATION/HDR\_TC\_PKT" name="SendDataType"/>

</GenericTypeMapSet>

</Interface>

…

</RequiredInterfaceSet>

…

</Component>

The <Component> in this example corresponds to the Device-Specific Access Protocol of SOIS. The part of the component that is visible above shows a required interface that uses a fully-qualified name to refer to the packet service interface in a separate instance of SEDS. The <GenericTypeMapSet> for this reference contains <GenericTypeMap> elements that associate with each abstract name, a concrete type defined in the “JENA/PACKET\_UTILIZATION” instance of SEDS.

If other types of packets appear in this interface, the software generated by the local tool chain should treat that as an error condition.

#### Alternate Generic Types

Below is an excerpt from the example file ccsds.sois.subnetwork.xml, which shows the abstraction of the memory access service interface provided by subnetwork by means of generic types.

<Package name="CCSDS/SOIS/Subnetwork">

<DataTypeSet>

<IntegerDataType name="Octet">

<Range>

<MinMaxRange min="0" max="255" rangeType="inclusiveMinInclusiveMax" />

</Range>

</IntegerDataType>

<IntegerDataType name="MemoryIDType">

<Range>

<MinMaxRange min="0" max="65535" rangeType="inclusiveMinInclusiveMax" />

</Range>

</IntegerDataType>

<IntegerDataType name="MemoryAddressType">

<Range>

<MinMaxRange min="0" max="4294967295" rangeType="inclusiveMinInclusiveMax" />

</Range>

</IntegerDataType>

<EnumeratedDataType name="StartStop">

<EnumerationList>

<Enumeration label="Start" value="0" />

<Enumeration label="Stop" value="1" />

</EnumerationList>

</EnumeratedDataType>

</DataTypeSet>

<DeclaredInterfaceSet>

…

<Interface name="MASInterfaceType" level="subnetwork" shortDescription="SOIS SUBNETWORK MEMORY ACCESS SERVICE from 852.0-M-1">

<GenericTypeSet>

<GenericType name="ReadMemoryIDType" baseType="MemoryIDType" />

<GenericType name="ReadMemoryAddressType" baseType="MemoryAddressType" />

<GenericType name="ReadDataType" />

<GenericType name="WriteMemoryIDType" baseType="MemoryIDType" />

<GenericType name="WriteMemoryAddressType" baseType="MemoryAddressType" />

<GenericType name="WriteDataType" />

<GenericType name="RmwMemoryIDType" baseType="MemoryIDType" />

<GenericType name="RmwMemoryAddressType" baseType="MemoryAddressType" />

<GenericType name="RmwDataType" />

</GenericTypeSet>

<CommandSet>

<Command name="read" mode="sync" …>

<Argument name="memoryID" type="ReadMemoryIDType" mode="inout" />

<Argument name="memoryAddress" type="ReadMemoryAddressType" mode="inout" />

<Argument name="data" type="ReadDataType" mode="out" dataUnit="true" />

</Command>

<Command name="write" mode="sync" …>

<Argument name="memoryID" type="WriteMemoryIDType" mode="in" />

<Argument name="memoryAddress" type="WriteMemoryAddressType" mode="in" />

<Argument name="data" type="WriteDataType" mode="in" dataUnit="true" />

</Command>

<Command name="unacknowledgedWrite" mode="async" …>

<Argument name="memoryID" type="WriteMemoryIDType" mode="in" />

<Argument name="memoryAddress" type="WriteMemoryAddressType" mode="in" />

<Argument name="data" type="WriteDataType" mode="in" dataUnit="true" />

</Command>

<Command name="readModifyWrite" mode="sync" …>

<Argument name="memoryID" type="RmwMemoryIDType" mode="in" />

<Argument name="memoryAddress" type="RmwMemoryAddressType" mode="in" />

<Argument name="mask" type="RmwDataType" mode="in" />

<Argument name="data" type="RmwDataType" mode="inout" dataUnit="true" />

</Command>

<Command name="unacknowledgedReadModifyWrite" mode="async" …>

<Argument name="memoryID" type="RmwMemoryIDType" mode="in" />

<Argument name="memoryAddress" type="RmwMemoryAddressType" mode="in" />

<Argument name="mask" type="RmwDataType" mode="in" />

<Argument name="data" type="RmwDataType" mode="in" dataUnit="true" />

</Command>

</CommandSet>

</Interface>

…

</DeclaredInterfaceSet>

</Package>

In the example above, some generic types are defined simply by naming them. Each of the names of generic data types appears later as the type attributes in command arguments. These names are the abstractions in the interface definitions, which make it possible to reuse the same interface definition for different vehicles that use different packet formats. The package name concatenated to the interface name, “CCSDS/SOIS/Subnetwork/MASInterfaceType”, is the fully qualified name by which other instances of SEDS can refer to the memory access service interface defined in this instance of SEDS.

Below is an excerpt from the example file jena\_star\_tracker.xml, which defines a concrete realization of the abstract interface above. The realization occurs by mapping the generic types above to concrete types below. In this case, there are alternate mappings.

<Component name="DSAP">

<ProvidedInterfaceSet>…</ProvidedInterfaceSet>

<RequiredInterfaceSet>

…

<Interface type="CCSDS/SOIS/Subnetwork/MASInterfaceType" name="subnetworkMAS">

<GenericTypeMapSet>

<GenericTypeMap fixedValue="0" type="CCSDS/SOIS/Subnetwork/MemoryAddressType" name="ReadMemoryAddressType"/>

<AlternateSet>

<Alternate>

<GenericTypeMap fixedValue="1" type="CCSDS/SOIS/Subnetwork/MemoryIDType" name="ReadMemoryIDType"/>

<GenericTypeMap type="TM\_HEALTH\_MESSAGE\_OVER\_MILBUS" name="ReadDataType"/>

</Alternate>

<Alternate>

<GenericTypeMap fixedValue="2" type="CCSDS/SOIS/Subnetwork/MemoryIDType" name="ReadMemoryIDType"/>

<GenericTypeMap type="TM\_STR\_IDENTIFICATION\_OVER\_MILBUS" name="ReadDataType"/>

</Alternate>

…

</AlternateSet>

</GenericTypeMapSet>

</Interface>

</RequiredInterfaceSet>

…

</Component>

The <Component> in this example corresponds to the Device-Specific Access Procedure of SOIS. The part of the component that is visible above shows a required interface that uses a fully-qualified name to refer to the memory access service interface in a separate instance of SEDS. The <GenericTypeMapSet> for this reference contains some <GenericTypeMap> elements that associate with some of the abstract names, a concrete type defined in the “CCSDS/SOIS/Subnetwork” instance of SEDS, and a fixed value of that concrete type. The other <GenericTypeMap> elements associate other abstract names with a concrete data type defined in this instance of SEDS.

So the first <Alternate> element above says that when ReadMemoryIDType has value 1, then the ReadDataType will be “TM\_HEALTH\_MESSAGE\_OVER\_MILBUS”. The second <Alternate> element above says that when ReadMemoryIDType has value 2, then the ReadDataType will be TM\_STR\_IDENTIFICATION\_OVER\_MILBUS”. If other types of containers appear in this interface, the software generated by the local tool chain should treat that as an error condition.

#### Handling Errors Detected in the Subnetwork

The tool chain may generate code to respond to exceptional events from the subnetwork, treating the events as parameters or commands that drive a state machine. A primitive event marked with the attribute failed=”true” indicates an explicit point where error conditions may be detected. Receipt of an event for which there is no matching transition from the current state, can be treated as implicit detection of an error condition. A manufacturer’s device SEDS would not contain such a state machine, because the state machine would be defined by the agency that implements the tool chain, and because the exceptional events produced in the subnetwork would be defined and implemented by the agency that implements the tool chain.

### List or Array

It is possible to define an array data type, but it is also possible to define a list inside a container. What is the difference? The difference is in the specification of the size of the array or list. For an array, the size is specified in the SEDS that defines it. For a list, the size is specified by one of the entries in the container where the list appears. So an array has a fixed size at run time, and a list has a size that can vary from instance to instance of a container at run time.

A definition of a list from the example file jena\_star\_tracker.xml appears below. The example defines a PUS packet for a telecommand that loads a list of bytes as a patch.

<ContainerDataType baseType="JENA/PACKET\_UTILISATION/HDR\_TC\_PKT" name="TC\_LOAD\_PATCH">

<ConstraintSet>

<ValueConstraint value="14" entry="serviceSubType"/>

<ValueConstraint value="221" entry="serviceType"/>

</ConstraintSet>

<EntryList>

<Entry type="CCSDS/SOIS/SEDS/UINT16" name="byteCount">

<ValidRange>

<MinMaxRange min="0" max="65527" rangeType="inclusiveMinInclusiveMax"/>

</ValidRange>

</Entry>

<ListEntry listLengthField="byteCount" type="CCSDS/SOIS/SEDS/UINT8" name="data"/>

</EntryList>

</ContainerDataType>

The list is a series of data items defined by the <ListEntry> element. In this example, the data items are unsigned 8-bit integers. The count of data items in the series is in the entry of the container named “byteCount”, specified by the listLengthField attribute of the <ListEntry> element. The container entry named “byteCount” is a 16-bit unsigned integer. To allow for software to interpret a container, the listLengthField must appear earlier in the container than the list that it constrains.

A definition of an array from the example file jena\_star\_tracker.xml appears below. The example defines a PUS packet for a telecommand that carries a fixed-length array of guide star indices.

<ContainerDataType baseType="JENA/PACKET\_UTILISATION/HDR\_TC\_PKT" name="TC\_LOAD\_GSC\_INDICES">

<ConstraintSet>

<ValueConstraint value="441" entry="packetDataLength"/>

<ValueConstraint value="13" entry="serviceSubType"/>

<ValueConstraint value="221" entry="serviceType"/>

</ConstraintSet>

<EntryList>

<Entry type="CCSDS/SOIS/SEDS/BOOLEAN" name="accumulate"/>

<Entry type="CCSDS/SOIS/SEDS/UINT16" name="byteOffset">

<IntegerDataEncoding encoding="unsigned" sizeInBits="15"/>

</Entry>

<Entry type="CCSDS/SOIS/SEDS/UINT16" name="byteCount"/>

<Entry type="guideStarCatalogIndex" name="guideStarIndex">

<ArrayDimensions>

<Dimension size="216"/>

</ArrayDimensions>

</Entry>

</EntryList>

</ContainerDataType>

The array is a series of data items defined by the <Entry … name=”guideStarIndex”> element. In this example, the data items are a data type called “guideStarCatalogIndex”. The count of data items in the series is 216, specified by the size attribute of the <Dimension> element. The container has an entry named “byteCount”, which is a 16-bit unsigned integer, but this entry does not control the size of the array. There are always 216 items in the array. The “byteCount” could be used to indicate that only the first n items contain useful data, and the remaining elements of the array would just be dead space.

### Containers that Contain their Length

Packets often have their length as one of their data fields. Often the length is measured in octets, but sometimes there is a different unit of measure. SEDS makes it possible to state how to convert the length entry in a container into the length in octets of the container.

The CCSDS space packet primary header [10] provides an example of length calibration below. The example is taken from the file ccsds.spacelink.packet.xml, and it has been edited to leave only the details that concern the calibration of the length entry.

<ContainerDataType abstract="true" name="PRIMARY\_HEADER">

<EntryList>

<FixedValueEntry fixedValue="0" type="CCSDS/SOIS/SEDS/UINT8" name="packetVersionNumber" …>

…

<IntegerDataEncoding encoding="unsigned" sizeInBits="3"/>

</FixedValueEntry>

<Entry type="CCSDS/SOIS/SEDS/UINT8" name="packetType" …>

…

<IntegerDataEncoding encoding="unsigned" sizeInBits="1"/>

</Entry>

<Entry type="CCSDS/SOIS/SEDS/BOOLEAN" name="secondaryHeaderFlag" …>

…

</Entry>

<Entry type="CCSDS/SOIS/SEDS/UINT16" name="apid" …>

…

<IntegerDataEncoding encoding="unsigned" sizeInBits="11"/>

</Entry>

<Entry type="CCSDS/SOIS/SEDS/UINT8" name="sequenceFlags" …>

…

<IntegerDataEncoding encoding="unsigned" sizeInBits="2"/>

</Entry>

<Entry type="CCSDS/SOIS/SEDS/UINT16" name="packetSequenceCount" …>

…

<IntegerDataEncoding encoding="unsigned" sizeInBits="14"/>

</Entry>

<LengthEntry type="CCSDS/SOIS/SEDS/UINT16" name="packetDataLength" …>

…

<PolynomialCalibrator>

<Term coefficient="7.0" exponent="0"/>

<Term coefficient="1.0" exponent="1"/>

</PolynomialCalibrator>

</LengthEntry>

</EntryList>

</ContainerDataType>

In the example above, the <LengthEntry> element specifies the number of octets in the packet, not counting the header, and decremented by 1, according to the CCSDS Recommendation for Space Packet Protocol.

The SEDS above describes only the packet primary header, so the <ContainerDataType> element specifies that the data type is abstract. Being abstract means that the data type is incomplete, and a derived data type is needed to define a container that can represent a complete container. The length that will appear for the length entry in a concrete packet is the sum of the length of the header and the length of the additional entries defined by the derived container data type.

In SEDS containers, the length entry must be interpreted as the length of the whole container, in octets. The sum of the lengths in the abstract header container is 6 octets. To compute the length of a whole concrete packet from the length entry in the primary header, it is necessary to add 6 for the header and to add 1 for the “less by 1”. This addition gives the coefficient of 7 in the term for exponent 0. The <PolynomialCalibrator> element above says to compute the following expression, where x is the value found in the length entry: x+7.

If no calibrator is present in a <LengthEntry> element, then the interpretation is that the length entry contains exactly the number of octets in the container.

### Array Length by Index Type

There may be some situations in which the size of an array exactly matches the range of possible values of an integer data type. An example of this situation is a data type that has been explicitly tailored for use as an index for the array. In this case, it is possible to specify the size of an array dimension by reference to the integer data type, as shown in the example below.

<ArrayDimensions>

<Dimension indexTypeRef="CCSDS/SOIS/SEDS/UINT8"/>

</ArrayDimensions>

The dimension element above refers to the package named “CCSDS/SOIS/SEDS” (See the package file “ccsds.sois.seds.xml”.), which defines an integer data type named “UINT8”. That data type has a maximum value of 255, and the minimum value of that data type is 0. So, the size of the array is 1 plus maximum minus minimum, or 256.

A consequence of determining the size of an array from the upper and lower bounds of an index data type is the use of negative integer values as indices. When the size of a dimension is stated explicitly through a size attribute of the Dimension element, the index is zero-based. When the indexTypeRef attribute is used, then the first element of the array is indexed by the minimum value of the referenced type.

See Section 3.9.3 in the SEDS blue book [3].

### Overriding Types in Declared Interfaces

When an author of a SEDS is describing a declared interface, there may be a parameter or an argument of a command whose type almost agrees with a type of data defined elsewhere, except for a few properties. The example below concerns an integer argument of a command in the jena\_star\_tracker.xml file. A close definition of the integer type appears in the ccsds.sois.seds.xml file, as reproduced below.

<IntegerDataType name="UINT16" shortDescription="Dimensionless unsigned Integer">

<IntegerDataEncoding encoding="unsigned" sizeInBits="16"/>

<Range>

<MinMaxRange min="0" max="65535" rangeType="inclusiveMinInclusiveMax"/>

</Range>

</IntegerDataType>

The integer data encoding above agrees with the command argument, but the range of values in the definition above is too wide. The jena\_star\_tracker.xml file contains the following definition of a command argument that uses the definition above with an adjustment to the range of values.

<Argument mode="in" type="CCSDS/SOIS/SEDS/UINT16" name="code" shortDescription="coding number">

…

<ValidRange>

<MinMaxRange min="0" max="2047" rangeType="inclusiveMinInclusiveMax"/>

</ValidRange>

</Argument>

Using a validity range makes this internal constraint visible to a user of the interface, allowing the value to be checked before sending the command. When generating the implementation of this interface, a tool chain may generate software that verifies that the argument is in its valid range. Alternatively, if this kind of validity checking is unwanted at run time, then the tool chain can skip those checks.

### Acquiring Telemetered Data

This section tells how to write the part of a device SEDS for collecting telemetry from the device and presenting that data in an onboard service interface. The basic structure of telemetry collection follows the diagram in Figure 4‑3, so this section will be talking about the transfer of data across interfaces, and from one interface of a component to another interface of the same component.

One of the fundamental actions performed by a component is to accept data at a required interface, and to deliver a result computed from that data at a provided interface. A special subset of that class of fundamental actions consists of copying items of telemetry data from a required interface to a provided interface. SEDS provides ways to describe that special subset simply or in detail.

The simple description is called a parameter map. There is a slightly more complex version of a parameter map, called a parameter activity map, in which some processing can be applied to the data as it is copied, such as calibration. See Section 3.14 in the SEDS blue book [3]. The detailed description consists of activities and state machines. See Section 3.15 and 3.16 in the SEDS blue book [3].

An interface provided by an onboard device service presents data from a device, hiding the complexity of network protocols and special behavioral treatments required by the device.

For simple devices, this service interface can often be accomplished by means of a parameter map in the component that implements the device-specific access protocol (DSAP). A functional interface generally presents data in engineering units, so a parameter activity map can be used to perform that conversion in the component that implements the device abstraction control procedure (DACP). See Figure 4‑3 for the concepts of DSAP and DACP.

Before showing an example of the simple description in Section 4.3.12.3, we first show the complexity that the simple description can sometimes replace in Section 4.3.12.2. The techniques in the latter section can be applied to write a SEDS for a complex device. Three interfaces will be crossed in those two sections, and the example data that crosses those interfaces is a telemetry packet from a star tracker.

#### Describing the Interfaces

The part of SEDS that is common to both simple and complex components is the description of their interfaces. This section tells how to write the parts of a device SEDS that describes those interfaces.

The telemetry packet must first cross the interface provided by the subnetwork, where it is simply a packet of data. The subnetwork interface is defined in the package file named “ccsds.sois.subnetwork.xml”. An excerpt from that file appears below, showing only the part of the interface that is relevant for telemetry.

<PackageFile …>

<Package name="CCSDS/SOIS/Subnetwork">

…

<DeclaredInterfaceSet>

<Interface name="PSInterfaceType" level="subnetwork" …>

<GenericTypeSet>

…

<GenericType name="ReceiveDataType" />

…

</GenericTypeSet>

<CommandSet>

…

<Command name="receive" mode="async" …>

<Argument name="data" type="ReceiveDataType" mode="out" dataUnit="true" />

</Command>

…

</CommandSet>

</Interface>

…

</DeclaredInterfaceSet>

</Package>

</PackageFile>

The name of the subnetwork interface is PSInterfaceType, and the telemetry packet arrives through an asynchronous indication represented by a command argument in this example. The command argument is marked as a protocol data unit by the dataUnit attribute. A parameter could have been used here more concisely, since it is implicitly a protocol data unit in this context, but the SEDS syntax for commands works, too.

The generic type named “ReceiveDataType” is a handle for connecting the subnetwork indication data to the actual data type in a DSAP required interface of a SEDS for a device. Any device that produces telemetry as packets can use the interface declaration above in its SEDS and tailor it to its needs by means of the “ReceiveDataType” handle.

An excerpt from the jena\_star\_tracker.xml SEDS appears below, showing the elements that are relevant to the DSAP side of the subnetwork interface. The subnetwork interface declared above is referenced in the context of the DSAP as a required interface below. The reference below is “CCSDS/SOIS/Subnetwork/PSInterfaceType”, which concatenates the package name above with the interface name above.

<DataSheet …>

…

<Package name="JENA/AS400">

<DataTypeSet>

…

<ContainerDataType baseType=”JENA/PACKET\_UTILISATION/PUS\_HK\_REPORT” name=”TM\_ADB”>

<ConstraintSet>…</ConstraintSet>

<EntryList>

…

<Entry type="CCSDS/SOIS/SEDS/INT16" name="rateX" unit="radianPerSecond">

<PolynomialCalibrator>

<Term coefficient="0.027976454840372228" exponent="1"/>

</PolynomialCalibrator>

</Entry>

…

</EntryList>

…

</ContainerDataType>

…

</DataTypeSet>

…

<ComponentSet>

<Component name="DSAP">

…

<RequiredInterfaceSet>

<Interface type="CCSDS/SOIS/Subnetwork/PSInterfaceType" name="subnetworkPS">

<GenericTypeMapSet>

<GenericTypeMap type="JENA/PACKET\_UTILISATION/HDR\_TM\_PKT" name="ReceiveDataType"/>

…

</GenericTypeMapSet>

</Interface>

…

</RequiredInterfaceSet>

…

</Component>

…

</ComponentSet>

</Package>

</DataSheet>

The “ReceiveDataType” handle appears above in a generic type map that associates the handle with a specific type “JENA/PACKET\_UTILISATION/HDR\_TM\_PKT”, which is defined in file “jena.packet.utilisation.xml” as an abstract type. The HDR\_TM\_PKT abstraction is realized by concrete types in the “jena\_star\_tracker.xml” file, one of which appears above. The concrete type that we will follow in this section appears above as a container data type named “TM\_ADB”. The concrete type description contains a constraint set, which tells how to recognize a packet of that type. The concrete type description also contains a list of entries, of which we show only one, named “rateX”, which we will follow later in this section.

At this point, the description above tells how a “TM\_ADB” packet from the star tracker appears at the subnetwork interface required by the DSAP. Section 4.3.12.2 describes what the DSAP does explicitly to move the data in that packet to the interface provided by the DSAP. Section 4.3.12.3 provides a simpler implicit description of the same thing. This section skips those details and steps directly to the interface provided by the DSAP, shown below in a slightly different excerpt from the “jena\_star\_tracker.xml” file.

<DataSheet …>

…

<Package name="JENA/AS400">

<DataTypeSet>…</DataTypeSet>

<DeclaredInterfaceSet>

<Interface abstract="false" level="access" name="DSAI"…>

<ParameterSet>

…

<Parameter mode="async" readOnly="true" type="TM\_ADB" name="TM\_ADB"/>

…

</ParameterSet>

<CommandSet>…</CommandSet>

</Interface>

…

</DeclaredInterfaceSet>

<ComponentSet>

<Component name="DSAP">

<ProvidedInterfaceSet>

<Interface type="JENA/AS400/DSAI" name="dsai"/>

</ProvidedInterfaceSet>

<RequiredInterfaceSet>…</RequiredInterfaceSet>

…

</Component>

…

</ComponentSet>

</Package>

</DataSheet>

The data sheet excerpt above declares an interface named DSAI, which can asynchronously present a parameter of type “TM\_ADB”. The DSAP component refers to this interface declaration by name as an interface that it provides. The task of the DSAP component will include moving a packet known as “data” in the subnetwork interface to the DSAI interface declared above, where it will appear as the TM\_ADB parameter.

At this point, the description above tells how a “TM\_ADB” packet from the star tracker appears at the DSAI interface provided by the DSAP. Section 4.3.12.2 describes what the DACP does explicitly to move the data in that packet to the interface provided by the DACP. Section 4.3.12.3 provides a simpler implicit description of the same thing. This part of the text skips those details and steps directly to the interface provided by the DACP, shown below with yet another slightly different excerpt from the “jena\_star\_tracker.xml” file.

<DataSheet …>

…

<Package name="JENA/AS400">

<DataTypeSet>…</DataTypeSet>

<DeclaredInterfaceSet>

…

<Interface abstract="false" level="functional" name="DSFI" …>

<ParameterSet>

…

<Parameter mode="async" readOnly="true" type="CCSDS/SOIS/SEDS/RadianRate" name="TM\_ADB\_rateX"/>

…

</ParameterSet>

<CommandSet>…</CommandSet>

</Interface>

…

</DeclaredInterfaceSet>

<ComponentSet>

…

<Component name="DACP">

<ProvidedInterfaceSet>

<Interface type="JENA/AS400/DSFI" name="dsfi"/>

</ProvidedInterfaceSet>

<RequiredInterfaceSet>

<Interface type="JENA/AS400/DSAI" name="dsai"/>

</RequiredInterfaceSet>

…

</Component>

</ComponentSet>

</Package>

</DataSheet>

The data sheet excerpt above declares an interface named DSFI, which can asynchronously present a parameter of type “CCSDS/SOIS/SEDS/RadianRate” named “TM\_ADB\_rateX”. The DACP component refers to this interface declaration by name as an interface that it provides. The task of the DACP component will include moving an item named rateX from the TM\_ADB packet in the DSAI interface to the DSFI interface declared above, where it will appear as the TM\_ADB\_rateX parameter.

#### Explicit Movement of Data from Required to Provided Interfaces

This section describes in explicit detail the movement of telemetry across the DSAP and DACP components, which is necessary to understand the simple implicit description of the same process in the next section.

##### Crossing the Device-Specific Access Protocol Component

First, the telemetry data crosses the DSAP as described below in an excerpt from the jena\_star\_tracker.xml file, which shows only the implementation of the DSAP component for the TM\_ADB packet.

<DataSheet …>

…

<Package name="JENA/AS400">

…

<ComponentSet>

<Component name="DSAP">

…

<Implementation>

<VariableSet>

…

<Variable type="TM\_ADB" name="working\_tm\_adb"/>

…

</VariableSet>

<ActivitySet>

…

<Activity name="update\_TM\_ADB">

<Body>

<SendParameterPrimitive operation="get" interface="dsai" parameter="TM\_ADB">

<ArgumentValue>

<VariableRef variableRef="working\_tm\_adb"/>

</ArgumentValue>

</SendParameterPrimitive>

</Body>

</Activity>

…

</ActivitySet>

<StateMachineSet>

…

<StateMachine defaultEntryState="Idle" name="ASYNC" …>

<State name="Idle"/>

…

<Transition fromState="Idle" toState="Idle" name="RECV\_TM\_ADB">

<OnCommandPrimitive interface="subnetworkPS" command="receive">

<ArgumentValue name="data" outputVariableRef="working\_tm\_adb"/>

</OnCommandPrimitive>

<Guard>

<TypeCondition>

<FirstOperand variableRef="data"/>

<TypeOperand>TM\_ADB</TypeOperand>

</TypeCondition>

</Guard>

<Do activity="update\_TM\_ADB"/>

</Transition>

…

</StateMachine>

…

</StateMachineSet>

</Implementation>

</Component>

…

</ComponentSet>

</Package>

</DataSheet>

The set of variables used internally by DSAP includes one named “working\_tm\_adb”, which can hold a copy of the TM\_ADB packet taken from the subnetwork interface.

The set of activities that can be executed internally by DSAP includes one named “update\_TM\_ADB”, which sends the content of the “working\_tm\_adb” variable to the parameter named “TM\_ADB” in the interface “dsai”.

The set of state machines, which are always ready to respond to conditions internally in DSAP, includes one named “ASYNC”. The ASYNC state machine responds to indications received from the subnetwork interface. This state machine has a single state named “Idle”, and there are a number of transitions that originate in “Idle” and terminate in “Idle”. One of those transitions appears in the excerpt above. The transition is triggered when data appears on the subnetwork interface and the guard condition is fulfilled. The guard condition checks that the data matches the type named “TM\_ADB”; that is, it checks the constraints on the the “TM\_ADB” container definition. When the transition is triggered, its “OnCommandPrimitive” element responds to the asynchronous “receive” command in the subnetwork interface by placing the data in the command argument into the variable “working\_tm\_adb”. After the command primitive response, the transition executes the activity specified in its “Do” element, which copies the packet to the parameter named “TM\_ADB” in the interface “dsai”.

In summary, a TM\_ADB packet crosses the DSAP component when it appears asynchronously in the required “subnetworkPS” interface by triggering a transition in a state machine by matching the container type constraints, and the transition copies the packet to the provided interface, “dsai”. The essential service provided by DSAP in this example is to identify the packet, and to provide it in the “dsai” interface according to packet type.

##### Crossing the Device Abstraction Control Procedure Component

For many spacecraft, crossing the DSAP component is sufficient for onboard operations. The SEDS for the star tracker also describes the DACP component, but leaves the choice of where to implement that component up to the mission designers. The DACP could be implemented on board the spacecraft or in the mission control center.

The telemetry data crosses the DACP as described below in an excerpt from the jena\_star\_tracker.xml file, which shows only the implementation of the DACP component for the TM\_ADB packet.

<DataSheet …>

…

<Package name="JENA/AS400">

…

<ComponentSet>

…

<Component name="DACP">

…

<Implementation>

<ParameterActivityMapSet>

…

<ParameterActivityMap>

<Provided name="local\_tm\_adb\_ratex" interface="dsfi" parameter="TM\_ADB\_rateX"/>

<Required name="local\_tm\_adb" interface="dsai" parameter="TM\_ADB"/>

<GetActivity>

<Calibration outputVariableRef="local\_tm\_adb\_ratex" inputVariableRef="local\_tm\_adb.rateX">

<PolynomialCalibrator>

<Term coefficient="0.027976454840372228" exponent="1"/>

</PolynomialCalibrator>

</Calibration>

</GetActivity>

</ParameterActivityMap>

…

</ParameterActivityMapSet>

</Implementation>

</Component>

</ComponentSet>

</Package>

</DataSheet>

This representation of the transfer of data across the DACP component is considerably simpler than the transfer of a packet across the DSAP component in the previous section. Here, each item of data that appears in the provided interface “dsfi” is transferred from the required interface “dsai” according to a “ParameterActivityMap” element like the one above. See Sections 3.14.15 through 3.14.21 in the SEDS blue book [3].

The example above specifies that the TM\_ADB packet in the required interface is the source of the transfer. The calibration element in the “GetActivity” element specifies to take the “rateX” entry in the TM\_ADB packet, to multiply it by a specific number, and to put the product into the “TM\_ADB\_rateX” parameter in the provided interface. The names of implicit intermediate variables start with “local\_”; these names can be used in the “GetActivity”.

While a parameter activity map can specify complicated activity, the map above is simply copying an item from a telemetry packet with calibration, and the calibrator in the “GetActivity” element is the same as the calibrator in the definition of the “rateX” entry in the definition of the “TM\_ADB” container data type. This observation suggests that additional simplification is possible, as will be shown in the next section.

#### Implicit Movement of Data from Required to Provided Interface

This section describes the movement of telemetry across the DSAP and DACP components in an implicit convention that is equivalent to the explicit description in the preceding section.

These conventions for simplification are not stated in the SEDS blue book [3], so a manufacturer cannot use them in the SEDS instances for their products that are expected to be used by multiple agencies. However, these conventions could be used within a project for bespoke devices.

##### Implicitly Crossing the Device-Specific Access Protocol Component

The specifications in Section 4.3.12.2.1 can be summarized in the following list.

* There is a telemetry item to be transferred across the DSAP component.
* As a prerequisite, the data type of the telemetry item includes the means to identify it in a subnetwork interface (e.g., the constraints in a container type).
* The declaration of the provided interface of the DSAP component contains a parameter of the data type of the telemetry item.
* Define an intermediate variable of the proper type to hold an instance of the telemetry item.
* Define an activity that copies the telemetry item from the intermediate variable to the parameter in the provided interface.
* Define a state machine with a transition that recognizes the telemetry item in the subnetwork interface and copies it to the intermediate variable.

The list above constitutes a basic routine that can be assumed as the explicit implementation of a more concise expression that appears in the SEDS fragment below. A tool chain would expand the implicit form below. The subnetwork packet interface definition would need to be modified by replacing the command-argument form with the equivalent parameter form for the “data” in the PACKET\_RECEIVE.indication, and in the state machine transitions that trigger on that indication.

<DataSheet …>

…

<Package name="JENA/AS400">

…

<ComponentSet>

<Component name="DSAP">

…

<Implementation>

<VariableSet>

…

<Variable type="TM\_ADB" name="working\_tm\_adb"/>

…

</VariableSet>

<ParameterMapSet>

…

<ParameterMap interface=”dsai” parameter=”TM\_ADB” variableRef=”working\_tm\_adb”/>

<ParameterMap interface=”subnetworkPS” parameter=”data” variableRef=”working\_tm\_adb”/> …

</ParameterMapSet>

…

</Implementation>

</Component>

…

</ComponentSet>

</Package>

</DataSheet>

The SEDS fragment above describes the transfer of a TM\_ADB packet across the DSAP component in an implicit simple form, given the convention of the explicit expansion in Section 4.3.12.2.1. The parameter map transfer occurs when the source data changes, so the transfer from the subnet interface would occur before the transfer to the “dsai” interface. The type of the variable implies the type to be detected by the implicit state machine for transfer from the subnet interface to the variable. Note that by making the state machine implicit, the opportunity to specify timing constraints is lost. See Sections 3.14.7 through 3.14.12 in the SEDS blue book [3].

In a future revision, SEDS might enable in- and out-parameters in “ParameterMap” elements, which would make the variable implicit, so the parameter map can be expressed in a single element.

##### Implicitly Crossing the Device Abstraction Control Procedure Component

The specifications in section 4.3.12.2.2 can be summarized in the following list.

* There is a telemetry item field to be transferred across the DACP component.
* As a prerequisite, the data type of the telemetry item includes the means to convert the field to the type in the provided interface (e.g., the calibrator for an entry in a container type).
* The declaration of the provided interface of the DACP component contains a parameter of the data type of the telemetry item field.
* The declaration of the required interface of the DACP component contains the parameter with the field.

This list constitutes a basic routine that can be assumed as the explicit implementation of a more concise expression that appears in the SEDS fragment below. A tool chain would expand the implicit form below.

<DataSheet …>

…

<Package name="JENA/AS400">

…

<ComponentSet>

…

<Component name="DACP">

…

<Implementation>

<ParameterActivityMapSet>

…

<ParameterActivityMap>

<Provided name="local\_tm\_adb\_ratex" interface="dsfi" parameter="TM\_ADB\_rateX"/>

<Required name="local\_tm\_adb" interface="dsai" parameter="TM\_ADB"/>

<GetActivity>

<Assignment outputVariableRef=”local\_tm\_adb\_ratex”>

<VariableRef variableRef=”local\_tm\_adb.rateX”/>

</Assignment>

</GetActivity>

</ParameterActivityMap>

…

</ParameterActivityMapSet>

</Implementation>

</Component>

</ComponentSet>

</Package>

</DataSheet>

The simplification above consists of making the calibrator implicit. The tool chain implicitly decides to use the calibrator in the assignment when the unit of measure in the source field matches that of the target parameter. The unit of measure appears in a “Semantics” element in the element for the field. An explicit calibrator is needed to change the unit of measure.

In a future revision, SEDS might enable reference to fields in “ParameterMap” elements, which would make it possible to use the more concise “ParameterMapSet” in place of the “ParameterActivityMapSet” above.

### Commanding a Device

This section tells how to write a SEDS to deliver a telecommand to a device. The familiar example of the Jena star tracker appears again here. The device abstraction control procedure (DACP) is omitted in this section, and only the device-specific access protocol (DSAP) is described. See Figure 4‑3.

#### Describing the Interfaces

This section tells how to write the parts of a device SEDS that describes the interfaces through which commands flow. These are the same interfaces that were seen in Section 4.3.12.1, but this section shows the commands in those interfaces.

The command packet must ultimately cross the interface provided by the subnetwork, where it is simply a packet of data. The subnetwork interface is defined in the package file named “ccsds.sois.subnetwork.xml”. An excerpt from that file appears below, showing only the part of the interface that is relevant for telecommands.

<PackageFile …>

<Package name="CCSDS/SOIS/Subnetwork">

…

<DeclaredInterfaceSet>

<Interface name="PSInterfaceType" level="subnetwork" …>

<GenericTypeSet>

<GenericType name="SendDataType" />

…

</GenericTypeSet>

<CommandSet>

<Command name="send" mode="async" …>

<Argument name="data" type="SendDataType" mode="in" dataUnit="true" />

</Command>

…

</CommandSet>

</Interface>

…

</DeclaredInterfaceSet>

</Package>

</PackageFile>

The name of the subnetwork interface is PSInterfaceType, and the command packet is sent through an asynchronous indication represented by a command argument in this example. The command argument is marked as a protocol data unit by the dataUnit attribute.

The generic type named “SendDataType” is a handle for connecting the subnetwork request data to the actual data type in a DSAP required interface of a SEDS for a device. Any device that receives commands as packets can use the interface declaration above in its SEDS and tailor it to its needs by means of the “SendDataType” handle.

An excerpt from the jena\_star\_tracker.xml SEDS appears below, showing the elements that are relevant to the DSAP side of the subnetwork interface. The subnetwork interface declared above is referenced in the context of the DSAP as a required interface below. The reference below is “CCSDS/SOIS/Subnetwork/PSInterfaceType”, which concatenates the package name above with the interface name above.

<DataSheet …>

…

<Package name="JENA/AS400">

<DataTypeSet>

…

<ContainerDataType baseType="JENA/PACKET\_UTILISATION/HDR\_TC\_PKT" name="TC\_ABERRATION">

<ConstraintSet>…</ConstraintSet>

<EntryList>

…

<Entry type="CCSDS/SOIS/SEDS/UINT16" name="apid" …>

<IntegerDataEncoding encoding="unsigned" sizeInBits="11"/>

</Entry>

…

</EntryList>

…

</ContainerDataType>

…

</DataTypeSet>

…

<ComponentSet>

…

<Component name="DSAP">

<RequiredInterfaceSet>

<Interface type="CCSDS/SOIS/Subnetwork/PSInterfaceType" name="subnetworkPS">

<GenericTypeMapSet>

…

<GenericTypeMap type="JENA/PACKET\_UTILISATION/HDR\_TC\_PKT" name="SendDataType"/>

</GenericTypeMapSet>

</Interface>

…

</RequiredInterfaceSet>

…

</Component>

…

</ComponentSet>

</Package>

</DataSheet>

The “SendDataType” handle appears above in a generic type map that associates the handle with a specific type “JENA/PACKET\_UTILISATION/HDR\_TC\_PKT”, which is defined in file “jena.packet.utilisation.xml” as an abstract type. The HDR\_TC\_PKT abstraction is realized by concrete types in the “jena\_star\_tracker.xml” file, one of which appears above. The concrete type that we will follow in this section appears above as a container data type named “TC\_ABERRATION”. The concrete type description contains a constraint set, which tells how to recognize a packet of that type. The concrete type description also contains a list of entries, of which we show only one, named “apid”, which we will follow later in this section.

At this point, the description above tells how a “TC\_ABERRATION” packet for the star tracker appears at the subnetwork interface required by the DSAP. Section 4.3.13.2 describes what the DSAP does explicitly to move the data in that packet from the interface provided by the DSAP. This section skips those details and steps directly to the interface provided by the DSAP, shown below in a slightly different excerpt from the “jena\_star\_tracker.xml” file.

<DataSheet …>

…

<Package name="JENA/AS400">

<DataTypeSet>…</DataTypeSet>

<DeclaredInterfaceSet>

<Interface abstract="false" level="access" name="DSAI"…>

…

<CommandSet>

<Command mode="async" name="TC\_ABERRATION">

<Argument mode="in" type="CCSDS/SOIS/SEDS/UINT16" name="apid" …>

…

<ValidRange>

<MinMaxRange min="0" max="2047" rangeType=inclusiveMinInclusiveMax"/>

</ValidRange>

</Argument>

…

</Command>

…

</CommandSet>

</Interface>

…

</DeclaredInterfaceSet>

<ComponentSet>

<Component name="DSAP">

<ProvidedInterfaceSet>

<Interface type="JENA/AS400/DSAI" name="dsai"/>

</ProvidedInterfaceSet>

<RequiredInterfaceSet>…</RequiredInterfaceSet>

…

</Component>

…

</ComponentSet>

</Package>

</DataSheet>

The data sheet excerpt above declares an interface named DSAI, which can accept a command of type “TC\_ABERRATION”. The DSAP component refers to this interface declaration by name as an interface that it provides. The task of the DSAP component will include constructing a packet of type “TC\_ABERRATION” from the arguments of the “TC\_ABERRATION” command in its provided DSAI interface. The DSAP component will then move the packet to an argument known as “data” in the subnetwork interface.

#### Explicit Movement of Data from Provided to Required Interfaces

This section describes the movement of command data across the DSAP component.

First, the command data crosses the DSAP as described below in an excerpt from the jena\_star\_tracker.xml file, which shows only the implementation of the DSAP component for the TC\_ABERRATION command.

<DataSheet …>

…

<Package name="JENA/AS400">

…

<ComponentSet>

<Component name="DSAP">

…

<Implementation>

<VariableSet>

…

<Variable type="TC\_ABERRATION" name="working\_tc\_aberration"/>

…

</VariableSet>

<ActivitySet>

…

<Activity name="send\_tc\_aberration">

<Body>

<SendCommandPrimitive interface="subnetworkPS" command="send">

<ArgumentValue name="data">

<VariableRef variableRef="working\_tc\_aberration"/>

</ArgumentValue>

</SendCommandPrimitive>

</Body>

</Activity>

…

</ActivitySet>

<StateMachineSet>

…

<StateMachine defaultEntryState="Idle" name="NO\_REPLY" ...>

<State name="Idle"/>

...

<Transition fromState="Idle" toState="Idle" name="RQST\_TC\_ABERRATION">

<OnCommandPrimitive interface="dsai" command="TC\_ABERRATION">

<ArgumentValue name="apid" outputVariableRef="working\_tc\_aberration.apid"/>

...

</OnCommandPrimitive>

<Do activity="send\_tc\_aberration"/>

</Transition>

</StateMachine>

…

</StateMachineSet>

</Implementation>

</Component>

…

</ComponentSet>

</Package>

</DataSheet>

The set of variables used internally by DSAP includes one named “working\_tc\_aberration”, which can hold a copy of the TC\_ABERRATION packet to be sent through the subnetwork interface.

The set of activities that can be executed internally by DSAP includes one named “send\_tc\_aberration”, which sends the content of the “working\_tc\_aberration” variable to the parameter named “data” in the interface “subnetworkPS”.

The set of state machines, which are always ready to respond to conditions internally in DSAP, includes one named “NO\_REPLY”. The NO\_REPLY state machine responds to commands received from the “dsai” interface. This state machine has a single state named “Idle”, and there are a number of transitions that originate in “Idle” and terminate in “Idle”. One of those transitions appears in the excerpt above. The transition is triggered when command “TC\_ABERRATION” is invoked on the “dsai” interface. When the transition is triggered, its “OnCommandPrimitive” element responds to the “TC\_ABERRATION” command in the “dsai” interface by placing the data in the command arguments into the entries of the variable “working\_tc\_aberration”. We show one such argument named “apid” being put into “working\_tc\_aberration.apid”; other arguments fill other entries in the container. After the command primitive, the transition executes the activity specified in its “Do” element, which copies the packet to the parameter named “data” in the interface “subnetworkPS”.

In summary, a TC\_ABERRATION command crosses the DSAP component when it is invoked in the provided “dsai” interface by triggering a transition in a state machine, and the transition constructs a “TC\_ABERRATION” packet from the command arguments and sends the packet to the required interface, “subnetworkPS”. The essential service provided by DSAP in this example is to assemble the packet, and to send it in the “subnetworkPS” interface.

### Interpreting Fixed-Point Binary Integers

The CCSDS time codes contain binary integers that represent fractional seconds. The unit of measure is seconds, but the computation of a number for that unit of measure requires multiplication by the inverse of the maximum value in the range of the integer.

The computation is something that could occur onboard a spacecraft, or its computation could wait until the data has been sent to a mission control center. The choice of when to compute seconds in this situation is parallel to the choice for when to compute engineering units from raw data.

No matter when a mission designer plans to compute fractional seconds, the EDS contains the information necessary for the computation. The essence of the computation is a calibrator, like the one below, which is for a 24-bit CCSDS unsegmented code, without a preamble.

<PolynomialCalibrator>

<Term coefficient="5.9604648328104515558750364705942e-8" exponent="1"/>

</PolynomialCalibrator>

The coefficient is the inverse of the highest unsigned integer that can be represented in 24 bits, and the exponent says that the coefficient should multiply the value of the time code.

There are a couple ways in which a calibrator like the one above can be put into the context of a SEDS. The simplest way is to decorate a data item where it appears in a container type. Often, the data item will appear in a container, so the example below shows a packet utilization standard header with a decorated time stamp.

<ContainerDataType name="PUS\_TM\_TIMESTAMP">

<EntryList>

<Entry name="coarse" shortDescription="Coarse OBT" type="CCSDS/SOIS/SEDS/UINT32"

unit="second" quanityKind="timeQK"/>

<Entry name="fine" shortDescription="Fine OBT" type="CCSDS/SOIS/SEDS/UINT32"

unit="second" quanityKind="timeQK">

<IntegerDataEncoding encoding="unsigned" sizeInBits="24"/>

<ValidRange>

<MinMaxRange max="16777215" min="0" rangeType="inclusiveMinInclusiveMax"/>

</ValidRange>

<PolynomialCalibrator>

<Term coefficient="5.9604648328104515558750364705942e-8" exponent="1"/>

</PolynomialCalibrator>

</Entry>

</EntryList>

</ContainerDataType>

Another place in a SEDS where it might be useful to place a calibrator for a fixed-point binary number is in an activity. The activity could be used in a parameter activity map to convert between the DSAP and the DACP. The example below shows such an activity.

<Implementation>

<VariableSet>

<Variable name="cuc" type="PUS\_TM\_TIMESTAMP"

unit="count" quantityKind="timeQK"/>

<Variable name="time" type="CCSDS/SOIS/SEDS/FLOAT32"

unit="second" quantityKind="timeQK"/>

</VariableSet>

<ActivitySet>

<Activity name="convertCucToTime">

<Body>

<Calibration inputVariableRef="cuc" outputVariableRef="time">

<PolynomialCalibrator>

<Term coefficient="5.9604648328104515558750364705942e-8" exponent="1"/>

</PolynomialCalibrator>

</Calibration>

</Body>

</Activity>

</ActivitySet>

</Implementation>

It is interesting to compare the semantic tags for the data items in the two examples above. In the second example, both the in and out data items are present, so it is possible to show that the input data item is a time quantity kind measured in units of counts, and the output data item is a time quantity kind measured in units of seconds. In the first example, we have only one data item, which represents the raw data; the convention in this case is to show the units after application of the calibrator, instead of the raw units.

### Strings

#### General

The attribute fixedlength="true" indicates a fixed amount of space allocated for a string, specified by the length attribute. The string in that space could vary in length by termination byte or by Pascal length. The content of any of the fixed amount of space after the end of the string is undefined. If there is no termination byte and there is no Pascal length, then the string exactly fills the fixed amount of space.

The attribute fixedlength="false" implies that the string length needs to be determined by one of the methods below. For a string with fixedlength="false" in a container, the offset to the next entry in the container is immediately after the string entry. The length attribute is the maximum number of bytes in the string.

* If the data type is in a container of known length, and it is the only variable-length entry in the container, then the lengths of all fixed-length entries are subtracted from the length of container, and the difference is the length of the string. A LengthEntry can provide the length of the container.
* Pascal length: An integer in the same container with the string can be designated to represent the number of bytes (not necessarily the same as characters) in the string. The advantage of this representation (and of the preceding) is that it can be determined in constant time.
* The space allocated for the string can be scanned, seeking a termination byte. If that byte is present, it is just beyond the end of the string, and it is not counted in the length of the string. If that byte is absent or undefined, and there is no Pascal length, then the string exactly fills the maximum space defined by the length attribute.

After the length of the string is known, then it is possible to interpret the content of the string as characters, according to the encoding attribute.

#### Specifying a Pascal Length

To indicate that an integer in a container contains the length of a string data type in the same container, the following attributes can be inserted into the entry for the integer:

*<… unit="byte" quantityKind="informationQK" subject="…"/>*

The ellipsis in the subject attribute above is replaced with the name of the entry for the string.

# Flight Software

SEDS that describe flight software services are generally simpler than those that describe devices, because the behaviour is often omitted for software services. Note that, in general, algorithms for software services are better expressed in a domain-specific programming language. Interface behavior, such as timing constraints should be explicit in a datasheet. However, software service SEDS have their own kind of complexity, because they may be parameterized for use in different missions.

Subsection 5.1 describes a scenario for usage of SEDS that describe flight software services.

Subsection 5.2 examines an example SEDS for a flight software service.

## Service Interoperability across Projects

Status: Tested for one project; proposed for second project using same SEDS.

See Figure 5‑1. This scenario is similar to the one described in Section 4.1.1, but instead of making a device interoperable on different platforms, this scenario makes a software product interoperable on different platforms. The software product is called a “service” in this context, because its value as an interoperable entity is the service that it can deliver.

The difference between the scenarios is in the means of adaptation. For a device, a “device service” is generated to act as a proxy for the device, hiding communications from the applications that use the device. For a “service”, an adaptation is generated to allow the software to execute and to communicate on a specific platform. In other words, the service is existing and is not generated from the SEDS; what is needed is to embed the service implementation in different platforms, and this step is configured via SEDS. The adaptation may have the form of a wrapper that calls a library function, or it may have the form of a binary object compiled to execute on the onboard platform. In the first case, the wrapper is distinguishable from the library function that it calls; in the second case, the adaptation is indistinguishable from the binary object that provides the service.

EDS for Service

x

FSW

Adapter for x

Service x

generation flow

communication

Figure 5‑1 Service Interoperability

## Service Data Sheet Example

A service data sheet differs from a device data sheet in the following details.

* The primary element of a device data sheet is named DataSheet, but the primary element of a service data sheet is named PackageFile.
* The Metadata element of a device data sheet appears in its Device element, but the Metadata element of a service data sheet appears after the Package element in the PackageFile element.
* A device data sheet may have multiple components. For example, the components in a device data sheet could represent the relationship between an interface with raw data and an interface with engineering units. A service data sheet generally contains a single component.
* A service data sheet describes the interface(s) for a service, but need not describe the implementation of the service.

A service data sheet provided by a third-party software vendor would likely contain “include” elements in the “xi” namespace, as shown in the example in Section 4.3.1, because the vendor wants specific inclusions. Where a service data sheet defines software written within an agency, or within a project, the use of such explicit “include” elements can be eliminated for package files provided within the agency or within the project, because the local tool chain can resolve the references from a file system directory that contains all the relevant package files.

An outline of a SEDS for the CCSDS Asynchronous Message Service (AMS) service appears below.

<PackageFile xmlns=<http://www.ccsds.org/schema/sois/seds>>

The Package element that follows provides a number of interfaces that organize the commands and indications of AMS into functional groups.

<Package name=”ccsds/sis/ams” shortDescription=”Asynchronous Messaging Service”>

The DataTypeSet element that follows defines the structure and content of data that passes through the interfaces defined in this data sheet.

<DataTypeSet>

The ContainterDataType elements that follow define the data that appears in indications of the basic service interface of AMS.

<ContainerDataType name=”FaultType”>…</…>

<ContainerDataType name=”MessageType”>…</…>

</DataTypeSet>

The DeclaredInterfaceSet element that follows defines the commands and parameters of AMS.

<DeclaredInterfaceSet>

The Interface elements that follow divide the commands and parameters of AMS into functional groups. These divisions are not essential to the implementation of AMS; all the commands and parameters could be defined in a single interface element. The divisions along functional boundaries make it easier to direct a tool chain to omit parts of AMS. For example, the private multicast functional group might be omitted for a specific mission.

<Interface name=”BasicService” level=”application”>

The ParameterSet element that follows defines the indications that may appear for an application that uses the basic service interface of AMS.

<ParameterSet>

The Parameter elements that follow name the indications that may appear for an application that uses the basic service interface of AMS, flag them as asynchronous events, and refer to types defined in the DataTypeSet element above.

<Parameter name=”Message” mode=”async” type=”MessageType”/>

<Parameter name=”Fault” mode=”async” type=”FaultType”/>

</ParameterSet>

The CommandSet element that follows defines the commands that may be used by an application that uses the basic service interface of AMS.

<CommandSet>

The Command element that follows names the command that may be used by an application that uses the basic service interface of AMS, and flags it as an asynchronous operation. Argument elements within the command element refer to types defined in the DataTypeSet element above.

<Command name=”Send” mode=”async”>…</…>

</CommandSet>

</Interface>

<Interface name=”SynchronousMessaging”…>…</…>

<Interface name=”Multicast”…>…</…>

<Interface name=”MulticastPrivate”…>…</…>

<Interface name=”MetaAMS”…>…</…>

</DeclaredInterfaceSet>

</Package>

The Metadata element that follows is not expanded for this example.

<Metadata>…</Metadata>

</PackageFile>

A tool chain can use the information above to generate C header files or interface declarations for other languages.

## Access to Component Variables

Often it is useful to monitor internal variables of flight software during execution. An example application is a current values table (CVT), which may be used by a limit checker. To describe this situation, the SEDS instance for a monitored application would contain one or more Component elements; each Component element would contain an Implementation element; and each Implementation element would contain a VariableSet element. The variables in the variable sets describe internal variables that may be monitored.

The method and timing for accessing the variables to be monitored is determined by tool chain and systemn architecture, not by specification in SEDS. When the monitoring application is present in the same processor with the application to be monitored, access is typically by memory access. Between separate processors, messaging would typically be used.

Given the knowledge of this list of variables, and the architecture of the target system, it will be possible for tooling to generate whatever artifacts are required by the target system to support this. This could be any of:

* a simple table of memory addresses and types
* code that registered those variables with a data pool, or other mechanism.
* code that periodically broadcast the current values of those variables as a message.
* code declaring and implementing a corresponding interface and/or implementation.
* a SEDS package file. This would specify an interface to those variables, with one parameter per variable. It could, in addition, specify an implementation whereby the parameters can be read from the variables, or, optionally, the variables can be set from the parameters.
* configuration for an editing tool allowed the default values of those variables to be displayed and set. This could be used to generate a configuration file, or table, read by the application onboard.

# System Design

This section provides an end-to-end walkthrough the development of a Mission illustrating the different use cases where SEDS and Common DoT may participate.

Subsection 6.1 is a summary of a system design process.

Subsection 6.2 explains the parts of the process in more detail.

Subsection 6.3 describes how to write SEDS instances that contain parameters that are shared by all SEDS instances in a project.

## Overall Data-Flow/Process

A summary of some representative use cases for electronic data sheets appears in Figure 6‑1. More use cases could be added, but this set is representative of the diverse opportunities to reduce costs when transferring descriptions of components through a mission development process.

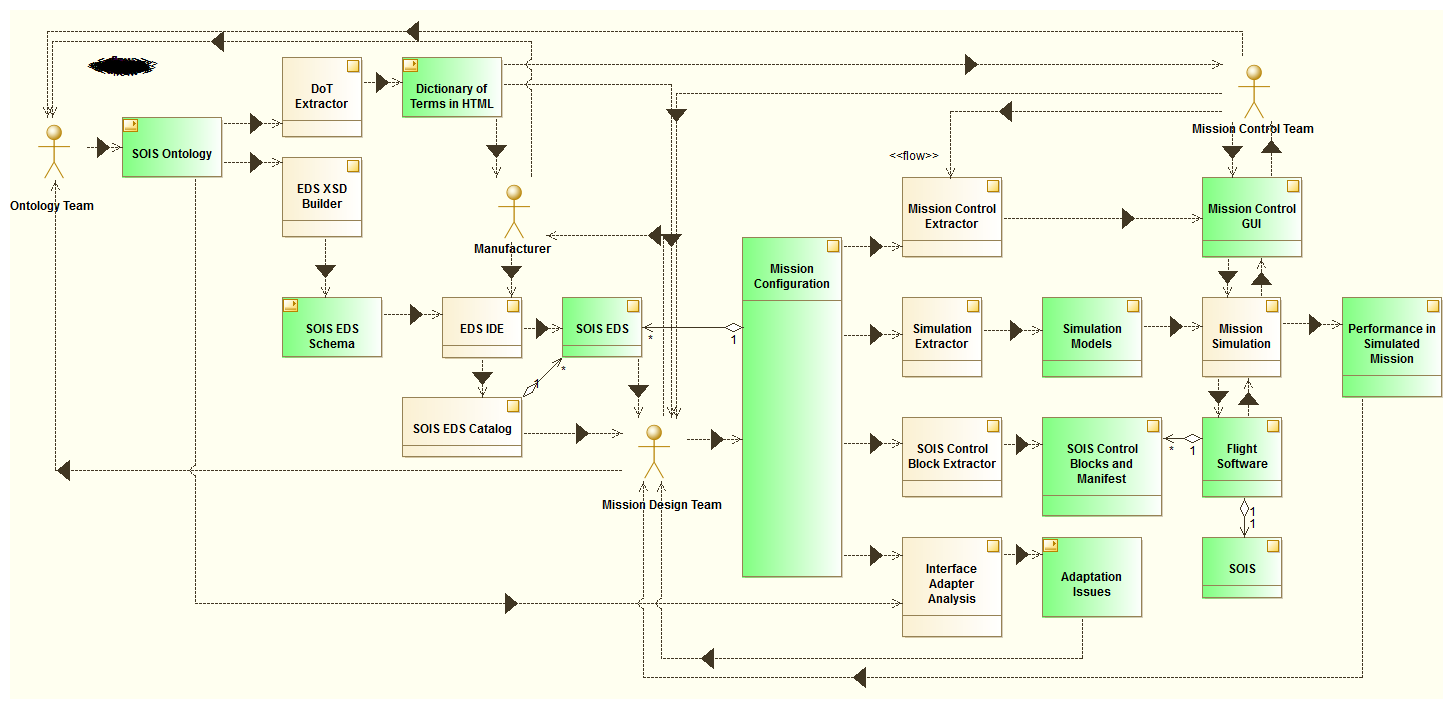


Figure 6‑1 Some Use Cases for Electronic Data Sheets.

The processes in Figure 6‑1 are rendered as beige blocks, while the data artifacts are green blocks. Four teams of people appear in Figure 6‑1; three of those teams would appear in a traditional mission development process.

* Manufacturer: This team represents the people who provide parts for the mission vehicles. This may include builders of secondary batteries, builders of star trackers, and writers of attitude control software, among many others. A manufacturer need not be external to the agency that is overseeing the mission; the concept can include in-house instrument builders and software engineers.
* Mission Designers: This team consists of people who decide how to compose parts into the assemblies needed for a mission. Starting with mission objectives, the designers may invent one or more vehicles or networks to accomplish the mission.
* Mission Control: This team includes the people who operate the mission vehicles. These people must track and understand the status of parts of the vehicles, and they must command or guide the vehicles to conduct the mission.
* Ontology Team: This team stays in the background, coming out when necessary to facilitate the use of terms in electronic data sheets. The team can serve the needs of multiple projects by publishing terms in a global location that enables a part to be used in any project, with the same interpretation for its electronic data sheet in each project. Some large projects, or a project with a short time line, may have a local ontologist in order to provide locally-defined terms that drive the project tool chain for parts that do not need to be portable across projects.

The diagram in Figure 6‑1 will be treated piecemeal in section 6.2. The software tools described in that section are expected to develop in response to the availability of consistent descriptions of parts in electronic data sheets. The tools are described conceptually, and it is left to the ingenuity and economics of the industry to implement them.

## Mission Development Phases

This section breaks a mission down into development phases and discusses topics on the use of SEDS and Common DoT, including tooling support. The phases appear in chronological order. The following table distinguishes those phases that have been tested from those that are more speculative.

|  |  |
| --- | --- |
| **Phase** | **Status** |
| 6.2.1 Selecting Parts | speculative |
| 6.2.2 Validating a Mission Configuration | speculative |
| 6.2.3 Generating Flight Software | tested |
| 6.2.4 Mission Control Software | speculative |
| 6.2.5 Simulation | tested |
| 6.2.6 Reality In the Loop | speculative |
| 6.2.7 Device Development | speculative |
| 6.2.8 Maintenance of Terminology | speculative |
| 6.2.8 Generating Documentation | tested |

### Selecting Parts

This section starts where a group of engineers are figuring how to implement a mission. The engineers have gotten to the point of selecting the parts to build a mission vehicle, given some practical constraints that they have derived from mission requirements. In a traditional process, there would be a search of possible suppliers, using various media, including telephones, internet, advertisements received in mail, and memory of prior projects. The sense that no stone has been left unturned is seldom achieved, because this search occupies valuable time and talent and so must often be curtailed.

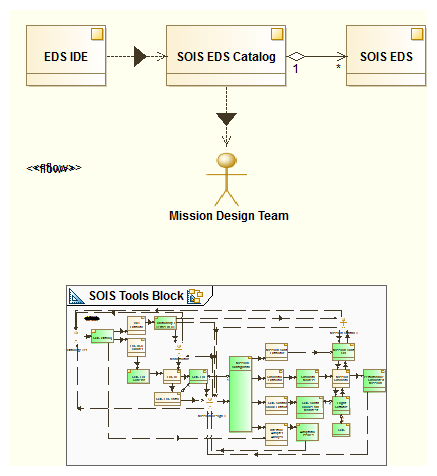


Figure 6‑2 Rapid Search for Candidate Parts

SOIS electronic data sheets can facilitate this search by making each manufacturer’s wares accessible to algorithmic search engines. The manufacturers have the privilege of keeping their SEDS instances confidential while presenting their SEDS instances to the search engines for indexing. The mission designers can rapidly obtain a list of candidate parts by using the practical constraints that they derived from mission requirements as search arguments.

The kind of search engine that uses constraints for search arguments can provide more precise results than the generic search engines that operate on unstructured text arguments to find web pages. The precision is possible due to the fact that electronic data sheets use simple syntax and terms defined in the global ontology.

Some working prototypes of this kind of search engine have been developed, using various techniques to optimize an engineer’s time. Standard constraint solver technology can whittle away at physical resource allocations and reduce other features of the design space as engineers make choices of parts. Randomized search can explore a design space with various combinations of parts until a viable combination is found.

At the time of this writing, the development of electronic data sheets has concentrated on the description of data interfaces. However, prototypical research has been conducted on the description of physical interfaces, including mounting planes, mass properties, power utilization, thermal properties, and radiation requirements. The electronic data sheet syntax allows for a “Metadata” element that can be used by defining terms in the ontology to describe physical interfaces of devices. The physical interface description is important to the process of selecting components, and later in flight simulation.

### Validating a Mission Configuration

When the mission designers have a bill of material, they can work out the connections and relations among the parts, building a mission configuration file. Standard modelling tools, such as MOF, UML, and SysML, can be used to describe the mission configuration in an algorithmically accessible manner. This brings up another activity that occupies the time of engineers on the design team. They must tediously check the mission configuration to be sure that every part has the resources that it needs.

There are two classes of questions about resources. Here are representatives of those classes: Is there a publisher of attitude data on board that the attitude control application can use? Can the solar panels keep the battery charged, given the orbit and the mission operations? The first class of questions can be answered in a straight-forward manner as described in Figure 6‑3. The second class of questions requires a system-wide view of the mission, which will be explained in section 6.2.6.

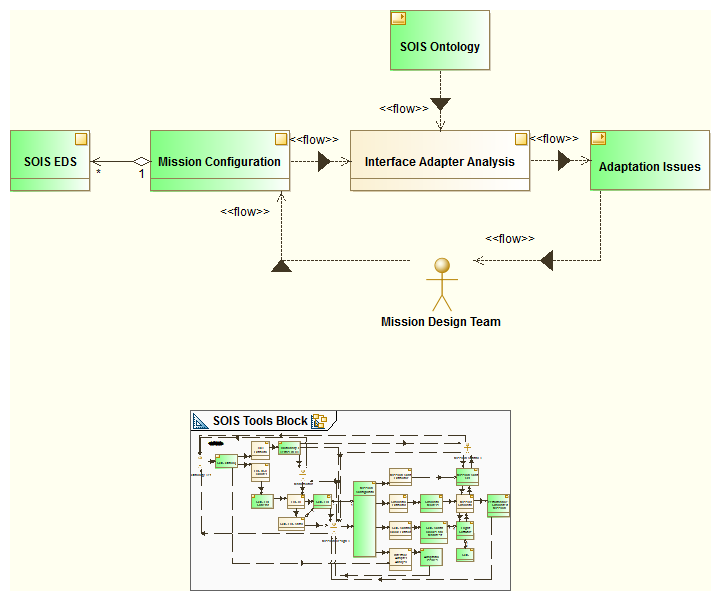


Figure 6‑3 First-Order Validation of Design

The interface adapter analysis tool aids mission designers in validating a design, using the set of electronic data sheets referenced by the mission configuration and their relationships in the configuration. In many cases, the validation of resources for a part is simply done by checking that the terms describing required interfaces in the part’s SEDS match the terms describing provided interfaces in another part’s SEDS.

Sometimes, validation is complicated by partial or cryptic agreement between interfaces. The tool uses the ontology to make the more complicated comparisons. Here is an example. A designer may have chosen an inertial rate unit that presents its data as raw counts from an analog to digital converter, while the attitude control software application requires an interface that provides rates in radians per second. The quantity kind (rotation rate) of the data match, but the units of measure (counts versus radians per second) do not. The electronic data sheet for the inertial rate unit describes a calibrator that transforms the counts to engineering units in the device abstraction control procedure. In case the calibrator produces degrees per second, the ontology describes conversions between engineering units, so the validation algorithm can recognize that a conversion between radians and degrees is needed to make an exact match. The calibrator could be adjusted when generating the driver for the inertial rate unit in section 4.3.5, but for the purpose of validating the mission configuration, it is only necessary to indicate the nature of mismatch.

The interface adapter analysis produces a list of interfaces that are required but not provided in the mission configuration, as well as a list of adaptations that are needed to satisfy weakly matching interfaces. The mission design team treats the report of the interface adapter analysis as a list of unfinished work.

Given a standard syntax and semantics for describing interfaces of parts, it is natural to want to extend the benefits of standardization to the interfaces themselves. Such an extension would greatly simplify the task of design, because the question of matching provided and required interfaces would be reduced to matching the names of standard interfaces. Furthermore, standard interfaces are essential for making parts into commodities, rather than custom-built items, which reduces costs for a project. However, non-standard interfaces are necessary for innovation. SEDS are agnostic to the concept of standard interfaces.

### Generating Flight Software

An electronic data sheet for a device describes how to transform between messages in sub-network layers of a protocol stack and data presented at a functional interface for use by applications. The description is not strongly procedural, so it accommodates a variety of implementations. See section 4.

Figure 6‑4 shows a software tool that extracts SOIS control blocks from the mission configuration and the SEDS that it references.

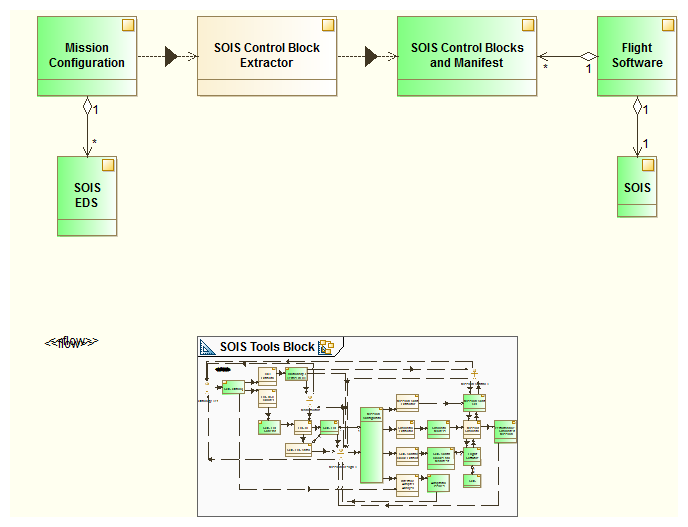


Figure 6‑4 Extracting Flight Software Tables from Mission Configuration

Prototypes for generating device drivers from SEDS have been built for multiple platforms, including a SOIS reference implementation and a flight executive organized around a software bus. See section 5.

### Mission Control Software

Electronic data sheets provide an efficient source for loading mission control databases and for configuring displays. Using the electronic data sheets referenced by a mission configuration, a software tool builds a model of the data that will be visible to controllers in a set of screens. The mission control team guides the tool in selecting the data to be displayed in each screen. The software tool can produce xaml files, or equivalent tables. It also produces the database behind the display, which constitutes the definition of the mission control view for operating the mission. See Figure 6‑5.

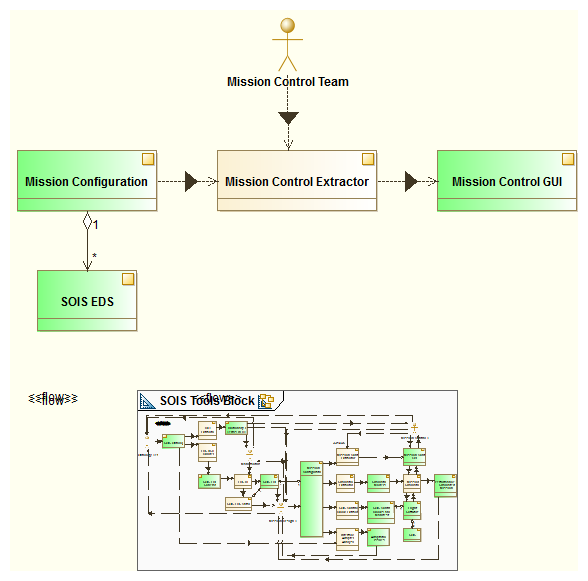


Figure 6‑5 Extracting a Mission Control Database

Two locations may be configured for mission operations decisions. The traditional location is a mission control center on the ground, as described in Section 6.2.4.1. For more autonomous missions, some decisions may be made on board the vehicle, as described in Section 6.2.4.2.

#### Interoperability with Mission Operations

Status: Tested for XTCE.

See Figure 6‑6. A significant part of the work in preparing for a mission is the compilation of a database that describes the transducers and software onboard a vehicle. SEDS can provide the raw material for building such a database.

In this example, two forms of description for mission operations are considered. XTCE describes telecommands and telemetry.

EDS for Device x

EDS for Service y

Device Service for x on A

MAL, XTCE for x

Adapter for y on A

Service y

Vehicle A

FSW for Vehicle A

MOC for Project A

GSW for Project A

MAL, XTCE for y

Device x

generation flow

communication

Figure 6‑6 Interoperability with Mission Operations

#### Interoperability with Spacecraft Monitoring and Control

Status: Proposed.

See Figure 6‑8. In this example, some functions of mission operations have been placed onboard a vehicle; often this is done as a part of a design for some degree of autonomous operation of the vehicle.

EDS for Device x

EDS for Service y

Device Service for x on A

MAL, ASN.1 for x

Adapter for y on A

Service y

Vehicle A

FSW for Vehicle A

SM&C for Vehicle A

MAL, ASN.1 for y

Device x

generation flow

communication

Figure 6‑7 Interoperability with Spacecraft Monitoring and Control

### Simulation

When the design for a mission vehicle has gathered enough detail to be testable, flight simulation can suggest whether the design can be viable in operation. Figure 6‑9 shows a software tool that extracts a simulation model from a mission configuration, including the SEDS referenced by the design.

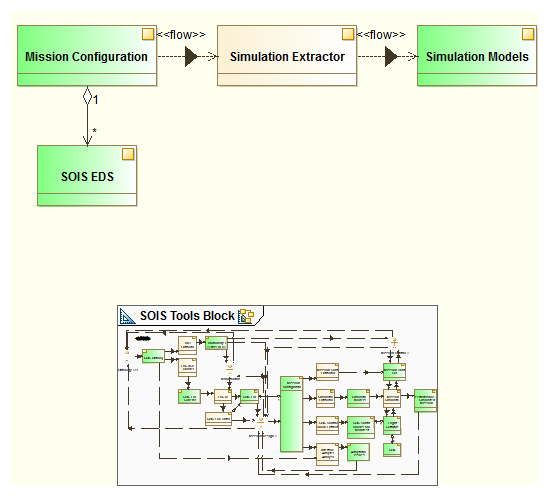


Figure 6‑8 Extracting a Simulation Model from a Mission Configuration

There are at least two levels of simulation models that can exploit SEDS to simulate a day in the life of a mission vehicle. The mission configuration provides the highest level model of a vehicle; it can be converted into a simulation model that represents the parts of a vehicle and the relations among the parts, as described in Section 6.2.5.2. The SEDS referenced by the mission configuration provide parameters for simulation models of parts of the vehicle, which constitute a more detailed level of simulation, as described in Section 6.2.5.1.

#### Simulation Testing of Device Services

Status: Tested for the case of assuming simple transformation of behaviour; proposed for the general case.

This scenario uses a SEDS for a device in two ways within a single project. One usage generates a device service to represent the device for applications in an onboard computer, as shown in the left side of Figure 4‑1. The other usage generates a simulation of the device to provide messages for testing in a flat-sat environment before the manufacturer delivers the physical device, as shown in Figure 6‑9.

Some explanation is needed to assure the reader that this scenario does not constitute a self-fulfilling prediction of a successful test. The diagrams in Figure 4‑1 and in Figure 6‑9 show two kinds of information extracted from a single SEDS instance. One kind of information is the behaviour of the device; the other is the behaviour of the device service. Both behaviours share a common interface across a subnetwork. SEDS describes the features of implementation of the device service (using state machines and activities) that are appropriate to communicate with the device in a manner that is consistent with its protocol and timing.

In general, a separate model is needed to simulate the internal behaviour of the device, but for preliminary testing there is a simple transformation between behavioural models so only the device service behaviour need be defined explicitly. For the cases in which a realistic simulation of the device is needed, there is no simple transformation, and the behaviour of the device must be described in a separate simulation model provided by the manufacturer with the SEDS. For the cases in which a simple transformation is adequate, the device service behaviour defined in the SEDS can be used to derive the device behaviour by simple operations, such as changing “send” to “receive”. The simple transformation can be useful for showing that software can execute an interface before a realistic simulation is available, but once a device or its emulator is available, the realistic simulation use case should be used.

The SEDS defines the features of behaviour of the device, but it does not prescribe a test of that behaviour. Instead, the simulator that implements the behaviour of the device must provide the capability to alter the values of behavioural parameters in order to test the integration of the device service and the device. A bridge may provide the capability to degrade the quality of service for transmission of data between device simulation and device service.

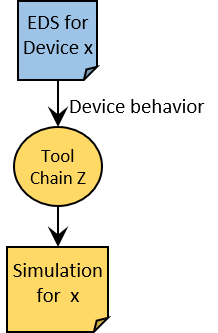


Figure 6‑9 Simulation Testing of Device Service

#### Reality In the Loop

Status: Proposed

Given the consistent description of mission vehicle parts that SEDS provide, it is practical to simulate the mission with real hardware and software parts, and with real people participating as part of the simulation, as shown in Figure 6‑7.

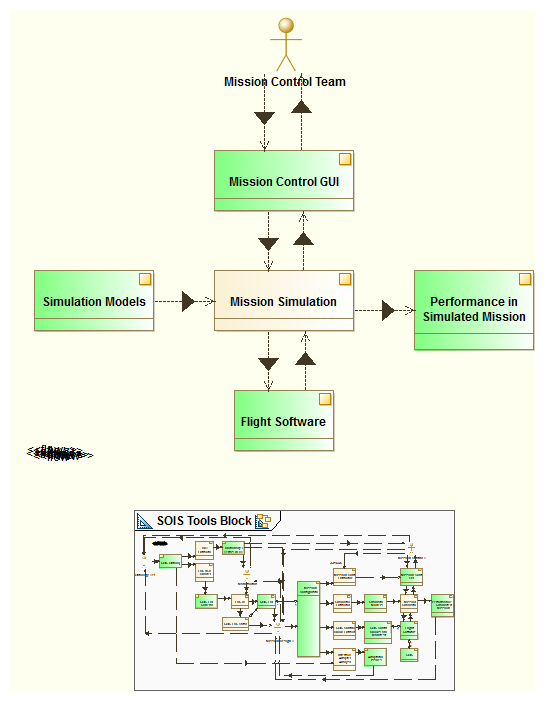


Figure 6‑10 Simulating a Mission with Reality in the Loop

The simulator prototypes described in Section 6.2.5.1 allow for flight software parts or actual device parts to participate actively in simulation. These additional configuration options for simulation are called “flight software in the loop” and “hardware in the loop”. The prototypes also included the mission control center in simulations. These options allow for system-testing of parts before flight, to identify difficulties that could arise during real operation.

### Device Development

The process of designing a vehicle can produce interface control documents, which can be expressed in the form of SEDS. The constraints on interfaces required by other components in the vehicle serve as specifications for the components that have yet to be manufactured.

### Maintenance of Terminology

In addition to the standard machine-readable and human-readable syntax of SEDS, the consistent use of terminology across all SEDS instances is essential for the algorithmic usage of SEDS. Terminology is a moving target, and the DoT relies upon feedback from the users of terms to keep a useful set of terms across time and space, as shown in Figure 6‑8.

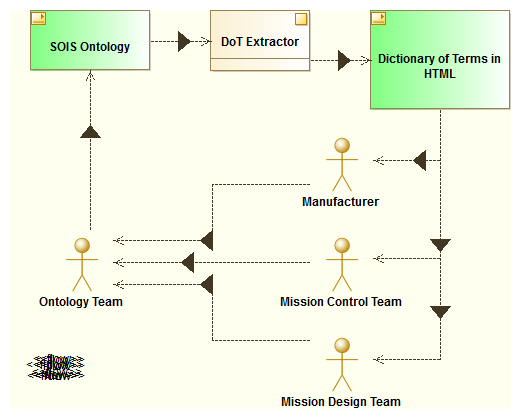


Figure 6‑11 Ontology Feedback Loop

The ontology team must listen, understand, and reconcile a variety of vocabularies and views of the subject matter, rather than to dictate a particular view of the subject matter. The manufacturer may report problems in expressions to the ontology team. The mission controllers may report ambiguities and errors to the ontology team. The designers report issues of expression and ambiguity to the ontology team.

Terms that represent synonyms or alternative factorizations may be necessary:

* e.g., “gyrometer” and “IRU”
* e.g., layered single-processor architecture and distributed message-bus architecture
* e.g., procedural specification and specification of mission objectives

Where synonyms and alternative factorizations become a part of the DoT, it becomes necessary that software tools use the ontology form of the DoT in order to have a clear representation of these relations among terms when interpreting SEDS.

### Generating Documentation

Status: Tested

The production of human-readable artefacts from SEDS files is generally a straightforward process. The Extensible Stylesheet Language technology [6] can often be used for this purpose. Figure 6‑12 is an example in which a style sheet generates an ICD from a device data sheet. If greater control is needed than can be obtained through XSL, the SEDS file could be read into a document object model (DOM), using a general-purpose programming language, and the artefact to be generated could be built from the DOM.

EDS for Device x

ICD for x

Figure 6‑12 An Example of Generating Documentation from SEDS

## Project Parameters

This section describes how SEDS can parameterize a project, simplifying the task of a large category of design changes, in which a parameter is changed.

Subsection 6.3.1 describes the general idea of project parameters.

Subsection 6.3.2 places the project parameters into the context of metadata.

Subsection 6.3.3 is a placeholder for future data sheets that describe the manifest, topology, and schedules of a vehicle.

Subsection 6.3.4 describes SEDS that are tailored for individual instances of hardware from a product line.

### Parametric Data Sheets

In some cases, it is convenient for datasheets to be created in a parameterized form, where they reference externally-sourced data not available at the time they are created. Here are some of the use cases for this feature:

* The flight software for a mission executes on a particular computer platform, which has specific sizes of words that can be handled efficiently by its processor. An author’s time would be best spent writing with general word sizes, expecting that actual word sizes will be inserted (and checked for adequate size) when the computer platform for a mission has been determined.
* The sizes of tables used to manage or configure generic flight software may depends on the requirements of a mission; to reuse an instance of SEDS across missions, this kind of parameter can be inserted after authoring is complete.
* Calibrators for devices generally vary between instances of the device. To make a data sheet that corresponds to a particular instance of a device, the calibration coefficients can be inserted at the same time as insertion of the serial number, after authoring is complete.
* Specific jumpers, DIP switch settings, or other initialization parameters for a device may be specified by the designers of a mission.

#### 6.3.1.1 Parametric Data Sheet Concepts

In order to serve the use cases above, a special syntax may appear in a SEDS for the purpose of deferring the specification of such values until the platform, mission, or other context is known. Such external references are simple named text strings that are processed by the tooling at a very early stage in the parsing process. This allows them to be used in places the XML schema would not normally allow an arbitrary string, and ensures the actual values supplied will be subject to full syntactic and semantic checking.

The source parameterized data sheets contain special strings that indicate where to insert referenced values. In order for these SEDS instances to be used across a variety of tool chains, the syntax of the insertion points is specified as part of the SOIS EDS standard. The syntax uses ${ and } as brackets around the name of the reference. The whole string ${…} is replaced by the value of the reference.

A common capability of data sheet tooling will be to perform that textual substitution, transforming such a parametric datasheet into a normal datasheet, with explicitly specified values for all external references.

#### Parametric Data Sheet Example

An fragment of a project parametric data sheet appears below.

<MetadataValueSet>

<StringValue name=”SIGNED\_INTEGER\_ENCODING” value=”twosComplement”/>

…

</MetadataValueSet>

A fragment of a SEDS into which a parameter above should be inserted appears below.

<IntegerDataType name="int8" shortDescription="Signed 8 bit integer">

<IntegerDataEncoding sizeInBits="8" encoding="${SIGNED\_INTEGER\_ENCODING}"/>

</IntegerDataType>

### Context of Usage

The context of usage of a device or a service is described in a <Metadata> element, which is the sole element in a <Device> element, as outlined below. In a package file, the <Metadata> element appears after the <Package> element.

<Device name=”starTracker”>

<Metadata>

…

</Metadata>

</Device>

The context of usage consists of four optional parts: a model of production, a configuration management section, a model of operation, and parametric data.

#### Model of Production

A model of production describes the provenance of a device or service, as shown below.

<Category name=”Sodern” memberOf=”Manufacturer”/>

<Category name=”Hydra” memberOf=”ManufacturersModel”/>

<MetadataValueSet>

<StringValue name=”serial” value=”123456” memberOf=”SerialNumber”/>

</MetadataValueSet>

</Category>

</Category>

The basic information needed are the name of the manufacturer and the model number. For devices that have individual characteristics, the manufacturer will provide a SEDS for each instance, which contains a serial number in the model of production, and which contains calibration coefficients for the specific instance. These instance SEDS can be generated by substitution of parameters, using a short package file for each device instance and a device model data sheet with substitution points for the parameters. This mechanism makes it unnecessary for most data sheet files distributed with devices to contain unsubstituted parameters.

#### Configuration Management

A configuration management section tracks the development of the content of a SEDS, as shown below.

<Category name=”CM” memberOf=”ConfigurationManagement”/>

<MetadataValueSet>

<StringValue name=”version” value=”0.1.0” memberOf=”SEDSInstanceVersion”/>

</MetadataValueSet>

</Category>

The basic information is the name of the SEDS file and the version number.

#### Model of Operation

A model of operation identifies objects related to the device or service, which must be known in order to interpret elements of data. An example for a hypothetical star tracker appears below.

<Category name=”imager” memberOf=”ModelOfOperation”>

<MetadataValueSet>

<StringValue name=”opticalAxis” value=”hasA”/>

<StringValue name=”focalPlane” value=”hasA”/>

…

</MetadataValueSet>

<Category name=”opticalAxis”>…</Category>

<Category name=”focalPlane”>…</Category>

…

</Category>

The example lists a number of parts of a star tracker, along with simple relations to the device as a whole. More information about a part can be provided in an imbedded <Category> with the same name. The purpose of this structure is to enable a semantic tag on an item of data to refer to part of the device. For example, to say that a particular temperature measurement in a housekeeping packet is the temperature of the focal plane, the following semantic tags would be used.

… subject=”imager.hasA.focalPlane” quantityKind=”temperatureQK” …

If the name “focalPlane” is unique in the model of operation, then the “imager.hasA.” can be omitted. In case there is more than one instance of “focalPlane” in the model of operation, the tree-traversal syntax with “.” separating objects and relationships identifies a particular node in the tree by traversing a path from the trunk to the node.

The structure of the model of operations <Category> is recursive, so a tree graph of any depth can be constructed.

### Deployment Description Data Sheet Place-Holder

This section is a place-holder for deployment description, because the topic has not yet been sufficiently discussed and developed.

### Placement of Instance Data

Earlier sections have described the content of SEDS, and the capability for references to external package files. The content of SEDS includes information about a product line, and information about specific instances of products. This section explains how to organize the content of SEDS so instance data is useful, and so product line data is duplicated minimally.

Instance data may appear in the following parts of a data sheet:

* A calibrator for conversion from raw analogue-to-digital converter counts to engineering units of measure generally contains coefficients that are unique to the particular hardware that produces the counts.
* Information in the model of production (Section 6.3.2.1) such as serial number identifies a particular instance of a device.
* Some construction details of made-to-order devices, such as quantity and placement of thermistors, may be unique for a device instance.
* Mass properties and rotation from mounting bracket to optical axis, which may be present in metadata, are additional examples of instance data.

All instance data is kept together in a SEDS DataSheet file, one per instance. All product line information may be kept in the same file and duplicated for each instance, but there is an option to put some product line information into a single package file that is referenced by each instance DataSheet file.

Typically, a DataSheet file will refer to external package files that describe standard structures, such as SOIS subnetwork interfaces, CCSDS packet headers, and time codes, which are available at the SANA web site (see Annex B). The references are to types in the standard package files, expressed as the name of the package concatenated to the name of the type within the package, with “/” between. A similar pattern would be used for reference to product line types in a separate package file.

# Security

One of the questions that arises when considering the security of a system whose interfaces are defined by SEDS, is whether the reduction in difficulty of connecting parts makes it easier for an intruder to gain access to the system. Where this concern is paramount, SEDS should be protected along with other engineering documents. For commodity parts, interfaces must be well known, and protection of SEDS would interfere with marketing. In any case, hiding information is not a complete strategy for security; other measures must be included where security is needed, such as access control and encryption.

1. ACRONYMS and Abbreviations

This annex identifies and defines the acronyms and abbreviations used in this Report.

API Application Programming Interface

CCSDS Consultative Committee for Space Data Systems

CDAS Command and Data Acquisition Services

DACP Device Abstraction Control Procedure

DSAP Device-Specific Access Protocol

DAS Device Access Service, the aggregation of DSAP’s for each device

DoT Dictionary of Terms

DVS Device Virtualisation Service, aggregation of DACP’s for each device

ECSS European Cooperation for Space Standardization

EDS Electronic Data Sheet

EGSE Electrical Ground Support Equipment

FDIR Failure Detection, Isolation, and Recovery

ICD Interface Control Document

IRD Interface Requirement Document

ISO International Standards Organisation

MCS Mission Control System

MO CCSDS Mission Operations Services

MIB Management Information Base

MTS Message Transfer Service

OSI Open Systems Interconnection

PDU Protocol Data Unit

S/C Spacecraft

SEDS SOIS Electronic Data Sheet

SOIS Spacecraft Onboard Interface Services

SVF Software Verification Facility

XML eXtensible Markup Language

XSL eXtensible Stylesheet Language

1. SOURCE of SEDS Software

The source material for SEDS is located in the Space Assigned Numbers Authority (SANA) web site at the following URL: <http://sanaregistry.org/r/sois/sois.html>, as described in Appendix B of the SEDS blue book [3].

When writing software for a tool chain, consult the programming guide for the language and integrated development environment (IDE) of your choice for using the SEDS schema. The SEDS schema imports a few standard schemas that may be obtained from SANA, or may be loaded from a local file system. The IDE will likely have controls for specifying the source of imported schemas at development time, and the programming language will likely have shared libraries for controlling the source of schemas at execution time. Often, no specification is needed.