SAE Aerospace
An SAE International Group

AEROSPACE STANDARD

SAE AS5684

Issued 2008-12

JAUS Service Interface Definition Language

RATIONALE

This document defines a formal specification language for specifying services in the unmanned systems domain. The language is machine readable and provides precise syntax and semantics. The precise syntax and semantics eliminates the potential for misinterpretation of service specifications, and allows for the development of tools that can be used to automate various aspects of the development of an unmanned system.

INTRODUCTION

Services and service oriented architectures (SOAs) are widely used in the IT domain. The core principle of SOAs is the design of systems by means of distributed capabilities that are self contained, loosely coupled, and have well defined interfaces. A service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners. This principle fits very well in the unmanned systems domain. The JAUS Service Interface Definition Language (JSIDL) is a top-level specification that defines a language for describing unmanned system capabilities. It uses the Relax NG Compact [rng] syntax as a means to formally specify a schema that is to be used to create JAUS Service Definitions (JSDs) for all defined capabilities. Each JSD must be written in XML and must be validated against the JSIDL schema. The machine readable schema permits the development and use of automated tools, and consequently reduces integration time and provides for technology insertion.

This document is intended for anyone who wants to build or test JAUS Service Definitions. It is also intended for anyone who wants to build tools for designing, testing, documenting, and/or implementing JAUS Service Definitions. The JAUS Service Definitions are intended for those who want to build or test a system that uses JAUS specifications.

TABLE OF CONTENTS

1. SCOPE 3
1.1 JAUS Document Organization 3
2. REFERENCES 4
2.1 Applicable Documents 4
2.2 Definitions 4
2.3 List of Acronyms 5
3. FRAMEWORK FOR DEFINING PROTOCOLS / SERVICE DEFINITIONS 5
4. ATTRIBUTES AND ELEMENTS OF A JAUS SERVICE DEFINITION 6
5. MESSAGE ENCODING 7
5.1 Composite Fields 9
5.2 Simple Fields 16
5.2.1 Primitive Fields 17
5.2.2 String Fields 19
5.2.3 BLOB Fields 19
5.3 Meta Fields 20
5.4 Field Information Attributes and Elements 22
5.4.1 Field Information Attributes 22
5.4.2 Field Information Elements 23
5.5 Declared Types and Constants 26

TO PLACE A DOCUMENT ORDER: Tel: 877-606-7323 (inside USA and Canada)
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
SAE WEB ADDRESS: http://www.sae.org

SAE values your input. To provide feedback on this Technical Report, please visit
http://www.sae.org/technical/standards/AS5684

6. BEHAVIOR 30
6.1 Separation of Protocol Behavior from Application Behavior 30
6.2 Protocol Behavior 31
6.3 States 33
6.3.1 Entry and Exit Actions 33
6.4 Transitions 34
6.4.1 Simple Transitions 35
6.4.2 Loopback Transitions 36
6.4.3 Push Transitions 36
6.4.4 Pop Transitions 37
6.5 Default States and Default Transitions 37
7. SERVICE REFERENCE RELATIONSHIPS 39
7.1 Inheritance 39
7.2 Client 43
8. AN EXAMPLE OF A SERVICE DEFINITION 44
9. VERSIONING AND BACKWARDS COMPATIBILITY 49
10. COMPLIANCE 49
11. NOTES 50

APPENDIX A SCHEMA 51

FIGURE 1 SCOPE OF JSIDL WITH REFERENCE TO OTHER FRAMEWORKS AND ARCHITECTURES 3
FIGURE 2 TOP LEVEL STRUCTURE OF A SERVICE DEFINITION 6
FIGURE 3 SERIALIZED VERSION OF A SECTION OF A JAUS MESSAGE 8
FIGURE 4 MESSAGE_DEF – A THREE PART STRUCTURE 8
FIGURE 5 DATA FIELD ELEMENTS - REPRESENT ENCODED DATA 9
FIGURE 6 ARRAY – A HOMOGENOUS SEQUENCE OF SIMPLE FIELDS 10
FIGURE 7 RECORD – A HETEROGENEOUS SET OF SIMPLE FIELDS (INCLUDING ARRAYS) THAT IS ORDERED 11
FIGURE 8 LIST – A VARIABLE SIZED SEQUENCE OF SIMPLE OR COMPOSITE FIELDS OF THE SAME TYPE 12
FIGURE 9 VARIANT – A CONTAINER OF RUN-TIME SELECTABLE TYPES 13
FIGURE 10 AN EMPTY VARIANT – CREATES THE POSSIBILITY OF SPECIFYING NO DATA 13
FIGURE 11 SEQUENCE – A HETEROGENEOUS SEQUENCE OF COMPOSITE FIELDS (EXCLUDING ARRAYS) 14
FIGURE 12 EXAMPLE OF A FIXED-SIZE/SHAPE N-ARY TREE 14
FIGURE 13 EXAMPLE OF AN ADJACENCY LIST REPRESENTATION OF A GRAPH 15
FIGURE 14 FIELD INFORMATION ATTRIBUTES AND ELEMENTS - PROVIDE INFORMATION ABOUT ENCODED DATA 22
FIGURE 15 EXAMPLE OF A REFERENCE TO A DECLARED_TYPE 28
FIGURE 16 SEPARATION OF PROTOCOL BEHAVIOR FROM APPLICATION BEHAVIOR 31
FIGURE 17 STRUCTURE OF THE PROTOCOL_BEHAVIOR ELEMENT 32
FIGURE 18 THE INHERITS-FROM ELEMENT 39
FIGURE 19 ADDITION OF FEATURES – NESTED STATES 40
FIGURE 20 ADDITION OF FEATURES – CONCURRENT STATE MACHINES 41
FIGURE 21 ADDITION OF FEATURES - TRANSITION 42
FIGURE 22 THE CLIENT-OF ELEMENT 43
FIGURE 23 UML STATE DIAGRAM OF THE ACCESS CONTROL SERVICE 45
FIGURE 24 UML STATE DIAGRAM OF THE EXTENDED ACCESS CONTROL SERVICE 47

TABLE 1 SIZE AND REPRESENTATION OF PRIMITIVE DATA TYPES 22
TABLE 2 TYPES OF TRANSITIONS FOR WHICH ENTRY/EXIT ACTIONS ARE EXECUTED [SMC] 33

1. SCOPE

The SAE Aerospace Information Report AIR5315 – Generic Open Architecture (GOA) defines “a framework to identify interface classes for applying open systems to the design of a specific hardware/software system.” [sae] JAUS Service (Interface) Definition Language defines an XML schema for the interface definition of services at the Class 4L, or Application Layer, and Class 3L, or System Services Layer, of the Generic Open Architecture stack (See Figure 1 below). The specification of JAUS services shall be defined according to the JAUS Service (Interface) Definition Language document.

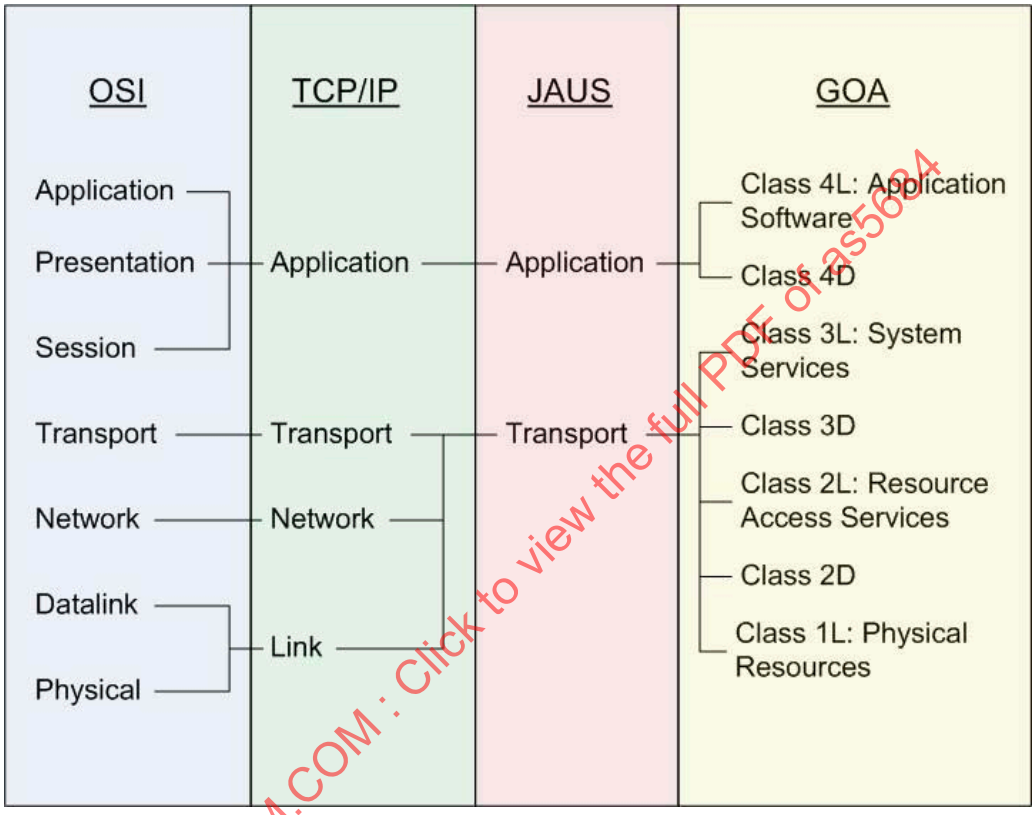

FIGURE 1 - SCOPE OF JSIDL WITH REFERENCE TO OTHER FRAMEWORKS AND ARCHITECTURES

1.1 JAUS Document Organization

The layout of this document is as follows. Section 2 lists external references. Section 3 and Section 4 describe the elements of a JAUS Service Definition: description, assumptions, message set, message encoding and protocol behavior. The bulk of the JAUS Service Definition lies in the last two elements, Message Encoding and Behavior. These two elements are described in detail in Section 5 and Section 6. Section 7 describes two service reference relationships that allow for the reuse of existing service definitions. Section 8 contains an example of a JAUS Service Definition. Version control rules for the JSIDL and all Service Definitions are presented in Section 9. Section 10 contains a short note on compliance. Appendix A contains the complete JSIDL.

Almost all figures in the document use UML 2.0 notation [uml].

2. REFERENCES

2.1 Applicable Documents

[ansic] Brian W. Kernighan. Dennis M. Ritchie. The C Programming Language (ANSI C). Prentice Hall. 1997.

[bnt] Barry N. Taylor. The International System of Units (SI), National Institute of Standards and Technology Special Publication 330, 1991 Edition. [http://physics.nist.gov/Document/sp330.pdf]

[greg] John D. McGregor, Douglas M. Dyer, A Note on Inheritance and State Machines, ACM SIGSOFT, Software Engineering Notes vol. 18 no. 4, Oct 1993 pg 61.

[holz] Gerard J. Holzmann. Design and Validation of Computer Protocols. Prentice Hall Software Series. 1991.

[lsp] Barbara Liskov. Data abstraction and hierarchy. Conference on Object Oriented Programming Systems Languages and Applications. P.17 – 34, 1987.

[lynch] Nancy A. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers. April, 1997.

[rng] Relax NG: [http://www.oasis-open.org/committees/relax-ng/spec-20011203.html] Standard lightweight XML schema language.

[sae] Overview and Rationale for GOA Framework Standard, SAE AIR5315, July 1997

[smc] State Machine Compiler (SMC) [http://smc.sourceforge.net/] .

[trans] JAUS Transport Specification, SAE AS5669 Revision 0.4 Draft..

[uml] Unified Modeling Language (UML), version 2.1.1, formal/2007-02-05 [http://www.omg.org/docs/formal/07-02-05.pdf]

[uri] Uniform Resource Identifiers (URI): Generic Syntax [http://ftp.ics.uci.edu/pub/ietf/uri/rfc2396.txt] .

[uuid] A Universally Unique IDentifier (UUID) URN Namespace [http://www.ietf.org/rfc/rfc4122.txt] .

[xml] Extensible Markup Language (XML) [http://www.w3.org/XML/] .

2.2 Definitions

2.2.1 Identifier (id)

An identifier is an object that can act as a reference to something that has identity [uri].

A basic requirement for identifiers is that they must be unique within the scope in which they are defined.

2.2.2 Distributed System

A Distributed System is one in which the power of several computing entities is aggregated to collaboratively run a single computational task in a transparent and coherent way, so that they appear as a single, centralized system.

2.2.3 Service

A service is an interface to a coherent function. A service specifies only the messaging syntax, semantics, and protocol necessary to use the function. It does not specify how the function is implemented.

2.3 List of Acronyms

ANSI American National Standards Institute

Id Identifier

API Application Programming Interface

ASCII American Standard Code for Information Interchange

BLOB Binary Large Object

DOM Document Object Model

IEEE Institute of Electrical and Electronics Engineers

JAUS Joint Architecture for Unmanned Systems

JSD JAUS Service Definition

JSIDL JAUS Service (Interface) Definition Language

NIST National Institute of Standards and Technology

SMC State Machine Compiler

UML Unified Modeling Language

URL Uniform Resource Locator

URN Uniform Resource Name

URI Uniform Resource Identifier

UUID Universally Unique Identifier

XML Extensible Markup Language

3. FRAMEWORK FOR DEFINING PROTOCOLS / SERVICE DEFINITIONS

This specification employs the framework for protocol definition developed by Holzmann. In this specification, the term “service” is equivalent to Holzmann’s use of the term “protocol.” In outline form, a complete and unambiguous specification of a protocol requires:

“A protocol specification consists of five distinct parts. To be complete, each specification should include explicitly:

1. The service to be provided by the protocol
2. The assumptions about the environment in which the protocol is executed
3. The vocabulary of messages used to implement the protocol
4. The encoding (format) of each message in the vocabulary
5. The procedure rules guarding the consistency of message exchanges

The fifth element of a protocol specification is the most difficult to design" [holz p. 21]

The JAUS Service (Interface) Definition Language provides a means for defining each of these five elements.

4. ATTRIBUTES AND ELEMENTS OF A JAUS SERVICE DEFINITION

This section describes the elements of a JAUS Service Definition. The actual schema is presented in Appendix A. Figure 2 below illustrates the top-level structure of a service definition. The root element of the definition is service_def. All the attributes and elements of a service_ def are required, except for the declared_type_set element which is optional. All JAUS Service Definitions must be specified under a JSIDL namespace Uniform Resource Identifier (or URI) [uri].

```
<service_def
   xmlns="urn:jaus:jsidl:0.11"
   name="service_name"
   id="service_uri"
   version="1.1">

  <description> </description>

  <assumptions> </assumptions>

  <references> </references>

  <declared_const_set> </declared_const_set>
  <declared_type_set> </declared_type_set>

  <message_set>
    <input_set> </input_set>
    <output_set> </output_set>
  </message_set>

  <internal_events_set> </internal_events_set>

  <protocol_behavior> </protocol_behavior>
</service_def>
```

FIGURE 2 - TOP LEVEL STRUCTURE OF A SERVICE DEFINITION

Attributes

- name: The name of the service. The name must be a valid identifier (refer Appendix A.4): It must also be unique among all service names used within the scope of the namespace in which the service is specified.

- id: The namespace of a service is defined by its id. The id is a globally unique URI that uniquely identifies a particular service. All versions of a particular service must have the same id. The chosen URI can be either a Uniform Resource Locator (URL) or a Uniform Resource Name (URN). The choice will depend on whether a location dependent id is preferred or a location independent id is preferred. For example, an organization that does not have or want to publish and maintain a web page for each service would prefer to use URNs over URLs. For services that are either experimental or designed for temporary use, a particular type of URN called a Universally Unique Identifer (UUID) [uuid] may be more applicable. UUIDs are 16 byte identifiers that are generated algorithmically and are globally unique in space and time.

- version: The version number of a service. Details about a service's version number is provided in Section 9: Versioning and Backwards Compatibility.

Elements

- description: A required element that must be used to provide a brief textual description of the service and its functionality. The description element has an attribute called xml:space. Basic textual formatting can be preserved in the description element if the value of this attribute is set to “preserve”.

- assumptions: A required element that must be used to provide a brief textual description about the assumptions made about the nature of the communication channels the service uses to communicate with its clients.

- references: An optional element that can be used to provide a list of service definitions that the service references. Section 7 describes service reference relationships in detail.

- declared_const_set: An optional element that can be used to define a reusable set of constant definitions (see Section 5.5).

- declared_type_set: An optional element that can be used to define a reusable set of data type definitions (see Section 5.5).

- message_set: A required element that must be used to define the input and output message sets that make up the external vocabulary of the service. When a service definition references other service definitions, the vocabulary of the service is determined by the nature of the reference used (see Section 7). Each message definition within the message_set contains a detailed encoding of the structure of the message data. Section 5 provides a detailed description on encoding, while Appendix A.2 contains the schema.

- Internal_events_set: The internal events set is used to define events[1] like timouts that occur internal to a service and have a direct effect on the protocol behavior of the service. Like message definitions, event definitions can contain a detailed specification of the encoding of data that may be associated with the event. Section 5 provides a detailed description on encoding, while Appendix A.2 contains the schema.

- protocol_behavior: The protocol or message exchange behavior of a service is defined in the form of a finite state machine. The definition of a service is structured around the separation of the behavior that is related to the exchange of messages (protocol) and the application behavior application behavior (see Section 6 on seperation of protocol and application behavior). The behavior specified by the service definition is that of the protocol and not the application. For example, the behavior element of a waypoint driver service definition must describe only the message interchange behavior between the service and its clients. It should not specify the internal computation required to say, generate wrench commands used to reach a given waypoint. A detailed description on specifying service behavior is provided in Section 6, while Appendix A.3 contains the schema for specifying protocol behavior.

5. MESSAGE ENCODING

The format of a JAUS message is defined using a message_def element. Although the message structure and format is specified in XML, the format of the actual on-the-wire JAUS message is a binary format containing only the serialized data. Figure 3 below shows a section of a serialized JAUS message. This message contains a sequence of twenty bytes. The first two bytes represent a short integer that is interpreted as the waypoint number. The next twelve bytes represent a sequence of three 4-byte integers interpreted as lattitude, longitude and elevation. The last six bytes represent three short integers that are interpreted as the roll, pitch and yaw. For inter-process communication, all multi-byte data like the short integer and 4-byte integer in a JAUS message that is not opaque data (like a JPEG image for example) is encoded in little endian where the low byte comes first. As an exception to this rule, the native format of the processor may be used for encoding messages that are used for communicating data within the same processor.

[1] The structure of the event_def element is similar to the structure of a message_def element except that it does not have message_id and is_command attributes.

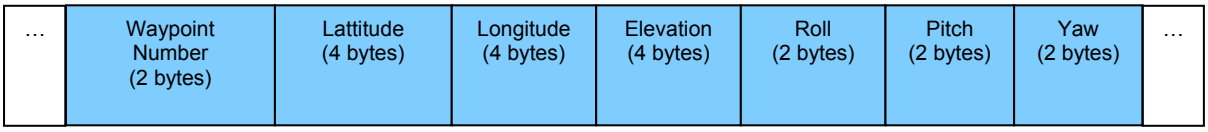

FIGURE 3 - SERIALIZED VERSION OF A SECTION OF A JAUS MESSAGE

The specification of such a format using the message_def element includes all the information that is necessary to format and interpret the message unambiguously. The message_def element has a three part structure: header, body and footer, each of which may be specified as being empty. It also contains an element called description that can be used to provide a brief textual description of the message. The description element provides a basic white space formatting capability through its attribute called xml:space. If this attribute is assigned the value “default”, then XML applications (readers and editors) will not adhere to the white space formatting that may have been used. If this attribute is assigned the value “preserve”, then XML applications will adhere to the white space formatting that may have been used in the description element. The message_def element has attributes for specifying the message name, a message id (in hexadecimal) and whether the message is a command message or not. The message ids assigned to messages belonging to the input and output set are globally unique, while those assigned to messages belonging to the internal set need to be unique within the scope of the service definition in which they are used.

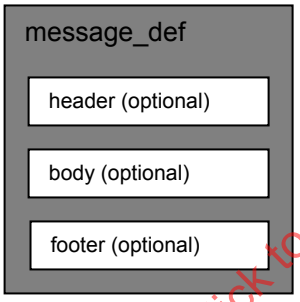

FIGURE 4 - MESSAGE_DEF – A THREE PART STRUCTURE

The JSIDL provides several elements and attributes for defining the structure and content of the header, body and footer of a message. These elements and attributes are broadly classified into two groups: 1) Data Field Elements, and 2) Field Information Elements and Attributes. Data Field Elements represent encoded data and are further divided into three groups – Composite Fields, Simple Fields and Meta Fields. These groups are illustrated in Figure 5 and defined in sections 5.1, 5.2 and 5.3 respectively. These fields have been designed for bandwidth optimization and to provide flexibility to the designer of Service Definitions. Section 5.4 contains the definitions of the Field Information Elements and Attributes. These elements and attributes are used in data field definitions to provide information about the encoded data. Finally, Section 5.5 describes the specification of messages and their contents as declared types for the purpose of reuse.

Examples:

```
<message_def name="Request_Control" message_id="000d" is_command="true">
  <description xml:space=”preserve”>
    This message is used to request exclusive control of...
  </description>
  <header>
     ...
  <header>
  <body>
    ...
  </body>
  <footer/>
</message_def>
```

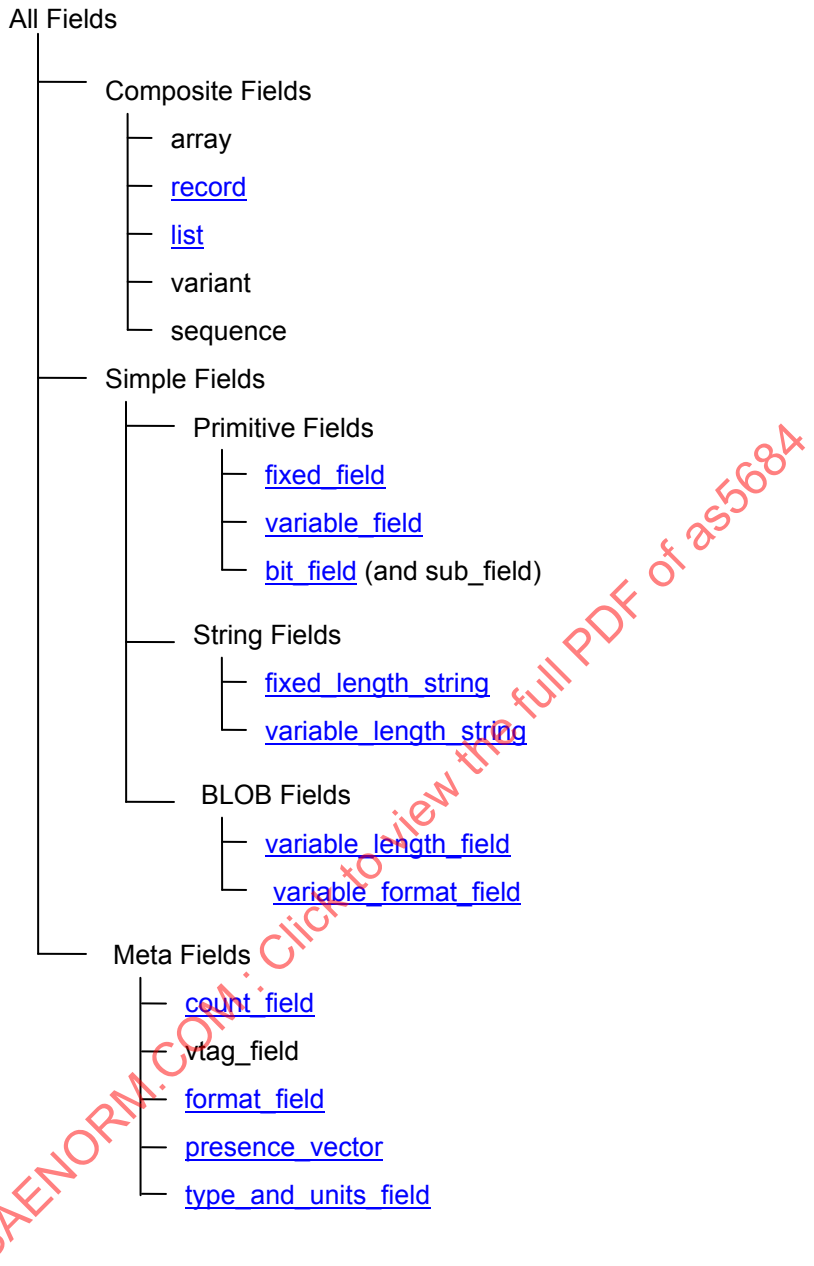

FIGURE 5 - DATA FIELD ELEMENTS - REPRESENT ENCODED DATA

5.1 Composite Fields

Composite Fields are containers that provide a means to create complex data structures by combining a set Simple Fields in various ways. There are five types of complex fields: array, record, list, variant and sequence. arrays can only be contained by records and records can only be contained by lists, variants and sequences. In addition to records, lists, variants and sequences can contain lists, variants and sequences.

array

An array field represents a rectangular multi-dimensional array composed of Simple Fields of the same type. Its structure is illustrated in Figure 6 below. The definition for an array must contain a Simple Field template for the elements of the array. It must also contain definitions for the dimensions of the array.

The array has a name attribute that is required, and an optional interpretation attribute that may be used to provide a brief textual interpretation for the array. The name of the array must be a valid identifier (refer Appendix A.4) and must be unique within the scope of the encapsulating record element. The dimensions of the array are specified using a sub-element called dimension, Each dimension definition consists of the name that is attributed to the dimension, the size of the dimension, and an brief textual interpretation that is optional. The name of the dimension must be a valid identifier (refer Appendix A.4) and must be unique within the scope of the array. The size of the dimension must be an integer constant or a reference to an integer constant (see Section 5.5).

The order in which the dimensions are specified indicates how the array is read. If the order of the dimensions of an m-dimensional array is d_1 x d_2 x .. x d_{m-1} x d_m, it implies that the subarray d_1 x d_2 x ... x d_{m-1} is repeated d_m times. When applied recursively, d_1 x d_2 x ... x d_{m-2} is repeated d_{m-1} x d_m times. Each two-dimensional sub-array is read row first.

An array defined within a record can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire array may not be included in the record if it is not required (see definition of record). The semantics of the attribute called optional that is contained in the encapsulated Simple Field is undefined when the Simple Field definition is used within an array definition.

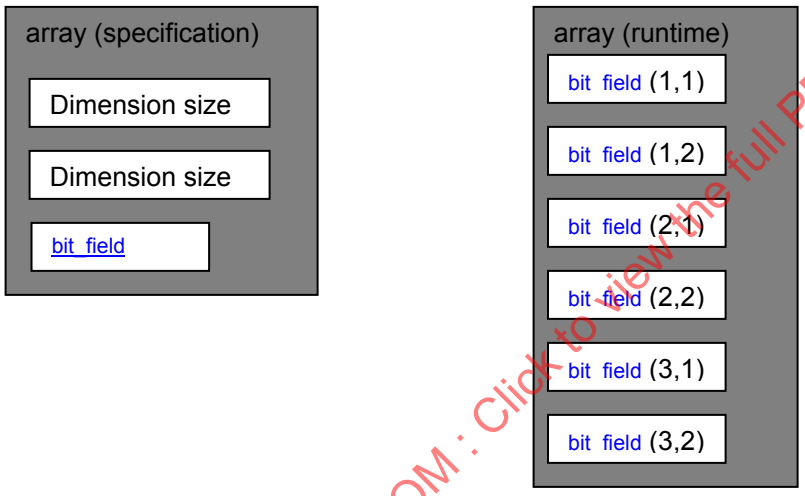

FIGURE 6 - ARRAY – A HOMOGENOUS SEQUENCE OF SIMPLE FIELDS

Examples:

```
<array name="Raster_Data" optional="false" interpretation="A 2-Dimensional scan....">
  <!—the semantics of the optional attribute of pixel is undefined -->
  <fixed_field name="Pixel" field_type="byte" field_units="one" optional="false" interpretation="A
single pixel value"/>
  <dimension name="RGB" size="3" interpretation="Red, Blue and Green Values"/>
  <dimension name="Color_Vector" size="1" interpretation="RGB Color vector"/>
  <dimension name="width" size="100"/>
  <dimension name="height" size="100"/>
</array>
```

This stands for a color vector repeated 100 x 100 times. Each 3 x 1 color vector contains three unsigned bytes standing for red, green and blue values[2].

The serialization of this array to obtain the on-the-wire format as shown step-wise below.

When the first two-dimensional array is serialized, we get
{r_1, g_1, b_1}.

[2] Note that a dimension of size 1 has no effect on the serialization of the array. Its purpose is to clearly define (or identify) elements and sub-elements of the array as vectors.

In the 100x100 matrix, this represents row1, column1.

Adding the next dimension we get a new two-dimensional array, ((r_1, g_1, b_1), 100) which when serialized yields,
{r_1, g_1, b_1, r_2, g_2, b_2, ..., r_100, g_100, b_100}.
In the 100x100 matrix, this represents the entire row1.

Adding the next dimension we get a new two-dimensional array,
((r_1, g_1, b_1, r_2, g_2, b_2, ..., r_100, g_100, b_100), 100)
which when serialized yields,
{r_1, g_1, b_1, r_2, g_2, b_2, ..., r_100, g_100, b_100, ..., r_10000, g_10000, b_10000}.
In the 100x100 matrix, this represents rows 1 to 100.

record

A record represents an arrangement of one or more Simple Fields or arrays leading to a heterogeneous or homogeneous set that is ordered. Its structure is illustrated in Figure 7 below.

The Simple Fields and arrays contained within a record can be optional at runtime. That is, they may or may not be included in the message. If any of the Simple Fields or arrays within a record are specified as being optional, the record must have a Meta Field called the presence_vector as its first field. This Meta Field is used to indicate the presence or absence of the optional Simple Fields or arrays within the record.

The record has a name attribute that is required, and an optional interpretation attribute that may be used to provide a brief textual interpretation for the record. The name of the record must be a valid identifier (refer Appendix A.4) and must be unique within the scope of the encasulating body or list parent element.

A record defined within a sequence can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire record may not be included in the sequence if it is not required (see definition of sequence).

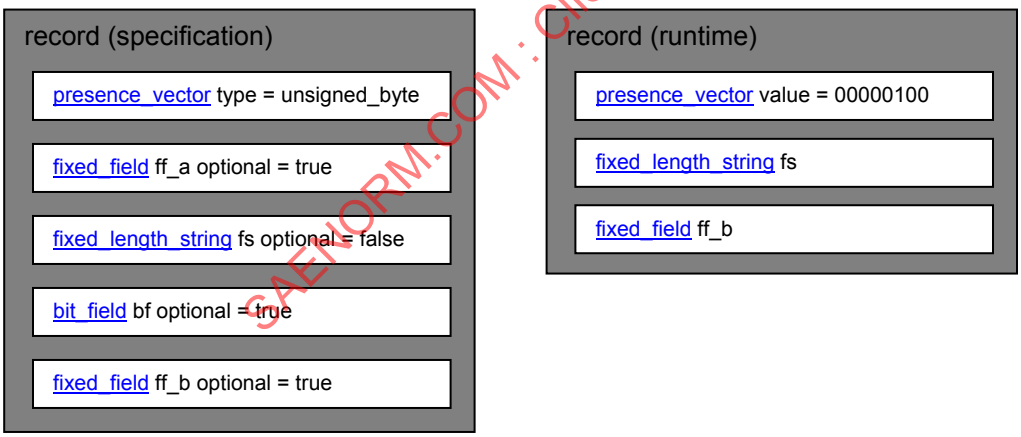

FIGURE 7 - RECORD – A HETEROGENEOUS SET OF SIMPLE FIELDS (INCLUDING ARRAYS) THAT IS ORDERED

Examples:

```
<!-- A record with a single Simple Field. -->
<record name="a_record">
  <fixed_field>...</fixed_field>
</record>

<!-- A record containing a presence vector followed by five
    Simple Fields, some or all of which are optional (since
    the presence vector is present). -->
<record name="some_record">
```

```
  <presence_vector field_type_unsigned="unsigned_byte"/>
  <fixed_field ...> ... </fixed_field>
  <fixed_field ...> ... </fixed_field>
  <fixed_length_string ...optional="true">
  ...
  </fixed_length_string>
  <fixed_field ...> ... </fixed_field>
  <variable_field ...optional="true">
  <array...>...</array>
  </variable_field>
  <fixed_field ...> ... </fixed_field>
</record>
```

list

A list field represents a variable sized sequence of Composite fields of the same type, with the exception of array. The size of the sequence is determined at runtime from the value a Meta field called count_field which is the first field in the list.

A list defined within a sequence can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire list may not be included in the sequence if it is not required (see definition of sequence).

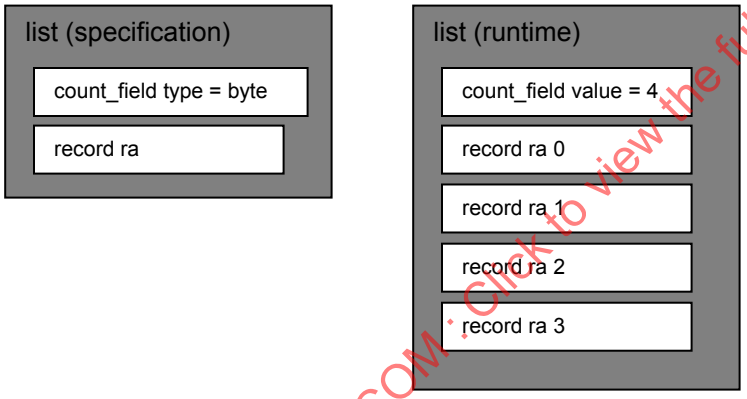

FIGURE 8 - LIST – A VARIABLE SIZED SEQUENCE OF SIMPLE OR COMPOSITE FIELDS OF THE SAME TYPE

Examples :

```
  <list name="subsystem_list" interpretation="list of subsystems in the system">
    <count_field field_type_unsigned="unsigned byte"/>
    <record name="subsystem_rec" interpretation="name,id pair">
      <fixed_field name="subsystem_id" field_type="unsigned byte" field_units="one"
optional="false"/>
      <fixed_length_string name="subsystem_name" string_length="80" optional="false"/>
    </record>
  </list>
```

variant

A variant field represents a composite field that can hold zero or one of several different types of pre-defined composite fields at runtime. The pre-defined set of composite fields are specified as a sequence and the type of field contained within the variant at runtime is determined from the value a Meta field called the vtag_field which is the first field in the variant. If the n^{th} composite field in the sequence is chosen, then the value of the vtag_field must be set to n, where n=0 if the first composite field in the sequence is chosen at runtime.

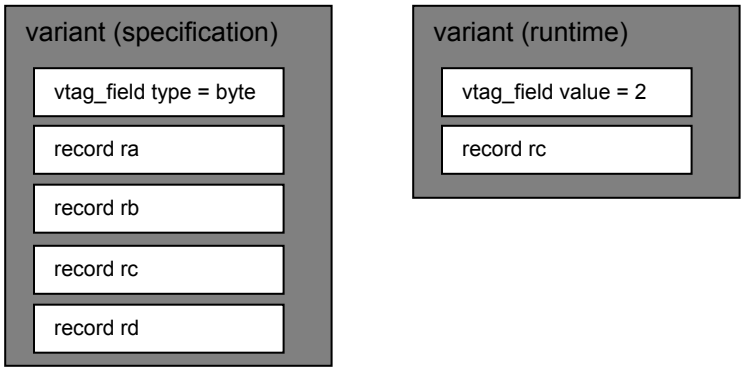

FIGURE 9 - VARIANT – A CONTAINER OF RUN-TIME SELECTABLE TYPES

A variant may be specified as being empty. That is, it does not specify a sequence of composite fields (or the sequence is of size zero). This case is useful when one of the possible choices of a variant has no data at all. This can be specified by including an empty variant within a variant. In this case, the vtag field of the empty variant must be set to 0 as shown in Figure 10 below.

A variant defined within a sequence can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire variant may not be included in the sequence if it is not required (see definition of sequence).

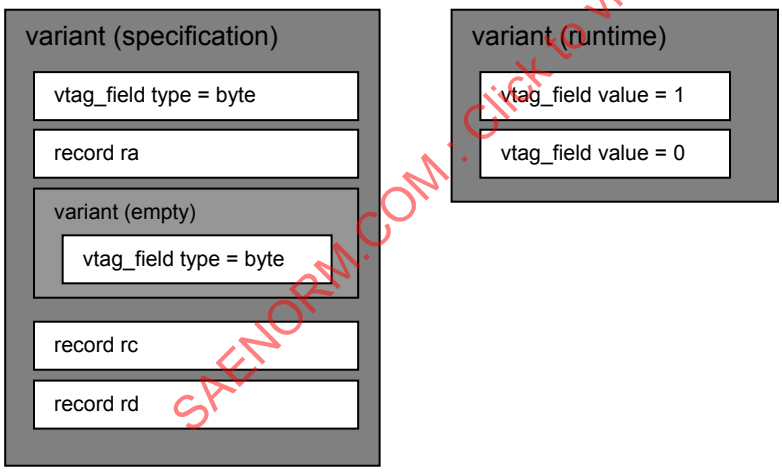

FIGURE 10 - AN EMPTY VARIANT – CREATES THE POSSIBILITY OF SPECIFYING NO DATA

Examples :

```
<variant name="shape">
  <vtag_field field_type_unsigned="unsigned byte" min_count="0" max_count="2"/>
  <record name="rectangle">
    <fixed_field name="length" field_type="byte" field_units="one" optional="false"/>
    <fixed_field name="height" field_type="byte" field_units="one" optional="false"/>
  </record>
  <record name="square">
    <fixed_field name="side" field_type="byte" field_units="one" optional="false"/>
  </record>
  <record name="circle">
```

```
      <fixed_field name="radius" field_type="byte" field_units="one" optional="false"/>
    </record>
  </variant>
```

sequence

As the name suggests, a sequence represents an arrangement of one or more Composite Fields (excluding arrays) leading to a heterogeneous or homogeneous sequence. Its structure is illustrated in Figure 11 below.

The Composite Fields contained within a sequence can be optional at runtime. That is, they may or may not be included in the message. If any of the composite fields within a sequence are specified as being optional, the sequence must have a Meta Field called the presence_vector as its first field. This Meta Field is used to indicate the presence or absence of the optional Composite Fields within the record.

A sequence defined within another sequence can be optional at runtime. This is specified by the boolean attribute called optional. If this attribute is set to true, the entire encapsulated sequence may not be included in the container sequence if it is not required.

The recursive structure of list, variant and sequence fields can be used to define data structures like graphs and trees. Examples of these structures are presented in Figures 12 and 13 below.

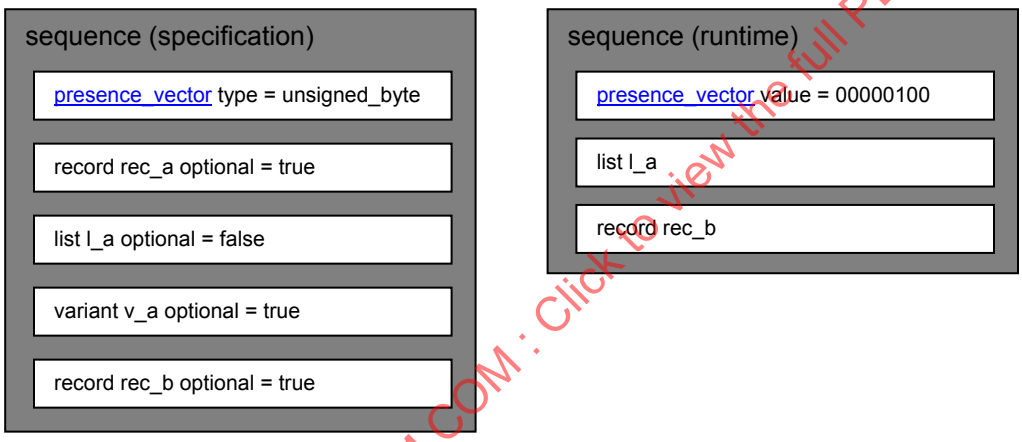

FIGURE 11 - SEQUENCE – A HETEROGENEOUS SEQUENCE OF COMPOSITE FIELDS (EXCLUDING ARRAYS)

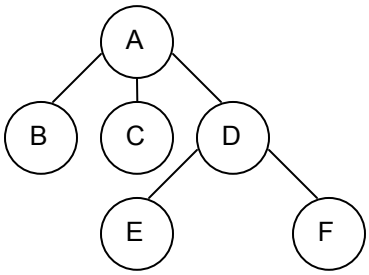

FIGURE 12 - EXAMPLE OF A FIXED-SIZE/SHAPE N-ARY TREE

Examples :

```
<sequence name="A">
  <record name="a">
    <fixed_field name="a" field_type="byte" field_units="one" optional="false"/>
    <fixed_field name="b" field_type="byte" field_units="one" optional="false"/>
  </record>
  <sequence name="B">
    <record name="b">
      <fixed_field name="b" field_type="byte" field_units="one" optional="false"/>
    </record>
  </sequence>
  <sequence name="C">
    <record name="c">
      <fixed_field name="c" field_type="byte" field_units="one" optional="false"/>
    </record>
  </sequence>
  <sequence name="D">
    <sequence name="E">
      <record name="e">
        <fixed_field name="e" field_type="byte" field_units="one" optional="false"/>
      </record>
    </sequence>
    <sequence name="F">
      <record name="f">
        <presence_vector field_type_unsigned="unsigned byte"/>
        <fixed_field name="f" field_type="byte" field_units="one" optional="true"/>
        <fixed_field name="g" field_type="byte" field_units="one" optional="true"/>
      </record>
    </sequence>
  </sequence>
</sequence>
```

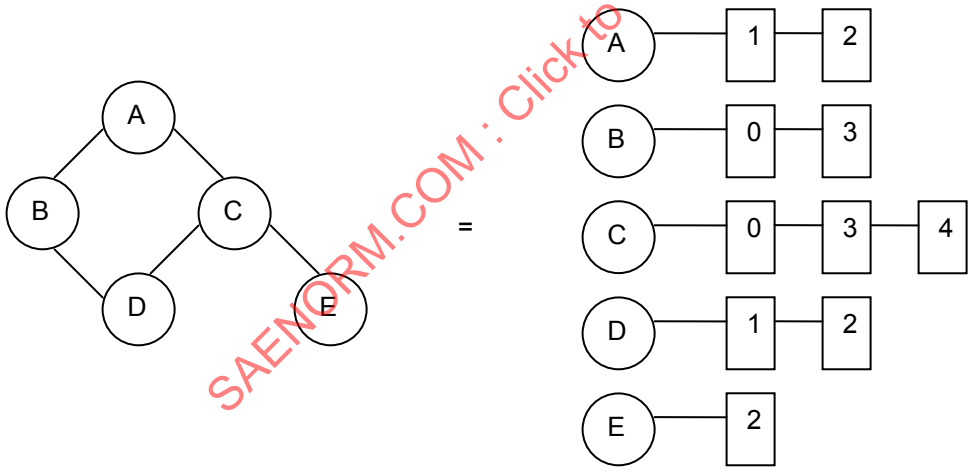

FIGURE 13 - EXAMPLE OF AN ADJACENCY LIST REPRESENTATION OF A GRAPH

```
<sequence name="adjacencyListGraph">
  <sequence name="A">
    <record name="a">
      <fixed_field name="a" field_type="byte" field_units="one" optional="false"/>
      <fixed_field name="b" field_type="byte" field_units="one" optional="false"/>
    </record>
    <list name="edges">
      <count_field field_type_unsigned="unsigned byte"/>
      <record name="index_rec">
        <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
      </record>
    </list>
  </sequence>
```

```
    <sequence name="B">
      <record name="b">
        <fixed_field name="b" field_type="byte" field_units="one" optional="false"/>
      </record>
      <list name="edges">
        <count_field field_type_unsigned="unsigned byte"/>
        <record name="index_rec">
          <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
        </record>
      </list>
    </sequence>
    <sequence name="C">
      <record name="c">
        <fixed_field name="c" field_type="byte" field_units="one" optional="false"/>
      </record>
      <list name="edges">
        <count_field field_type_unsigned="unsigned byte"/>
        <record name="index_rec">
          <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
        </record>
      </list>
    </sequence>
    <sequence name="D">
      <record name="d">
        <fixed_field name="d" field_type="byte" field_units="one" optional="false"/>
      </record>
      <list name="edges">
        <count_field field_type_unsigned="unsigned byte"/>
        <record name="index_rec">
          <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
        </record>
      </list>
    </sequence>
    <sequence name="E">
      <record name="e">
        <fixed_field name="e" field_type="byte" field_units="one" optional="false"/>
      </record>
      <list name="edges">
        <count_field field_type_unsigned="unsigned byte"/>
        <record name="f">
          <fixed_field name="index" field_type="byte" field_units="one" optional="false"/>
        </record>
      </list>
    </sequence>
  </sequence>

  <list name="variable_size_shape_n_ary_tree" interpretation="A variable size and shape n-ary tree
and graph. All nodes in this graph hold the same type of data. But, this is not necessary. Nodes can
be heterogeneous.">
    <count_field field_type_unsigned="unsigned byte" interpretation="node count"/>
    <sequence name="node">
      <record name="node_data">
        <fixed_field name="data" field_type="byte" optional="false" field_units="one"/>
      </record>
      <list name="edges">
        <count_field field_type_unsigned="unsigned byte" interpretation="edge count"/>
        <record name="edge_index">
          <fixed_field name="edge_index" field_type="byte" field_units="one" optional="false"/>
        </record>
      </list>
    </sequence>
  </list>
```


5.2 Simple Fields

A Simple Field is an abstract data type that must be used to specify a single cohesive unit of data that cannot be further divided into two or more Simple or Composite Fields. Simple Fields may contain one or more Meta Fields. There are three kinds of Simple Fields: Primitive Fields, String Fields and Binary Large Object (BLOB) Fields.

5.2.1 Primitive Fields

Primitive Fields represent data with primitive types such as byte, integer, float etc. These types are described in the Field Information attribute field_type.

fixed_field

A fixed_field represents a primitive data type whose type and units are fixed. The type of the fixed_field is specified by the field_type attribute and its units are specified by the field_units attribute.

The default value range of a fixed_field is the range of the chosen field_type. If the values of a fixed_field needs to be further constrained, the child element must be used.

If the fixed_field represents a scaled integer, the range for the scale must be specified with the scale_range child element.

A fixed_field used within a record can be optional at runtime. This is specified by setting the attribute called optional to true, else it must be set to false.

The name of the fixed_field must be specified using its name attribute. The name must be a valid identifier (refer Appendix A.4) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the fixed_field.

Examples:

```
<fixed_field field_type="short integer" field_units="radian" name="Roll" optional="true">
  <scale_range real_lower_limit="-PI" real_upper_limit="PI" integer_function="round"/>
</fixed_field>

<fixed_field field_type="unsigned short integer" name="Waypoint_Number" field_units="one"
optional="false">
  <value_set offset_to_lower_limit="false">
    <value_range lower_limit="0" upper_limit="65535" lower_limit_type="inclusive"
      upper_limit_type="inclusive"/>
  </value_set>
</fixed_field>

<fixed_field name="priority" field_type="byte" field_units="one" optional="false">
  <value_set offset_to_lower_limit="false">
    <value_range lower_limit="0" upper_limit="9" lower_limit_type="inclusive"
      upper_limit_type="inclusive"/>
    <value_enum enum_index="0" enum_const="'low priority'" interpretation="implies a low
      priority message."/>
    <value_enum enum_index="5" enum_const="'medium priority'" interpretation="implies
      medium priority message."/>
    <value_enum enum_index="9" enum_const="'high priority'" interpretation="implies a
      high priority message."/>
  </value_set>
</fixed_field>
```

variable_field

A variable_field is a Simple Field whose type and units can vary at runtime. This field is always preceded by a Meta Field called type_and_units_field that specifies its type and units.

A variable_field used within a record can be optional at runtime. This is specified by setting the attribute called optional to true, else it must be set to false.

The name of the variable_field must be specified using its name attribute. The name must be a valid identifier (refer Appendix A.4) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the variable_field.

Examples:

```
<variable_field name="temperature" interpretation="CPU temperature" optional="false">
  <type_and_units_field>
    <type_and_units_enum index="0" field_type="short integer" field_units="degree Celsius"/>
    <type_and_units_enum index="1" field_type="short integer" field_units="kelvin"/>
    <type_and_units_enum index="2" field_type="float" field_units="degree Celsius"/>
    <type_and_units_enum index="3" field_type="float" field_units="kelvin"/>
  </type_and_units_field>
</variable_field>
```

bit_field

The bit_field is a Simple Field that can be used to specify one or more bit-level fields within a single primitive data type whose type must be specified by the field_type attribute.

Bit-level fields within a bit_field are specified by an ordered list of Field Information elements called sub_field. The unspecified bits of a bit_field must always be set to 0.

A bit_field used within a record can be optional at runtime. This is specified by setting the attribute called optional to true, else it must be set to false.

The name of the bit_field must be specified using its name attribute. The name must be a valid identifier (refer Appendix A.4) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the bit_field.

Examples:

```
<bit_field name="two_sub_fields" field_type_unsigned="unsigned byte" optional="false">
  <sub_field name="sub_1">
    <bit_range from_index="0" to_index="3"/>
    <value_set offset_to_lower_limit="false">
      <value_range lower_limit="0" upper_limit="12" lower_limit_type="inclusive"
upper_limit_type="inclusive"/>
    </value_set>
  </sub_field>
  <sub_field name="sub_2">
    <bit_range from_index="4" to_index="7"/>
    <value_set offset_to_lower_limit="false">
      <value_enum enum_index="0" enum_const="'str const 0'" interpretation="index holds the integer
value of the enum"/>
      <value_enum enum_index="1" enum_const="'str const 1'" interpretation="value holds a string
constant that maps to the index"/>
      <value_enum enum_index="2" enum_const="'str const 2'"/>
    </value_set>
  </sub_field>
</bit_field>
```

sub_field

Each field contained within a bit_field is specified by a sub_field element. sub_fields use the bit_range element to specify the bits used for each field within a bit_field, and the value_set element to specify the values that each field can take.

The name of a sub_field must be specified by its name attribute and must be a valid identifier (refer Appendix A.4). sub_field also has an optional attribute for a textual interpretation of the specified field.

Examples: Refer bit_field examples

5.2.2 String Fields

String Fields represent character string data derived from the American Standard Code for Information Interchange (ASCII) character set.

fixed_length_string

A fixed_length_string is a Simple Field that contains a character string whose maximum length is fixed. The maximum length of the string must be specified by an attribute called string_length whose value must be an integer constant or a reference to an integer constant (see Section 5.5). The value of this field must be a character string of length shorter than the maximum length, where the remaining bytes are padded with the null character.

A fixed_length_string used within a record can be optional at runtime. This is specified by setting the attribute called optional to true, else it must be set to false.

The name of the fixed_length_string must be specified using its name attribute. The name must be a valid identifier (refer Appendix A.4) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the fixed_length_string.

Examples:

```
<fixed_length_string name="Node_Name" string_length="80" optional="false"/>
```

variable_length_string

A variable_length_string is a Simple Field that contains a character string whose maximum length is not fixed. This field is always preceded by a Meta Field called count_field that specifies its exact length in bytes.

A variable_length_string used within a record can be optional at runtime. This is specified by setting the attribute called optional to true, else it must be set to false.

The name of the variable_length_string must be specified using its name attribute. The name must be a valid identifier (refer Appendix A.4) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the variable_length_string.

Examples:

```
  <variable_length_string name="node_description" interpretation="a brief description of the node's
functions" optional="true">
    <count_field field_type_unsigned="unsigned byte"/>
  </variable_length_string>
```

5.2.3 BLOB Fields

BLOB Fields represent large binary data such as JPEG images.

variable_length_field

A variable_length_field is a Simple Field that contains a BLOB. This field is always preceded by a Meta Field called count_field that specifies the size of the BLOB in bytes.

A variable_length_field used within a record can be optional at runtime. This is specified by setting the attribute called optional to true, else it must be set to false.

The name of the variable_length_field must be specified using its name attribute. The name must be a valid identifier (refer Appendix A.4) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the variable_length_field.

Examples:

```
<variable_length_field name="JPEG_frame" field_format="JPEG" optional="false">
  <count_field field_type_unsigned="unsigned integer"/>
</variable_length_field>
```

variable_format_field

A variable_format_field is a Simple Field that contains a BLOB whose format can vary at runtime. This field is preceded by an ordered sequence of two Meta Fields called format_field and count_field in that order. These Meta Fields specify the format and size of the BLOB in bytes respectively.

A variable_format_field used within a record can be optional at runtime. This is specified by setting the attribute called optional to true, else it must be set to false.

The name of the variable_format_field must be specified using its name attribute. The name must be a valid identifier (refer Appendix A.4) and it must be unique within the scope of the encapsulating record. The interpretation attribute is an optional attribute that may be used to provide a textual description of the variable_format_field.

Examples:

```
<variable_format_field name="video_frames" optional="false">
  <format_field>
    <format_enum index="0" field_format="MJPEG"/>
    <format_enum index="1" field_format="MPEG-1"/>
    <format_enum index="2" field_format="MPEG-2"/>
    <format_enum index="3" field_format="MP4"/>
  </format_field>
  <count_field field_type_unsigned="unsigned short integer"/>
</variable_format_field>
```

5.3 Meta Fields

Meta Fields are additional fields that are required by some Composite and Simple Fields to encode meta-data, i.e. structural and/or semantic information about the encoded data. Meta Fields are used in fields whose structure and semantics can vary during run-time. The Meta Fields are defined below.

count_field

The count_field specifies the size of the Composite or Simple Field that it is associated with.

The type of the count_field must be specified by its field_type_unsigned attribute. The type must be one of the unsigned integer types: "byte", "unsigned short integer", "unsigned integer" or "unsigned long integer".

The count_field has optional attributes for specifying a minimum count, maximum count and a textual interpretation. The values of the attributes for specifying the maximum and minimum counts must be integer constants or references to integer constants (see Section 5.5)

Examples: Refer list *and* variable_length_field *examples.*

vtag_field

The vtag_field specifies the index of the chosen Composite Field from the sequence of Composite Fields within a variant.

The type of the vtag_field must be specified by its field_type_unsigned attribute. The type must be one of the unsigned integer types: "byte", "unsigned short integer", "unsigned integer" or "unsigned long integer".

The vtag_field has optional attributes for specifying a minimum count, maximum count and a textual interpretation. The values of the attributes for specifying the maximum and minimum counts must be integer constants or references to integer constants (see Section 5.5)

Examples: Refer variant *examples.*

type_and_units_field

A type_and_units_field is an unsigned byte that specifies the type and units of a variable_field at runtime. type_and_units_field uses a Field Information Element called type_enum to provide an enumeration of allowed combinations of type and units for the variable_field.

Examples: Refer variable_field *examples.*

format_field

A format_field is an unsigned byte that specifies the format of a variable_format_field at runtime. format_field uses a Field Information Element called format_enum to provide an enumeration of allowed BLOB formats for the variable_format_field.

Examples: Refer variable_format_field *examples.*

presence_vector

The presence_vector is an optional field that is used in a record and a sequence. When used, it must be the first field within the record or sequence. It is used to indicate the presence or absence of optional Simple Fields when used in a record, or groups of Simple Fields (encapsulated in a record, list, variant or a sub-sequence) when used in a sequence. Simple Fields or groups of Simple Fields that are optional must be marked as being optional by setting their boolean attribute called optional to “true”. The first optional Simple Field in the record maps onto the least significant bit of the presence_vector. Subsequent optional Simple Fields map onto subsequent bits of the presence_vector in the order in which they occur in the record. The same mapping rule applies when the presence_vector is used in a sequence containing groups of Simple Fields. The number of bits in the presence_vector must be greater than or equal to the number of optional Simple Fields in the record.

The presence_vector must be of one of the unsigned integer types that can be specified by its field_type_unsigned attribute. It can map up to a maximum of 64 optional Simple Fields when its data type is set to an unsigned long integer. If a record contains more than 64 optional Simple Fields, it must be split into two or more consecutive records.

Examples:


```
<service_def xmlns="urn:jaus:jsidl:1.0" name="Service_X" id="http://xyz.com" version="1.1">
  <description/>
  <assumptions/>
  <message_set>
    <message_def name="Report_Global_Pose" message_id="0002" type="output" is_command="false">
      <description/>
      <body>
        <record name="record_w_optional_fields">
          <presence_vector field_type_unsigned="unsigned byte"/>
          <!–roll, pitch and yaw map to the first three bits of the presence vector -->
          <fixed_field name="Roll" field_type="short integer" field_units="radian" optional="true">
            <scale_range real_lower_limit="-PI" real_upper_limit="PI" integer_function="round"/>
          </fixed_field>
          <fixed_field name="pitch" field_type="short integer" field_units="radian" optional="true">
            <scale_range real_lower_limit="-PI" real_upper_limit="PI" integer_function="round"/>
          </fixed_field>
          <fixed_field name="yaw" field_type="short integer" field_units="radian" optional="true">
            <scale_range real_lower_limit="-PI" real_upper_limit="PI" integer_function="round"/>
          </fixed_field>
        </record>
```

```
      </body>
    </message_def>
  </message_set>
  <protocol_behavior/>
</service_def>
```

More Examples: Refer record examples.

5.4 Field Information Attributes and Elements

Field Information Attributes and Elements are used in Data Field Elements to provide information about the encoded data. Figure 14 lists these attributes and elements.

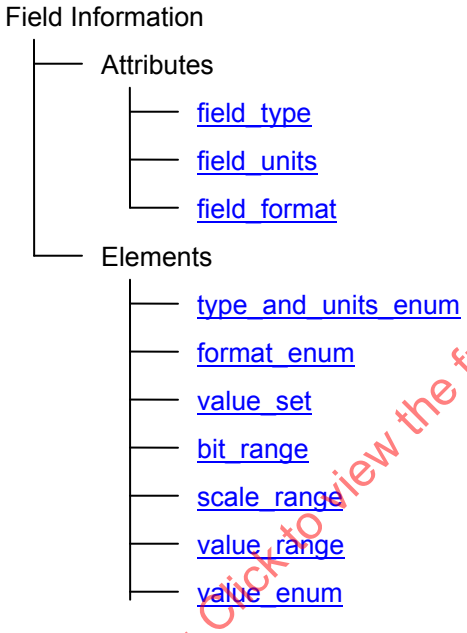

FIGURE 14 - FIELD INFORMATION ATTRIBUTES AND ELEMENTS - PROVIDE INFORMATION ABOUT ENCODED DATA

5.4.1 Field Information Attributes

field_type

The field_type attribute specifies primitive data types as listed in table 1 below.

TABLE 1 - SIZE AND REPRESENTATION OF PRIMITIVE DATA TYPES

XS	Data Type	Size	Representation
xsd:byte	byte	1	8 bit signed integer
xsd:short	short integer	2	16 bit signed integer
xsd:integer	integer	4	32 bit signed integer
xsd:long	long integer	8	64 bit signed integer
xsd:unsignedByte	unsigned byte	1	8 bit unsigned integer
xsd:unsignedShort	unsigned short integer	2	16 bit unsigned integer
xsd:unsignedInt	unsigned integer	4	32 bit unsigned integer
xsd:unsignedLong	unsigned long integer	8	64 bit unsigned integer
xsd:float	float	4	32 bit floating point number (IEEE 754)
xsd:double	long float	8	64 bit floating point number (IEEE 754)

Examples: Refer fixed_field *examples.*

field_type_unsigned

The field_type_unsigned attribute specifies the sub-set of non-negative primitive data types from table 1 above.

Examples: Refer variable_format_field *examples.*

field_units

The field_units attribute specifies the units of data fields that are associated with units. JAUS mandates the use of the International System of Units as specified in NIST Special Publication 330, 1991 Edition [bnt]. See also Appendix A.2.

Examples: Refer fixed_field *, and* variable_field *examples.*

field_format

The field_format attribute specifies the format of message fields containing BLOB data.

AU
BMP
JPEG
MJPEG
MPEG-1
MPEG-2
MP2
MP3
MP4
RAW
WAV
JAUS_MESSAGE
XML
RNC
RNG
XSD
or, any other user defined format

Examples: Refer variable_length_field *and* variable_format_field *examples.*

5.4.2 Field Information Elements

type_and_units_enum

Each type_and_units_enum element specifies an enumeration constant whose value is given by the index attribute. Each constant value is mapped onto a type and units pair that are specified by the field_type and field_units attributes. Each type_and_units_enum can have a value_set or scale_range defined for the corresponding field_type and field_units pair.

Examples: Refer variable_field *examples.*

format_enum

Each format_enum element specifies an enumeration constant whose value is given by the index attribute. Each constant value is mapped onto a BLOB format that is specified by the field_format attribute.

Examples: Refer variable_format_field examples.

value_set

A value_set specifies one or more value ranges and/or value enumerations using value_range and value_enum Field Information Elements. When a value_set is used in a field definition, it restricts the values that the field can take to the values defined by the encapsulated value_ranges and value_enums. When the value_set contains both value ranges and value enumerations, the value_set is the union of both. Value enumerations that intersect with value ranges, merely add mnemonics to to the intersecting values. If multiple value_ranges are specified, then the union of those ranges must be taken. A value_set containing two value_enums with the same index value is not allowed.

value_set contains an attribute called offset_to_lower_limit whose value is of type boolean. When the value of this attribute is set to true, it implies that the lower limit of the value_set is mapped to the smallest value that the field can hold. The example below shows a case of a fixed_field that can hold values in the range (-128,127), but whose actual range is (2000, 2100). In this case, year 2000 is mapped onto -128, and a value of -78 implies year 2050 or the enum constant "Age of Cyborgs". The offset_to_lower_limit attribute is a bandwidth optimization feature that allows fields that have large values and small net ranges to be encoded into small data types.

Examples:

```
<fixed_field name="year" field_type="byte" field_units="one" optional="false">
  <value_set offset_to_lower_limit="true">
    <value_range lower_limit="2000" lower_limit_type="inclusive" upper_limit="2100"
upper_limit_type="inclusive"/>
    <value_enum enum_index="2000" enum_const="'Robotic Revolution'"/>
    <value_enum enum_index="2050" enum_const="'Age of Cyborgs'"/>
    <value_enum enum_index="2100" enum_const="'Star Wars'"/>
  </value_set>
</fixed_field>
```

More Examples: Refer fixed_field examples.

bit_range

A bit_range specifies the range of bits for each sub_field contained in a bit_field. The from_index and to_index attributes give this range and the indices are always numbered such that 0 corresponds to the least significant bit (refer Message Encoding). A textual interpretation of the bit range may be provided by the optional interpretation attribute.

Examples: Refer bit_field examples.

scale_range

When this element is used in the definition of a data field, it implies that the field represents a scaled integer. Scaled integers are integer types that represent real values. The real values are first bounded by upper and lower limits and are then mapped onto the integer range of the integer field type used for the scaled integer. Each scaled integer is associated with a bias and scale factor. The bias and scale factor of a real number represented as an unsigned integer is as shown below. In addition to this, the lower and upper limit types must be set to either "inclusive" or "exclusive". An attribute called integer_function must be used to specify the type of rounding that is performed when the real value is converted to an integer. The allowable types are "round", "floor" and "ceiling".

$$\text{Integer\_Range} = 2^{(\text{number\_of\_bits})} - 1 \qquad \text{(Eq. 1)}$$

$$\text{Scale\_Factor} = \frac{(\text{Real\_Upper\_Limit} - \text{Real\_Lower\_Limit})}{\text{Integer\_Range}}$$

$$\text{Bias} = \text{Real\_Lower\_Limit}$$

Once the scale factor and bias are computed from the real limits, the conversion from scaled value to real value and vice versa can be done using the following equations.

$$\text{Scaled\_Value} = \text{Round}\left(\frac{\text{Real\_Value} - \text{Bias}}{\text{Scale\_Factor}}\right) \quad \text{(Eq. 2)}$$

$$\text{Real\_Value} = \text{Scaled\_Value} \times \text{Scale\_Factor} + \text{Bias}$$

Examples:

An example of using a scaled integer value follows. Given a lower limit value of –100, and an upper limit value of 100, the scale factor and bias for a short integer type would be.

$$\text{Scale\_Factor} = \frac{(100 - (-100))}{2^{16} - 1} = 0.00305180 \quad \text{(Eq. 3)}$$

$$\text{Bias} = -100$$

To convert the real value of 30 to its scaled integer representation, the following formula is used.

$$\text{Scaled\_Value} = \frac{(30.0 - \text{Bias})}{\text{Scale\_Factor}} = 42598 \quad \text{(Eq. 4)}$$

The entity that receives the scaled value (42598) can determine the real number value with the following calculation.

$$\text{Real\_Value} = 42598 \times \text{Scale\_Factor} + \text{Bias} = 30.0005764 \quad \text{(Eq. 5)}$$

The value of the lower and upper limits of a scale_range must be valid integer constant or a reference to a valid declared constant of an integer type.

Refer fixed_field *examples.*

value_range

A value_range specifies a range of values for a data field. The range is given by the lower_limit and upper_limit attributes. In addition to this, the lower and upper limit types must be set to either “inclusive” or “exclusive”. A textual interpretation of the value range may be provided by the optional interpretation attribute.

Examples: Refer fixed_field *examples.*

value_enum

Each value_enum element specifies a mapping between an enumeration string constant given by the enum_const attribute, and an integer value given by the enum_index attribute. A textual interpretation of the enumeration constant may be provided by the optional interpretation attribute. The value of the enum_const attribute can be either a string literal, or a reference to a constant definition (see Section 5.5) that defines a string literal. If the value is a string literal, it must be wrapped in single quotes (for example, enum_const =“ ‘string literal’ ”). If the value is a reference to a constant definition it must not (for example, enum_const = “ name reference ”).

Examples: Refer fixed_field *examples.*

5.5 Declared Types and Constants

Common data types and constants that are used frequently in several services, or in several places within a service may be declared within the declared_type_set and declared_const_set elements and then referenced where needed. These elements may be defined within a service definition as a child element of service_def, or they may be defined in a separate XML document independent of a service definition as shown in the example below. Declared types can be message definitions and sections of message definitions like header, body, footer, and Simple and Composite fields. A declared constant can be used to specify integer, floating point and string constants. Type and constant definitions that are defined within a service definition fall under the namespace and scope of the service definition's id and version. It is therefore an error if an id and version pair that is different from that of the encapsulating service definition is specified for the declared_type or declared_const elements. In genereal, it is not required to respecify the id and version pair for the declared_types and declared_const elements when these element are specified within a service definition, but, the attirbute pair must be specified when these elements are defined independent of a service definition.

Examples:

```
<!-- a declared types and constant declared within a service definition -->
<service_def xmlns="urn:jaus:jsidl:1.0" name="GlobalPose" version="0.1" id="urn:jaus:GlobalPose">
  <description>
    Describe GlobalPose message set here.
  </description>
  <assumptions/>

  <declared_const_set name="basic_constants">

    <const_def name="PI" const_type="long float" const_value="3.14159265358979323846"
field_units="one" interpretation="ratio of any circle's circumference to its diameter"/>

    <const_def name="e" const_type="long float" const_value="2.7182818284590452354"
field_units="one" interpretation="base of the natural logarithm"/>

    <const_def name="PLUS_100" const_type="long float" field_units="one" const_value="100.0"/>

    <const_def name="MINUS_100" const_type="long float" field_units="one" const_value="-100.0"/>
  </declared_const_set>

  <declared_type_set name="global_pose">
    <fixed_field name="positionRms" field_type="unsigned integer" field_units="meter"
optional="true" interpretation="A RMS value indicating the validity; of the position data.">
      <scale_range integer_function="round" real_lower_limit="MINUS_100" real_upper_limit="PLUS_100"
/>
    </fixed_field>
  </declared_type_set>

  <message_set>

    <input_set> ... </input_set>
    <output_set>
      <message_def is_command="true" message_id="4402" name="ReportGlobalPose">
        <description>Report Global Pose</description>
        <header/>
        <body>
          <record name="GlobalPoseRec" optional="false">
            <declared_fixed_field name="positionRMS" optional="true"/>
            ...
          </record>
        </body>
        <footer/>
      </message_def>
    </output_set>
    <internal_set/>
  </message_set>
```

```
  <protocol_behavior>
    ...
  </protocol_behavior>
  <service_def>
```

Service definitions and declared_type_sets can reference other declared type sets and declared constant sets using the declared_type_set_ref and declared_const_set_ref child elements of declared_type_sets. A declared_const_set can reference other declared_const_sets, but not a declared_type_set. The referenced name is resolved using the "." operator. As shown in the example below for BasicTypes.GlobalPoseWithTimeStamp. Here BasicTypes is the name of the declared_type_set reference and GlobalPoseWithTimeStamp is the name of the declared_type that is being referenced. In general, the name of a declared_type_set_ref attribute is of the form (declared_type_set_ref name.)*declared_type name, and this conforms with the namespaced identifier patter called identifier_ns (see pattern.rnc). When making such references, a new name that is unique within the scope of the referencing declared_type_set or declared_const_set must be assigned to the reference and then used locally as an alias to the referenced definition.

A constant definition must specify the field units for the constant. This is important for the validation against the use of the constant in a field. A field definition that references a constant must agree in type and units to the constant definition. In cases where there is type coercion, type agreement can follow the type conversion rules of ansi C [ansic].

The definition of a declared_type cannot be overriden in a service definition.

Examples:

```
<!-- a declared type set defined in a separate file -->
<declared_type_set name="BasicTypes" xmlns="urn:jaus:jsidl:1.0" id="some_uri" version="1.1">

   <!-- example of a complex declared type -->
  <bit_field name="TimeStamp" field_type_unsigned="unsigned integer" optional="true"
    interpretation="5 subfields">
    <sub_field name="milliseconds" interpretation="0-999">
      <bit_range from_index="0" to_index="9"/>
      <value_set offset_to_lower_limit="false">
        <value_range lower_limit="0" upper_limit="999" lower_limit_type="inclusive"
          upper_limit_type="inclusive"/>
      </value_set>
    </sub_field>
    <sub_field name="seconds" interpretation="0-59">
      <bit_range from_index="10" to_index="15"/>
      <value_set offset_to_lower_limit="false">
        <value_range lower_limit="0" upper_limit="59" lower_limit_type="inclusive"
          upper_limit_type="inclusive"/>
      </value_set>
    </sub_field>
    <sub_field name="minutes" interpretation="0-59">
      <bit_range from_index="16" to_index="21"/>
      <value_set offset_to_lower_limit="false">
        <value_range lower_limit="0" upper_limit="59" lower_limit_type="inclusive"
          upper_limit_type="inclusive"/>
      </value_set>
    </sub_field>
    <sub_field name="hours" interpretation="0-23">
      <bit_range from_index="22" to_index="26"/>
      <value_set offset_to_lower_limit="false">
        <value_range lower_limit="0" upper_limit="23" lower_limit_type="inclusive"
          upper_limit_type="inclusive"/>
      </value_set>
    </sub_field>
    <sub_field name="days" interpretation="1-31">
      <bit_range from_index="27" to_index="31"/>
      <value_set offset_to_lower_limit="false">
        <value_range lower_limit="1" upper_limit="31" lower_limit_type="inclusive"
          upper_limit_type="inclusive"/>
      </value_set>
    </sub_field>
  </bit_field>
```

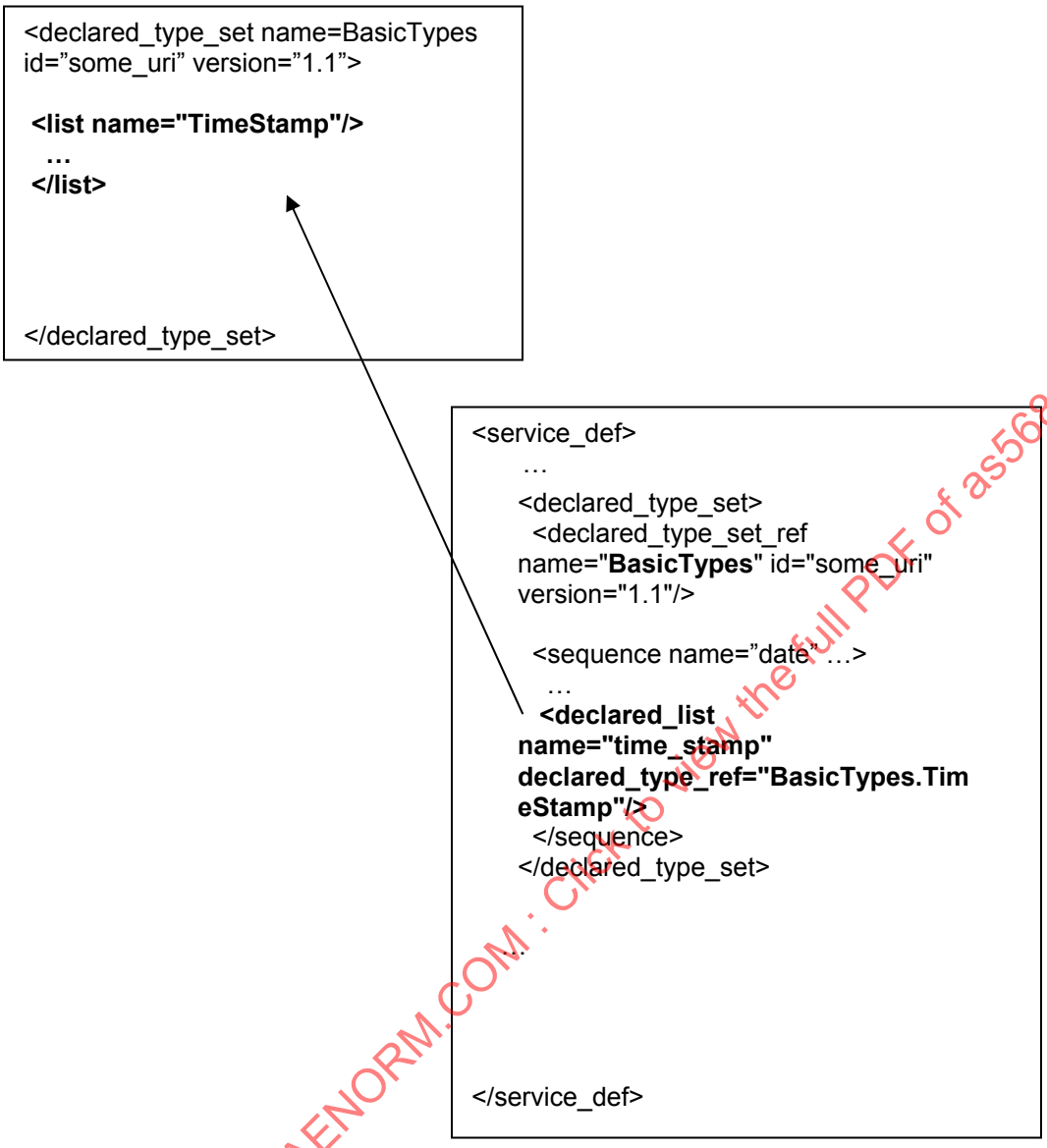

FIGURE 15 - EXAMPLE OF A REFERENCE TO A DECLARED_TYPE

```
<!-- examples of declared records -->
<record name="GeodeticCoordinates" optional="false">
  <presence_vector field_type_unsigned="unsigned byte"/>
  <fixed_field name="Latitude" field_type="integer" field_units="degree" optional="true">
    <scale_range real_lower_limit="-90" real_upper_limit="90" integer_function="round"/>
  </fixed_field>
  <fixed_field name="Longitude" field_type="integer" field_units="degree" optional="true">
    <scale_range real_lower_limit="-180" real_upper_limit="180" integer_function="round"/>
  </fixed_field>
  <fixed_field name="Elevation" field_type="integer" field_units="meter" optional="true">
    <scale_range real_lower_limit="-10000" real_upper_limit="35000" integer_function="round"/>
  </fixed_field>
</record>
```

```
  <record name="Orientation" optional="false">
    <presence_vector field_type_unsigned="unsigned byte"/>
    <fixed_field name="Roll" field_type="short integer" field_units="radian" optional="true">
      <scale_range real_lower_limit="-PI" real_upper_limit="PI" integer_function="round"/>
    </fixed_field>
    <fixed_field name="Pitch" field_type="short integer" field_units="radian" optional="true">
      <scale_range real_lower_limit="-PI" real_upper_limit="PI" integer_function="round"/>
    </fixed_field>
    <fixed_field name="Yaw" field_type="short integer" field_units="radian" optional="true">
      <scale_range real_lower_limit="-PI" real_upper_limit="PI" integer_function="round"/>
    </fixed_field>
  </record>

  <!-- example of a declared type that is composed of other declared types -->
  <sequence name="GlobalPose" optional="false">
      <declared_record name="Position" declared_type_ref="GeodeticCoordinates"/>
      <declared_record name="Orientation" declared_type_ref="Orientation"/>
    </list>
  </sequence>

  <!-- example of declared type that is composed of other declared types -->
  <sequence name="GlobalPoseWithTimeStamp" optional="false">
    <record name="TimeStampRec" optional="false">
      <declared_bit_field name="TimeStamp" declared_type_ref="TimeStamp" optional="false"/>
    </record>
    <list name="PositionList" optional="false">
      <count_field field_type_unsigned="unsigned byte"/>
      <declared_record name="Position" declared_type_ref="GeodeticCoordinates"/>
    </list>
    <list name="OrientationList" optional="false">
      <count_field field_type_unsigned="unsigned byte"/>
      <declared_record name="Orientation" declared_type_ref="Orientation"/>
    </list>
  </sequence>

</declared_type_set>

<!--A service that uses declared data types -->
<service_def xmlns="urn:jaus:jsidl:1.0" id="" name="uses_declared_types" version="1.1">
  <description>…</description>
  <assumptions/>

<!-- declared types defined within the service definition -->
  <declared_type_set name="declared_types">
    <declared_type_set_ref name="BasicTypes" id="some_uri" version="1.1"/>

    <!-- examples of declared records -->
    <record name="geodeticCoordinates" optional="false">
      <presence_vector field_type_unsigned="unsigned byte"/>
      <fixed_field name="Latitude" field_type="integer" field_units="degree" optional="true">
        <scale_range real_lower_limit="-90" real_upper_limit="90" integer_function="round"/>
      </fixed_field>
      <fixed_field name="Longitude" field_type="integer" field_units="degree" optional="true">
        <scale_range real_lower_limit="-180" real_upper_limit="180" integer_function="round"/>
      </fixed_field>
      <fixed_field name="Elevation" field_type="integer" field_units="meter" optional="true">
        <scale_range real_lower_limit="-10000" real_upper_limit="35000" integer_function="round"/>
      </fixed_field>
    </record>
  </declared_type_set>
```

```
  <!-- examples of referenced declared types -->
  <message_set>
    <input_set/>
    <output_set>
      <message_def name="report_global_pose" message_id="0001" is_command="false">
        <description/>
        <header/>
        <body>
          <declared_list name="global_pose_with_time_stamp"
declared_type_ref="BasicTypes.GlobalPoseWithTimeStamp"/>
        </body>
        <footer/>
      </message_def>
      <message_def name="report_local_and_global_pose" message_id="0002" is_command="false">
        <description/>
        <header/>
        <body>
          <sequence name="local_and_global_pose_seq" optional="false">
            <record name="local_pose" optional="false">
              <fixed_field name="x" field_type="integer" field_units="meter" optional="false">
                <scale_range real_lower_limit="0" real_upper_limit="10000" integer_function="round"
                />
              </fixed_field>
              <fixed_field name="y" field_type="integer" field_units="meter" optional="false">
                <scale_range real_lower_limit="0" real_upper_limit="10000" integer_function="round"
                />
              </fixed_field>
              <fixed_field name="heading" field_type="integer" field_units="radian"
optional="false">
                <scale_range real_lower_limit="0" real_upper_limit="6.2831" integer_function="round"
                />
              </fixed_field>
            </record>
            <!-- a declared_type defined external to this service -->
            <declared_list name="global_pose" declared_type_ref="BasicTypes.GlobalPose"/>
          </sequence>
        </body>
        <footer/>
      </message_def>
      <message_def name="report_global_coordinates" message_id="0003" is_command="false">
        <description/>
        <header/>
        <body>
          <!-- a declared type defined within this service -->
          <declared_record name="position_rec" declared_type_name="geodetic_coordinates"/>
        </body>
        <footer/>
      </message_def>
    </output_set>
    <internal_set/>
  </message_set>
  <protocol_behavior is_stateless="true">
   ...
   </protocol_behavior>
</service_def>
```


6. BEHAVIOR

6.1 Separation of Protocol Behavior from Application Behavior

The behavior of a service consists of a set of rules that govern the exchange and processing of messages between the service and its environment. This behavior can be modeled using a typical finite state machine model that consists of a set of states, where each state consists of a set of state transitions, and each state transition further consists of a sequence of transition actions.

Since JSDs are only required to define the portion of a service's behavior that is directly related to the exchange of messages, the behavior of a service is separated into two parts called protocol behavior and application behavior. The protocol behavior part consists of the rules that govern the exchange of messages, while the application behavior part consists of the rules that govern the processing of the messages being exchanged. This separation is made by introducing an interface between the two behaviors as shown in Figure 16 below. On one side, the protocol behavior part interfaces with the service's environment via the input and output message sets. On the other side, it interfaces with the application behavior via transitions that are triggered by internal events defined by the service and transition actions. This separation allows the service definition to abstract away the application behavior part.

Since the communication across the interface between the protocol behavior and the application behavior part cannot be tested externally (across-the-wire), this communication is considered informative. The communication between the service's protocol behavior partition and its envoronment can be tested externally and is therefore considered normative.

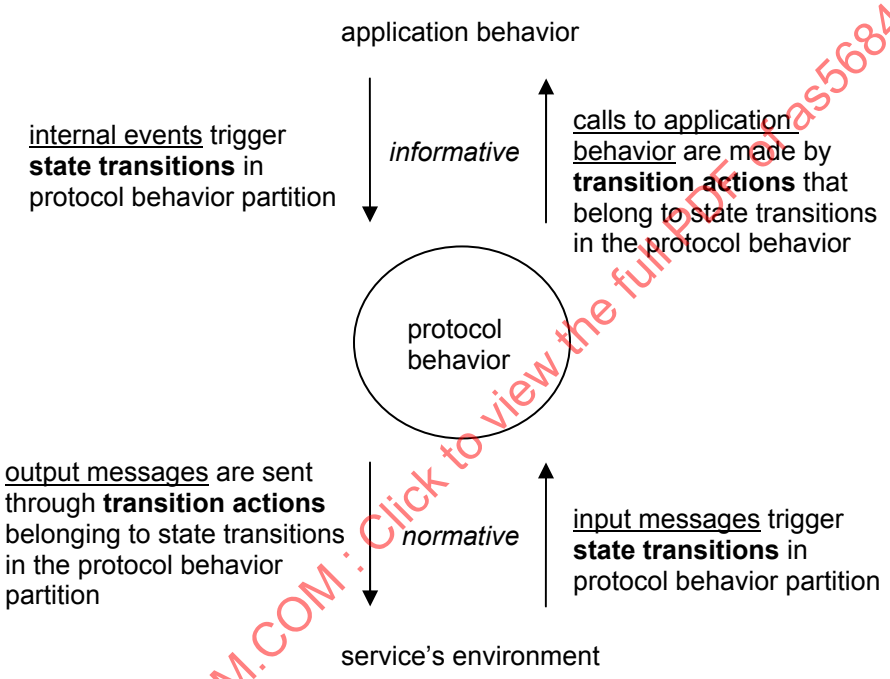

FIGURE 16 - SEPARATION OF PROTOCOL BEHAVIOR FROM APPLICATION BEHAVIOR

6.2 Protocol Behavior

In keeping with Holzmann's [holz] definition (refer Section 3) of a complete protocol, the protocol behavior of a service is defined using a finite state machine model. A Relax NG Compact schema of this model is presented in Appendix A.3. The root element for the definition of the finite state machine is called protocol_behavior. This element may contain nested and concurrent finite state machines. The protocol_behavior element has a boolean attribute called is_stateless whose value must be set to true only if the service does not store data received from its clients during the course of its execution. If a service is stateless, clients to the service may interact with any instance of the service and may even switch from one instance of the service to another during the course of their interaction with the service at runtime. Clients of stateful services must be bound at runtime to particular instances of the service from the beginning to the end of their interaction with the service. This is because the protocol behavior of the service is dependent on the client data that is collected and stored by the service.

```
<protocol_behavior is_stateless="true">

  <!—A set of start states -->
  <start state_machine_name="FSM_A" state_name="Init"/>
  <start state_machine_name="FSM_B" state_name="Init"/>

 <!—A set of concurrent state machines -->
 <state_machine name="FSM_A">
   <state name="Init">
     <transition name="Message1"> … </transition>
     <transition name="Message2"> … </transition>
   </state>
   <state name="Running">
     <transition name="Message1"> … </transition>
     <transition name="InternalEvent2"> … </transition>
   </state>
 </state_machine>

 <state_machine name="FSM_B">
   <state name="Init">
     <transition name="Message1"> … </transition>
     <transition name="Message2"> … </transition>
   </state>
   <state name="Running">
     <transition name="Message1"> … </transition>
     <transition name="InternalEvent2"> … </transition>
     …
   </state>
 </state_machine>
 …
```


FIGURE 17 - STRUCTURE OF THE PROTOCOL_BEHAVIOR ELEMENT

The protocol_behavior element has two child elements, start and state machine. start defines the set of start states of the service, one per state machine. The protocol behavior can consist of one or more concurrent state machines that are completely independent of one another. Two or more concurrent state machine may be used to break down large and complex behavior into simple pieces.

The start element specifies the names of the states that each of the state machines must be in when the service first comes to life.

Each state machine element has an attribute called name that must also be a valid namespaced identifier (refer Appendix A.4) of the form (service_name.)*state_machine_name. The state machine name must be unique within the scope of its parent protocol_behavior element and is required only if the state machine of an inherited service is being extended. Similarly, the name of a state must be unique within the scope of its parent state machine. The protocol_behavior element must contain at least one state machine, and each state machine must contain at least one state. Each state may also contain an optional nested state called the default state which is described in Section 6.3.

6.3 States

Each state element has a name attribute that must be a valid identifier (refer Appendix A.4) and an optional interpretation attribute that may be used to provide a brief textual interpretation for the state. The name of a state must also be unique within the scope of its parent element which may be the state machine, or another state element if the state is a nested state. The state element may contain zero or more outgoing transitions and at most one default transition. If a state does not define a transition, the state is said to be quiescent and may potentially be a state at which the finite state machine comes to rest. States also have child elements for defining entry and exit actions which are described below.

If a state contains nested states, it must specify the name of the initial nested state using its the attribute called initial_state. All transitions to the enclosing state are transitions to the nested state that is identified as the initial state. Each state may specify at most one special nested state called the default_state. The default_state is described in detail in Section 6.5 on default states and default transitions.

Examples:

```
<!-- A state with a simple transition -->
<state name="Suspended">
  <transition name="Start">
    <simple>
      <end_state state="Running"/>
    </simple>
    <!-- actions go here -->
  </transition>
</state>

<!-- A quiescent state -->
<state name="Deleted">
  …
</state>

<!-- nested states -->
<state name="On" initial_state="Display">
   <state name="Hide">
    …
   </state>
   <state name="Display">
   …
   </state>
</state>
```


6.3.1 Entry and Exit Actions

As their names suggest, entry and exit actions are associated with a state being entered or exited. Entry and exit actions are atomic actions that may be used to specify abstract application behavior like that of setting up a timeout or performing operations on a stored data. The only protocol related action that can be specified by an entry and exit action is that of sending an ouput message. This action must be specified by a special action called a send_action.

Entry and exit actions may or may not be executed depending on the type of transition that resulted in a state being entered or exited. The table below shows the types of transitions that affect the execution of entry and exit actions.

TABLE 2 - TYPES OF TRANSITIONS FOR WHICH ENTRY/EXIT ACTIONS ARE EXECUTED [SMC]

Transition Type	Execute "from" state's Exit Actions?	Execute "to" state's Entry Actions?
Simple Transition	Yes	Yes
Loop back Transition	No	No
Push Transition	No	Yes
Pop Transition	Yes	No

Entry and exit actions are executed in the order in which they are defined. Each action has a name attribute whose value must be a valid identifier (refer Appendix A.4), and must indicate the type of action being performed. For instance, an entry action that requires a camera to be initialized may be named initCamera. An ordered list of arguments may be specified for each entry and exit action. The values of these arguments are limited to primitive constants or string constants, both of which must be encased in double quotes and then in single quotes as shown below. The name along with the argument list makes an entry or exit action resemble a function call with no return type as in traditional programming languages like C and Java. Refer to Section 6.4 for more about entry and exit actions.

If a transition results in a nested state being entered, the entry and exit actions of the super-states are executed in the order of the nesting if and only if these actions are found to be executable based on the type of transition.

Examples:

```
<state name="some_state">
  <entry>
    <action name="Start_Timer" interpretation="Start the timer for…">
      <argument constant= ' "Idle" '/>
      <argument constant=" '1' "/>
    </action>

    <!– send a Report Ready message -->
    <send_action name="Report_Ready"/>

  </entry>
  ...
</state>
```

6.4 Transitions

Transitions are defined within a state (the parent state) and indicate a state change from the enclosing state to a target (or end) state. Transitions are triggered only by the receipt of input message and internal events that are defined by the service (refer Section 6.1). The target state of a transition must be in the same state machine as the parent state[3]. A transition definition has a name attribute that is required, and may contain a parameter list. The value of the name attribute must be a valid namespaced identifier (refer Appendix A.4) of the form (service_name.)*(message_name | internal_event_name). The name of the message used denotes the trigger for the transition. The name along with the parameter list makes a transition resemble a function definition with no return type as in traditional programming languages like C and Java. The parameter list contains name references to data types which can include the input message or internal event that triggers the transition, or data types defined within the input message or internal event. Each parameter definition has two attributes: type and value. The type attribute must be assigned the name of the type being referenced, message_def, record etc. The value attribute must be assigned the name specified for the data type. (refer Appendix A.4). Transitions have an optional interpretation attribute that may be used to provide a brief textual interpretation.

Transitions may also have guards associated with them. Guard conditions are defined with the optional child attribute called guard. The guard condition must be a conditional expression made up of a set of one or more action names concatenated with the equality operators == and !=, the binary operators && (written as & in XML) and ||, and the unary negation operator !. If the actions use a variable name in their argument list, then that name must be declared in the transition's parameter list. The actions may also take constants as arguments. If an argument is a constant, it must be encased in double quotes and then in single quotes. The names of the actions must indicate the condition that is to be evaluated. Guarded transitions are executable only when they are triggered and their guard condition evaluates to true. The name, parameter list and guard condition together form the signature of a transition. The signatures of transitions defined within a state must be unique in at least one of these three components.

[3] No forking or joining operations are allowed between the concurrent states.

Transitions can also have one or more actions associated with them. These actions are executed just before the target state of the transition is entered and in the order in which they are defined. Each action has a name attribute whose value must be a valid identifier (refer Appendix A.4). A sequence of arguments may be specified for each transition action. If these actions specify a variable name as part of their arguments, then that variable must be defined in the transition's parameter list. The actions may also take constants as arguments. If an argument is a constant, it must be encased in double quotes and then in single quotes. Similar to entry and exit actions, the sequence of transition actions may include the special send_action for sending out an output message. Refer Section 6.1 for more about transition actions.

A transition in one state machine can be used to invoke an internal event in a concurrent state machine if the action of the invoking transition is named after the internal event for which a transition has been defined in the concurrent state machine. Transitions for the internal event used in this case must not be defined in the state machine that contains the invoking transition. If the internal event is defined for more than one concurrent state machine, all of these state machine's transitions are invoked simultaneously.

Examples:

```
<!-- A state with a simple transition containing transition parameters, a guard and transition
actions -->
<state name="Idle">
  <transition name="Run">
    <parameter type="message_def" variable="Run"/>
    <guard condition="is_Processor_Available() == true &amp;&amp; isValid( Run )"/>
    <simple>
      <end_state state="Running"/>
    </simple>
    <action name="Stop_Timer">
      <!-- String constant encased in double quotes and then in single quotes -->
      <argument constant= " 'Idle' "/>
    </action>
    <action name="Do_Work">
      <argument variable="Run"/>
    </action>
  </transition>

  <transition name="Stop">
    <simple>
      <end_state state="Idle"/>
    </simple>
    <action name="Clean_Up"/>
  </transition>
</state>
```


There are four basic types of transitions: Simple Transitions, Loopback Transitions, Push Transitions and Pop Transitions, and one special type called the default transition. The basic types are described in detail in later sub-sections, while default transitions are described in Section 6.5.

Transitions are executed when they are found to be executable. Once a state is entered, its transitions are evaluated for executability in the order in which they are defined. If two or more transitions are executable at the same time, the transition that is defined highest in a top-down order among the ones that are executable is the transition that is chosen to be executed.

The following implicit rule is made to help simplify the protocol behavior description of service definitions: If a transition is not defined for a particular (state, input message, guard condition) or (state, internal event, guard condition) triple, then it implies that the receipt of the input message or internal event with the given guard condition has no effect (do nothing) on the protocol behavior for that state.

6.4.1 Simple Transitions

A simple transition as the name suggests, changes the state of the finite state machine from the state in which it is defined to a target or end_state. Simple transitions are specified with the transition's child element called simple. This element contains an end_state child element with which a target state can be specified.

Examples:

```
<!-- A state with a simple transition -->
<state name="Suspended">
  <transition name="Start">
    <simple>
      <end_state state="Running"/>
    </simple>
    <!-- actions go here -->
     ...
  </transition>
</state>
```

6.4.2 Loopback Transitions

A simple transition in which no end_state is defined is called a loopback transition. A loopback transition, causes the state in which it is defined to be re-entered. Loopback transitions may also be defined by specifying the end_state of the transition to be the parent state, the state in which the transition is defined.

Examples:

```
<!-- The second transition of the Idle state is a loop back
    transition -->
<state name="Idle">
  <transition name="Run">
    <guard condition="is_Processor_Available() == true &amp;amp;&amp;amp; connectionIsOpen()"/>
    <simple>
      <end_state state="Running"/>
    </simple>
    <action name="Stop_Timer">
      <argument constant= ' "Idle" ' />
    </action>
    <action name="Do_Work"/>
  </transition>
  <transition name="Run">
    <simple>
      <end_state state="Idle"/>
    </simple>
    <action name="Reject_Request"/>
  </transition>
</state>
```

6.4.3 Push Transitions

Push transitions are used to transition from the enclosing state to the specified end state such that the end state is re-entered once with every successive push and exited once with every pop transition and for the number of times a push transition was triggered. This operation is similar to the push and pop operations performed on a stack. Push transitions are defined by the transition element's child element called push. A push transition may have a simple transition defined within it. When specified, the simple transition is executed first and immediately followed by the execution of the actual push transition. The simple transition is executed only once and for the first time that the push transition is executed. This mechanism is provided in order to alter the pop transition behavior by specifying different end state than the parent state of the push transition.

Examples:

```
<state name="Running">
  <transition name="Blocked">
    <push>
      <end_state state="Blocked"/>
      <simple>
        <end_state state="Block_Pop"/>
      </simple>
    </push>
```

```
    <action name="Get_Resource"/>
  </transition>
</state>
```

This causes the state machine to:

1. Transition to the BlockPop state
2. Execute the BlockPop entry actions
3. Push to the Blocked state
4. Execute the Blocked entry actions

When the state machine issues a pop transition, control is returned to Block_Pop and the secondary pop transition is issued from there.

6.4.4 Pop Transitions

The pop transition is used to transition out of a state into which the state machine was pushed using a push transition. It is defined by the transition element's child element called pop. The pop transition is defined in the state into which the state machine is pushed and does not specify an end state. That is because the pop transition will return to whatever state issued the corresponding push transition. The pop transition has an optional attribute called transition which can be used to specify the name of a secondary transition that needs to be executed once a pop transition has been issued for every push transitions that was issued. This name must be the name of a transition defined in the enclosing state of the pop transition. If this secondary transition takes a sequence of arguments, the pop transition must specify the values of these arguments. The values of the arguments may be primitive constants, string constants or variable names. If the argument is a constant, it must be encased in double quotes and then in single quotes in order to make it distinguishable from a variable name. If the argument contains a variable name, it must be declared in the parent transition's parameter list.

Examples:

```
<!-- A state with two pop transitions -->
<state name="Waiting">
  <!-- In this example, if the input message is the message called Granted, then the state machine
returns to the corresponding state that did the push and then takes that state's OK transition. If
the input message is the message called Denied, the same thing happens except the FAILED transition
is taken -->
  <transition name="Granted">
    <pop transition="OK"/>
  </transition>
  <transition name="Denied">
    <pop transition="FAILED"/>
  </transition>
</state>
```


6.5 Default States and Default Transitions

Default States and Default Transitions are two ways of specifying default transitions. Default transitions are used to handle message receipts and internal events that have not been enabled in a state, that is, input messages and internal events that have no corresponding transitions defined. Another more useful application for default transitions is to make a single definition, rather than multiple definitions of a transition that is used in more than one state.

Every state contains an optional special nested state called default_state. Like other states, the default state may contain transition definitions within it. If a state receives an event for which it has no transition defined, the default state's transitions are evaluated. If the default state specifies a transition for the event then that transition is executed.

Examples:

```
<!-- A state with a default state -->
<state name="Task">
  <state name="Suspended">
    ...
  </state>
  <state name="Running">
```

```
    ...
  </state>
  <state name="Waiting">
    ...
  </state>
  <default_state>
    <!-- Valid run request but transition occurred in an
      invalid state. Send a reject reply to valid
      messages -->
    <transition name="Run">
      <parameter type="message_def" variable="Run"/>
      <guard condition="is_Processor_Available() == true &amp;amp;&amp;amp; is_Valid( msg ) == true"/>
      <!-- loop back transition -->
      <simple/>
      <action name="Reject_Request">
        <argument variable="Run"/>
      </action>
    </transition>

    <!-- Ignore invalid messages when received in invalid states -->
    <transition name="Run">
      <parameter type="message_def" variable="Run"/>
      <!-- loop back transition -->
      <simple/>
    </transition>

    <transition name="Shutdown">
      <simple>
        <end_state state="Shutdown"/>
      </simple>
      <action name="Start_Shutdown"/>
    </transition>
  </default_state>
</state>
```

Every state including the default state contains an optional special transition called default_transition. Again, if a state receives an event for which it has no transition defined, the default transition is executed if it is defined. Since any transition can fall through to a default transition, the default transition has no parameter list. Aside from this difference, a default transition is like any other transition.

Examples:

```
<state name="Connecting">
  <!-- We are now connected to the far-end. Now we can log on -->
  <transition name="Connected">
    <simple>
      <end_state state="Connected"/>
    </simple>
    <action name="Logon"/>
  </transition>
  <!-- Any other transition at this point is an error. -->
  <!-- Stop the connecting process and retry later -->
  <default_transition>
    <simple>
      <end_state state="Retry_Connection"/>
    </simple>
    <action name="Stop_Connecting"/>
  </default_transition>
</state>
```

If the nested default state of a state contains a default transition, it means that all transitions will be handled by that state. The default transition within the default state is the transition definition of last resort. The following list shows the transition precedence.

1. Guarded transition
2. Unguarded transition
3. The Default State's guarded transition
4. The Default State's unguarded transition
5. The current State's guarded Default transition
6. The current State's unguarded Default transition
7. The Default State's guarded Default transition
8. The Default State's unguarded Default transition

7. SERVICE REFERENCE RELATIONSHIPS

Services may reuse the definitions of other services through two reference relationships: inheritance and client.

7.1 Inheritance

The inheritance relationship provides a *reuse* mechanism by which a definition of one service can be reused in the definition of another service such that the resultant definition is the union of both. This relationship can help minimize the repetition that is caused especially when the same protocol behavior is required in multiple services. Without such a reuse mechanism, the same behavior will need to be respecified over and over again, wasting time, introducing inconsistencies and risking errors.

The service that is reused is called a base service and the service that reuses the base service is called the derived service. This relationship allows for the reuse of service definitions in a manner that guarantees that the base service is substitutable with the derived service as stated by the Liskov Substitution Principle [lsp]. The figure below shows the syntax that is required for one service to inherit from another. The id and version attributes are used to uniquely identify the base service that is being inherited. The name attribute acts as an alias for the inherited service's (id, version) pair. It must be assigned a name that is unique within the scope of the referencing service.

```
<service_def
  xmlns="urn:jaus:jsidl:0.11"
  name="specific_driver"
  id="urn:jaus:specific_driver"
  version="1.1">

  <description> … </description>

  <assumptions> … </assumptions>

  <references>
    <inherits_from
      name="generic_driver"
      id="urn:jaus:generic_driver"
      version="1.1"/>
  </references>
   ….
</service_def>
```

FIGURE 18 - THE INHERITS-FROM ELEMENT

The following rules apply to the inheritance relationship:

Single Inheritance

A service can inherit from at most one other service. But, multiple levels of inheritance is allowed. This means that a service can inherit from a service that inherits from another service, and so on.

Examples

Refer Section 8.

Addition of features

Messages defined in base services cannot be overridden by the derived service. But, the vocabulary of the base services can be extended by defining new messages within the derived service's message set element. The new resultant message set is the union of the message sets of the related services.

A derived service may be used to add nested states to any of the states of the base service state machines [(see Section 6.3 for a description of state definitions)]. To add nested states using the inheritance relationship, only the new set of nested states need to be defined with reference to the base service's states as shown in Figure 19 below.

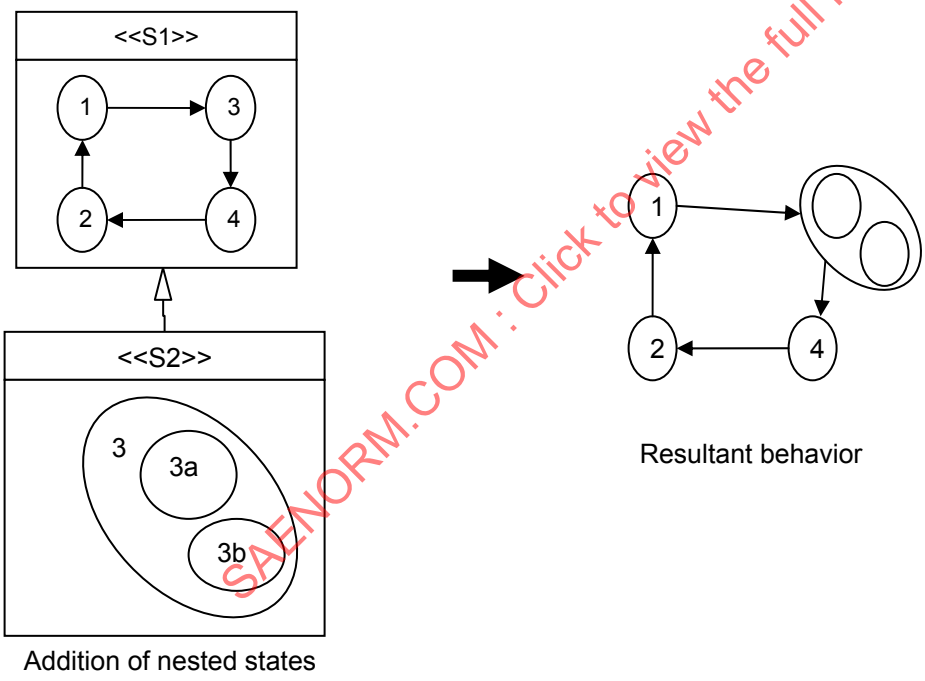

FIGURE 19 - ADDITION OF FEATURES – NESTED STATES

The derived service may be used to add a new concurrent state machine to the set of concurrent state machines defined by its base services. To add a concurrent state machine, only the new concurrent state machine needs to be defined in the derived service as shown in Figure 20 below.

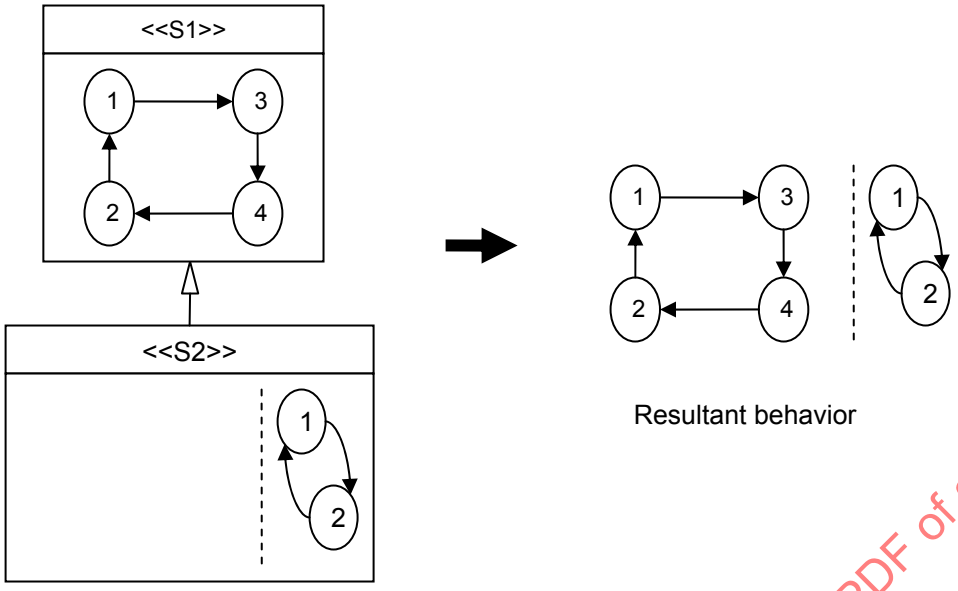

FIGURE 20 - ADDITION OF FEATURES – CONCURRENT STATE MACHINES

The derived service may be used to add new transitions to any one of the concurrent state machines of the base services. New transitions can be added with no restrictions for only the new set of input messages and events that the derived service defines (see Section 6.4 for the specification of transitions). But, these new transitions may send output messages defined by the base services as part of their transition actions. To add a new transition to a base service's state machine, only the new transition needs to be defined in relation to the base service's state machine as shown in Figure 21 below. Existing transitions of the base service do not have to be respecified in the derived service. New transitions may also be added for input messages and events defined in a base service and for which the base service does not explicitly specify the transition. Since the addition of such transitions replaces the implicit 'do_nothing' behavior specified in the base services, the addition of these transitions is considered as feature overriding. Overriding of features is described in the next sub-section.

If the trigger of an added transition is an internal event, care must be taken to ensure that the internal event is first initialized by another transition that is triggered by a newly defined input message. That is, the new internal event must be causally and temporally related to a new input message such that the internal event executes only after a transition to the related input message has been executed at least once. This constraint guarantees that services that are clients to only the base services are not exposed to the behavior associated with the internal events of the derived service. With this constraint, a service that is a client to the base service can still communicate with the base service without encountering new and unexpected behavior from a derived service. For better clarity, it is recommended that the causal relationship to the input message is specified in the interpretation attribute of the transition that is triggered by the internal event[4].

A derived service cannot remove any feature the belongs to its base services. This means that the derived service must use the same assumptions made by its base services.

Examples

Refer Section 8.

[4] Such a relation can be automatically validated.

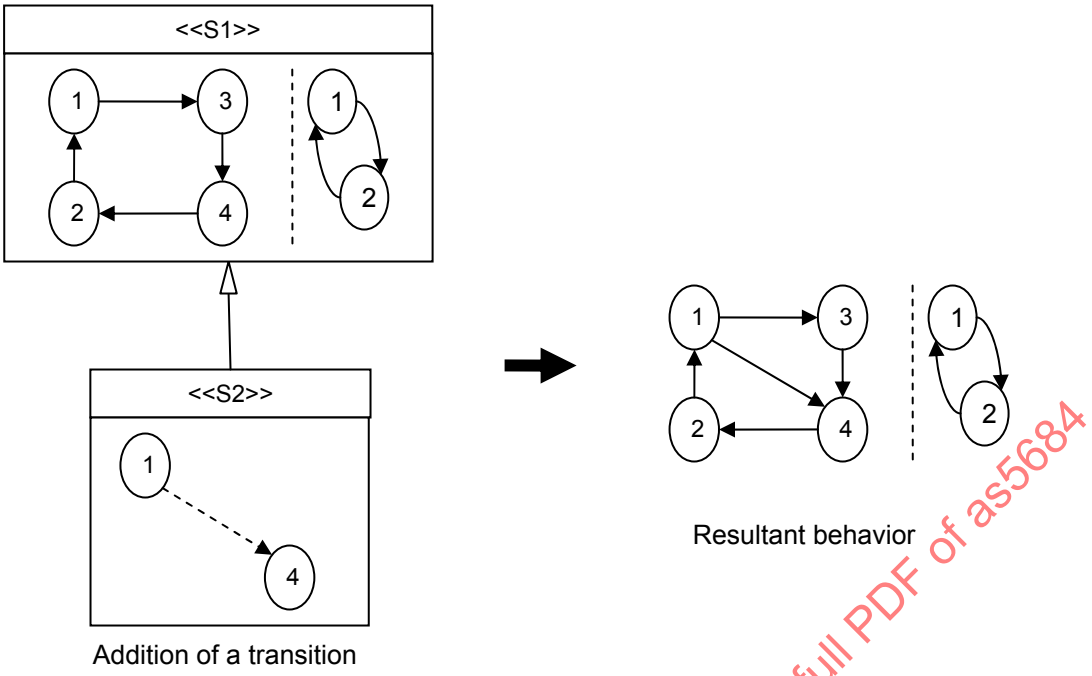

FIGURE 21 - ADDITION OF FEATURES - TRANSITION

Overriding of features

The only feature that may be overriden is transitions. In order for a base service to be subtype compatible with the derived service, the pre-condition on a transition in the derived service may only be weakened (made less stringent) relative to the pre-condition on the same transition in the base service. The post-condition on a transition in the derived service may only be strengthened (made more stringent) relative to the post-condition on the same transition in the base service.

The pre-condition of a transition consists of its source state, the data contained within the input message or internal event that triggers the transition and the guard condition of the transition. The post-condition of a transition consists of the transition actions and its end state.

The overriding transition in the derived service replaces the definition of the transition that it overrides in the base service. If multiple derived services in an inheritance chain override the same transition, then the overriding transition belonging to the last derived service to override the transition is the transition that replaces all other overriding transitions.

Examples

Refer Section 8.

```
<!-- transitions in the base service -->
<transition name="Login">
  <parameter type="Login" value="msg"/>
  <guard condition="isMember( msg )"/>
  <simple/>
  <action name="ReportStatus">
    <argument value=" 'LOGGED IN' "/>
  </action>
</transition>

<!-- overridden transition in a derived service with a stronger post-condition -->
<transition name="Login">
  <parameter type="Login" value="msg"/>
  <guard condition="isMember( msg )"/>
  <simple/>
  <action name="ReportStatus">
```

```
    <argument value=" 'LOGGED IN' "/>
  </action>
  <action name="startClock">
    <argument value=" '15' " interpretation="15 minutes access time"/>
  </action>
</transition>
```

7.2 Client

The client relationship may be used when the behavior of a service is dependent on the behavior of another service (the referenced service), but it is not necessary to make the signature of the other (referenced) service a part of the public interface of the referencing service. That is, the referenced service is completely hidden from (or private to) the users of the referencing service. But, by virtue of the reference, the referencing service will contain as part of its vocabulary, the message set of the referenced service. The input messages of the referenced service become output messages of the referencing service and vice versa. The state machine of the referenced service will also contain transitions corresponding to this vocabulary. The client service may also be used to define the protocol behavior of a pure client to another service.

Client references are defined using the client-of element shown in Figure 22 below. The id and version attributes are used to uniquely identify the service being referenced. The name attribute is used solely as an alias for the referenced service's (id, version) pair. This alias is used through out the service definition to make references to the service and its elements. The value of the name attribute must be unique within the scope of the service definition and among all service names used.


```
<service_def
  xmlns="urn:jaus:jsidl:0.11"
  name="driver_client"
  id="urn:jaus:driver_client"
  version="1.1">

  <description> … </description>

  <assumptions> … </assumptions>

  <references>
    <client_of
      name="driver"
      id="urn:jaus:driver"
      version="1.1"/>
  </references>
   ….
</service_def>
```

FIGURE 22 - THE CLIENT-OF ELEMENT

Example

```
<!-- A service that is also a client of another service -->
<?xml version="1.0" encoding="UTF-8"?>
<service_def xmlns="urn:jaus:jsidl:0.10" name="Waypoint_Driver" version="1.1"
id="urn:jaus:usc:Waypoint_Driver">
  <description>
    ...
  </description>
  <assumptions>
    ...
  </assumptions>
  <references>
    <client_of name="gps" id="urn:jaus:usc:Global_Pose_Sensor" version="1.1"/>
  </references>
```

```
  <message_set>
    ...
  </message_set>
  <protocol_behavior>
    ...
  </protocol_behavior>
</service_def>

<!-- A pure client definition -->
<service_def xmlns="urn:jaus:jsidl:0.10" name="Waypoint_Driver_Client" version="1.1"
id="urn:jaus:usc:Waypoint_Driver_Client">
  <description>
    ...
  </description>
  <assumptions>
    ...
  </assumptions>
  <references>
     <client_of name="w_driver" id="urn:jaus:usc:Waypoint_Driver" version="1.1"/>
  </references>
  <!-- a pure client service has an empty message_set -->
  <message_set/>
  <internal_event_set/>
  <protocol_behavior>
    ...
  </protocol_behavior>
</service_def>
```

8. AN EXAMPLE OF A SERVICE DEFINITION

This section contains two examples, one of a simple JSD and another that extends the simple JSD by adding new features and overriding existing features. The base service in this example allows a user to log in and out of a system. The protocol behavior ensures that there can be at most one user logged in at any given time. Figure 23 illustrates the protocol behavior with the help of a UML State Machine diagram. The code listing that follows contains the entire service definition that is compliant with the schema in Appendix A.

The protocol behavior as a single state machine containing two states, LOGGED_OUT and LOGGED_IN. On initialization, it enters the LOGGED_OUT state and performs an entry action init( ) that initializes the system. The messages that belong to the service vocabulary are LOGIN, LOGOUT and LOGIN_ERROR. Internal actions include init( ) and isError(LOGIN). The state transitions are described in the following clauses.

- If a LOGIN message is received while the system is in the LOGGED_OUT state, and the isError guard condition evaluates to false, then the system transitions to the LOGGED_IN state.
- If a LOGIN message is received while the system is in the LOGGED_OUT state, and the isError guard condition evaluates to true, then the system outputs a LOGIN_ERROR message and transitions back to the LOGGED_OUT state. Note that the entry action init( ) is executed during this self-transition.
- If a LOGIN message is received while the system is in the LOGGED_IN state, the system outputs a LOGIN_ERROR message and transitions back to the LOGGED_IN state.
- If a LOGOUT message is received while the system is in the LOGGED_OUT state, the system does nothing and simply transitions back to the LOGGED_OUT state.
- If a LOGOUT message is received while the system is in the LOGGED_IN state, the system transitions back to the LOGGED_OUT state.

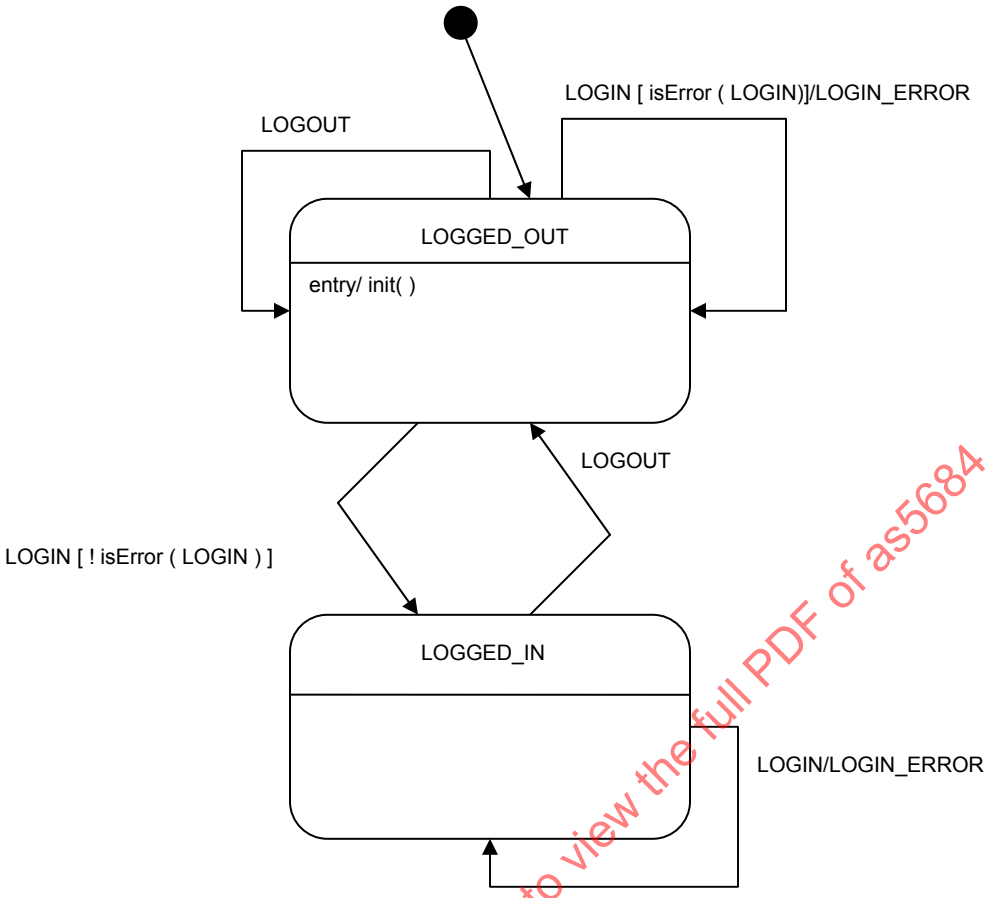

FIGURE 23 - UML STATE DIAGRAM OF THE ACCESS CONTROL SERVICE


```
<?xml version="1.0" encoding="UTF-8"?>
<service_def name="Access_Control" version="2.1" id="http://xyz.org/usc/core/Access_Control"
xmlns="urn:jaus:jsidl:1.0">
  <description>
    The Access Control service allows at most one user to be logged into a system.
  </description>
  <assumptions>
    This service is assumed to work with unreliable communication links where messages may be lost
or re-ordered.
  </assumptions>
  <message_set>
    <input_set>
      <message_def name="LOGIN" message_id="000d" is_command="true">
        <description>A command message.</description>
        <header name="emptyHeader"/>
        <body name="loginBody">
          <record name="User_Info_Rec" optional="false">
            <fixed_length_string name="User_Name" string_length="15" optional="false"/>
            <fixed_length_string name="Password" string_length="15" optional="false"/>
          </record>
        </body>
        <footer name="emptyFooter"/>
      </message_def>
      <message_def name="LOGOUT" message_id="000e" is_command="true">
        <description>A logout command message.</description>
        <header name="emptyHeader"/>
        <body name="emptyBody"/>
        <footer name="emptyFooter"/>
      </message_def >
    </input_set>
```

```
    <output_set>
      <message_def name="LOGIN_ERROR" message_id="000f" is_command="false">
        <description>A message for reporting error conditions</description>
        <header name="emptyHeader"/>
        <body name="emptyBody"/>
        <footer name="emptyFooter"/>
      </message_def >
    </output_set>
  </message_set>
  <internal_events_set/>
  <protocol_behavior>
    <start state_name="LOGGED_OUT" state_machine_name="FSM1"/>
    <state_machine name="FSM1">
      <state name="LOGGED_OUT">
        <entry>
          <action name="init"/>
        </entry>
        <transition name="LOGIN">
          <parameter type="LOGIN" value="LOGIN"/>
          <guard condition="isError(LOGIN)"/>
          <simple>
            <end_state state="LOGGED_OUT"/>
          </simple>
          <!-- send a LOGIN_ERROR message -->
          <send_action name="LOGIN_ERROR"/>
        </transition>
        <transition name="LOGIN">
          <parameter type="LOGIN" value="LOGIN"/>
          <guard condition="!isError(LOGIN)"/>
          <simple>
            <end_state state="LOGGED_IN"/>
          </simple>
        </transition>
        <transition name="LOGOUT">
          <simple>
            <end_state state="LOGGED_OUT"/>
          </simple>
        </transition>
      </state>
      <state name="LOGGED_IN">
        <transition name="LOGIN">
          <simple>
            <end_state state="LOGGED_IN"/>
          </simple>
          <send_action name="LOGIN_ERROR"/>
        </transition>
        <transition name="LOGOUT">
          <simple>
            <end_state state="LOGGED_OUT"/>
          </simple>
        </transition>
      </state>
    </state_machine>
  </protocol_behavior>
</service_def>
```


This simple Login service above can be extended using the inherits_from relationship to include additional behavior as shown below. In defining the extended behavior, only the new and overidden features (shown in black) need to be specified in the derived service. The remaining features of the base service (shown in gray), do not need to be respecified in the XML definition of the derived service, except where references need to be made between the added and overidden features and those of the base service.

The derived service adds nested states to both of the base service's states. While in the LOGGED_OUT state, the service can now be IDLE or BUSY. Two internal events BUSY and IDLE are defined to allow for the transition between these two nested states. While in the LOGGED_IN state, the service can now be in a LIMITED_ACCESS or FULL_ACCESS state depending on whether a regular LOGIN or the new GUEST_LOGIN took place.

The transition for the LOGIN message in the LOGGED_OUT state is overidden for the nested BUSY state. Note that the pre-condition of the overidding transition has been weakened to an implicit *true*.

In addition, transitions have been specified for the new message GUEST_LOGIN and internal events BUSY and IDLE.

Since no transitions have been specified for the BUSY internal event in the LOGGED_IN.BUSY state and for the IDLE internal event in the LOGGED_IN.IDLE internal state, the implicit *do_nothing* action must be assumed for these cases.

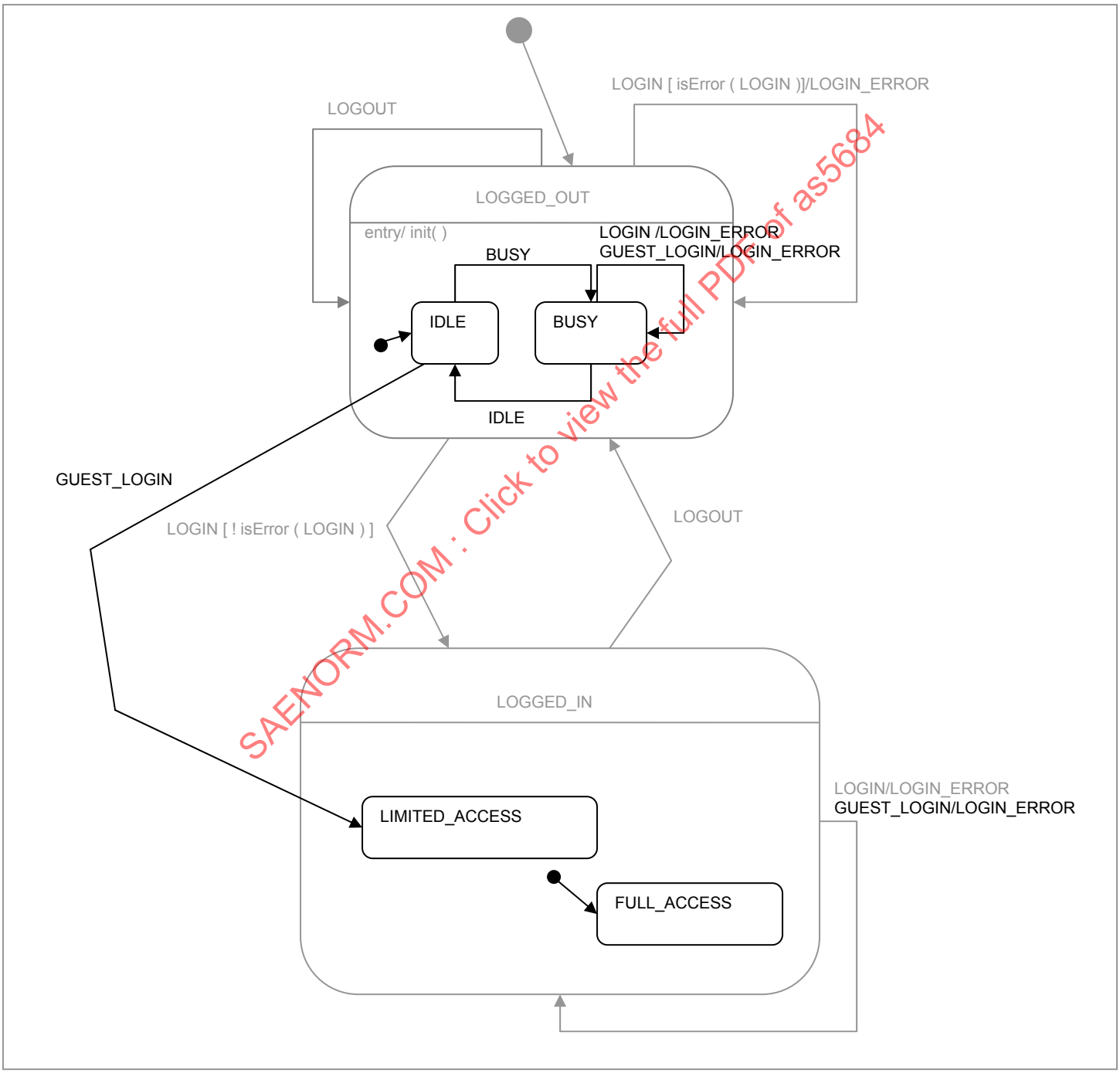

FIGURE 24 - UML STATE DIAGRAM OF THE EXTENDED ACCESS CONTROL SERVICE

```
<?xml version="1.0" encoding="UTF-8"?>
<service_def name="Access_Control_Enhanced" version="1.0"
id="http://xyz.org/usc/core/Access_Control_Enhanced" xmlns="urn:jaus:jsidl:1.0">
  <description>
    The Access Control service allows at most one user to be logged into a system. In addition to
basic access control, this service allows for guests to login in with limited access.
  </description>
  <assumptions>
    This service is assumed to work with unreliable communication links where messages may be lost
or re-ordered.
  </assumptions>
  <references>
    <inherits_from name="access_control" id="http://xyz.org/usc/core/Access_Control" version="2.1"/>
  </references>
  <message_set>
    <input_set>
      <message_def name="GUEST_LOGIN" message_id="1000" is_command="true">
        <description>A command message.</description>
        <header name="emptyHeader"/>
        <body name="emptyBody"/>
        <footer name="emptyFooter"/>
      </message_def>
    </input_set>
    <output_set/>
  </message_set>
  <internal_events_set>
    <event_def name="BUSY">
      <header name="emptyHeader"/>
      <body name="emptyBody"/>
      <footer name="emptyFooter"/>
    </event_def>
    <event_def name="IDLE">
      <header name="emptyHeader"/>
      <body name="emptyBody"/>
      <footer name="emptyFooter"/>
    </event_def>
  </internal_events_set>
  <protocol_behavior>
    <start state_name="LOGGED_OUT" state_machine_name="access_control.FSM1"/>
    <state_machine name="access_control.FSM1">
      <state name="LOGGED_OUT" initial_state="IDLE">
        <state name="IDLE">
          <transition name="BUSY" interpretation="busy internal event">
            <simple>
              <end_state state="LOGGED_OUT.BUSY"/>
            </simple>
          </transition>
          <transition name="GUEST_LOGIN">
            <simple>
              <end_state state="LOGGED_IN.LIMITED_ACCESS"/>
            </simple>
          </transition>
        </state>
        <state name="BUSY">
          <transition name="LOGIN" interpretation="overrides base transition">
            <simple/>
            <send_action name="LOGIN_ERROR"/>
          </transition>
          <transition name="GUEST_LOGIN">
            <simple/>
            <send_action name="LOGIN_ERROR"/>
          </transition>
          <transition name="IDLE" interpretation="idle internal event">
            <simple>
              <end_state state="LOGGED_OUT.IDLE"/>
            </simple>
          </transition>
        </state>
      </state>
      <state name="LOGGED_IN" initial_state="FULL_ACCESS">
```

```
        <transition name="GUEST_LOGIN">
          <simple/>
          <send_action name="LOGIN_ERROR"/>
        </transition>
        <state name="FULL_ACCESS"/>
        <state name="LIMITED_ACCESS"/>
      </state>
    </state_machine>
  </protocol_behavior>
  </service_def>
```

9. VERSIONING AND BACKWARDS COMPATIBILITY

This section describes the rules for versioning and the version numbers that are used at the application level.

- Version numbers are of the form M.N [5]. M is called the major number and N is called the minor number.
- M and N must be non-negative integers (0,1,2,…).
- It is recommended that both major and minor numbers be changed in increments of 1 for each change or change set, but numbers may be skipped.
- Any version M.N, with N>0, must pass compliance tests for (i.e. be backwards compatible with) version M.0.
- Any version M.N+1, with N>1, must pass compliance tests for (i.e. be backwards compatible with) version M.N.
- Any service version M.N that breaks backwards compatibility must either continue with a new service name and new version number, or increment the major number to M+1 and reset its minor number to 0.

The version numbers described above are used for two purposes at the application level.

- For versioning the JSIDL document (this document). This version number reflects changes made to the JSIDL document. It is relevant only to those who are creating or maintaining service definitions. This version number must not be forked. Also, it must be added to the end of the JSIDL schema namespace URI. For example, when the JSIDL version number is 1.0, the JSIDL schema namespace must be set to “urn:jaus:jsidl:1.0”.
- For versioning individual service definitions.

10. COMPLIANCE

A service definition that is based on a particular version of the JSIDL is said to be compliant with the JSIDL if it validates against the schema in the JSIDL [rng]. Compliance to the JSIDL does not guarantee the correctness and/or completeness of a service definition. Communication with a JSIDL defined service is governed by the definition of that service, and compliance with that definition is the sole criteria for determining compliance.

[5] Branching of version numbers was deliberately left out for the sake of simplicity. It is assumed that this feature will not be required.

11. NOTES

11.1 A change bar (l) located in the left margin is for the convenience of the user in locating areas where technical revisions, not editorial changes, have been made to the previous issue of this document. An (R) symbol to the left of the document title indicates a complete revision of the document, including technical revisions. Change bars and (R) are not used in original publications, nor in documents that contain editorial changes only.

APPENDIX A - SCHEMA

The schema for all JAUS Service Definitions is defined in jaus_service_interface_definition_language.rnc. The schema is broken up into smaller schemas for the purpose of modularity. This decomposition is shown in Figure 25. jaus_service_interface_definition_language.rnc includes message_set.rnc and protocol_behavior.rnc. message_set.rnc and protocol_behavior.rnc further include patterns.rnc. message_set.rnc defines the productions for message vocabulary and encoding, while protocol_behavior.rnc defines the productions for specifying protocol behavior using state machine descriptions. patterns.rnc defines a set of string patterns (refer Appendix A.4).

Message vocabularies and protocol behavior must not be specified outside of a service definition. They must be encapsulated by the Service Definition to which they belong. The schemas are defined in a single namespace urn:jaus:jsidl:M.N. The M.N at the end of the namespace is the JSIDL document version number. Details on the document version number are provided in Section 10: Versioning and Backwards Compatibility.

Relax NG [rng] notation specifies cardinality by appending a token with one of several special characters. A token followed by + means one or more occurences of that token are allowed. A token followed by a * means zero or more occurences of that token are allowed, and a token followed by a ? means that the token is optional. If a token is not followed by any of the above mentioned characters, it means that one and only one occurence of the token is allowed.

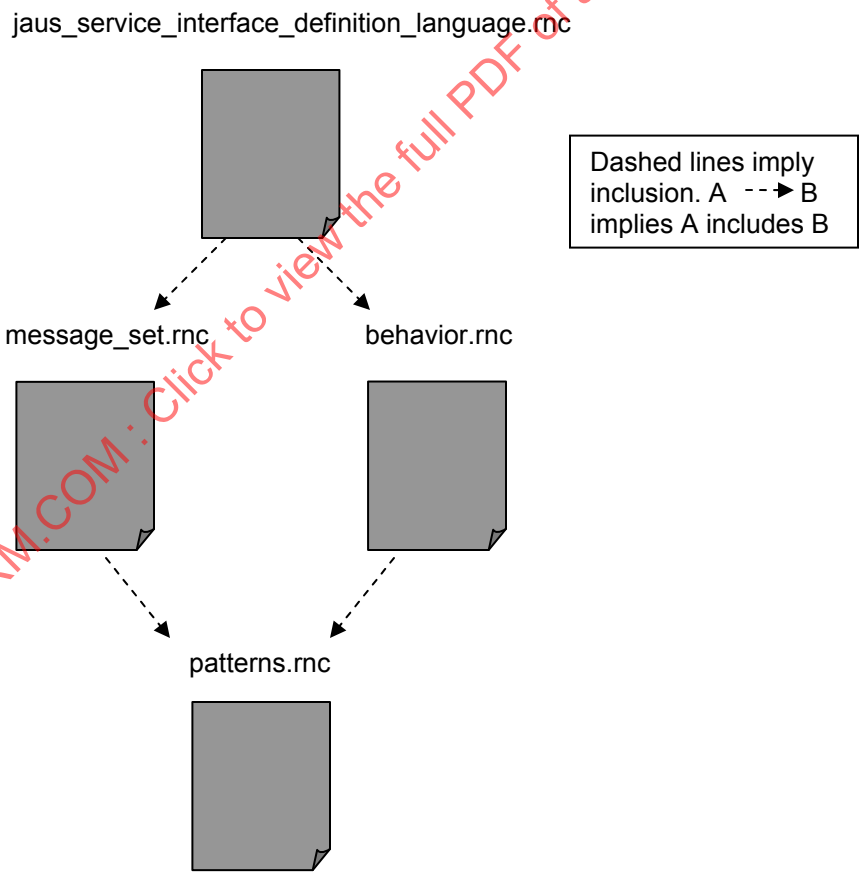

FIGURE A1 - AN ILLUSTRATION OF THE BREAKUP OF THE SCHEMAS.