JAUS Tool Set

User's Guide

INTEROPERABILITY FOR UNMANNED SYSTEMS

Table of Contents

Table of Figures

1 About this guide

This document is divided into the following chapters:

- **Chapter 1**, “About this Guide”.
- **Chapter 2**, “Introduction” gives an overview of the key features.
- **Chapter 3**, “JTS System Description”, explains requirements and structure
- **Chapter 4**, "Installation and Getting Started", describes first steps with JTS.
- **Chapter 5, “**Persistence”
- **Chapter 6**, “Validation”
- **Chapter 7**, “A Short Tour" is a quick- start guide that describes how a complete client-server implementation can be built in under 10 minutes using JTS.
- **Chapter 8**, “Creating your First Service: Ping”, describes how to build the simplest single Component application.
- **Chapter 9**, “Creating Your Second Service: Addition”, describes how to create a multi-component client/server application.
- **Chapter 10**, “Defining Message Elements”, describes how to create various message elements through the JTS GUI.
- **Chapter 11,** "Defining Messages", describes how to composite message elements into message definitions.
- **Chapter 12**, “Defining Protocol Behavior”, creating application behavior with the JTS Finite State Machine editor.
- **Chapter 13**, “Other Examples”, details a Waypoint Driver example and an Environmental Sensing example

- **Chapter 14**, "Search", describes how to use the advanced search capabilities built into JTS**10**, “Defining a Service Definition”, Describes how a Service Definition is put together.
- **Chapter 15,** “JSIDL Input/Output”
- **Chapter 16,** “Tree view”
- **Chapter 17,** “Software Framework", describes the generated code and how to modify it, and how to run the communications component.
- **Chapter 18,** “Document Generator”
- **Chapter 19,** “Wireshark Plugin”
- **Chapter 20,** “Protocol Validation”

1.1 Who Should Use It

This guide is intended for users with different degrees of knowledge and experience with JAUS/AS-4 and the JAUS Tool Set:

- **System Designers**: System designers can use JTS to specify JAUS Components and Service Definitions.
- **Software Developers**: Software developers can use JTS to implement JAUS compliant components and services. Software developers can also extend the functionality of JTS to meet their (or the industry’s) specific needs. Software developers who wish to extend JTS functionality are encouraged to read the JTS Developer's Guide, included within the JTS distribution.
- **Software Testers**: Software testers can use the runtime monitoring tool Wireshark along with the plugin provided (see Section 19) in JTS to monitor and analyze message exchanges in real time.

1.2 Typographical Conventions

This document uses the following typographical conventions:

- **Command and option names** appear in bold type in definitions and examples. The names of directories, files, machines, partitions, and volumes also appear in bold.
- *Variable information* appears in italic type. This includes user-supplied information on command lines.

```
Screen output and code samples appear in monospace type.
```

In addition, the following symbols appear in command syntax definitions.

- Square brackets [ ] surround optional items.
- Angle brackets < > surround user-supplied values.
- Percentage sign % represents the regular command shell prompt.
- Pipe symbol | separates mutually exclusive values for an argument.

2 Introduction

2.1 Overview

The ***JAUS Tool Set*** (JTS) is a set of open source software specification and development tools accompanied by an open source software framework to develop Joint Architecture for Unmanned Systems (JAS) the design and development of JAUS[1] compliant implementations for simulations and control of robotic components per SAE-AS4 standards. JTS consists of the components below:

GUI based Service Editor: The Service Editor (referred to as the GUI in this document) provides a user friendly interface with which a system designer can specify and analyze formal specifications of Components and Services defined using the JAUS Service Interface Definition Language (JSIDL) [5684].

Validator: A syntactic and semantic validator provides on-the-fly validation of specifications entered (or imported) by the user with respect to JSIDL syntax and semantics is integrated into the GUI.

Specification Repository: A repository (or database) that is integrated into the GUI that allows for the storage of and encourages the reuse of existing formal specifications.

Code Generator: The Code Generator automatically generates C++, Java, or C# code that has a 1:1 mapping to the formal specifications. The generated code includes all aspects of the service, including the implementations of marshallers and unmarshallers for messages, and implementations of finite state machines for protocol behaviors that are effectively decoupled from application behavior.

Document Generator: The Document Generator automatically generates documentation for sets of Service Definitions. Documents may be generated in several formats.

Software Framework: The software framework implements the transport layer specification AS5669A [5669A], and provides the interfaces necessary to integrate the auto-generated C++, Java, and C# code with the transport layer implementation.

[1] Joint Architecture for Unmanned Systems - now under the auspices of SAE-AS4 is a two-layer architecture for robots.

Wireshark Plugin: The Wireshark plugin implements a plugin to the popular network protocol analyzer called Wireshark. This plugin allows for the live capture and offline analysis of JAUS message-based communication at runtime(http://www.wireshark.org/).

Protocol Validation: The protocol validation consists of generating PROMELA (Protocol Meta Language) source code. Together with user added code, this produces a model of the main protocol for a service set. The model is then interpreted by SPIN (Simple PROMELA Interpreter) and used for validating the model.

2.2 Purpose

JTS has been designed and developed for the purpose of expediting the development of robust JAUS compliant implementations either from the ground up or through the integration of JAUS interfaces with existing (possibly proprietary) implementations. The developers of JTS believe that this tool set will reduce the time and subsequent cost required to specify, implement and test JAUS compliant systems to a fraction of the time and cost required to do the same manually.

This document provides a detailed description of JTS functionality, and instructions for its use. This includes step-by-step instructions for building services, with additional guidance for accessing more advanced features. This guide covers importing and exporting service definitions and automatic generation of C++, Java, and C# code. The document also touches on some concepts and the philosophy on which JTS is built. Finally, a notional workflow is provided to show how the JAUS Tool Set can be used within new or existing programs to quickly create standards-compliant services.

Readers interested in information about the overall architecture of the toolset, modifying the source code or becoming involved in bug fixes and patches should consult the JTS Developer's Guide.

2.3 Scope

This document is intended for the users of the JAUS Tool Set. Users need not have a complete understanding of SAE JAUS, but some familiarity is expected. In addition, a technical background in the design, development, and integration of distributed software components for unmanned systems is assumed.

2.4 References

The SAE JAUS documents are available from SAE International, 400 Commonwealth Drive, Warrendale, PA 15096-0001, Tel: 877-606-7323 (inside USA and Canada) or 724-776-4970 (outside USA), Web address: www.sae.org. While detailed knowledge of this standard is not required to use the JAUS Tool Set, some familiarity with these documents will make the software tool easier to use.

[5665] AIR5665 Architecture Framework for Unmanned Systems,

[5669A] AS5669A JAUS Transport Specification, Revision A

[5684] AS5684 JAUS Service Interface Definition Language

[5710] AS5710 JAUS Core Service Set

2.5 Glossary and Terminology

2.5.1 Glossary

API: Application Programming Interface
ASCII: American Standard Code for Information Interchange
BLOB: Binary Large Object
ID: Identifier
JAUS: Joint Architecture for Unmanned Systems
JSD: JAUS Service Definition
JSIDL: JAUS Service (Interface) Definition Language
PROMELA: Protocol Meta Language
RA: (JAUS) Reference Architecture
SMC: State Machine Compiler
SPIN: Simple PROMELA Interpreter
UML: Unified Modeling Language
URL: Uniform Resource Locator
URN: Uniform Resource Name
URI: Uniform Resource Identifier
UUID: Universally Unique Identifier
XML: Extensible Markup Language

2.5.2 Common Terms

The JAUS Tool Set and associated documentation uses several terms defined by the SAE JAUS standard. For convenience, these definitions are summarized here.

Component: A component is a software element in a JAUS system. A component that provides a *service* is called a server. A component that uses one or more services is called a client. [AS5710] Although not required, a component is typically a process.

Node: A node is an independent and distinct unit within a subsystem and is made up of a logical grouping of components. Although not required, a node is typically a single computer/processor.

Subsystem: A subsystem is an independent and distinct unit within a system and is made up of a logical grouping of nodes. Although not required, a subsystem is typically a unique distributed device or platform

Service Definition: A Service Definition is a textual and/or XML representation of a service interface. Any Service Definition shall conform to the JAUS Service Interface Definition Language Schema defined in AS5684 [5684]. Each Service Definition contains a service identifier, version, message set, protocol, and associated information. [AS5710]

Service Identifier: A service identifier is a globally unique string that identifies a specific *Service Definition*. Since a Service Definition mandates a message set and associated protocol, the service identifier and version number are sufficient to uniquely identify the service interface. Service Identifiers are based on a unique URI, and are specified for each service published by SAE AS-4.

JAUS Identifier: The JAUS Identifier is a 4-byte unsigned integer that corresponds to a communication end point. Messages are sent to, and received from, a JAUS Identifier. The Discovery Service permits the run-time determination of the mapping between JAUS Identifiers and Service Identifiers. A JAUS Identifier has the form {SubsystemID, NodeID, ComponentID}. [AS5710]

Transport Layer: The JAUS Transport Layer is responsible for providing resources and mechanisms for the routing and delivery of messages over a variety of available transport domains. While a Service Definition is independent of the underlying transport, it is designed for integration with AS5669A [5669A]. Other transport layers are possible, provided they meet the requirements described in the Transport Service [5710].

Message Code: The Message Code, sometimes called a Message ID or Command Code, is an identifier globally unique to each message. This code, along with the associated service version, allows receiving entities to know the type, intent, and structure of an incoming message. The Message Code is serialized in the same position within each message [5710].

3 JTS System Description

3.1 Key Features

The key features of JTS include:

- **Graphical tools** to formally specify SAE JAUS service definitions.
- **JSIDL import and export** of service definitions, enabling JTS to be used as a standards creation tool.
- **Validator** to ensure correctness of formal specifications.
- **Persistence** of created services and associated elements within a database, enabling rapid re-use of services.
- **Automated code generation** of C++ Component code to handle protocol sequencing and message marshalling and un-marshalling.
- **Automated documentation generation** of HTML documents of sets of service definitions.
- **Communications Component** (formerly Node Manager) for inter-process and inter-processor communication per AS-5669A [5669A].
- **Runtime Verification** of JAUS message traffic and data across-the-wire through the wireshark network protocol analyzer.
- **Protocol Inspection** via Wireshark

3.2 Environment

Table 1 - Generated Code Supported Systems provides a reference of the systems the generated C++, Java, and C# code and JTS have been tested on.

Table 1 - Generated Code Supported Systems

	C++	Java	C#
Windows	Using Visual Studio C++ compiler.	Win XP/7 using Java Runtime Environment.	Win XP/7 using Visual Studio C# compiler and .NET 3.5
Linux	Ubuntu 8.04/9.04	Ubuntu 10.04 using Java Runtime Environment.	Untested
Cygwin	Version 1.5 on XP version 1.7 on Vista/7	Not supported.	Untested.
OS X	Leopard and Snow Leopard with GNU g++	Untested.	Untested.

Note: JTS does not support generating Java code through Cygwin due to known issues with Cygwin loading native DLL's.

JTS is built using Java, and therefore requires a Java Run-Time Environment. Additional details on how to download and install Java are available from http://www.java.com. Version 6 (sometimes called 1.6) is recommended.

Note: If building from source, the Java Developer's Kit SE Edition is required, downloadable from http://developer.sun.com

The JAUS Tool Set is built upon several open-source tools. These tools are required for the correct operation of the toolset, and have been included in the installation package. No user action is required:

- **State Machine Compiler: http://smc.sourceforge.net**. This is used to generate all the state management, transition, and action code. JTS only generates SMC description files (.sm files), and invokes SMC to generate C++ code.
- **JMatter Framework: http://www.jmatter.org/**. The jMatter framework is the basis for the GUI, database, and entire JTS structure. An in-depth description of jMatter is outside the scope of this document, but a basic understanding of what jMatter is, and what it provides, will be very helpful. All jMatter applications share a common GUI philosophy and architecture, and JTS follows this structure.

Note: Tutorial videos at http://www.jmatter.org/ provide an excellent overview of the jMatter GUI and we recommend that you view some of the introductory videos.

- **MxGraph: http://www.jgraph.com/mxgraph.html**. MxGraph is used to draw state machine diagrams using a drag-and-drop GUI.

3.3 Workflow

The JTS workflow differs from the typical development environment or IDE such as MS Dev Studio and Eclipse. **In particular, there is no notion of a project file**. This is a critical distinction, and one that may cause some initial confusion. When starting JTS, the user does not "Create a new project" as in Dev Studio or Eclipse. Rather, she just starts creating the service. There is also no "save" feature, since a backend database is used for object persistence specification.

Every specification created in JTS is stored in an internal database. This includes specifications of Simple Fields, Complex Fields, Message Definitions, Service Definitions, and Component specifications. The storage system uses Hibernate to map the domain model to a traditional relational database system that is automatically installed with JTS.

A significant advantage of this approach is easier service re-usability (See section on Persistence). Past Records, Message and Service Definitions that you have created will always be available for incorporation into new specifications.

At the highest level, a typical JTS workflow follows a sequence of service creation, code generation, code modification, and execution. This is illustrated in more detail below.

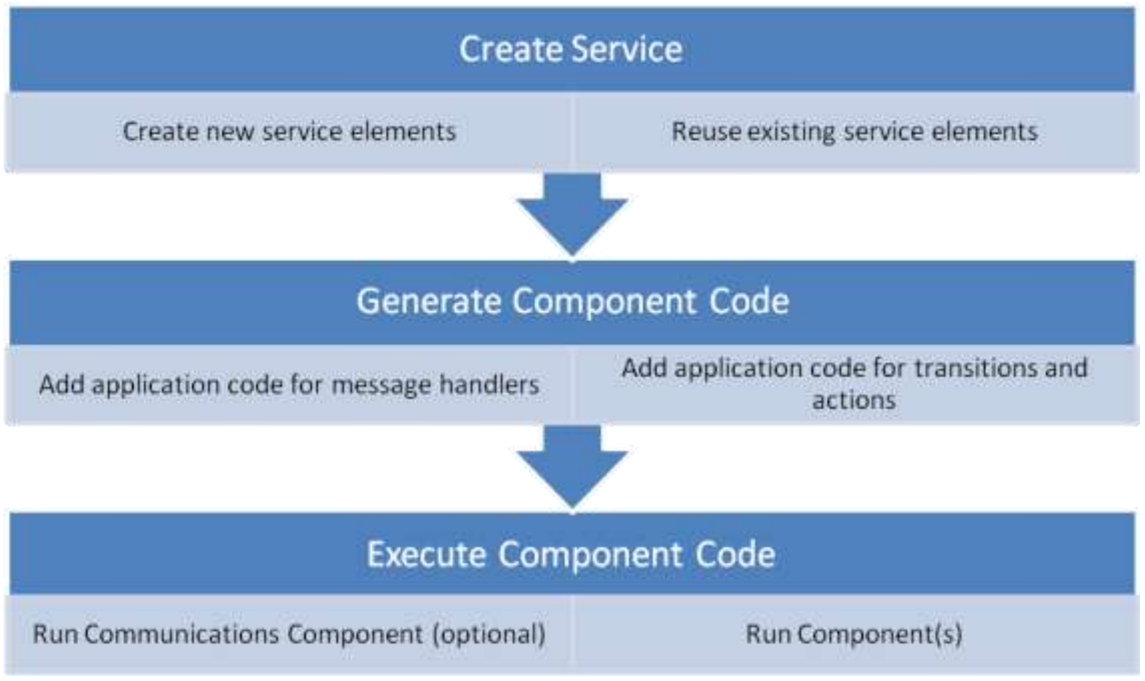

Create Service
Create new service elements
Reuse existing service elements
Generate Component Code
Add application code for message handlers
Add application code for transitions and actions
Execute Component Code
Run Communications Component (optional)
Run Component(s)

Figure 1: JTS Workflow

The steps for creating a service are explained in more detail in Section 4.5. Examples of creating services, modifying generated code, and executing generated code are presented in Sections 5, 9, and 7.

4 Installing and Getting Started

4.1 Obtaining and Installing JTS

Note: JTS is available as a binary distribution (JAR file) or as source.

The JAUS Tool Set Graphical User Interface is designed to be highly portable, and will run on any system that supports a Java Run-Time Environment. This section will describe the system requirements, including hardware and software prerequisites.

4.1.1 Getting Help

This User's Guide is intended to provide an overview with some detailed examples. You will need help at some point. We have assistance available for developers using JTS at http://www.jaustoolset.org/forums/. This includes a community support forum, FAQs, tutorial videos, and mini how-to guides.

4.1.2 System Requirements

JTS is not hardware intensive, and should run on most laptops and desktops. It is not intended for embedded devices with little memory, storage, or processing power:

- x86-compatible CPU
- 256 Mb RAM
- 500 Mb disk space (hard drive, compact flash, or USB memory)

At a minimum, JTS requires a Java Run-Time Environment. Compilation of generated source code will require a compiler, python interpreter, and SCons. See Sections 4.1.3 and 4.1.4 for details on obtaining these tools.

Note: We strongly recommend installing the SUN JAVA SE JDK directly from http://developer.sun.com rather than using the Java installation that may come with your native distribution or through distribution-specific package managers.

4.1.3 Microsoft Windows Preparation

Note: Source code generated by JTS is known to have compilation problems if the installation target directory contains spaces, e.g. c:\Program Files\JTS. Please install JTS to a target directory that does not include spaces, such as c:\JTS.

Note: When compiling C# code, if scons is unable to locate the CSharpCommon.dll, it is because the C# scons add-on failed to call the CSharpCommon sconstruct file.

The workaround for this is located in the folder <installation_dir>/templates/Common/libCSharp/ libCommon_CSharp/framework/ <1_1 or 1_0> where 1_1 and 1_0 are folders containing the libraries for transport versions 1.1 and 1.0. Navigate to the appropriate folder and locate two files: Sconstruct and Sconstruct.workaround. Change the names like so: Sconstruct -> Sconstruct.tmp Sconstruct.workaround ->Sconstruct.

Navigate to the directory in a console, and run the command scons. This will force the library to be built. When finished, rename the files back to their original names.

JTS and generated C++ and C# code are supported under Windows natively using the Microsoft Visual Studio (Express or full). Table 2 shows the environments each language has been tested on and the recommended compiler.

Table 2 - Generated Code Windows Support

	C++	Java	C#
Tested versions of Windows	XP/Vista/7	XP/7	XP/7
Cygwin	• XP/2000: 1.5.x and 1.7.x Vista/7: 1.7.x Using GNU g++ compiler	Not supported	Untested.
Recommended compiler	Visual Studio C++ compiler (VS Express or Full)	Java Development Kit	Visual Studio C# compiler (VS Express or Full) with .NET 3.5

Cygwin is available at http://www.cygwin.com.

Note: When installing Cygwin, we recommend installing the GNU g++ compiler toolchain.

In addition, JTS and all generated languages require the following tools:

- Java Development Kit SE (from http://developer.sun.com).
- Ant binary distribution (from http://ant.apache.org/bindownload.cgi)
- Python (from http://www.python.org). Note that Python V3.x is incompatible with scons; we recommend Version 2.7.
- Scons (from http://www.scons.org).

4.1.4 Linux Preparation

For Linux, the preparation steps are similar to Windows. If planning on generating C# code, please note that C# generated code has not been tested in a Linux environment. For Java and C++ code generation and JTS, install the following:

- Java Development Kit SE (from http://developer.sun.com).

Note: Do not install the JDK that typically comes with Ubuntu or other distributions. Use the one directly from Sun.

- Ant Version 1.7.x from your OS package manager, or http://ant.apache.org
- Scons Version 1.2 or later from your OS package manager, or http://www.scons.org

Note: JTS makes use of the antlr parser generator. Antlr is included with the JTS distribution. Some Linux distributions also include antlr by default. If yours includes antlr, please remove it using your distribution's package manager, otherwise compilation errors may occur.

4.1.5 Other Operating Systems

While installation instructions are provided only for Linux and Windows, the JAUS Tool Set should run on any platform with a Java SDK SE Edition, version 6 or higher. Please see the community forms on the support website for additional guidance.

4.1.6 Installing JTS from Binary Distributions

The JAUS Tool Set binary distribution can be downloaded from http://www.jaustoolset.org/forums/local_links.php.

Installing the binary distribution is straightforward:

- Unzip the distribution in the directory of your choice

- Set your environment variables as described in Section 4.1.8
- Double-click on JTS.jar to run JTS.

4.1.7 Installing JTS from Source

JTS compilation is the same for either Linux or Windows with Cygwin.

The JAUS Tool Set source distribution can be downloaded from http://www.jaustoolset.org/forums/local_links.php. We assume installation into /home/user/JTS/

```
ubuntu:~/> unzip JTS_1_1_0Source.zip
```

At this point, there are certain compilation and database preparation actions that need to be taken. This is done by invoking the appropriate ant target. This target handles database preparation and compilation.

```
ubuntu:~/> cd JTS/GUI
ubuntu:~/JTS/GUI> ant schema-export
```

4.1.8 Setting Environment Variables

Code generated by the JAUS Toolset uses a common software framework to provide basic services such as transport, thread handling, and definition for parent classes. Before compiling the generated code, the 'JTS_COMMON_PATH' environment variable must be set. This variable must reference the Common directory found in the JTS installation

If Installing from Source Distribution.

- WINDOWS: Right click "My Computer", select Properties. Select the "Advanced" tab and "Environmental Variables" button at the bottom. This brings up a new dialog. Select "New" under "System variables" and enter "JTS_COMMON_PATH" (no quotes) as the variable name. Enter "C:\<path to JTS install>\JTS\GUI\templates\Common" (no quotes) as the value. Press 'OK' until the boxes go away.
- LINUX or CYGWIN: Edit ~/.bashrc or ~/.bash_profile. Add "export JTS_COMMON_PATH='/<path to install>/JTS/GUI/templates/Common' (ignore double quotes, single quotes required around path name). Note that in cygwin, the path should be a cygpath, e.g. /cygdrive/c/<rest of the path>….

If Installing from Binary Distribution.

- WINDOWS: Right click "My Computer", select Properties. Select the "Advanced" tab and "Environmental Variables" button at the bottom. This brings up a new dialog. Select

“New” under “System variables” and enter “JTS_COMMON_PATH” (no quotes) as the variable name. Enter “C:\<path to JTS install>\JTS\templates\Common” (no quotes) as the value. Press ‘OK’ until the boxes go away.

- LINUX or CYGWIN: Edit ~/.bashrc or ~/.bash_profile. Add “export JTS_COMMON_PATH=’/<path to install>/JTS/templates/Common’ (ignore double quotes, single quotes required around path name). Note that in cygwin, the path should be a cygpath, e.g. /cygdrive/c/<rest of the path>….

Failure to set the JTS_COMMON_PATH environment variable will result in an error message when attempting to build the generated code.

4.1.9 Configuring the Run-Time Environment

By default, the generated code builds an executable and several dynamic (shared) libraries. To run the executable, the shared libraries must be available at run-time. The build scripts automatically copy the shared libraries to the bin directory, such that they are co-located with the executable. For Windows environments, including Cygwin, this is sufficient and no additional user action is required.

For many flavors of Linux, however, a co-located shared library may still not be found at run-time. For this reason, we recommend modifying the library path using the $LD_LIBRARY_PATH environment variable:

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:’.’

Otherwise, all shared libraries must be copied to a run-time accessible location (such as /lib or /usr/lib) to execute the generated code.

4.2 Starting the system (Source Distribution Only)

JTS is started through ant, as follows:

```
ubuntu:~/JTS/GUI> ant run
```

At this point, you should see a GUI with a blank window. Section 4.4 presents a brief overview of the GUI.

4.3 Ant Targets (Source Distribution Only)

The Java Ant build system is used to compile and run JTS. In addition, there are other build targets that are useful. Below is a listing of Ant targets.

Table 3 – Ant targets

Project	Target	Use
JTS GUI	**schema-export**	This is used to build JTS. It also has the side effect of clearing the database, so using this target will result in loss of all data in JTS.
JTS GUI	**schema-update**	This target is used to migrate the existing contents of a database to a new revision.
JTS GUI	**compile**	Does a full compile of JTS, without re-generating and wiping the database. Also does a full compile of the PromelaCodeGenerator which is required by the JTS GUI.
JTS GUI	**clean**	Does a full clean of all JTS build artifacts.
JTS GUI	**clean-database**	Wipes the database. Faster than doing a schema-update
JTS GUI	**backup-database**	Backs up the database in a directory specified via a prompt, located under db_backup directory. The user may also opt to have a directory name automatically generated based on a timestamp.
JTS GUI	**restore-database**	Restores a backup database from a directory under db_backup created by backup-database. The user must specify the backup directory name.
JTS GUI	**run**	Runs JTS
PromelaCodeGenerator	**compile**	Does a full compile of the PromelaCodeGenerator.
PromelaCodeGenerator	**clean**	Does a full clean of the PromelaCodeGenerator

		build artifacts.
PromelaCodeGenerator	**run**	Runs the PromelaCodeGenerator in a standalone mode.

4.4 Graphical User Interface Overview

The JTS GUI is based on the jMatter framework. All jMatter applications have a similar look and feel. jMatter applications have a container frame, in which multiple internal frames with content can be opened and worked on. The column on the left is a class list containing selectable icons for the top-level specification elements.

The figure below shows what you should see when you start JTS.

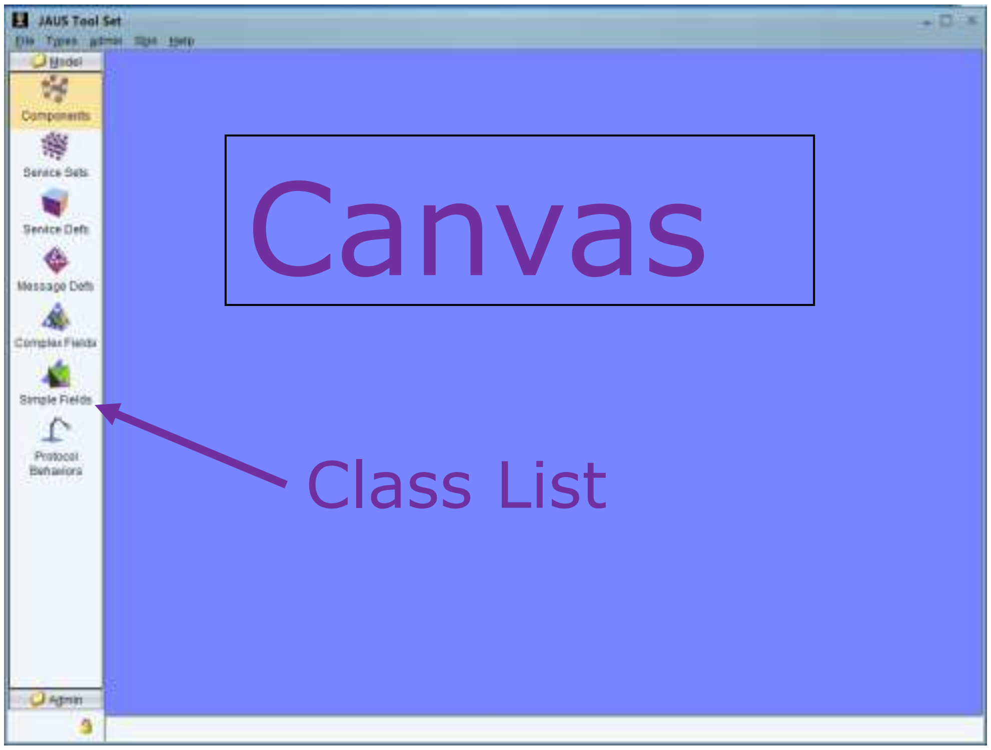


Figure 2: JTS Canvas

The GUI consists of the following:

- **Canvas:** This is the main work area where internal frames may be opened to create, read, update and delete specifications for simple and complex data fields, protocols, message definitions, etc.
- **Class List:** These icons represent the top-level specification elements (also called types) in JSIDL. This includes:
- **Simple Fields:** Atomic data types (Fixed Field, Bit Field etc)
- **Complex Fields:** Composite data types (Record, List etc.)
- **Message Definitions:** Message definitions consisting of header, body and footer.
- **Protocol Behavior:** Finite state machine descriptions of service behavior.
- **Service Definitions:** Service specifications
- **Service Set:** Sets of service specifications
- **Component:** Component specifications
- **Admin:** The Admin class list allows users to set typical administrative features. This is part of the jMatter framework and more information on its use may be found in the jMatter documentation.

Almost every JSIDL specification element is represented by a selectable icon. Since all of these icons cannot fit in the class list, they are made accessible through the menu. To see a complete list of types that may be accessed through the user interface, go to Types->Types->Browse.

Double clicking on these icons opens an internal frame that lists existing objects of the particular type in the database. Right clicking on the icons provides a context menu with options to create new objects of the type, open existing objects of the type or find objects of the type. An option called Manage Restrictions is also provided as part of the jMatter framework. More information on this option may be found in the jMatter documentation.

A fuller explanation of how to use the GUI to create complete Service Definitions is provided in Section 5 (Creating your First Service) and Section 10 (Defining Message Elements). In addition, there is an Overview Video Tutorial that highlights what each Element Icon does in JTS. This video tutorial is available at http://www.jaustoolset.org/forums

4.5 Service Creation Overview

JTS is intended to be a service creation tool. It uses a bottom-up philosophy for service creation, in which simple fields are used to compose complex fields which are embedded within message definitions, which are assigned to input/output sets, which are combined with protocol behaviors to create service definitions, which are aggregated into service sets, and then instantiated in components which lead to generated C++ executable software. The figure below illustrates this hierarchy of service creation.

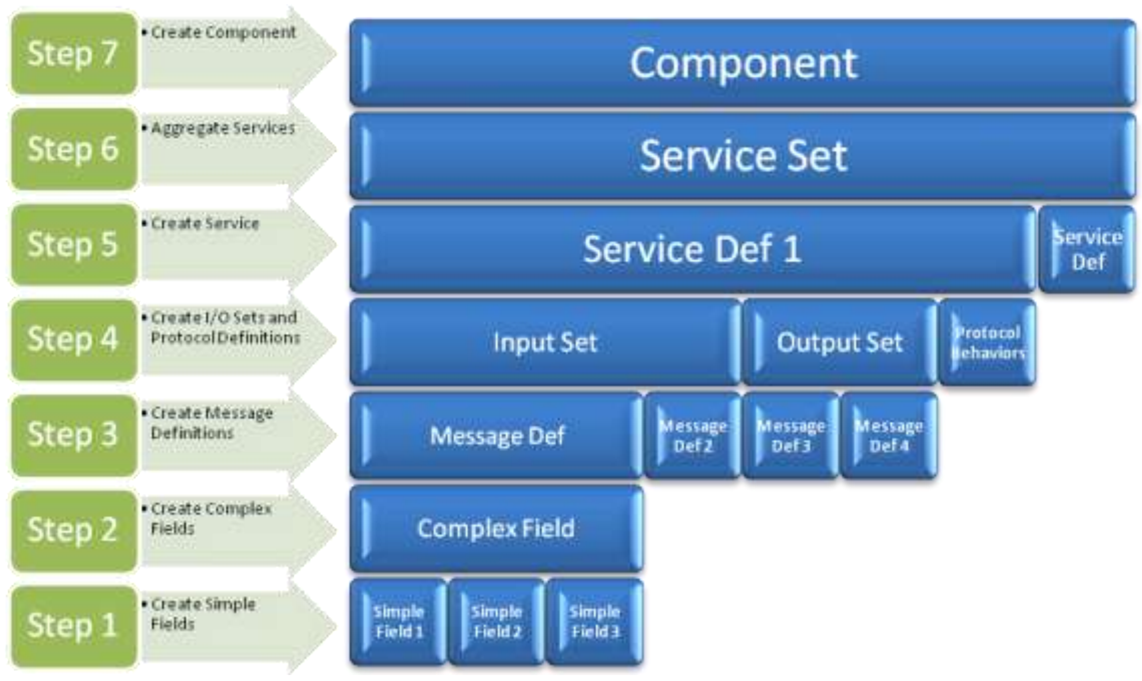


Figure 3: JTS Service Creation Hierarchy

JTS does not disallow the top-down approach, it only discourages it by virtue of its design. The bottom-up approach was favored to the typical top-down approach since it helps to tackle the hierarchical complexity of JSIDL specification elements in a more manageable way. For instance, it turns out that a SimpleField like a Fixed Field is not all that simple to specify completely in JSIDL. Aside from the attributes of a Fixed Field, one may need to specify a scale range, value ranges and value enumerations. The bottom-up approach gives focus to the element being specified without adding clutter from associated elements. So when the user is specifying a Simple Field, the user interface puts all the focus on the Simple Field and not the fields associated with the Simple Field. Once the Simple Field has been completely specified, its specification can be used in specifying higher level elements like Complex Fields, Message Definitions and so on. This approach is akin to designing and setting aside parts in a parts library and then building complex components from those parts in a tool like AutoCAD.

Working with JTS is significantly easier if you have some notion of the layout of the service you want to create prior to specification, and instantiate the service in the order of steps listed above. Once the service specification has been built, the document generator (See Section 18) may be used to obtain a "top-down" view of a service specification. The user interface also provides a tree-view that may be used to verify the top-down view.

Another significant benefit of the bottom-up approach comes from the class model of entities and associations that is required for the bottom-up methodology. In the class model, each class exists as a separate entity in the database (See Section 5). Associations tie entities together to make high level entities like Message Definitions and Service Definitions. As such, JTS is uniquely suited for design element re-use. Once simple fields, records, and message definitions have been created, it is extremely straightforward to re-use them in new services and components.

Therefore, while there may be some initial start-up cost with using JTS for service creation, there is a rapid gain in productivity that occurs from element re-use.

5 Persistence

A very important feature that is central to JTS is the repository or database feature. Its objective however is very simple. It is to encourage reuse of existing types. From the perspective of languages, the set of specifications within the database may be viewed as a domain language – the language of robotics. In this language, Simple Fields are the words (or vocabulary) of the language. The message definitions may be viewed as short meaningful phrases. Protocol behavior defines the grammar that must be applied in making valid sentences. Finally, the message traffic that is generated from an execution is a *conversation* that took place in the language of robotics (or JAUS in particular).

The database allows and encourages the user to reuse existing words, phrases and grammar. In doing so, it creates a convergence in the domain language towards a finite and comprehensive vocabulary.

Along with the database comes a simple yet powerful search mechanism which is described in Section 13. As of release 1.0 the search mechanism is limited to performing manual queries on the database.

5.1 Recursive Deletion

JTS provides the ability to recursively delete a service definition. In previous versions of JTS, a service definition could be deleted from the persistence database. However the various elements that made up the service definition, such as event definitions, simple fields, and complex fields would not be deleted when their containing service definition was deleted. Recursive deletion removes both a service definition and any elements that are uniquely contained within that service definition. This allows re-importing of services with modified sub-elements, where changes to sub-elements are guaranteed to be incorporated.

Recursive deletion may be invoked on a service definition either by clicking “Recursive Delete” on the service definition’s context menu (see Figure 4) or by clicking “Recursive Delete” in a service definition’s window in JTS.

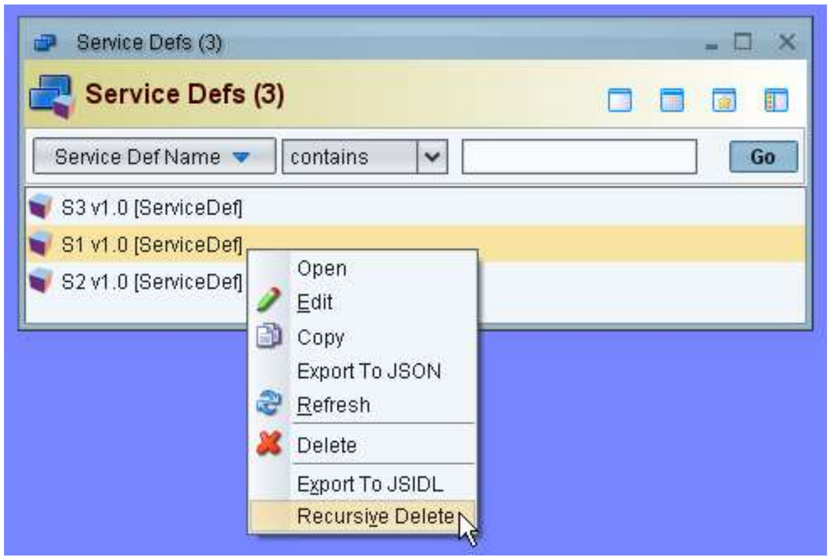
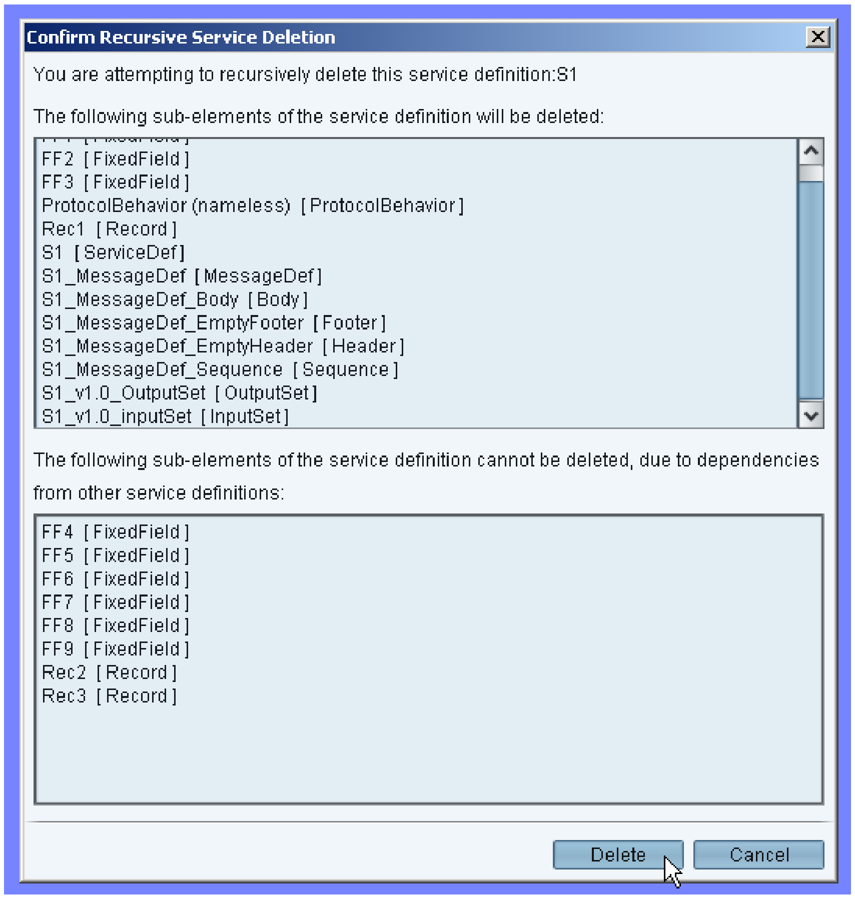


Figure 4: Invoking Recursive Delete

After clicking “Recursive Delete”, a dialog appears like that shown in Figure 5. In the figure, a service definition “S1” has been selected for recursive deletion. The upper list shows the elements comprising S1 that will be deleted as they are unique to S1, while the lower list shows elements comprising S1 that cannot currently be deleted, since other service definitions depend on them. Click “Delete” to delete the selected service definition and all elements displayed in the upper list.



You are attempting to recursively delete this service definition:S1

The following sub-elements of the service definition will be deleted:

```
FF2  [ FixedField ]
FF3  [ FixedField ]
ProtocolBehavior (nameless)  [ ProtocolBehavior ]
Rec1  [ Record ]
S1  [ ServiceDef ]
S1_MessageDef [ MessageDef ]
S1_MessageDef_Body  [ Body ]
S1_MessageDef_EmptyFooter  [ Footer ]
S1_MessageDef_EmptyHeader  [ Header ]
S1_MessageDef_Sequence  [ Sequence ]
S1_v1.0_OutputSet  [ OutputSet ]
S1_v1.0_inputSet  [ InputSet ]
```

The following sub-elements of the service definition cannot be deleted, due to dependencies from other service definitions:

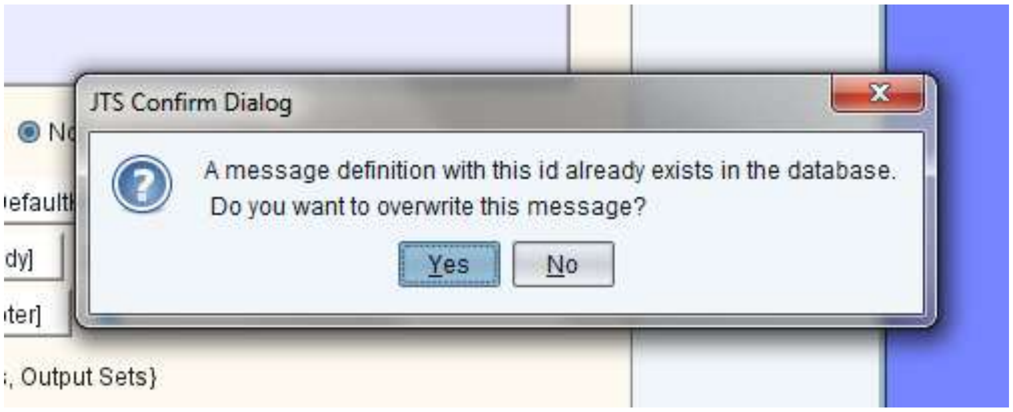
```
FF4  [ FixedField ]
FF5  [ FixedField ]
FF6  [ FixedField ]
FF7  [ FixedField ]
FF8  [ FixedField ]
FF9  [ FixedField ]
Rec2  [ Record ]
Rec3  [ Record ]
```

Figure 5: Confirming Recursive Deletion

Note: Currently, after performing recursive deletion, JTS element list windows such as the service definitions list, complex types list, and message definitions list are not automatically updated to show the results of recursive deletion. They need to be manually refreshed by the user clicking the ‘Go’ button in the list window with an appropriate filter selected.

5.2 Overwrite Message Definitions

JTS includes a feature where message ids are checked when trying to save a message definition. This is done to eliminate duplicate messages in the database. Whenever trying to save a message with the same message id as a message already stored in the database, a window will pop up to notify you of your options.

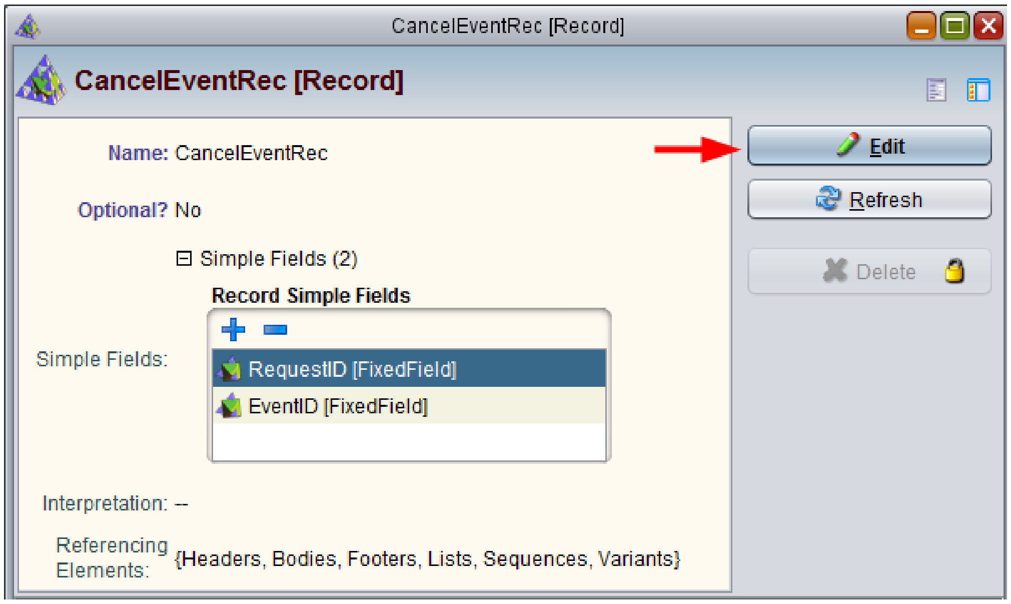


A message definition with this id already exists in the database.
Do you want to overwrite this message?

In most cases, like when you are modifying a previously saved message definition, you will want to overwrite the previous message definition. If this is the case, simply press the 'Yes' button and your changes will be saved. However, if you receive this pop up unexpectedly, a message with the same message ID already exists in the database. In this case you will most likely want to click the 'No' button and modify the message id to some other value unique to the system you are designing.

5.3 Editing Lists

When editing items in a list within a JTS window, special care must be taken to insure that the item is in the editing state so that validation of the definition can occur. If the window is not in the editing state, it is possible for a user to incorrectly specify items in a list which may lead to unspecified behavior. The proper way to add and remove items from an element is to press the Edit button. This is applicable to all windows that contain lists within JTS.



CancelEventRec [Record]

CancelEventRec [Record]

Name: CancelEventRec

Optional? No

Simple Fields (2)

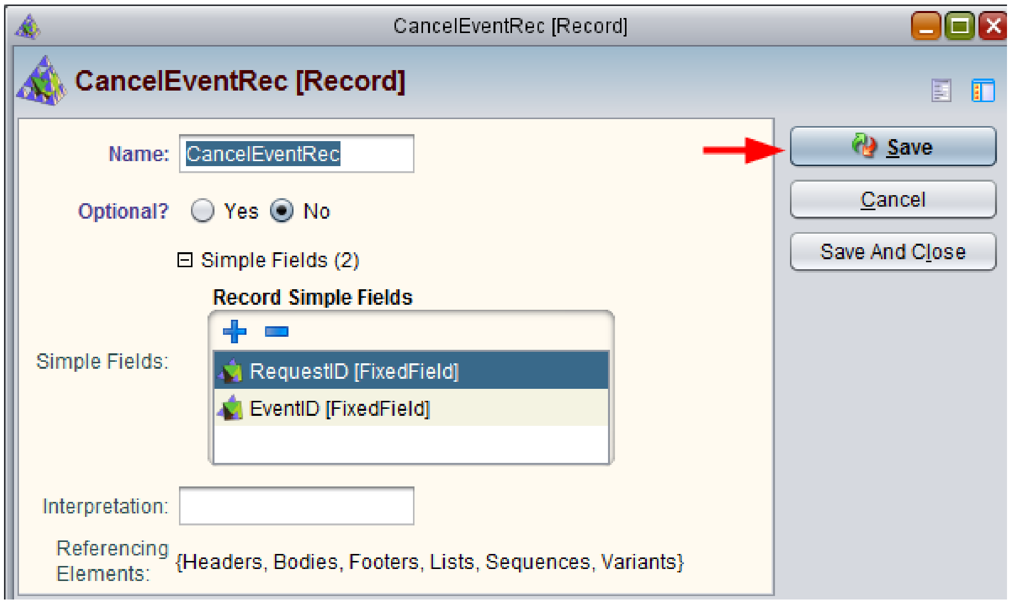
Record Simple Fields

Simple Fields:
- RequestID [FixedField]
- EventID [FixedField]

Interpretation: --

Referencing Elements: {Headers, Bodies, Footers, Lists, Sequences, Variants}

When the window is in viewing mode, do not modify any lists within the window. To edit a list, press the edit button which will change the button text to 'Save'. This specifies the window is in an editable state and that validation can occur when a save is triggered.



CancelEventRec [Record]

CancelEventRec [Record]

Name: CancelEventRec

Optional? Yes [✓ No]

Simple Fields (2)

Record Simple Fields

Simple Fields:
- RequestID [FixedField]
- EventID [FixedField]

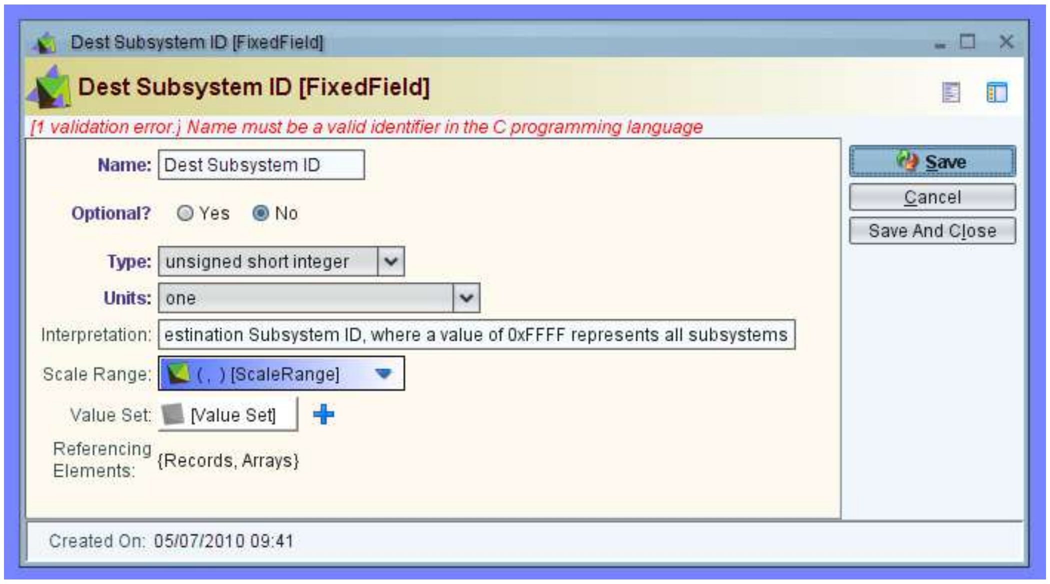
Interpretation:

Referencing Elements: {Headers, Bodies, Footers, Lists, Sequences, Variants}

6 Validation

One of the objectives of JTS was to allow the user to create well defined specifications with only a high level understanding and knowledge of JSIDL. The details such as the syntax and semantics behind each type have been built into the tool in the form of an on-the-fly validator. The validator prompts the user with an error each time the user makes a syntactically or semantically incorrect input either manually or through the “Import” option.

Manual entries are flagged using red embedded text as shown in the figure below. The text message is brief but aims to guide the user towards correct syntax and semantics as specified by JSIDL. Note that the validator catches most but not all semantic errors in release 1.0. This is especially true for the protocol behavior section. An exhaustive treatment of all semantic errors is left for a future release.


Dest Subsystem ID [FixedField]
Dest Subsystem ID [FixedField]
Save
Cancel
Save And Close

[1 validation error.] Name must be a valid identifier in the C programming language

Name: Dest Subsystem ID

Optional? Yes [✓ No]

Type: unsigned short integer

Units: one

Interpretation: estination Subsystem ID, where a value of 0xFFFF represents all subsystems

Scale Range: (,) [ScaleRange]

Value Set: [Value Set]

Referencing Elements: {Records, Arrays}

Created On: 05/07/2010 09:41

Figure 6: Validation message when incorrect input is detected

7 A Short Tour

This section is designed to help the user create a fully executable JAUS client-server implementation of a simple Ping service in under 10 minutes using JTS.

7.1 Create a Component Specification

a) Launch the application by typing "ant run" at the command prompt, or by double clicking on JTS.jar (binary distribution).

b) Import JAUSToolset\ examples\xml\Ping\PingClient.xml by right-clicking on the "Service Defs" icon on the main GUI frame and selecting "Import from JSIDL". This will import the PingClient service and PingServer service since PingClient is related to (client-of) PingServer. Double-click on "Service Defs" to view the imported services.

c) In JTS, a Component is built from a set of Service Sets, and a Service Set is built from a set of Service Definitions that are usually related by the inherits-from or client-of relationships. To build a Component for the Ping example, a Service Set needs to be built first by right-clicking on the Service Set icon on the main GUI frame and selecting “New”. Specify the name, id and version of the new Service Set as "PingServiceSet, urn:jts:PingServiceSet" and 1.0. To select the services for this Service Set, expand “Service Defs”, then click on the blue "+" button and select "Browse" in the pop-up menu. Now use the List Picker to select both PingServer and PingClient. Then press "Done". "Save and Close" PingServiceSet.

d) Next, create a new Component by right-clicking on the Component icon on the main GUI frame and selecting “New” from the pop-up menu. Set the name of the component as “PingComponent” and component ID = 120. Add the PingServiceSet to the component by expanding “Service Sets”, then clicking on the blue "+" button and selecting “Browse” to bring up the list picker. Press "Save" on the Ping component dialog box to complete the creation of the Ping component specification.

7.2 Auto-Generate a Component Implementation

a) To auto-generate the code for the component, simply press the "Auto-generate Code" button on the PingComponent window. Click “Browse” and select the path for the generated code to go. Note that the path should not have spaces in it, so if generating on Windows, avoid using folders with names similar to “My Documents”. The code is generated in the directory named ‘PingComponent_120’ under the specified path. By default, the code is generated in C++. To generate code in Java or C#, select the Java or C# radio buttons at the bottom of the pop-up window.

b) Now, the application behavior code has to be manually added to the server implementation that was auto-generated. First, we add the action handler that allows the Ping Server to send a Report Heartbeat Pulse using the built-in ‘sendJausMessage’ function. If generating C++ code, replace the following function in PingComponent_120\src\urn_jts_PingServer_1_0\PingServer_PingFSM.cpp

```
void PingServer_PingFSM::ReportHeartbeatPulseAction()
{
     /// Insert User Code HERE
}
```

with,

```
void PingServer_PingFSM::ReportHeartbeatPulseAction()
{
     // Send a ReportHeartbeatPulse message back to the local component.
     ReportHeartbeatPulse response;
     sendJausMessage(response, *jausRouter->getJausAddress());
}
```

If generating Java or C# code, the same changes must be made, but with slightly different syntax:

In PingComponent_120\src\urn_jts_PingServer_1_0\PingServer_PingFSM.java or .cs, replace:

```
public void ReportHeartbeatPulseAction()
{
     /// Insert User Code HERE
}
```

with:

```
public void ReportHeartbeatPulseAction()
{
     // Send a ReportHeartbeatPulse message back to the local component.
     ReportHeartbeatPulse response = new ReportHeartbeatPulse();
     sendJausMessage(response, jausRouter.getJausAddress());
}
```

c) Finally, the application behavior code has to be manually added to the client. The client has two actions: 1) Send a Query Heartbeat Pulse that will elicit a response from the Server, and 2) Print a message to the screen when a response is received.

e) In PingComponent_120\src\urn_jts_PingClient_1_0\PingClient_PingClientFSM.cpp, replace

```
void PingClient_PingClientFSM::QueryHeartBeatPulseAction()
{
     /// Insert User Code HERE
}

void PingClient_PingClientFSM::printToScreenAction()
{
     /// Insert User Code HERE
}
```

with,

```
void PingClient_PingClientFSM::QueryHeartBeatPulseAction()
{
     // Send the QueryHeartbeat message to the local component
     QueryHeartbeatPulse query;
     sendJausMessage( query, *jausRouter->getJausAddress());
}

void PingClient_PingClientFSM::printToScreenAction()
{
     printf("Hello World!\n");
}
```

For Java and C#, replace the following lines of code in

PingComponent_120\src\urn_jts_PingClient_1_0\PingClient_PingClientFSM.java or .cs:

```
public void QueryHeartBeatPulseAction()
{
     /// Insert User Code HERE
}

public void printToScreenAction()
{
     /// Insert User Code HERE
}
```

with this for Java,

```
public void QueryHeartBeatPulseAction()
{
     // Send the QueryHeartbeat message to the local component
     QueryHeartbeatPulse query = new QueryHeartbeatPulse();
     sendJausMessage( query, jausRouter.getJausAddress());
}

public void printToScreenAction()
{
     System.out.println("Hello World!");
}
```

and this for C#,

```
public void QueryHeartBeatPulseAction()
{
     // Send the QueryHeartbeat message to the local component
     QueryHeartbeatPulse query = new QueryHeartbeatPulse();
     sendJausMessage( query, jausRouter.getJausAddress());
}

public void printToScreenAction()
{
     Console.WriteLine("Hello World!");
}
```

7.3 Compile and Execute a Component

a) To build the component, simply type "scons" under the directory "PingComponent_120" at the command prompt.

b) If running C# and a compile error regarding the common library appears, please refer to the note in section 4.1.3 for a workaround.

c) If running C#, open a second command prompt and navigate to JTS/nodeManager, type “scons”. When it has finished compiling, navigate to JTS/nodeManager/bin and type “NodeManager.exe nm.cfg”. This is required by the C# code to run properly. It does not print any output while running. When finished with the example, use ctrl + c to exit the application.

d) Once the build process is completed, the PingComponent_120 executable is generated in the PingComponent_120/bin directory.

- To run C++, simply execute it from the command line:

```
$> cd bin
$> ./PingComponent_120
```

  for Linux or

```
$> cd bin
$> PingComponent_120.exe
```

  for Windows.

- To run Java, execute the following line from the command line in any system:

```
$> cd bin
$> java -jar PingComponent_120.jar
```

- To run C#, execute the following line from the command line in Windows:

```
$> cd bin
$> PingComponent_120.exe
```

Note that there may be a short delay and a warning message while the Framework attempts to contact the Node Manager. This is normal, and will be discussed further in Section 16.

e) The “Hello World!” message will be displayed on the screen.

f) End the program. For C++ and Java, hit Ctrl + c. For C# hit Ctrl + c followed by the escape key.

8 Creating your First Service: Ping

Note: In this Section, we repeat the Ping Example used in the Short Tour. Rather than importing the service however, the JTS User Interface is exercised to construct the service. Before starting this example, we recommend clearing the JTS database. If you are working from a source distribution, close any running instances of JTS, and from the command line, type: 'ant schema-export'. Use caution, since this will cause all elements defined in JTS to be erased. If you are working from a binary distribution, copy a blank database template (available at http://www.jaustoolset.org/forums) to the GUI/db/ location.

In this section, we present a guided walk-through of using the JAUS Tool Set to model, build, and run a simple component. The workflow for this example is a bottom-up approach broken into 7 steps (Steps 6-7 are variations on the same task):

1. Identify the services needed in the component, and their interfaces
2. Define the messages each service will use for inputs and outputs
3. Describe the protocol that governs the rules for message exchange for each service
4. Merging messages and protocol into service definitions
5. Build a single component from the services
6. Generate the C++ source code, and integrate user code *or*
7. Generate the Java source code, and integrate user code *or*
8. Generate C# source code, and integrate user code
9. Build and run the system

The goal of this section is not a detailed instruction manual on every feature and facet of the JTS. Rather, the emphasis is on covering the gamut of functionality, to give a basis for more in-depth study later in this document.

8.1 Identify the Interface

In this example, we consider a simple service which behaves similarly to the SAE JAUS "Liveness" Service [AS5710] or an ICMP "ping" server. When the service receives a ping request, called a QueryHeartbeatPulse message in SAE JAUS, it should reply to the sender with a ping response (ReportHeartbeatPulse). We can represent this behavior with a simple state machine diagram, as shown in the following Figure 7 below:

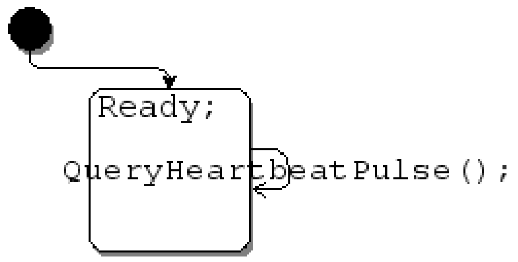


Figure 7 Protocol Definition for Ping Server

The state machine diagram shows a service with a single 'default' state along with a single loopback transition denoted as 'A'. This transition can be described by its triggering message, any associated guard conditions, and its action as shown in Table 3.

Table 3 Ping Server Actions and Guards

Label	**Trigger**	**Conditions**	**Actions**
A	QueryHeartbeatPulse	*none*	Send a Report Heartbeat Pulse message to the component that sent the query

Hence, when the service receives the QueryHeartbeatPulse message, it should reply back with a ReportHeartbeatPulse.

In order to test our ping server, we'll need another service to send messages to it. This will be our client. Our client should send a ping request (QueryHeartbeatPulse) when it first starts up. If it receives a message, it should print out an acknowledgement to the screen, as seen in Figure 8: Protocol Definition for Ping Client.

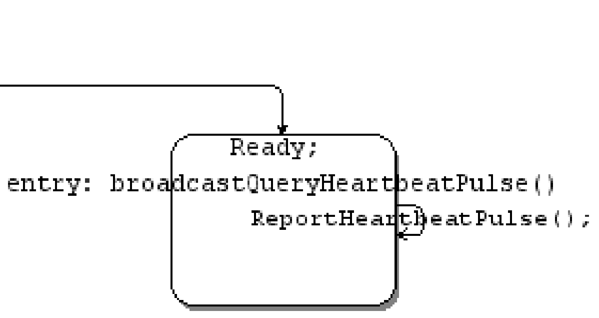


Figure 8: Protocol Definition for Ping Client

This protocol definition introduces the concept of 'entry' actions, which are similar to transition actions but are executed each time the state is entered. In this case, action 'A' is executed one time when the client starts, and action 'B' is executed when we receive a ReportHeatbeatPulse message.

Table 4 Ping Client Actions and Guards

Label	**Trigger**	**Conditions**	**Actions**
A	<entry>	*none*	Broadcast a Query Heartbeat Pulse message to all services in this component
B	ReportHeartbeatPulse	*none*	Print a message on the screen

Note that since transition 'B' is a loopback transition (the end state is the same as the start state), the entry action 'A' will not be executed. [JSIDL] specifies that entry and exit actions are not executed on loopback transitions. If we want our client to continuously send ping requests, we could add a second action to the 'B' transition to broadcast another query message. For this example, we will assume one broadcast at client start-up is sufficient.

8.2 Defining the Message Set

The ping client and server only require two messages: QueryHeartbeatPulse and ReportHeartbeatPulse. We'll use the JTS to model these messages.

1. Start the toolset using instructions in Section 4.2. The JTS user interface is broken into three areas: A menu bar at the top, a left hand column of icons that lists the Model Elements, and a large blue work area.

2. From the Model Elements list, right click the 'Message Defs' icon and select 'New'

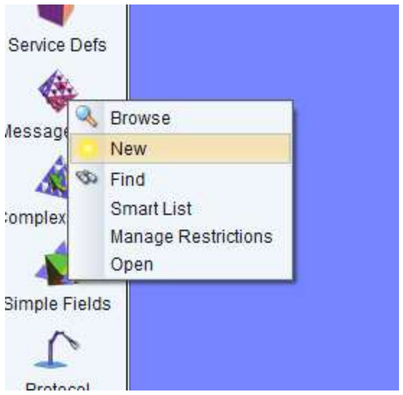


Figure 9: New Message Set Screenshot

3. This opens the 'New Message Def' dialog box. Here we define the message name and ID, give it an optional description, and assign fields (data) to the message. In the 'Name' box, enter 'QueryHeartbeatPulse'

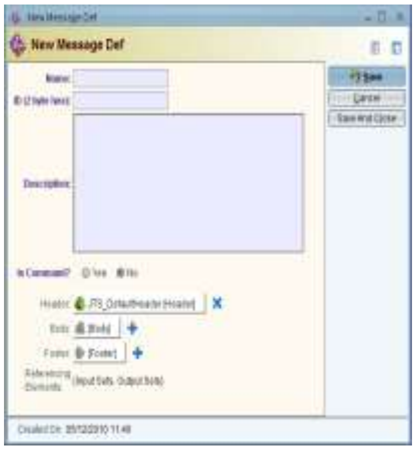
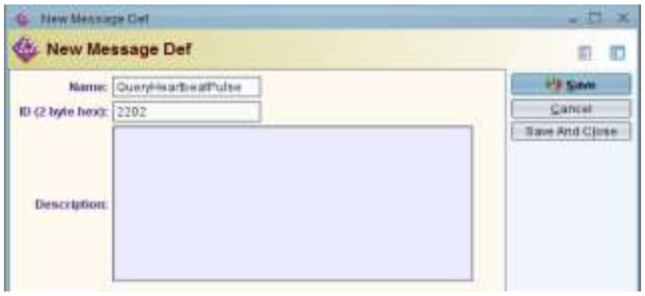


Figure 10: New Message Def Screenshot

4. The ID is a 2-byte hexadecimal value that uniquely identifies each message. Instead of the human-readable Name string, the ID allows a receiving entity to correctly identify the message type before processing it. SAE JAUS requires the ID to be included in each

message as the first two bytes. The JAUS Tool Set defaults to this setting, by automatically including the JTS_DefaultHeader for each new message. For this example, we'll use the SAE JAUS message identifier for the QueryHeatbeatPulse message (2202)

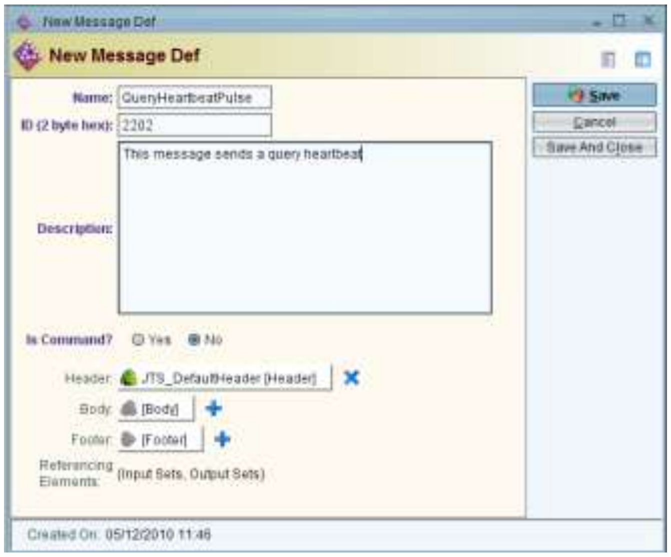


New Message Def

Name: QueryHeartbeatPulse
ID (2 byte hex): 2202
Description:

Figure 11: Populated Message Def Screenshot

5.	Now enter a text description.

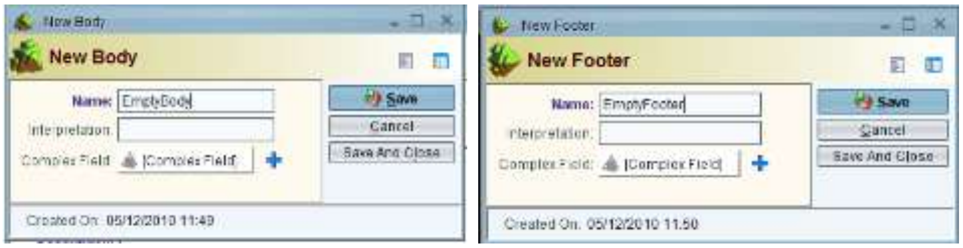


New Message Def

Name: QueryHeartbeatPulse
ID (2 byte hex): 2202
Description: This message sends a query heartbeat
Is Command? Yes [✓ No]
Header: JTS_DefaultHeader [Header]
Body: [Body]
Footer: [Footer]
Referencing Elements: (Input Sets, Output Sets)
Created On: 05/12/2010 11:46

Figure 12: New Message Def Description Screenshot

6.	Even if a message doesn't have data associated with it, we must provide it with an empty body, and empty footer. Create an EmptyBody by clicking on the blue "+" by Body, select "New", then create a new Body with Name "EmptyBody", leaving the other fields blank. Press “Save and Close” when done. Do the same for Footer, creating a new Footer with name "EmptyFooter".

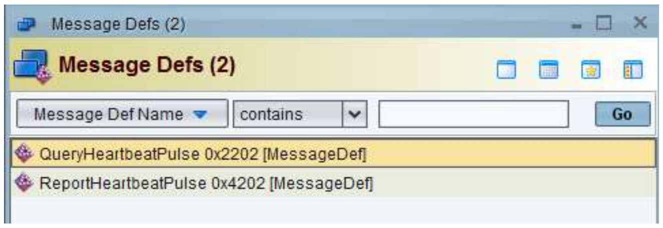

New Body
Name: EmptyBody
Interpretation:
Complex Field: [Complex Field]
Created On: 05/12/2010 11:49

New Footer
Name: EmptyFooter
Interpretation:
Complex Field: [Complex Field]
Created On: 05/12/2010 11:50

Figure 13: New Body / Footer Screenshot

7. Since our message has no additional data associated with it, select 'Save And Close'. This enters the new message in the database, which we'll use later to create our service definition.

8. Repeat steps 2-6 to create the ReportHeartbeatPulse message with an ID of 4202h. This time, however, you do not need to create a new EmptyBody and EmptyFooter, since they have already been defined. So, instead of selecting "New" when you click the blue "+", select "Browse", and pick the existing elements from the list. Once completed, double clicking on the 'Message Defs' icon in the Model Element column should show a dialog box with both messages defined:


Message Defs (2)

QueryHeartbeatPulse 0x2202 [MessageDef]
ReportHeartbeatPulse 0x4202 [MessageDef]

Figure 14: Message Def List Screenshot

Now that we've defined the messages (vocabulary) that our simple services uses, we need to model the protocol which dictates the rules for message exchange.

8.3 Describe the Protocol

The protocol behavior for our ping client and server were described in Section 8.1. Here we convert our design into a JTS models. We'll need two different models: one for the server, and one for the client.

1. From the Model Elements list, right click the 'Protocol Behaviors' icon and select 'New'

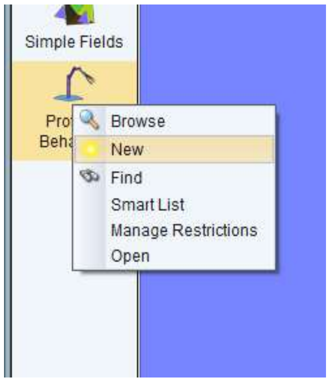


Figure 15: New Protocol Screenshot

2. This opens the ‘New Protocol Behavior’ layout tool. The layout tool provides a drag-and-drop utility to design protocol behaviors using the same state-machine like syntax used through the SAE JAUS documents. The layout tool provides a menu bar, behavior elements selection column, and a workspace

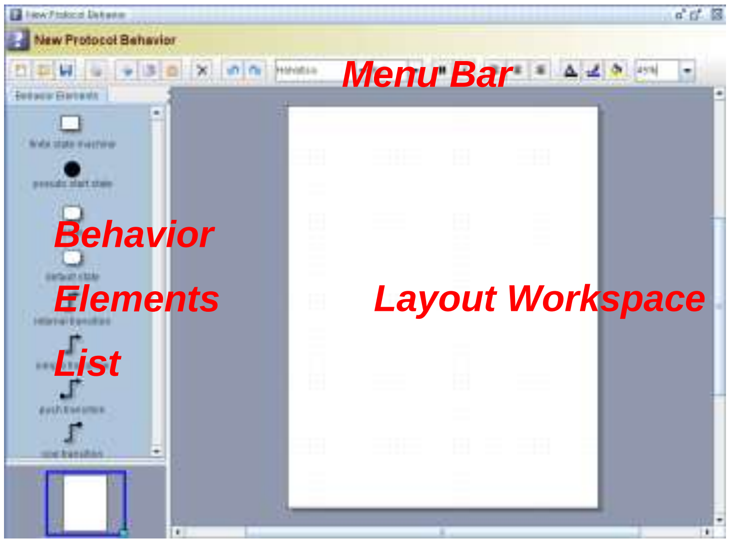


Figure 16: Protocol UI Screenshot

3. First, we have to create a new Finite State Machine to contain our states and transitions. Drag a "Finite State Machine" icon from the Behavior Elements List to the Layout Workspace. Drag and grow the box to increase the size, since it will contain our states.

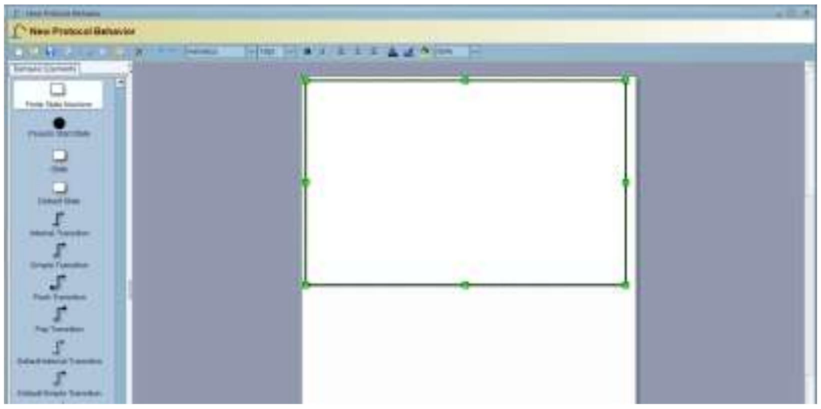


Figure 17: Insert FSM Screenshot

4. Double-Click on the FSM box in the Layout Workspace, and when the edit window appears, type in *"name = PingServerProtocol; isStateless = true"*.

5. Now, right-click on the FSM box in the Layout Workspace, and select *Shape->EnterGroup* from the context menu. This pushes us down the stack into the FSM, so we can add states.

6. Now, we have to add states for the ping server, which responds with a *ReportHeartbeatPulse* whenever a *QueryHeartbeatPulse* message is received. Drag a 'State' icon from the Behavior Elements List to the Layout Workspace and double click it to give it a name of "Ready"

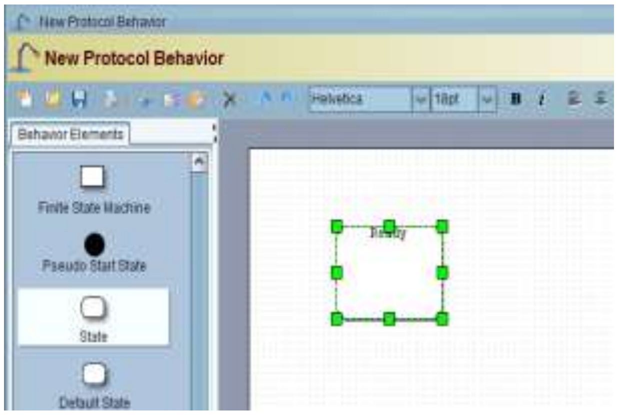


Figure 18: Add Ready State Screenshot

7.	Since this is the only state in our protocol definition, we need to tell the JTS to start in this state. This is achieved through a ‘Pseudo Start State’ icon and an automatic transition. Drag a ‘Pseudo Start State’ icon from the Behavior Elements List to the Layout workspace, then connect it to the Ready state with a Simple Transition:

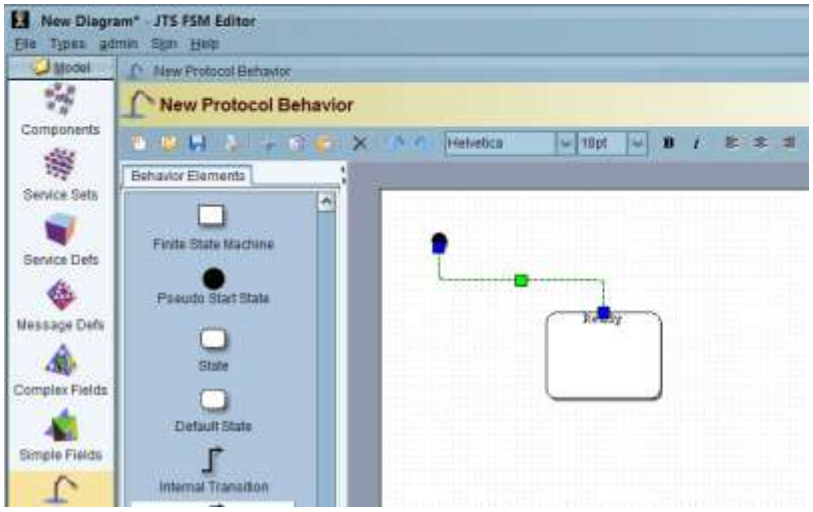


Figure 19: Add Pseudo Start State Screenshot

8.	We can now add an Internal Transition that starts and ends back at the Ready state

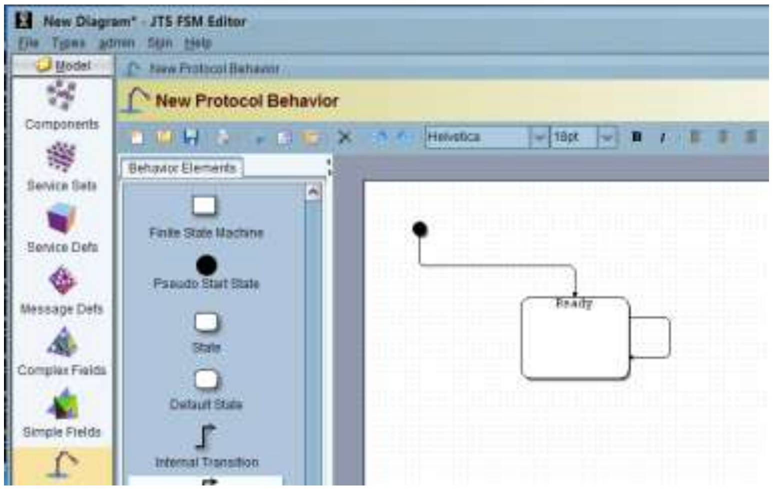


Figure 20: Add Internal Transition Screenshot

9. Next, we parameterize the transition, defining the trigger, guard, and actions design in as specified in Section 8.1 for our server. Double-clicking on the transition opens a text box for editing. The syntax for a transition is:

```
trigger(param_type value, …)[guard]/action(arg1,…)
```

where 'value', 'arg1', and 'guard' are all optional. For our server, the trigger is specified as *QueryHeartbeatPulse() / sendReportHeartbeatPulse()*. This means that the transition will be named "*QueryHeartbeatPulse"* and that when it is triggered, and action named "send*ReportHeartbeatPulse*" will occur. Double click on the transition link to bring up an edit field and enter the trigger specification text "QueryHeartbeatPulse() / sendReportHeartbeatPulse()". When you click outside the edit window, the displayed text is shortened to just the trigger:

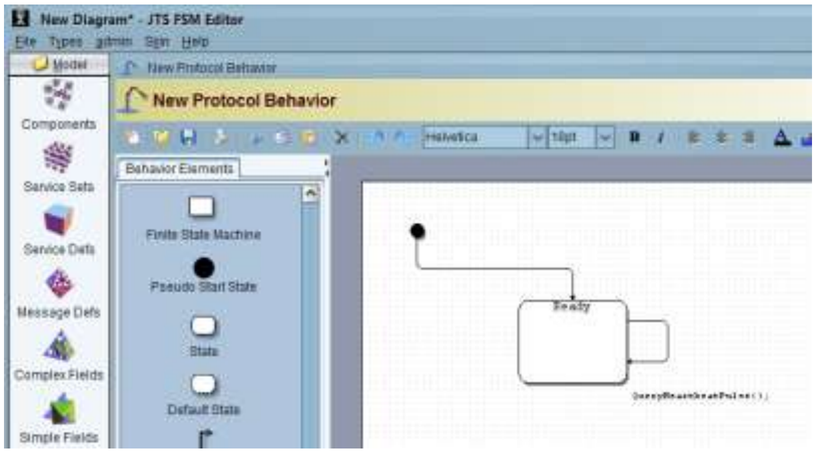


Figure 21: Add Trigger Screenshot

10. Now, our state machine for the Ping Server is complete. Right click on the Layout Window, and select *Shape->ExitGroup*, and you should see the following:

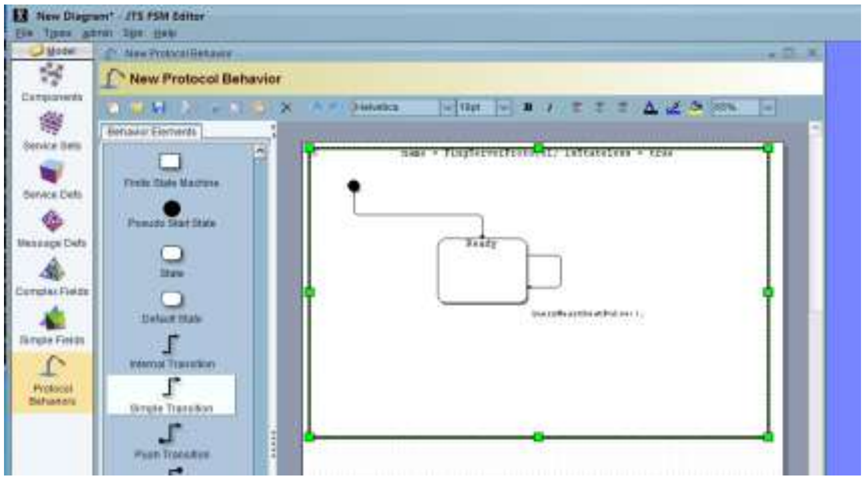


Figure 22: Complete State Screenshot

11. Now, we need to save the protocol. To do this, click on the save icon (the floppy disk in the upper left), and save the file as "PingServer.mxe" in a directory of your choice.

12. With the protocol behavior for the ping server complete, we can now model the protocol for the ping client. Repeat steps **1-10** as above, except name the FSM "PingClientProtocol", and use the client specific transition "ReportHeartbeatPulse() / printToScreen()":

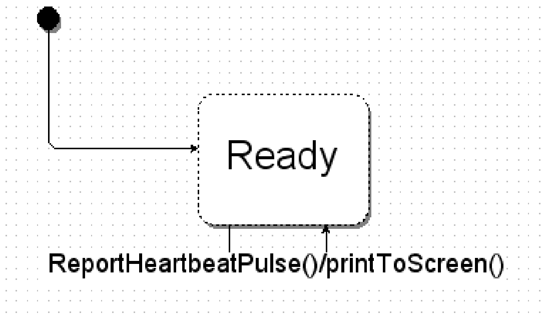


Figure 23: Ping Client State Layout Screenshot

13. This defines the protocol to handle an incoming ReportHeartbeatPulse message, but will still have to add the entry action that broadcasts the query when we enter the Ready state on start-up. This is added as a text tag to the state name. Double-click the state and edit the text to make it read *"Ready; entry: broadcastQueryHeartbeatPulse()"*:

Note: The "entry" keyword must be lower-case.

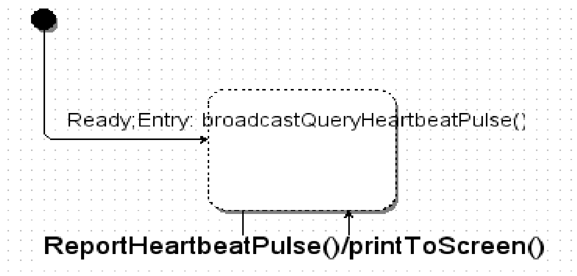
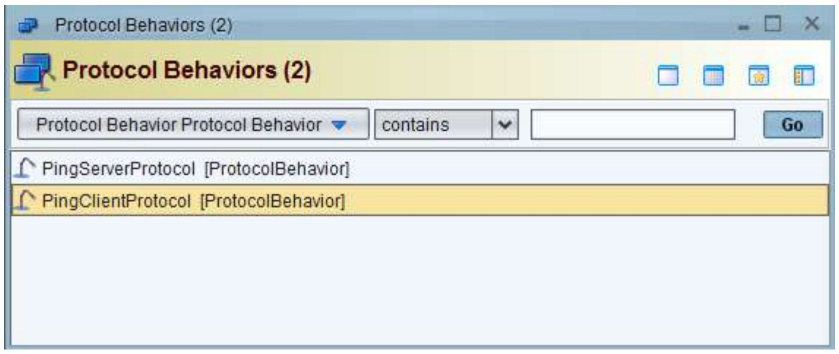

Ready;Entry: broadcastQueryHeartbeatPulse()
ReportHeartbeatPulse()/printToScreen()

Figure 24: Ping Client State Layout With Trigger Screenshot

14. Again as in step 12, save the protocol as PingClient.mxe, and close the window.

We now have models for both the client and server protocol. Double-click the 'Protocol Behaviors' icon in the Model Elements list to open the protocol behavior filter. This will show the two protocol state machines we just defined:


Protocol Behaviors (2)

Protocol Behaviors (2)

Protocol Behavior Protocol Behavior | contains | | Go

PingServerProtocol [ProtocolBehavior]
PingClientProtocol [ProtocolBehavior]

Figure 25: Protocol Behavior List Screenshot

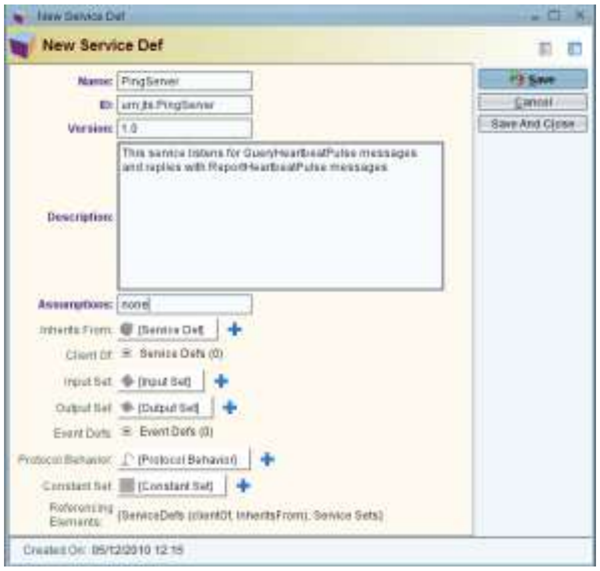
The next step will put it all together into a service, and ultimately a component for deployment.

8.4 Build the Service

With the messages and protocol defined, we can now combine them into a service definition.

1. From the Model Elements list, right click the 'Service Defs' icon and select 'New'

2. This opens the 'New Service Def' dialog box. Each service must have a unique name, urn-based identifier, and version. A description and assumptions field is also required. For our ping server, we'll use the following:


New Service Def

Name: PingServer

ID: urn:jts:PingServer

Version: 1.0

Description: This service listens for QueryHeartbeatPulse messages and replies with ReportHeartbeatPulse messages

Assumptions: none

Inherits From: [Service Def]

Client Of: Service Defs (0)

Input Set: [Input Set]

Output Set: [Output Set]

Event Defs: Event Defs (0)

Protocol Behavior: [Protocol Behavior]

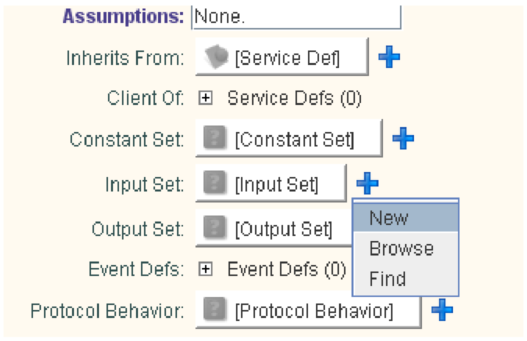
Constant Set: [Constant Set]

Referencing Elements: [ServiceDefs (clientOf, inheritsFrom), Service Sets]

Created On: 05/12/2010 12:18

Figure 26: New Service Def Screenshot

3. We must tell JTS which messages our service will handle. This is defined as an Input Set. Click the blue plus ("+") sign next to [Input Set] and select 'New'


Assumptions: None.

Inherits From: [Service Def]

Client Of: Service Defs (0)

Constant Set: [Constant Set]

Input Set: [Input Set]

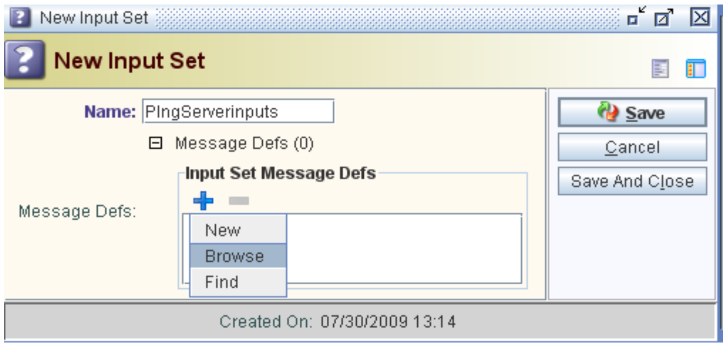
Output Set: [Output Set]

Event Defs: Event Defs (0)

Protocol Behavior: [Protocol Behavior]

Figure 27: New Input Set Creation Screenshot

4. This opens the New Input Set dialog box. We'll use "PingServerInputs" as our input set name. Then select the plus ("+") sign next to Message Defs to open the input set editor. In the editor, select the blue plus sign and select 'Browse'



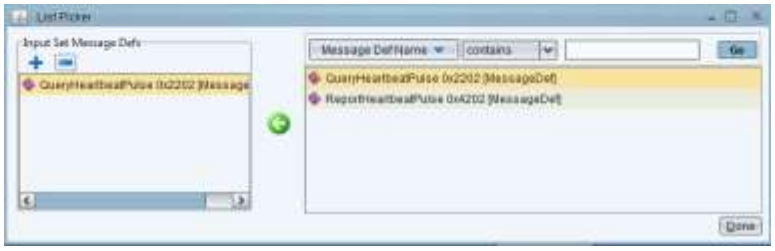
Name: PIngServerinputs

Message Defs: Message Defs (0)

Created On: 07/30/2009 13:14

Figure 28: New Input Set Screenshot

5. In the List Picker dialog, we can quickly select which messages form the input set. In this case, only the QueryHeartbeatPulse message is used. Highlight the query message, then press the green arrow to add it to the Input Set Message Defs list.



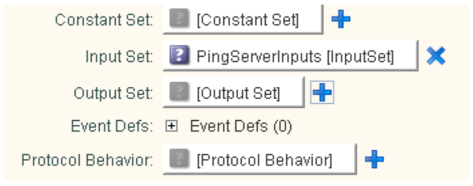
QueryHeartbeatPulse 0x2202 [Message

QueryHeartbeatPulse 0x2202 [MessageDef]

ReportHeartbeatPulse 0x4202 [MessageDef]

Figure 29: Input Set List Picker Screenshot

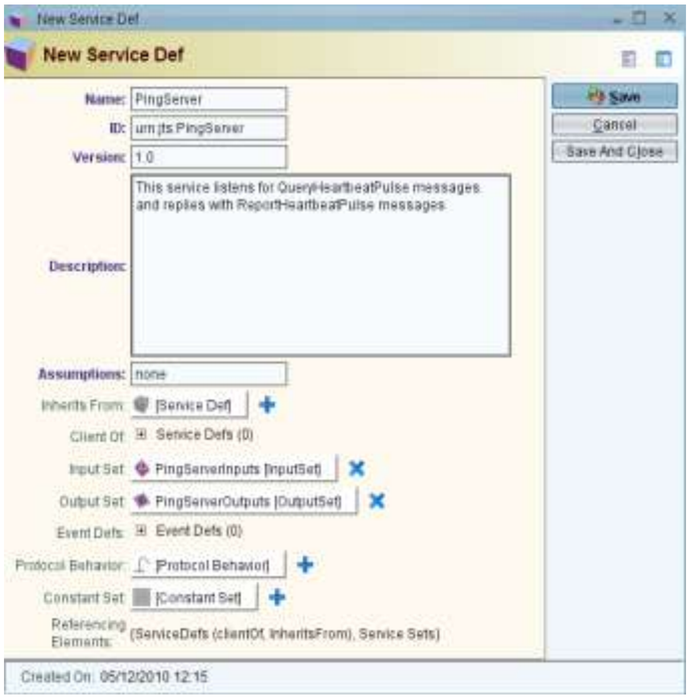
6. Select ‘Done’ in the List Picker dialog and ‘Save And Close’ on the New Input Set dialog. The PingServerInputs set should now be added as inputs to our service



Constant Set:	[Constant Set]
Input Set:	PingServerInputs [InputSet]
Output Set:	[Output Set]
Event Defs:	Event Defs (0)
Protocol Behavior:	[Protocol Behavior]

Figure 30: Ping Service Inputs Screenshot

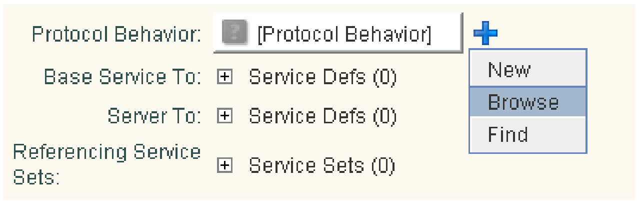
7. Repeat steps 3-6 to create a New Output Set called PingServerOutputs. This tells JTS which messages our new service will output. In this case, select ReportHeartbeatPulse from the List Picker.



Name: PingServer
ID: urn:jts:PingServer
Version: 1.0
Description: This service listens for QueryHeartbeatPulse messages and replies with ReportHeartbeatPulse messages.
Assumptions: none
Inherits From: [Service Def]
Client Of: Service Defs (0)
Input Set: PingServerInputs [InputSet]
Output Set: PingServerOutputs [OutputSet]
Event Defs: Event Defs (0)
Protocol Behavior: [Protocol Behavior]
Constant Set: [Constant Set]
Referencing Elements: (ServiceDefs (clientOf, inheritsFrom), Service Sets)
Created On: 05/12/2010 12:15

Figure 31: New Service Def Populated Screenshot

8. Finally, we associate the protocol behavior for our new service. Select the blue plus ("+") sign next to [Protocol Behavior] and select 'Browse'. From within the list picker, select 'PingServerProtocol' and click 'pick'.

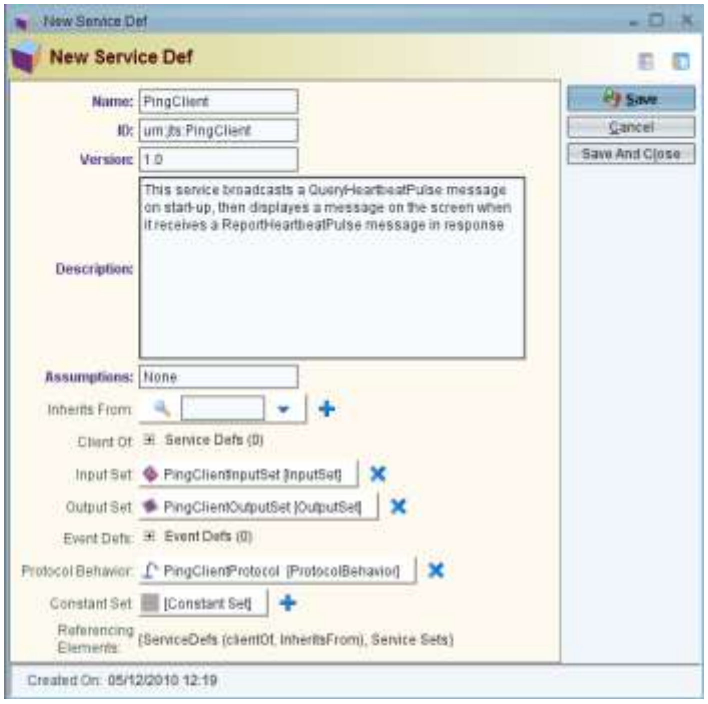


Protocol Behavior: [Protocol Behavior]
Base Service To: Service Defs (0)
Server To: Service Defs (0)
Referencing Service Sets: Service Sets (0)

Figure 32: Add Protocol Behavior Screenshot

9. Click 'Save and Close' to save this new service definition to the database.

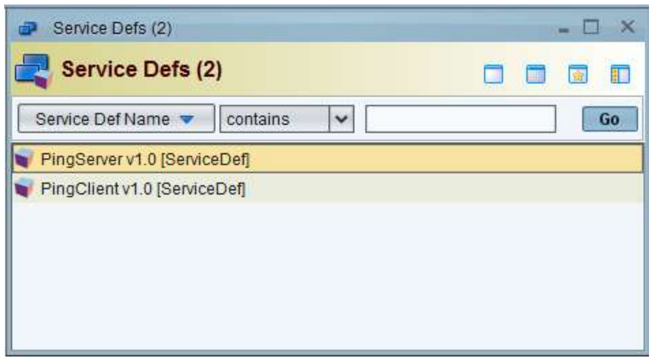
10. With the PingServer defined, repeat steps 1-9 to define the PingClient service. Note that the input set for the client consists only of the ReportHeartbeatPulse message, while the output set contains only the QueryHeartbeatPulse message. The Protocol Behavior should be 'PingClientProtocol'.



Name: PingClient
ID: urn:jts:PingClient
Version: 1.0
Description: This service broadcasts a QueryHeartbeatPulse message on start-up, then displayes a message on the screen when it receives a ReportHeartbeatPulse message in response
Assumptions: None
Inherits From:
Client Of: Service Defs (0)
Input Set: PingClientInputSet [InputSet]
Output Set: PingClientOutputSet [OutputSet]
Event Defs: Event Defs (0)
Protocol Behavior: PingClientProtocol [ProtocolBehavior]
Constant Set: [Constant Set]
Referencing Elements: {ServiceDefs (clientOf, InheritsFrom), Service Sets}
Created On: 05/12/2010 12:19

Figure 33: New Service Def Populated Screenshot

11. To verify that both server and client services have been defined, double-click the Service Defs icon from the Model Elements list to display the Service Defs list box:



PingServer v1.0 [ServiceDef]
PingClient v1.0 [ServiceDef]

Figure 34: Service Def List Screenshot

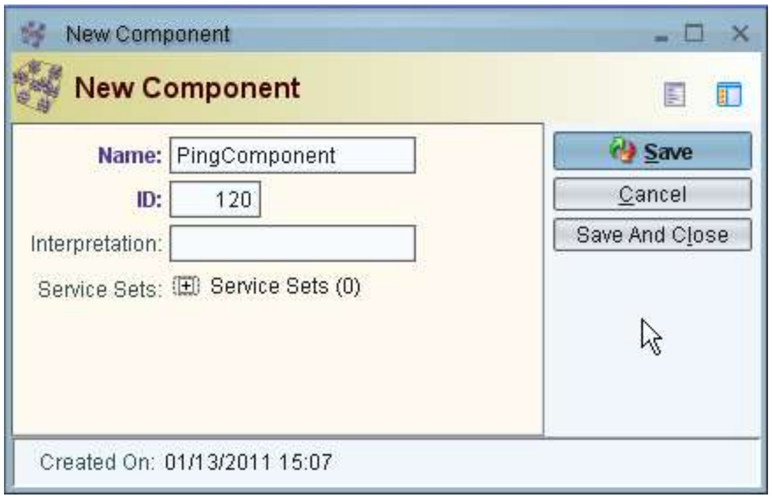
With our services defined, we next examine how to combine one or more services into a component before generating the code.

8.5 Create a Component

A component represents a collection of services available at a unique transport address. In practical terms, the JAUS Tool Set only generates code for a component; we must therefore combine our messages and protocol into services which in turn become part of a service set before being integrated into a component. This section provides step-by-step instructions for making a component that includes both the ping server and client.

From the Model Elements list, right click the 'Components' icon and select 'New'

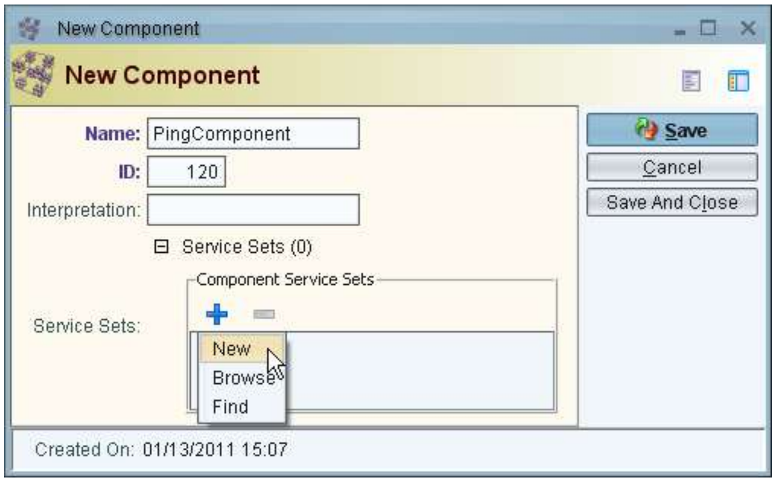
1. This opens the 'New Component' dialog box, shown in Figure 35: New Component Screenshot. Each component has a name and component ID. The component ID is a number between 1 and 254 that must be unique within the node. We'll give our PingComponent an ID of 120.



Name: PingComponent
ID: 120
Interpretation:
Service Sets: Service Sets (0)
Created On: 01/13/2011 15:07

Figure 35: New Component Screenshot

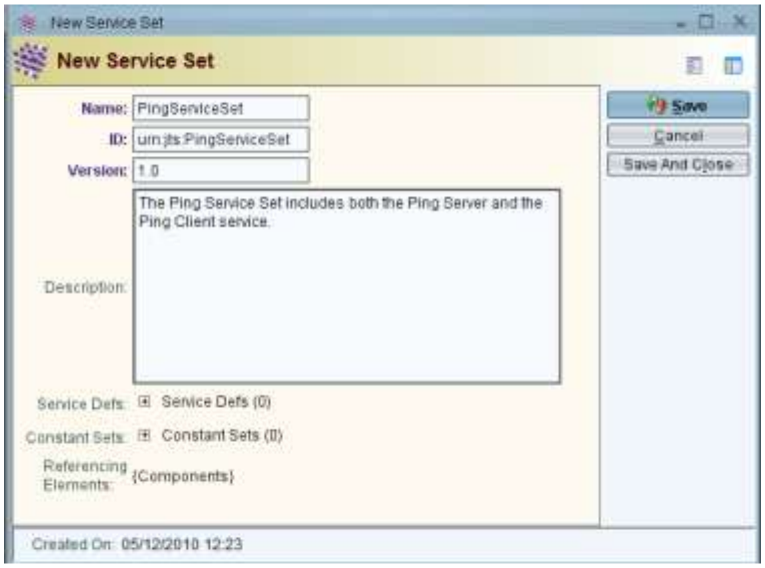
2. We must tell JTS which services our component will host. This is defined as a Service Set. Click the plus ("+") sign next to 'Service Sets' to open the Component Service Sets editor. Click the blue plus ("+") and select 'New' to define a new service set:



Name: PingComponent
ID: 120
Interpretation:
Service Sets:
Created On: 01/13/2011 15:07

Figure 36: New Component Populated Screenshot

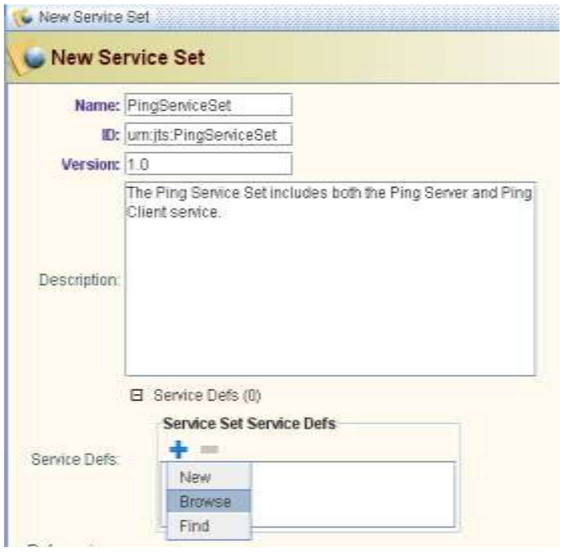
3. This opens the New Service Set dialog box. We'll use "PingServiceSet" as our service set name, adding a unique ID and meaningful description:



Name: PingServiceSet
ID: urn:jts:PingServiceSet
Version: 1.0
Description: The Ping Service Set includes both the Ping Server and the Ping Client service.
Service Defs:
Constant Sets:
Referencing Elements: {Components}
Created On: 05/12/2010 12:23

Figure 37: New Service Set Screenshot

4. Then select the plus ("+") sign next to Service Defs to open the Service Set Service Defs editor. In the editor, select the blue plus sign and select 'Browse'



New Service Set

Name: PingServiceSet

ID: urn:jts:PingServiceSet

Version: 1.0

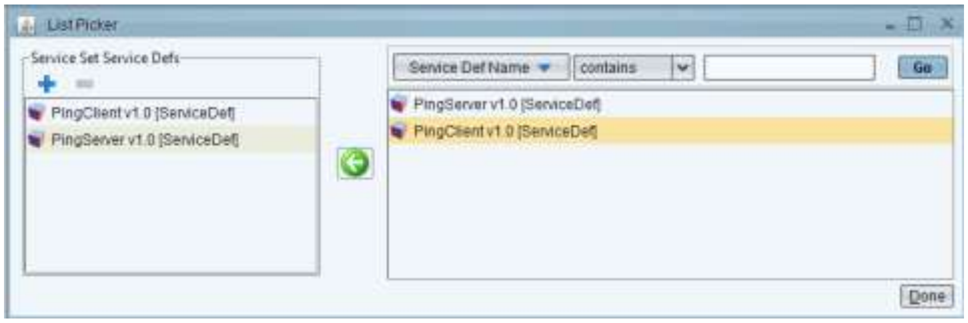
Description: The Ping Service Set includes both the Ping Server and Ping Client service.

Service Defs (0)

Service Defs:

Figure 38: Add Service Def Screenshot

5. In the List Picker dialog, we can quickly select which services definitions form the service set. In this case, we select both the PingServer and PingClient, then press the green arrow to add them to the Service Set Service Defs list.



List Picker

Service Set Service Defs

PingClient v1.0 [ServiceDef]

PingServer v1.0 [ServiceDef]

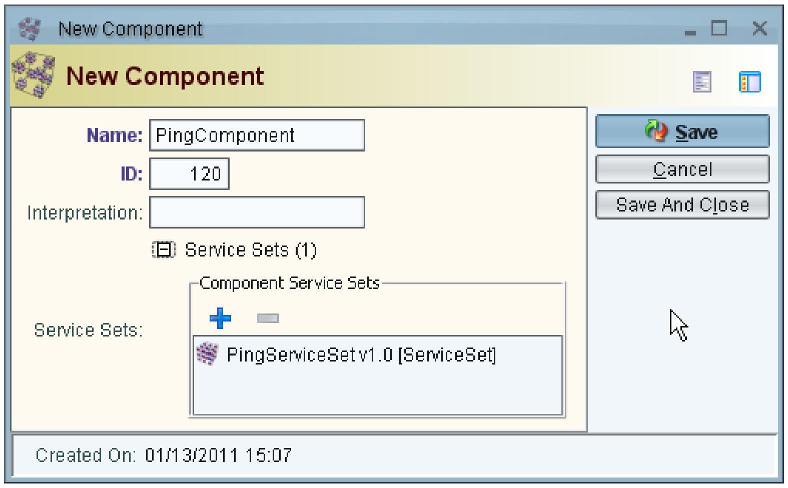
Service Def Name ▾ contains

PingServer v1.0 [ServiceDef]

PingClient v1.0 [ServiceDef]

Figure 39: Pick Service Def Screenshot

6. Select ‘Done’ in the List Picker dialog and ‘Save And Close’ on the New Service Set dialog. The PingServiceSet should now be added to our component:


New Component

Name: PingComponent

ID: 120

Interpretation:

Service Sets (1)

Service Sets:

Component Service Sets

PingServiceSet v1.0 [ServiceSet]

Created On: 01/13/2011 15:07

Figure 40: Populated New Component Screenshot

7. Click 'Save and Close' to save this new component to the database.

   If generating C++ code, continue to section 8.6. If generating Java code skip to section 8.7, and if generating C# skip to section 8.8.

8.6 Generate C++ Source and Integrate User Code

With our component modeled, we can use the JAUS Tool Set to generate source code that will adhere to the interface description we defined. The messages are created as classes, the framework is established that supports intra- and inter-component communication, and the protocol functions are defined as easy-to-understand stubs. The goal of generated code is to allow an engineer to quickly flesh out the protocol stubs and test a running system. This section describes the process for our simple ping client and server component in C++.

1. From the Model Elements list, double click the 'Components' icon to bring up the selection dialog box. Right click the PingComponent, and select "Auto Generate Code"

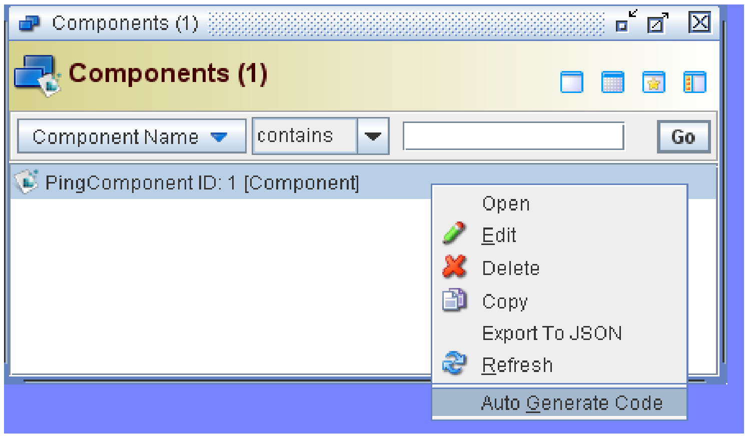
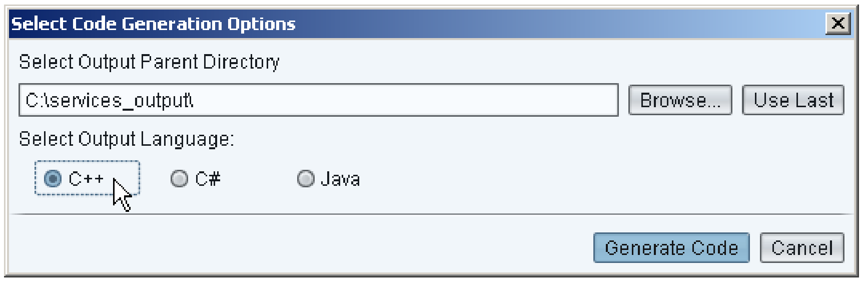


Figure 41: Generate Source Screenshot

2. The "Select Code Generation Options" dialog appears. Select the output directory and leave C++ as the selected output language. Note "Use last" restores the previous path used when generating code, and "Browse" will display a directory browse dialog so the user can avoid typing the output path by hand. Click "Generate Code" to continue.


Select Code Generation Options

Select Output Parent Directory

C:\services_output\

Select Output Language:

[✓ C++] C# Java

Figure 42: Select Code Generation Options Dialog

3. The generated code is placed in a directory named 'PingComponent_120', beneath the directory selected in the Select Code Generation Options dialog. Using "C:\services_output" as in Figure 42, generated code would be placed in C:\services_output\PingComponent_120.

4. We need to implement the behavior specified by the ping server protocol.

Examining *PingComponent_120/src/urn_jts_PingServer_1_0/PingServerProtocol.cpp*, we find the pre-generated function stub for the actions associated with the QueryHeartbeatPulse transition:

```
void PingServer::sendReportHeartbeatPulseAction()
{
     /// Insert User Code HERE
}
```

We must add the code that allows the server to respond with a Report Heartbeat Pulse message. In this case, we use the built-in sendJausMessage() function to send the response back to the local component:

```
void PingServer::sendReportHeartbeatPulseAction()
{
     // Send a RHP message to the local component.
     ReportHeartbeatPulse response;
     sendJausMessage(response, *jausRouter->getJausAddress());
}
```

5. With the server implementation complete, we now flesh out the client behavior. The relevant stubs defined in *src/urn_jts_PingClient_1_0/PingClientProtocol.cpp* are:

```
void PingClient::broadcastQueryHeartbeatPulseAction()
{
     /// Insert User Code HERE
}

void PingClient::printToScreenAction()
{
     /// Insert User Code HERE
}
```

These two actions are responsible for sending a Query Heartbeat Pulse message, as well as printing an output message when a response is received. As with the Server implementation, we use the built-in sendJausMessage() function to route the Query Heartbeat Pulse to the local component:

```
void PingClient::broadcastQueryHeartbeatPulseAction()
{
     QueryHeartbeatPulse query;
     sendJausMessage(query, *jausRouter->getJausAddress());
}

void PingClient::printToScreenAction()
{
     printf("Hello World!\n");
}
```

With our code complete, we're now ready to build and run our ping component as described in section 8.9.

8.7 Generate Java Source Code and Integrate User Code

With our component modeled, we can use the JAUS Tool Set to generate source code that will adhere to the interface description we defined. The messages are created as classes, the framework is established that supports intra- and inter-component communication, and the protocol functions are defined as easy-to-understand stubs. The goal of generated code is to allow an engineer to quickly flesh out the protocol stubs and test a running system. This section describes the process for our simple ping client and server component in Java.

1. From the Model Elements list, double click the 'Components' icon to bring up the selection dialog box. Right click the PingComponent, and select "Auto Generate Code"
2. The "Select Code Generation Options" dialog appears. Select the output directory and select Java as the selected output language. Note "Use last" restores the previous path used when generating code, and "Browse" will display a directory browse dialog so the user can avoid typing the output path by hand. Click "Generate Code" to continue.
3. The generated code is placed in a directory named 'PingComponent_120', beneath the directory selected in the Select Code Generation Options dialog. Using "C:\services_output" as in Figure 42, generated code would be placed in C:\services_output\PingComponent_120.
4. We need to implement the behavior specified by the ping server protocol.

   Examining *PingComponent_120/src/urn_jts_PingServer_1_0/PingServer_PingFSM.java*, we find the pre-generated function stub for the actions associated with the QueryHeartbeatPulse transition:

```
public void sendReportHeartbeatPulseAction()
{
     /// Insert User Code HERE
}
```

We must add the code that allows the server to respond with a Report Heartbeat Pulse message. In this case, we use the built-in sendJausMessage() function to send the response back to the local component:

```
public void sendReportHeartbeatPulseAction()
{
     // Send a RHP message to the local component.
     ReportHeartbeatPulse response = new ReportHeartbeatPulse();
     sendJausMessage(response, jausRouter.getJausAddress());
}
```

5. With the server implementation complete, we now flesh out the client behavior. The relevant stubs defined in *src/urn_jts_PingClient_1_0/PingClient_PingFSM.java* are:

```
public void broadcastQueryHeartbeatPulseAction()
{
     /// Insert User Code HERE
}

public void printToScreenAction()
{
     /// Insert User Code HERE
}
```

These two actions are responsible for sending a Query Heartbeat Pulse message, as well as printing an output message when a response is received. As with the Server implementation, we use the built-in sendJausMessage() function to route the Query Heartbeat Pulse to the local component:

```
public void broadcastQueryHeartbeatPulseAction()
{
     QueryHeartbeatPulse query = new QueryHeartbeatPulse();
     sendJausMessage(query, jausRouter.getJausAddress());
}

public void printToScreenAction()
{
     System.out.println("Hello World!");
}
```

With our code complete, we're now ready to build and run our ping component as described in section 8.9.

8.8 Generate C# Source Code and Integrate User Code

With our component modeled, we can use the JAUS Tool Set to generate source code that will adhere to the interface description we defined. The messages are created as classes, the framework is established that supports intra- and inter-component communication, and the protocol functions are defined as easy-to-understand stubs. The goal of generated code is to allow an engineer to quickly flesh out the protocol stubs and test a running system. This section describes the process for our simple ping client and server component in C#.

1. Follow steps 1 and 2 from section 8.7, selecting C# instead of Java in step 2.

2. The generated code is placed in a directory named 'PingComponent_120', beneath the directory selected in the Select Code Generation Options dialog. Using "C:\services_output" as in Figure 42, generated code would be placed in C:\services_output\PingComponent_120.

3. We need to implement the behavior specified by the ping server protocol.

Examining *PingComponent_120/src/urn_jts_PingServer_1_0/PingServer_PingFSM.cs*, we find the pre-generated function stub for the actions associated with the QueryHeartbeatPulse transition:

```
public void sendReportHeartbeatPulseAction()
{
     /// Insert User Code HERE
}
```

We must add the code that allows the server to respond with a Report Heartbeat Pulse message. In this case, we use the built-in sendJausMessage() function to send the response back to the local component:

```
public void sendReportHeartbeatPulseAction()
{
     // Send a RHP message to the local component.
     ReportHeartbeatPulse response = new ReportHeartbeatPulse();
     sendJausMessage(response, jausRouter.getJausAddress());
}
```

4. With the server implementation complete, we now flesh out the client behavior. The relevant stubs defined in *src/urn_jts_PingClient_1_0/ PingClient_PingFSM.cs* are:

```
public void broadcastQueryHeartbeatPulseAction()
{
     /// Insert User Code HERE
}

public void printToScreenAction()
{
     /// Insert User Code HERE
}
```

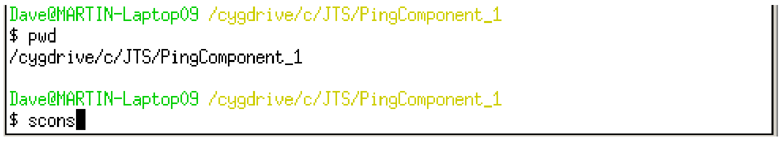
These two actions are responsible for sending a Query Heartbeat Pulse message, as well as printing an output message when a response is received. As with the Server implementation, we use the built-in sendJausMessage() function to route the Query Heartbeat Pulse to the local component:

```
public void broadcastQueryHeartbeatPulseAction()
{
     QueryHeartbeatPulse query = new QueryHeartbeatPulse();
     sendJausMessage(query, jausRouter.getJausAddress());
}

public void printToScreenAction()
{
     Console.WriteLine("Hello World!");
}
```

With our code complete, we're now ready to build and run our ping component.

8.9 Build the System

Compiling generated code requires a host of tools including python, scons, and standard C++ libraries for all generated languages. Generated C# and C++ will also require a compiler such as the one provided with Visual Studio, and Java requires the Java Runtime Environment. For a full list of supported operating systems and required tools, please see section 3.2 Environment. The Code Generator automatically creates the required SCons build scripts to make the component, all selected services, and the integrated framework. To start the build process, simply type 'scons' from the command line in the PingComponent_120 directory:



```
Dave@MARTIN-Laptop09 /cygdrive/c/JTS/PingComponent_1
$ pwd
/cygdrive/c/JTS/PingComponent_1

Dave@MARTIN-Laptop09 /cygdrive/c/JTS/PingComponent_1
$ scons
```

Figure 43: Ping Compile Command Screenshot

Once the build process is completed, the PingComponent_120 executable is generated in the PingComponent_120/bin directory. To run C++, simply execute it from the command line:

```
$> cd bin
$> ./PingComponent_120
```

for Linux or

```
$> cd bin
$> PingComponent_120.exe
```

for Windows.

To run Java, execute the following line from the command line in any system:

```
$> cd bin
$> java -jar PingComponent_120.jar
```

To run C#, execute the following line from the command line in Windows:

```
$> cd bin
$> PingComponent_120.exe
```

Note that there may be a short delay and a warning message while the Framework attempts to contact the Node Manager. This is normal, and will be discussed further in Section 16. Eventually, the “Hello World!” message will be displayed on the screen.

Now that our simple “Ping” example is complete, we’ll take a look at a more complex example that uses multiple components.

9 Creating your Second Service: Adding Two Numbers

In this section, we provide a walkthrough of how to create a service by importing existing JSIDL. The workflow is again bottom up, broken into the following steps:

1. Import JSIDL for the client and server service definitions
2. Create Service Sets for the client and server
3. Create Components for the client and server
4. Generate code for the client and server
5. Modify the generated code to add application behavior
6. Execute a communications component, and the client and server

By the end of this section, you should be comfortable importing JSIDL from the SAE services, and generating code which you modify for application specific behavior.

9.1 Identify the Interface

In this example, we consider a simple client/server that implements addition. The client sends two numbers to the server, the server adds them and returns the result.

The server supports the following messages:

- Input: *QueryAddition // has a body with two unsigned integers*
- Output: *ReportAddition // has a body with one unsigned integer*

The server behavior is represented with a simple state machine as shown below:

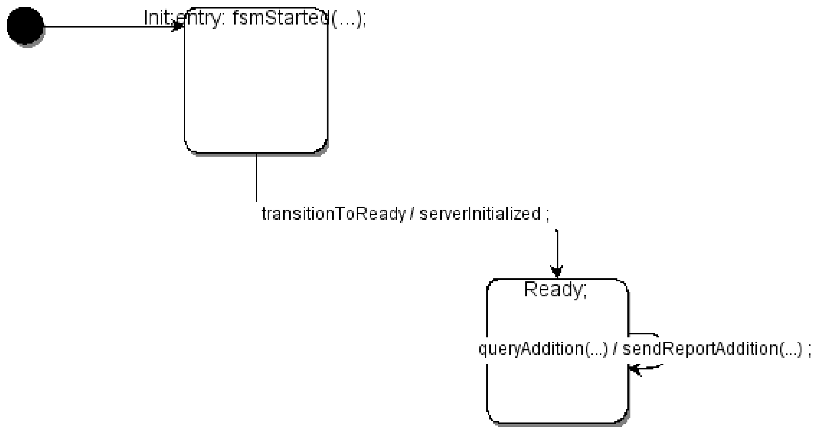

Init;entry: fsmStarted(...);
transitionToReady / serverInitialized ;
Ready;
queryAddition(...) / sendReportAddition(...) ;

Figure 44: Addition Server FSM

This FSM consists of two states: Init and Ready. There is an entry transition for Init, a transition from Init to Ready, and a self transition within Ready.

The client supports the following messages:

- Input: *ReportAddition*

The client behavior is represented with a simple state machine as shown below:

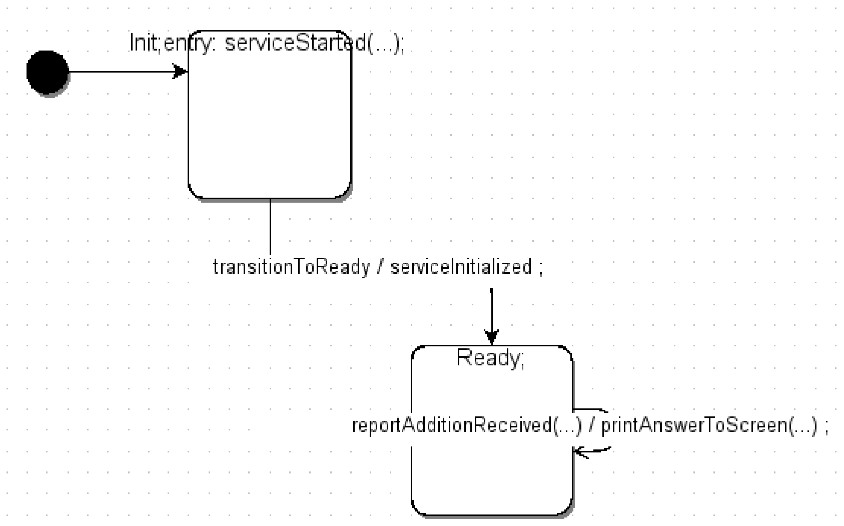

Init;entry: serviceStarted(...);
transitionToReady / serviceInitialized ;
Ready;
reportAdditionReceived(...) / printAnswerToScreen(...) ;

Figure 45: Addition Client FSM

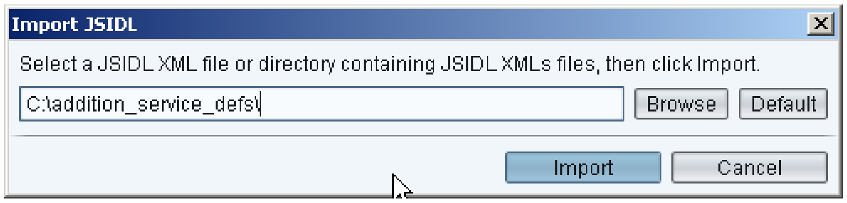
Again, we see two states: Init and Ready, with a similar transition structure to the server.

9.2 JSIDL Import

Note: The service definition files you will need for this section are located in: examples/xml/Addition/

JSIDL import for the client and server service definitions is straightforward. Upon starting JTS, do the following:

1. Right Click on "Service Defs" and select Import. In the Import JSIDL dialog that appears (Figure 46), browse to the directory containing the AdditionClient service def, select the client service def's JSIDL file, and then click OK.


Import JSIDL

Select a JSIDL XML file or directory containing JSIDL XMLs files, then click Import.

C:\addition_service_defs\

Figure 46: Import JSIDL Dialog

2. Right Click on "Service Defs" and select Import. Browse to the directory containing the AdditionServer service def, select the server service def's JSIDL, then click OK

As the services are imported, you will see flashing messages in the JTS window indicating successful import of various entities. Confirm this by double clicking on "Message Defs", "Complex Fields", and "Protocol Behaviors", and you should see the following, shown in Figure 47:

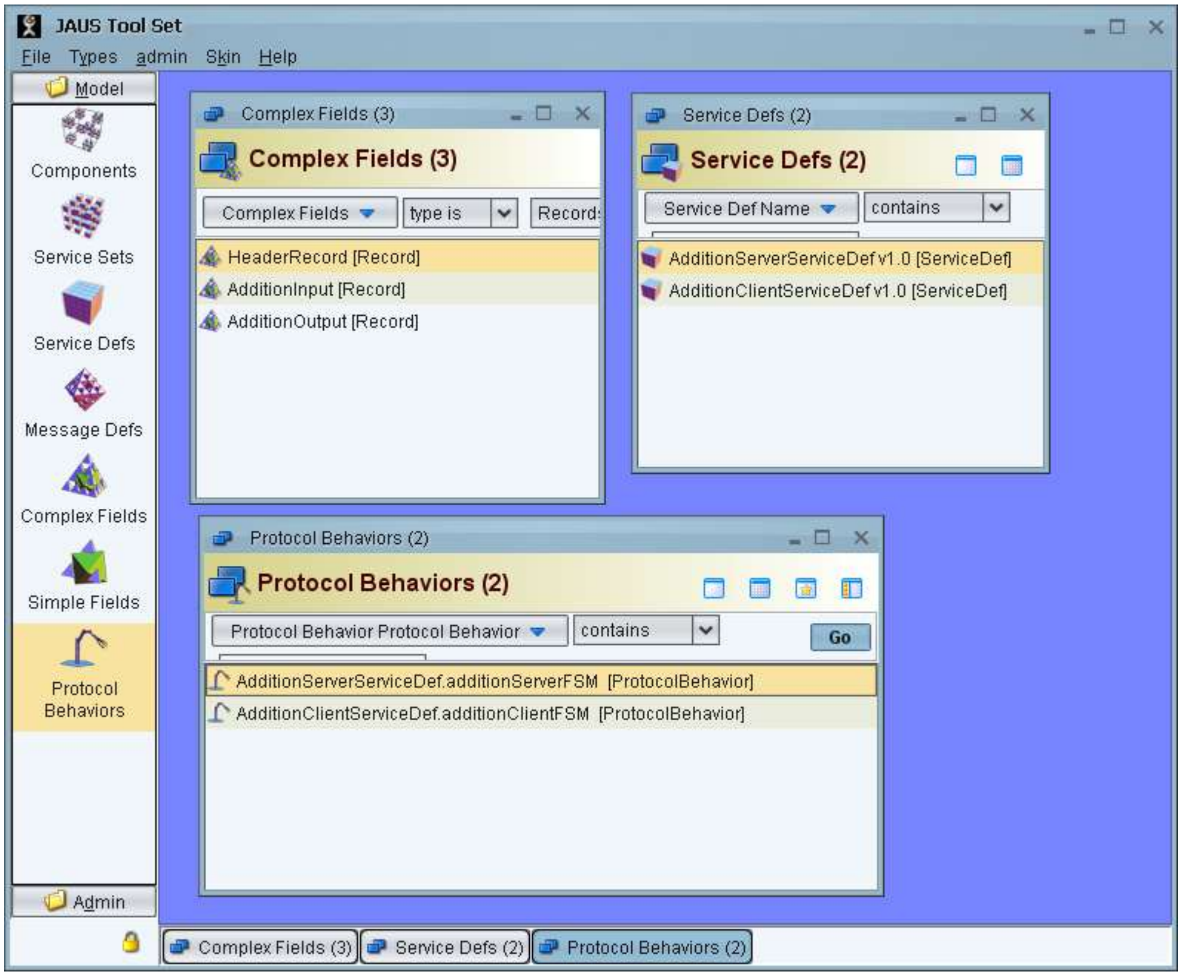

JAUS Tool Set
File Types admin Skin Help
Model
Components
Service Sets
Service Defs
Message Defs
Complex Fields
Simple Fields
Protocol Behaviors
Admin
Complex Fields (3)
Complex Fields type is Records
HeaderRecord [Record]
AdditionInput [Record]
AdditionOutput [Record]
Service Defs (2)
Service Def Name contains
AdditionServerServiceDef v1.0 [ServiceDef]
AdditionClientServiceDef v1.0 [ServiceDef]
Protocol Behaviors (2)
Protocol Behavior Protocol Behavior contains Go
AdditionServerServiceDef.additionServerFSM [ProtocolBehavior]
AdditionClientServiceDef.additionClientFSM [ProtocolBehavior]
Complex Fields (3) Service Defs (2) Protocol Behaviors (2)

Figure 47: Entities After Addition Client/Server Import

Note: We recommend you drill down into the various elements that have been defined to explore message content, service layout, protocol layout, etc.

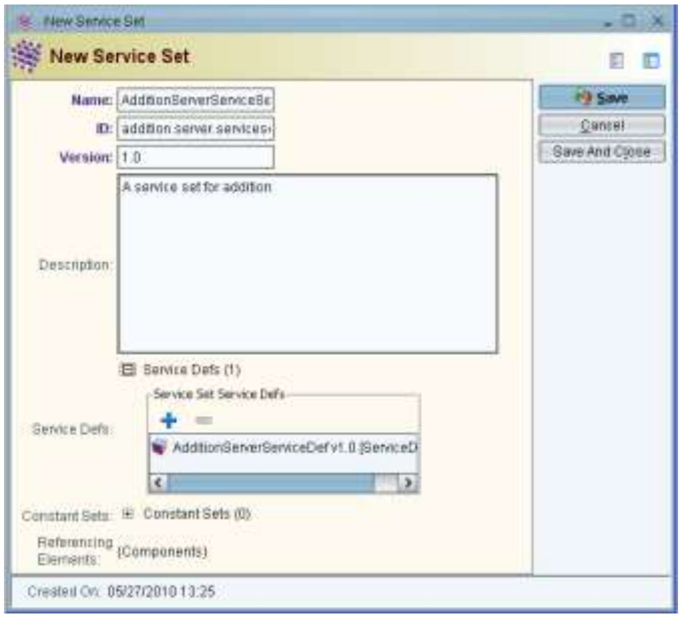
9.3 Creating the Service Sets

Once the service definitions are imported, the service sets have to be defined. A service set contains one or more service definitions. The service set for the server will contain the service definition for the server while the service set for the client will contain the service definition for the client. Create the service sets as follows:

1. For the Server: Right click on Service Set, and select "New". Fill in the fields as follows:
    - **Name:** AdditionServerServiceSet
    - **ID**: addition.server.serviceset

- **Version**: 1.0
- **Description**: A service set for addition
- **ServiceDef**: Add the *AdditionServerServiceDef* service definition to the list by clicking on the "+" icon, and selecting "Browse", the pick the correct server from the list.
- Click on "Save"

Once you are done, the New Service Set window should look like this:



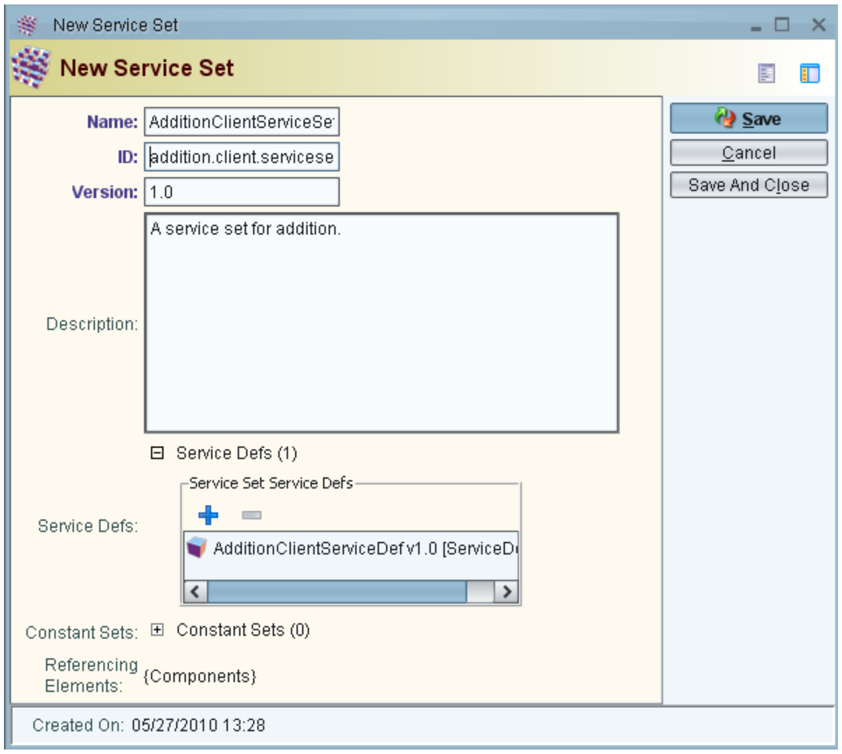
Name: AdditionServerServiceSe
ID: addition.server.services
Version: 1.0
Description: A service set for addition
Service Defs: Service Defs (1)
Service Set Service Defs
AdditionServerServiceDef v1.0 [ServiceD
Constant Sets: Constant Sets (0)
Referencing Elements: (Components)
Created On: 05/27/2010 13:25

Figure 48: New Addition Service Set Screenshot

2. For the Client: Follow the same procedure as above. Fill in the fields as follows:

- **Name**: AdditionClientServiceSet
- **ID**: addition.client.serviceset
- **Version**: 1.0
- **Description**: A service set for addition
- **ServiceDefs**: Again, click on the "+" icon, select "Browse", and add the *AdditionClientServiceDef* to the list. Click "Done".
- Click "Save"

Once you are done, the New Service Set window should look like this:



Name: AdditionClientServiceSe
ID: addition.client.servicese
Version: 1.0
Description: A service set for addition.
Service Defs: Service Defs (1)
Service Set Service Defs
AdditionClientServiceDef v1.0 [ServiceD
Constant Sets: Constant Sets (0)
Referencing Elements: {Components}
Created On: 05/27/2010 13:28

Figure 49: Populated Addition Server Set Screenshot

9.4 Create the Components

Now that the service sets are created, we need to create the actual components that will contain the service sets. We need to create a client and server component. Each component will be a self-contained executable, with a scons-based make system, and all common files included (i.e., no external dependencies).

To create the server component, do the following:

1. Right-click on Components, select "New". Then enter the following:

   - **Name:** AdditionServerComponent
   - **ID:** 150 (this is the JAUS Component ID of the component)
   - **ServiceSets:** Click on the "+", then click on the blue "+" and select "Browse". Then select "AdditionServerServiceSet", click the green arrow, and click "Done"

3. Click "Save"
4. Click "Auto Generate Code", and specify an output path in the dialog, keeping C++ as the output language. Click “Generate Code” when ready.

5. Now navigate to the output path directory from step 4, and you should see a new directory called "AdditionServerComponent_150". This contains the auto-generated files we will be modifying.

To create the Client component, do the following:

1. Right-click on Components, select "New". Then enter the following:
   - **Name:** AdditionClientComponent
   - **ID:** 151 (this is the JAUS Component ID of the component)
   - **ServiceSets:** Click on the "+", then click on the blue "+" and select "Browse". Then select "AdditionClientServiceSet", click the green arrow, and click "Done"
2. Click "Save"
3. Click "Auto Generate Code" and specify an output path in the dialog. If generating C++, leave the C++ radio button selected. Otherwise, select the appropriate radio button to generate Java or C# code. Click “Generate Code” when ready.
4. Now navigate to the output path directory from step 4, directory, and you should see a new directory called "AdditionClientComponent_151". This contains the auto-generated files we will be modifying.

9.5 Modify the C++ Server Component

Note: Working client/server code with all of the application behavior described below already available in *JTS/examples/AdditionClientComponent_151* and *JTS/examples/AdditionServerComponent_150.* The underlying directory structure (particularly where the *FSM.cpp file is stored might be slightly different than below). You can examine this if you run into issues or problems with modifying the code yourself.

Since the generated code only defines stub actions, we need to add application-level behavior to the server component. We will only need to examine / modify the following files:

```
AdditionServerComponent_150
   | src
     |urn_jaus_example_addition_server_1_0
        | AdditionServerServiceDef_additionServerFSM.cpp
        | AdditionServerServiceDefService.cpp
```

Note: By default, this example uses Subsystem 126, Node 1, Component ID 150.

Now, we want to modify the Finite State Machine to add behavior code. We therefore modify *AdditionServerServiceDef_additionServerFSM.cpp* to populate the entry action that is associated with the Init state. In this code snippet, we invoke an internal event to transition us from Init to Ready.

```
void AdditionServerServiceDef_additionServerFSM::fsmStartedAction()
{
  /// We now generate an internal event, which will be handled up
  /// above, resulting in a transition call to move us from
  /// Init to Ready
  std::cout << "Addition server started\n";
  ieHandler->invoke(new InitToReadyEventDef());
  std::cout << "Sent internal event to transition to Ready\n";
}
```

On system start-up, this code will fire off an internal event. The generated code will ‘catch’ this event, automatically transition the state machine from Init To Ready, and call the serviceInitializedAction(). For this example, there is no action to take in this transition, but we fill out the code for completeness as below.

```
void AdditionServerServiceDef_additionServerFSM::serverInitializedAction()
{
  /// Insert User Code HERE
  /// This is the action for the transitionToReady Transition.
  /// Add in whatever code is needed when transitioning from
  /// Init to Ready

  std::cout << "Transitioned from Init to Ready. Ready to begin adding!\n";

  // Nothing else needs to be done here. We'll sit in
  // READY until we get a QueryAddition
  // message. When that happens, we'll trigger a self-transition
  // back into READY that
  // computes the answer and sends it back to the requestor.
}
```

In the Ready state, we expect to receive a *QueryAddition* message. When this message is received, the generated code automatically executes the appropriate transition and calls the sendReportAdditionAction. This handler code is offered below, and does the following:

1. Extracts the two numbers to be added from the incoming QueryAddition message

2. Adds them

3. Creates a *ReportAddition* message with the result, and sends it to the component that sent us the *QueryAddition* message.

```
void AdditionServerServiceDef_additionServerFSM::sendReportAdditionAction(
                                  QueryAddition msg, unsigned int sender)
  {
    /// Insert User Code HERE
     int A1=msg.getAdditionInputBody()->getAdditionInput()->getA1();
     int A2=msg.getAdditionInputBody()->getAdditionInput()->getA2();

      // Now, let’s pull out the two numbers we received
     std::cout << " Need to add " << A1 << " + " << A2 << std::endl;

     // Now let's formulate a response
     int answer;
     answer = A1 + A2;
     ReportAddition theAnswer;
     theAnswer.getAdditionOutputBody()->
         getAdditionOutput()->setAdditionResult(answer);

     // Encode the response and send it back to the requestor.
     sendJausMessage( theAnswer, JausAddress(sender) );

     std::cout << "answer sent to client\n";
  }
```

This function takes two arguments: the message that triggered the transition as well as an unsigned integer representing the sender’s address. The code generator is able to resolve the message argument and pass in the appropriate value without any additional input from the user. However, when the function contains basic types (unsigned byte, unsigned short, unsigned int, or unsigned long), the code generator cannot automatically determine the value for these arguments. By default, basic types will be uninitialized and will display a warning message at run-time. In order to pass the proper value to the function, we need to modify some of the automatically generated code in *AdditionServerServiceDefService.cpp*.

This file represents the ‘wiring’ for the state machine; that is, the code generator automatically calls the appropriate state transition when the trigger is received. The “QueryAddition” trigger will contain the following lines:

```
unsigned int sender;
printf("WARNING!  Using parameter 'sender' without initialization!\n");
```

We need to initialize the 'sender' value with the 4-byte address of the client that sent the message. This can be found in the Receive event that triggered the transition. The Receive event stores the sender's address as three numbers: a two-byte subsystem identifier, a one-byte node identifier, and a one-byte component identifier. This is sometimes represented as an unsigned integer, where the subsystem id is stored in the highest order bits and the component id is stored in the lowest order bit. Hence, we initialize 'sender' using these values:

```
unsigned int sender =
  (casted_ie->getBody()->getReceiveRec()->getSrcSubsystemID() << 16) +
  (casted_ie->getBody()->getReceiveRec()->getSrcNodeID() << 8) +
  (casted_ie->getBody()->getReceiveRec()->getSrcComponentID());
```

It is important to note that this modification to the generated code is only required when transition arguments use primitive types. When possible, use messages and events as transition parameters, rather than primitive types. For example, by inheriting from a TransportService such as that defined by [AS5710], the sender's address can be represented by Receive.Body.ReceiveRec which will be handled automatically by the code generator.

9.6 Modify the C++ Client Component

Now, we need to add application-level behavior to the server component. We will only need to examine and modify the following file:

```
AdditionClientComponent_151
    |src
    |urn_jaus_example_addition_client_1_0
        |AdditionClientServiceDef_additionClientFSM.cpp
```

We want to modify the Finite State Machine in *AdditionClientServiceDef_additionClientFSM.cpp* to add behavior code. First, we have to populate the entry action that is associated with the Init state, just as we did in the server. In this code snippet, invoke an internal event to transition us from Init to Ready.

```
void AdditionClientServiceDef_additionClientFSM::serviceStartedAction()
{
     /// Insert User Code HERE
    std::cout << "Addition client started\n";

    /// We now generate an internal event, which will be handled up
    /// above, resulting in a transition call to move us from
    /// Init to Ready
    ieHandler->invoke(new InitToReadyEventDef());
    std::cout << "Sent internal event to transition to Ready\n";
}
```

As with the server, the internal event will cause the state machine to transition from Init to Ready. During this transition, we want to send a *QueryAddition* message to the server using the serviceInitializedAction() in the Finite State Machine. The code below creates the message, and sends a query message to add 500 + 500.

```
void AdditionClientServiceDef_additionClientFSM::serviceInitializedAction()
{
  /// Insert User Code HERE
  std::cout << "In Ready state. Let's start adding...\n";


  // This is the basic message type for our query.
  QueryAddition query;

  // The message contains a body, with a record.
  query.getAdditionInputBody()->getAdditionInput()->setA1(500);
  query.getAdditionInputBody()->getAdditionInput()->setA2(500);

  // Send the response to the server on this subsystem and node.  The
  // Component ID is fixed at 150.
  JausAddress server(jausRouter->getJausAddress()->getSubsystemID(),
                  jausRouter->getJausAddress()->getNodeID(),
                  150);

  // Encode the request and send it to the server.
  sendJausMessage( query, server );

  std::cout << "Send addition request\n";
}
```

When a *ReportAddition* message is received, the generated code will execute a self-transition back to the Ready state, calling the printAnswerToScreenAction as a result. This action takes the incoming message as a parameter, and prints the answer out to the screen:

```
void AdditionClientServiceDef_additionClientFSM::printAnswerToScreenAction(
                                                    ReportAddition msg)
{
  /// Insert User Code HERE
  std::cout << "Transitioned back to Ready\n";
  std::cout << "  The answer is "
     << msg.getAdditionOutputBody()->getAdditionOutput()->getAdditionResult()
     << std::endl;
}
```

Now that the code is written, we need to compile it, and execute the Node Manager, Server, and Client.

9.7 Modify the Java Server Component

Note: Working client/server code with all of the application behavior described below is already available in *JTS/examples/AdditionClientComponent_151* and *JTS/examples/AdditionServerComponent_150.* The generated folder names will be different than the examples below). You can examine this example code if you run into issues or problems with modifying the code yourself.

Since the generated code only defines stub actions, we need to add application-level behavior to the server component. We will only need to examine / modify the following files:

```
AdditionServerComponent_150
   | src
     |urn_jaus_example_addition_server_1_0
        | AdditionServerServiceDef_additionServerFSM.java
        | AdditionServerServiceDefService.java
```

Note: By default, this example uses Subsystem 126, Node 1, Component ID 150.

Now, we want to modify the Finite State Machine to add behavior code. We therefore modify *AdditionServerServiceDef_additionServerFSM.java* to populate the entry action that is associated with the Init state. In this code snippet, we invoke an internal event to transition us from Init to Ready.

```
  public void fsmStartedAction()
  {
    /// We now generate an internal event, which will be handled up
    /// above, resulting in a transition call to move us from
    /// Init to Ready
    System.out.println("Addition server started");
    ieHandler.invoke(new InitToReadyEventDef());
    System.out.println("Sent internal event to transition to Ready");
  }
```

On system start-up, this code will fire off an internal event. The generated code will 'catch' this event, automatically transition the state machine from Init To Ready, and call the serviceInitializedAction(). For this example, there is no action to take in this transition, but we fill out the code for completeness as below.

```
  public void serverInitializedAction()
  {
    /// This is the action for the transitionToReady Transition.
    /// Add in whatever code is needed when transitioning from
    /// Init to Ready

    System.out.println("Transitioned from Init to Ready. Ready to begin add-
ing!");

    // Nothing else needs to be done here. We'll sit in
    // READY until we get a QueryAddition
    // message. When that happens, we'll trigger a self-transition
    // back into READY that
    // computes the answer and sends it back to the requestor.
  }
```

In the Ready state, we expect to receive a *QueryAddition* message. When this message is received, the generated code automatically executes the appropriate transition and calls the sendReportAdditionAction. This handler code is offered below, and does the following:

2. Extracts the two numbers to be added from the incoming QueryAddition message

2. Adds the two numbers together.

3. Creates a *ReportAddition* message with the result, and sends it to the component that sent us the *QueryAddition* message.

```
public void sendReportAdditionAction(QueryAddition msg, long sender)
  {
     long A1=msg.getAdditionInputBody().getAdditionInput().getA1();
     long A2=msg.getAdditionInputBody().getAdditionInput().getA2();

      // Now, let's pull out the two numbers we received
     System.out.println(" Need to add " + A1 + " + " + A2 );

     // Now let's formulate a response
     long answer;
     answer = A1 + A2;
     ReportAddition theAnswer = new ReportAddition();
     theAnswer.getAdditionOutputBody()
               .getAdditionOutput().setAdditionResult(answer);

     // Encode the response and send it back to the requestor.
     sendJausMessage( theAnswer, new JausAddress(sender) );
```

```
      System.out.println("answer sent to client");
   }
```

This function takes two arguments: the message that triggered the transition as well as an unsigned integer representing the sender's address. The code generator is able to resolve the message argument and pass in the appropriate value without any additional input from the user. However, when the function contains basic types (unsigned byte, unsigned short, unsigned int, or unsigned long), the code generator cannot automatically determine the value for these arguments. By default, basic types will be uninitialized and will display a warning message at run-time. In order to pass the proper value to the function, we need to modify some of the automatically generated code in *AdditionServerServiceDefService.java*.

This file represents the 'wiring' for the state machine; that is, the code generator automatically calls the appropriate state transition when the trigger is received. The "QueryAddition" trigger will contain the following lines:

```
long sender;
System.out.println   ("WARNING!     Using   parameter   'sender'   without
initialization!\n");
```

We need to initialize the 'sender' value with the 4-byte address of the client that sent the message. This can be found in the Receive event that triggered the transition. The Receive event stores the sender's address as three numbers: a two-byte subsystem identifier, a one-byte node identifier, and a one-byte component identifier. This is sometimes represented as an unsigned integer, where the subsystem id is stored in the highest order bits and the component id is stored in the lowest order bit. Hence, we initialize 'sender' using these values:

```
long sender = (long)
   (casted_ie.getBody().getReceiveRec().getSrcSubsystemID() << 16) +
   (casted_ie.getBody().getReceiveRec().getSrcNodeID() << 8) +
   (casted_ie.getBody().getReceiveRec().getSrcComponentID());
```

It is important to note that this modification to the generated code is only required when transition arguments use primitive types. When possible, use messages and events as transition parameters, rather than primitive types. For example, by inheriting from a TransportService such as that defined by [AS5710], the sender's address can be represented by Receive.Body.ReceiveRec which will be handled automatically by the code generator.

9.8 Modify the Java Client Component

Now, we need to add application-level behavior to the server component. We will only need to examine and modify the following file:

```
AdditionClientComponent_151
    |src
    |urn_jaus_example_addition_client_1_0
         |AdditionClientServiceDef_additionClientFSM.java
```

We want to modify the Finite State Machine in *AdditionClientServiceDef_additionClientFSM.java* to add behavior code. First, we have to populate the entry action that is associated with the Init state, just as we did in the server. In this code snippet, invoke an internal event to transition us from Init to Ready.

```
public void serviceStartedAction()
{
     System.out.println( "Addition client started");

     /// We now generate an internal event, which will be handled up
     /// above, resulting in a transition call to move us from
     /// Init to Ready
     ieHandler.invoke(new InitToReadyEventDef());
     System.out.println("Sent internal event to transition to Ready");
}
```

As with the server, the internal event will cause the state machine to transition from Init to Ready. During this transition, we want to send a *QueryAddition* message to the server using the serviceInitializedAction() in the Finite State Machine. The code below creates the message, and sends a query message to add 500 + 500.

```
public void_ serviceInitializedAction()
{
   System.out.println("In Ready state. Let's start adding...");

   // This is the basic message type for our query.
   QueryAddition query = new QueryAddition();

   // The message contains a body, with a record.
   query.getAdditionInputBody().getAdditionInput().setA1((long)500);
   query.getAdditionInputBody().getAdditionInput().setA2((long)500);
```

```
  // Send the response to the server on this subsystem and node.  The
  // Component ID is fixed at 150.
  JausAddress server = new JausAddress(
                     jausRouter.getJausAddress().getSubsystemID(),
                     jausRouter.getJausAddress().getNodeID(),
                     (short) 150);

  // Encode the request and send it to the server.
  sendJausMessage( query, server );

  System.out.println("Send addition request");
}
```

When a *ReportAddition* message is received, the generated code will execute a self-transition back to the Ready state, calling the printAnswerToScreenAction as a result. This action takes the incoming message as a parameter, and prints the answer out to the screen:

```
public void printAnswerToScreenAction(
                                                          ReportAddition msg)
{
  System.out.println( "Transitioned back to Ready");
  System.out.println( "  The answer is "
      + msg.getAdditionOutputBody().getAdditionOutput().getAdditionResult());
}
```

Now that the code is written, we need to compile it, and execute the Node Manager, Server, and Client.

9.9 Modify the C# Server Component

Note: Working client/server code with all of the application behavior described below already available in *JTS/examples/AdditionClientComponent_151* and *JTS/examples/AdditionServerComponent_150.* The underlying directory structure (particularly where the *FSM.cs file is stored might be slightly different than below). You can examine this if you run into issues or problems with modifying the code yourself.

Since the generated code only defines stub actions, we need to add application-level behavior to the server component. We will only need to examine / modify the following files:

```
AdditionServerComponent_150
   | src
     |urn_jaus_example_addition_server_1_0
        | AdditionServerServiceDef_additionServerFSM.cs
        | AdditionServerServiceDefService.cs
```

Note: By default, this example uses Subsystem 126, Node 1, Component ID 150.

Now, we want to modify the Finite State Machine to add behavior code. We therefore modify *AdditionServerServiceDef_additionServerFSM.cs* to populate the entry action that is associated with the Init state. In this code snippet, we invoke an internal event to transition us from Init to Ready.

```
public void fsmStartedAction()
{
  /// We now generate an internal event, which will be handled up
  /// above, resulting in a transition call to move us from
  /// Init to Ready
  Console.WriteLine( "Addition server started ");
  ieHandler.invoke(new InitToReadyEventDef());
  Console.WriteLine( "Sent internal event to transition to Ready");
}
```

On system start-up, this code will fire off an internal event. The generated code will 'catch' this event, automatically transition the state machine from Init To Ready, and call the serverInitializedAction(). For this example, there is no action to take in this transition, but we fill out the code for completeness as below.

```
public void serverInitializedAction()
{
  /// This is the action for the transitionToReady Transition.
  /// Add in whatever code is needed when transitioning from
  /// Init to Ready

 Console.WriteLine("Transitioned from Init to Ready. Ready to begin adding");

  // Nothing else needs to be done here. We'll sit in
  // READY until we get a QueryAddition
  // message. When that happens, we'll trigger a self-transition
  // back into READY that
  // computes the answer and sends it back to the requestor.
}
```

In the Ready state, we expect to receive a *QueryAddition* message. When this message is received, the generated code automatically executes the appropriate transition and calls the sendReportAdditionAction. This handler code is offered below, and does the following:

3. Extracts the two numbers to be added from the incoming QueryAddition message

2. Adds them

3. Creates a *ReportAddition* message with the result, and sends it to the component that sent us the *QueryAddition* message.

```
public void _sendReportAdditionAction(
                                QueryAddition msg, uint sender)
  {
     uint A1=msg.getAdditionInputBody().getAdditionInput().getA1();
     uint A2=msg.getAdditionInputBody().getAdditionInput().getA2();

      // Now, let’s pull out the two numbers we received
     Console.WriteLine( " Need to add " + A1 + " + " + A2 );

     // Now let's formulate a response
     uint answer;
     answer = A1 + A2;
     ReportAddition theAnswer = new ReportAddition();
     theAnswer.getAdditionOutputBody().
              getAdditionOutput().setAdditionResult(answer);

     // Encode the response and send it back to the requestor.
     sendJausMessage( theAnswer, new JausAddress(sender) );

     Console.WriteLine( "answer sent to client");
  }
```

This function takes two arguments: the message that triggered the transition as well as an unsigned integer representing the sender’s address. The code generator is able to resolve the message argument and pass in the appropriate value without any additional input from the user. However, when the function contains basic types (unsigned byte, unsigned short, unsigned int, or unsigned long), the code generator cannot automatically determine the value for these arguments. By default, basic types will be uninitialized and will display a warning message at run-time. In order to pass the proper value to the function, we need to modify some of the automatically generated code in *AdditionServerServiceDefService.cs*.

This file represents the ‘wiring’ for the state machine; that is, the code generator automatically calls the appropriate state transition when the trigger is received. The “QueryAddition” trigger will contain the following lines:

```
uint sender;
Console.WriteLine   ("WARNING!     Using   parameter   'sender'   without
initialization!\n");
```

We need to initialize the ‘sender’ value with the 4-byte address of the client that sent the message. This can be found in the Receive event that triggered the transition. The Receive event stores the sender’s address as three numbers: a two-byte subsystem identifier, a one-byte node identifier, and a one-byte component identifier. This is sometimes represented as an unsigned integer, where the subsystem id is stored in the highest order bits and the component id is stored in the lowest order bit. Hence, we initialize ‘sender’ using these values:

```
uint sender = (uint)
   ((casted_ie.getBody().getReceiveRec().getSrcSubsystemID() << 16) +
    (casted_ie.getBody().getReceiveRec().getSrcNodeID() << 8) +
   (casted_ie.getBody().getReceiveRec().getSrcComponentID()));
```

It is important to note that this modification to the generated code is only required when transition arguments use primitive types. When possible, use messages and events as transition parameters, rather than primitive types. For example, by inheriting from a TransportService such as that defined by [AS5710], the sender’s address can be represented by Receive.Body.ReceiveRec which will be handled automatically by the code generator.

9.10 Modify the C# Client Component

Now, we need to add application-level behavior to the server component. We will only need to examine and modify the following file:

```
AdditionClientComponent_151
    |src
    |urn_jaus_example_addition_client_1_0
         |AdditionClientServiceDef_additionClientFSM.cs
```

We want to modify the Finite State Machine in *AdditionClientServiceDef_additionClientFSM.cs* to add behavior code. First, we have to populate the entry action that is associated with the Init state, just as we did in the server. In this code snippet, invoke an internal event to transition us from Init to Ready.

```
public void serviceStartedAction()
{
     Console.WriteLine( "Addition client started");

     /// We now generate an internal event, which will be handled up
```

```
    /// above, resulting in a transition call to move us from
    /// Init to Ready
    ieHandler.invoke(new InitToReadyEventDef());
    Console.WriteLine( "Sent internal event to transition to Ready");
}
```

As with the server, the internal event will cause the state machine to transition from Init to Ready. During this transition, we want to send a *QueryAddition* message to the server using the serviceInitializedAction() in the Finite State Machine. The code below creates the message, and sends a query message to add 500 + 500.

```
public void serviceInitializedAction()
{
  Console.WriteLine( "In Ready state. Let's start adding...");

  // This is the basic message type for our query.
  QueryAddition query = new QueryAddition();

  // The message contains a body, with a record.
  query.getAdditionInputBody().getAdditionInput().setA1(500);
  query.getAdditionInputBody().getAdditionInput().setA2(500);

  // Send the response to the server on this subsystem and node.  The
  // Component ID is fixed at 150.
  JausAddress server = new JausAddress(
                 jausRouter.getJausAddress().getSubsystemID(),
                 jausRouter.getJausAddress().getNodeID(),
                 150);

  // Encode the request and send it to the server.
  sendJausMessage( query, server );

  Console.WriteLine( "Send addition request");
}
```

When a *ReportAddition* message is received, the generated code will execute a self-transition back to the Ready state, calling the printAnswerToScreenAction as a result. This action takes the incoming message as a parameter, and prints the answer out to the screen:

```
public void printAnswerToScreenAction(
                                                    ReportAddition msg)
{
  Console.WriteLine( "Transitioned back to Ready");
  Console.WriteLine( "  The answer is "
     + msg.getAdditionOutputBody().getAdditionOutput().getAdditionResult());
}
```

Now that the code is written, we need to compile it, and execute the Node Manager, Server, and Client.

9.11 Compiling and Executing the Client and Server

The Code Generator generates a full build system, based on *scons*. Therefore, the user only has to execute scons to build the generated components. This is done as follows:

```
%> cd ~/JTS/GUI/Components/AdditionClientComponent_151
%> scons
%>
%> cd ~/JTS/GUI/Components/AdditionServerComponent_150
%> scons
```

In addition, the Node Manager needs to be compiled as well. The Node Manager is responsible for message passing between components located on the same computer or remotely. Compile the node manager using *scons*:

```
%> cd ~/JTS/nodeManager
%> scons
```

Now, open up three terminals, one each for the Node Manager, Server, and Client:

Start the Node Manager

```
%> cd ~/JTS/nodeManager/bin
%> ./NodeManager.exe nm.cfg
```

On Linux

```
%> cd /JTS/nodeManager/bin
%> NodeManager.exe nm.cfg
```

On Windows

By default, the Node Manager does not display output.

Start the Server

```
%> cd ~/JTS/GUI/Components/AdditionServerComponent_150/bin/
%> ./AdditionServerComponent_150.exe
```

In C++ on Linux,

```
%> cd ~/JTS/GUI/Components/AdditionServerComponent_150/bin/
%> java –jar AdditionServerComponent_150.jar
```

in Java,

```
%> cd /JTS/GUI/Components/AdditionServerComponent_150/bin/
%> AdditionServerComponent_150.exe
```

In C++ and C# on Windows,

You should see output similar to the following:

```
$ ./AdditionServerComponent_150.exe
Addition server started
Sent internal event to transition to Ready
  Transitioning to Ready
Transitioned from Init to Ready. Ready to begin adding!
```

Start the Client

```
%> cd ~/JTS/GUI/Component/AdditionClientComponent_151/bin/
%> ./AdditionClientComponent_151.exe
```

In C++ on Linux,

```
%> cd ~/JTS/GUI/Components/AdditionClientComponent_151/bin/
%> java –jar AdditionClientComponent_151.jar
```

in Java,

```
%> cd /JTS/GUI/Components/AdditionClientComponent_151/bin/
%> AdditionClientComponent_151.exe
```

In C++ and C# on Windows,

You should see output similar to:

```
Addition client started
Sent internal event to transition to Ready
  Transitioning to Ready
In Ready state. Let's start adding...
Send addition request
Transitioned to Ready
  The answer is 1000
```

While the server prints:

```
Need to add 500 + 500
answer sent to client
```

10 Defining Message Elements

In this section we outline how to interact with the GUI to create the elements that make up messages. This includes simple and complex fields, as defined in AS-5684.

We recommend you read the AS-5684 section titled "Message Encoding". In particular, reference Figure 5 for a hierarchy of field types.

10.1 Simple Fields

To create a simple field, right click on the simple field selection on the left hand of the GUI, and select "New". The first thing you will be asked to do is select the type of Simple Field you wish to create. Your options are:

- **Fixed Field:** The basic field that is used to contain ints, shorts, characters, etc.
- **Bit Field:** A bit field assigns specific meaning to individual bits within a primitive data type
- **Variable Length Field:** This is a simple field whose length can vary at run time. It is preceded by a meta field called a *count_field* that specifies the size of the data in bytes.
- **Fixed Length String:** A fixed length character array
- **Variable Length String:** A variable length character array. The data in a variable length string is preceded by a *count_field* that specifies the size of the data in bytes.
- **Variable Format Field:** This field contains a BLOB whose format can vary at runtime. It is preceded by 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.
- **Variable Field:** This field allows for run-time selection of the type and units. It is preceded by a *type_and_units_enum* that specifies type and units
- **Array**: An array is a single or multi-dimensional collection of simple fields.

Each simple field has a set of required and optional parameters in common. These are:

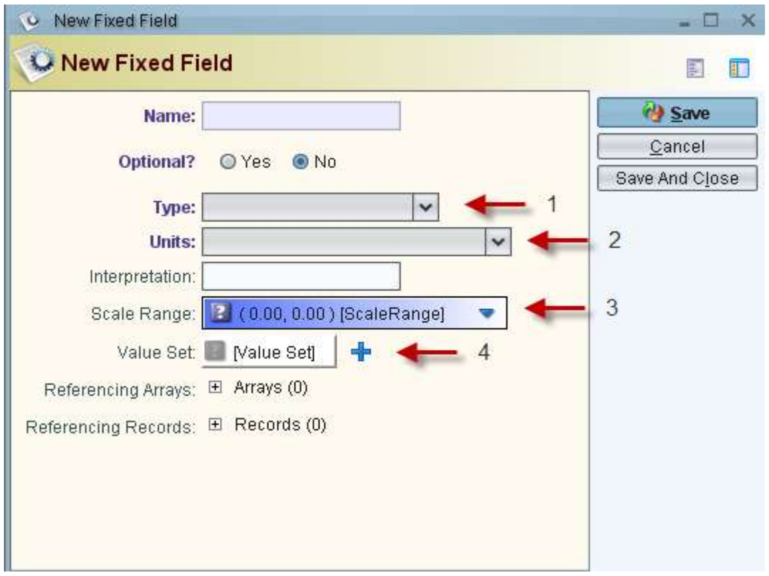
- **Name:** The unique name of the simple field (required)

- **Optional:** If true, a presence vector bit is assigned. If false, the field is required. (required)
- **Interpretation**: A textual interpretation of the field (optional)

In addition, you will see a list of all instances of higher level objects (containers such as records) where the simple field is used.

Specific instructions on field creation are below. Each field is described with a screenshot of the field creation window, with numbered annotations describing each entry unique to the field.

10.1.1 Fixed Field



Name:

Optional? Yes [✓ No]

Type:

Units:

Interpretation:

Scale Range: (0.00, 0.00) [ScaleRange]

Value Set: [Value Set]

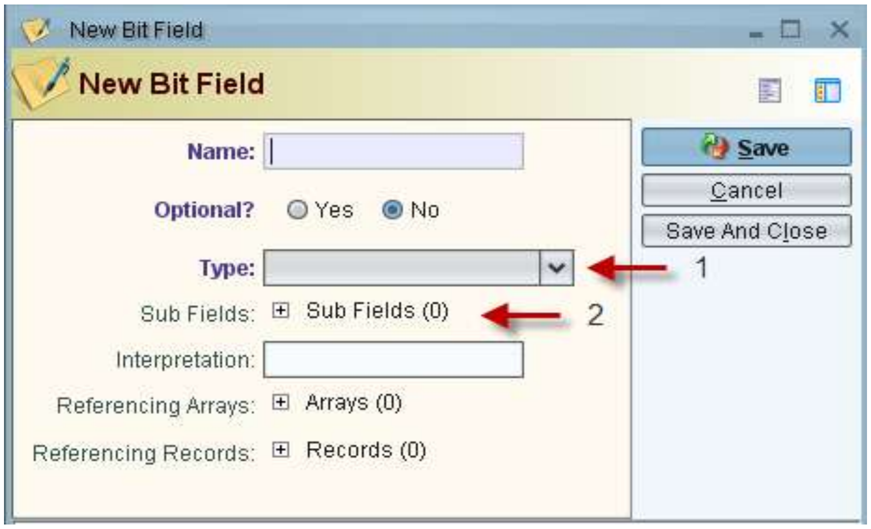
Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 50: Fixed Field Entry

1. The C primitive value type of the fixed field
2. The unit of the fixed field
3. An optional range that can be used for scales
4. Value Set: TODO->add description

10.1.2 Bit Field



Name:

Optional? Yes [✓ No]

Type:

Sub Fields: Sub Fields (0)

Interpretation:

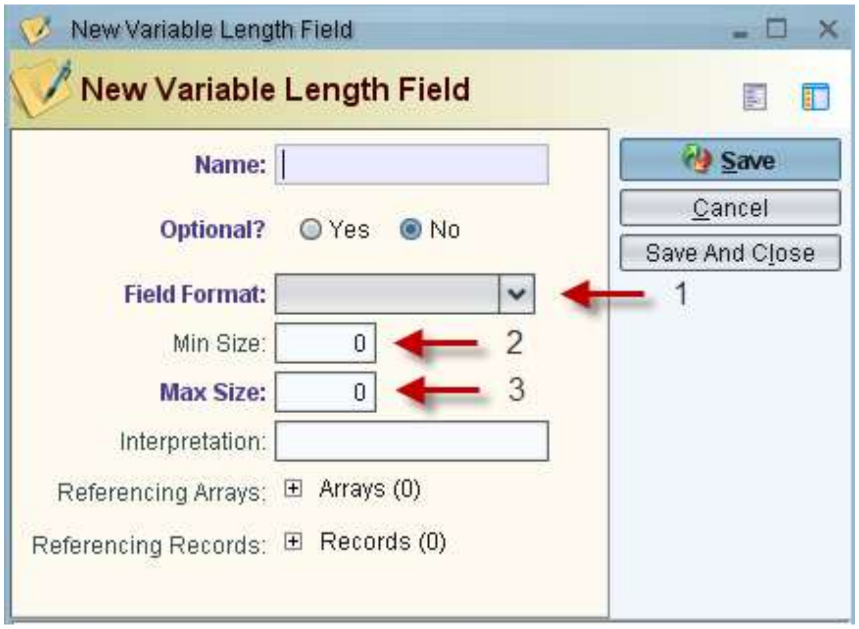
Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 51: Bit Field Entry

1. The C primitive value type of the fixed field
2. Sub Fields: The subfields describe individual bit fields in terms of bits they span and real values those bits can take on (e.g., bits 3-5 can take on values of 0-3)

10.1.3 Variable Length Field

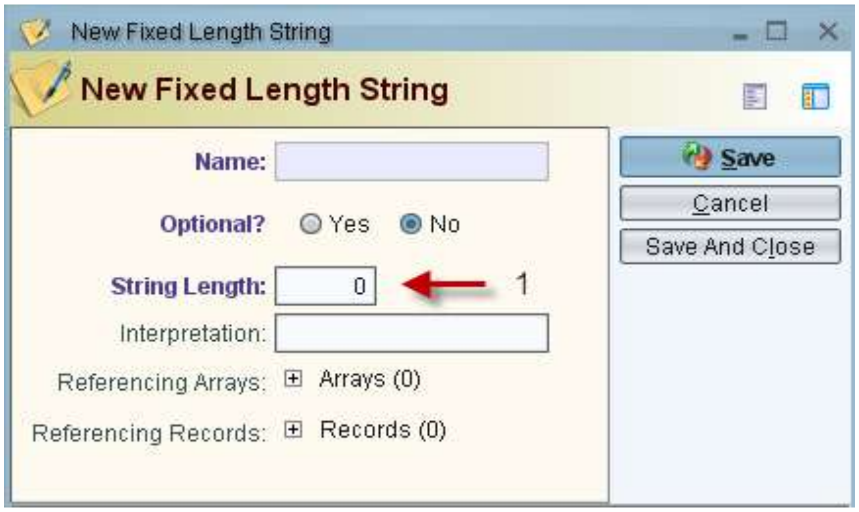


Name:
Optional? Yes [✓ No]
Field Format:
Min Size: 0
Max Size: 0
Interpretation:
Referencing Arrays: Arrays (0)
Referencing Records: Records (0)

Figure 52: Variable Length Field Entry

1. The format of the variable length field
2. The minimum size of the variable length field (must be greater than 1)
3. The maximum size of the variable length field (must be greater than minimum size)

10.1.4 Fixed Length String


New Fixed Length String

Name:

Optional? Yes [✓ No]

String Length: 0

Interpretation:

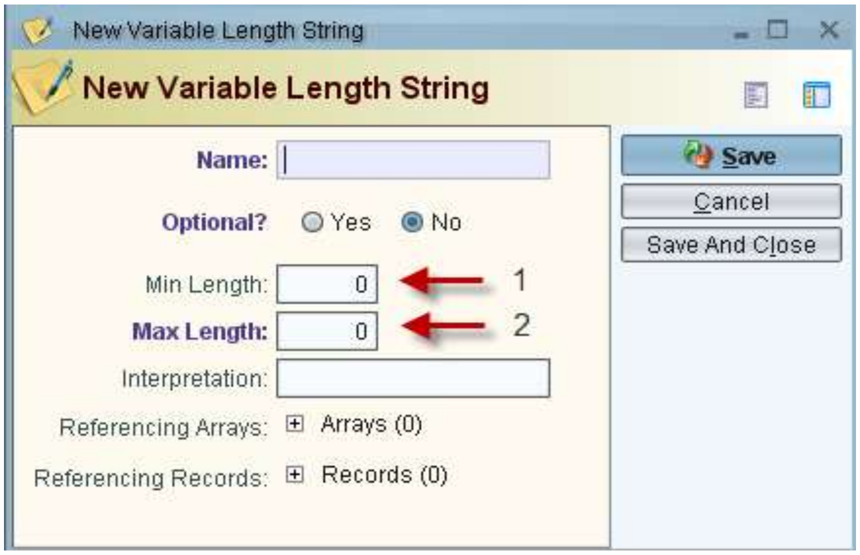
Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 53: Fixed Length String Entry

1. The length of the string. A string length must be greater than zero.

10.1.5 Variable Length String



Name:

Optional? Yes [✓ No]

Min Length: 0

Max Length: 0

Interpretation:

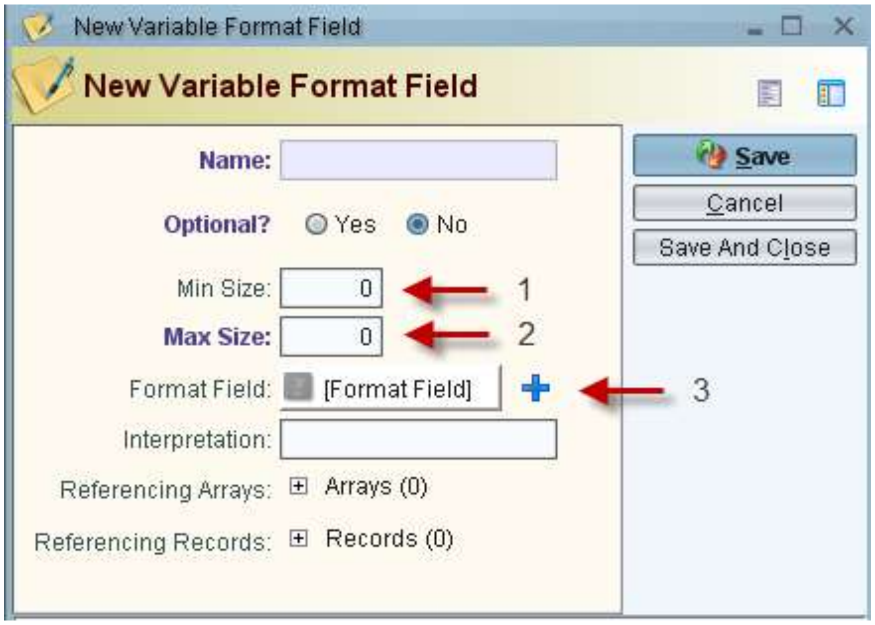
Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 54: Variable Length String Entry

1. The minimum length of the string. A string length must be greater than zero.
2. The maximum length of the string. Must be greater than the minimum string length.

10.1.6 Variable Format Field

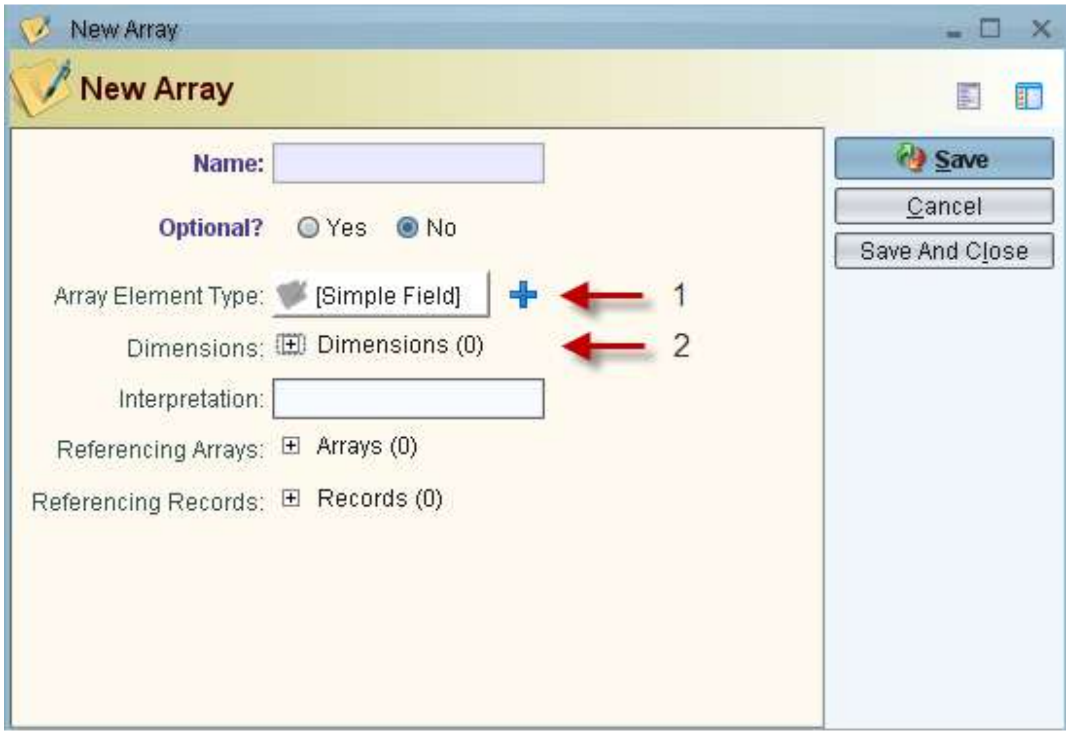


Name:	
Optional?	Yes [✓ No]
Min Size:	0
Max Size:	0
Format Field:	[Format Field]
Interpretation:	
Referencing Arrays:	Arrays (0)
Referencing Records:	Records (0)

Figure 55: Variable Format Field Entry

1. The minimum length of the string. A string length must be greater than zero.
2. The maximum length of the string. Must be greater than the minimum string length.
3. Format Field: The format field allows you to enter a list of allowable formats for this field.

10.1.7 Array



Name:

Optional? Yes [✓ No]

Array Element Type: [Simple Field]

Dimensions: Dimensions (0)

Interpretation:

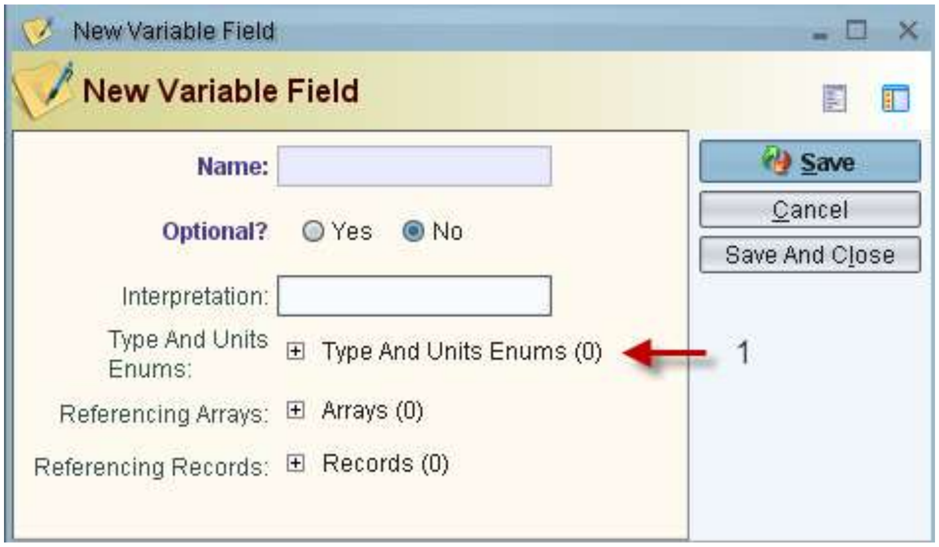
Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 56: Array Entry

1. The simple field type that the array is made up of
2. The dimensions of the array: The dimensions of the array range from 1…n.

10.1.8 Variable Field



Name:

Optional? Yes [✓ No]

Interpretation:

Type And Units Enums: ⊞ Type And Units Enums (0)

Referencing Arrays: ⊞ Arrays (0)

Referencing Records: ⊞ Records (0)

Figure 57: Variable Field Entry

1. A list of all types and unit enums defined for the variable field

10.2 Complex Fields

A complex field contains one or more simple fields. The most common complex field is the *Record*, which is used to hold a sequence of simple fields. The Record forms the basis for many SAE JAUS messages. JTS supports the following complex fields:

- **Record:** A record is an arrangement of one or more simple fields or arrays.
- **Lists:** A list is a variable sized sequence of composite fields, such as records, sequences, and variants.
- **Sequences:** A sequence is an arrangement of one or more composite fields (excluding arrays).
- **Variants:** A variant is a composite field that can hold zero or one of several different types of pre-defined composite fields at runtime.

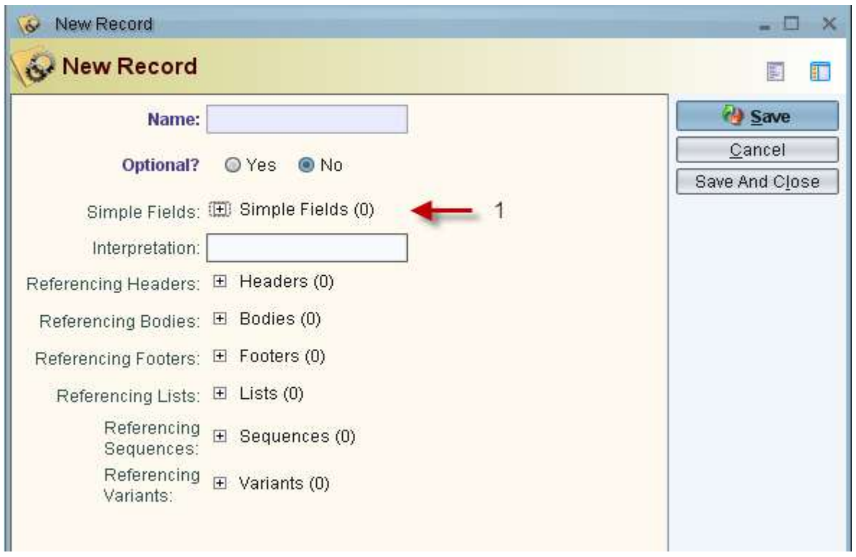
To create a complex field, right click on the simple field selection on the left hand of the GUI. All complex fields have a required name box that you enter the unique name of the simple field into

All complex fields also have an optional Boolean which allows you to set whether the simple field is optional when it is included in a record or array. All complex fields also have an optional interpretation box that can be used to describe the simple field for later reference

All complex fields have a list of all instances of where the complex field is used. The references on these lists can be modified by double clicking on the reference in question.

10.2.1 Record

A record represents an arrangement of one or more Simple Fields or arrays leading to a heterogeneous or homogeneous set that is ordered.



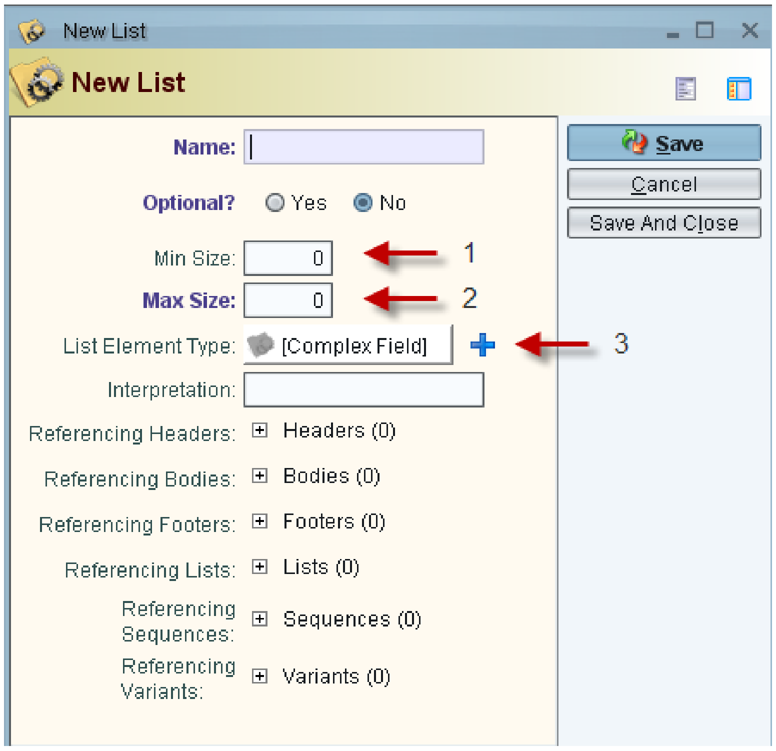
Name:
Optional? Yes [✓ No]
Simple Fields: Simple Fields (0)
Interpretation:
Referencing Headers: Headers (0)
Referencing Bodies: Bodies (0)
Referencing Footers: Footers (0)
Referencing Lists: Lists (0)
Referencing Sequences: Sequences (0)
Referencing Variants: Variants (0)

Figure 58: Record Entry

1. A list of simple fields defined for this record.

10.2.2 List

A list field represents a variable sized sequence of Composite fields of the same type, with the exception of array



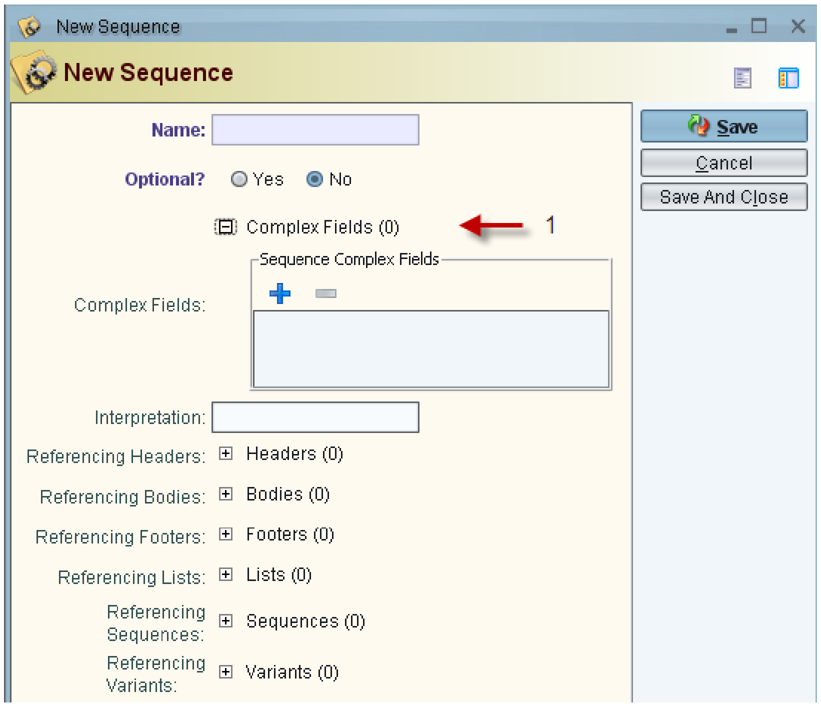
Name:
Optional? Yes [✓ No]
Min Size: 0
Max Size: 0
List Element Type: [Complex Field]
Interpretation:
Referencing Headers: Headers (0)
Referencing Bodies: Bodies (0)
Referencing Footers: Footers (0)
Referencing Lists: Lists (0)
Referencing Sequences: Sequences (0)
Referencing Variants: Variants (0)

Figure 59: List Entry

1. Required minimum size of the list. Must be greater than zero.
2. Required maximum size of the list. Must be greater than the minimum size.
3. The type of complex field that makes up the list

10.2.3 Sequence

A sequence represents an arrangement of one or more Composite Fields (excluding arrays) leading to a heterogeneous or homogeneous sequence



Name:

Optional? Yes [✓ No]

Complex Fields: Complex Fields (0)

Sequence Complex Fields

Interpretation:

Referencing Headers: Headers (0)

Referencing Bodies: Bodies (0)

Referencing Footers: Footers (0)

Referencing Lists: Lists (0)

Referencing Sequences: Sequences (0)

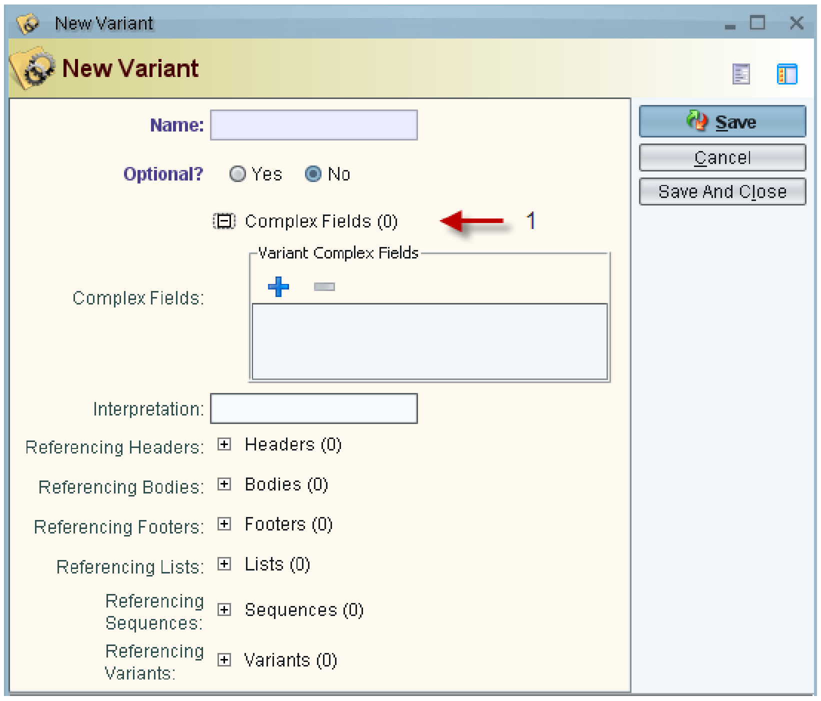
Referencing Variants: Variants (0)

Figure 60: Sequence Entry

1. A list of the complex fields defined for this sequence

10.2.4 Variant

A variant represents a composite that can hold zero or one of several different types of pre-defined composite fields at runtime


New Variant

New Variant

Name:

Optional? Yes [✓ No]

Complex Fields (0)

Complex Fields: Variant Complex Fields

Interpretation:

Referencing Headers: Headers (0)

Referencing Bodies: Bodies (0)

Referencing Footers: Footers (0)

Referencing Lists: Lists (0)

Referencing Sequences: Sequences (0)

Referencing Variants: Variants (0)

Figure 61: Variant Entry

1. A list of the complex fields defined for this variant

10.3 Complex Field Examples

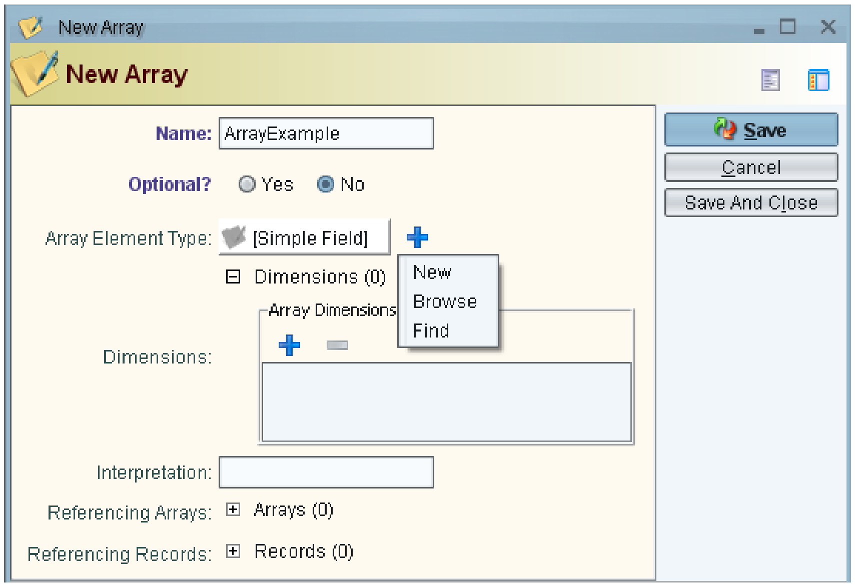
10.3.1 Array Example

Our example will be packing RGB information in a multidimensional array. Since we are using an array, we can only use one type of simple field for the array's primitive values. We will be using a bit field because it is the most logical choice for representing a choice of red, blue or green as byte values. The array we will be defining is shown below.

(R, G, B)	(R, G, B)	…	…	(1000th)
(R, G, B)	…			
…		…		
…			…	
(1000th)				(1000th)

This is a 1000x1000 array with a 3x1 vector at each index

This array can be thought of as storing the information needed to display a picture to a screen. Each 3x1 vector or R,G,B values stores the color value that needs to be displayed at a certain index on the screen. We need to create a simple field of array type


New Array

New Array

Name: ArrayExample

Optional? Yes [✓ No]

Array Element Type: [Simple Field]

Dimensions: Dimensions (0)

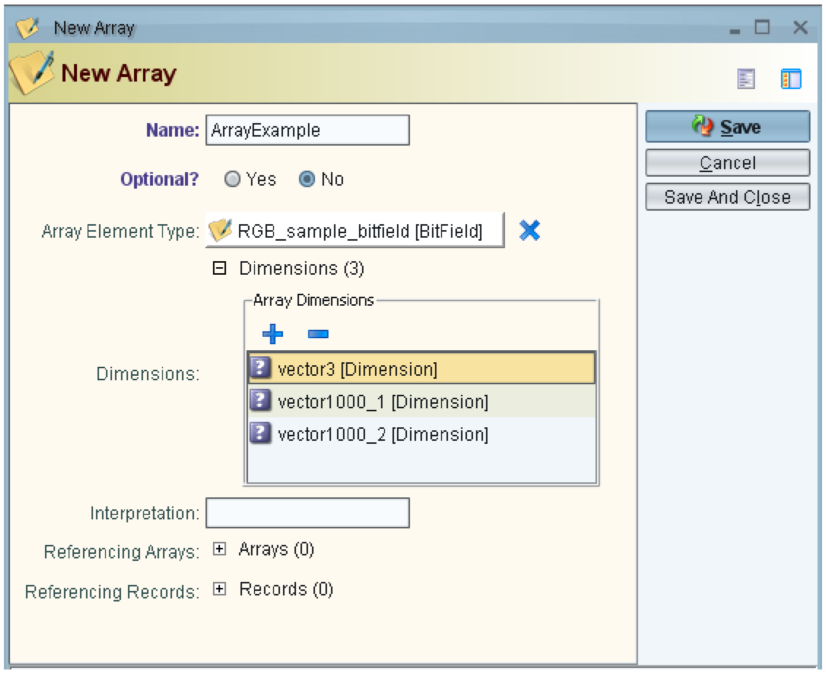
Interpretation:

Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 62: New Array Example Screenshot

We can next set the Array Element Type as a bit field which will hold our R,G,B values. Now we add the array dimensions. There are 4 dimensions that needed to be added because we are storing an array of size 3x1 in an array size 1000



New Array

Name: ArrayExample

Optional? Yes [✓ No]

Array Element Type: RGB_sample_bitfield [BitField]

Dimensions (3)

Dimensions:

Array Dimensions

vector3 [Dimension]
vector1000_1 [Dimension]
vector1000_2 [Dimension]

Interpretation:

Referencing Arrays: Arrays (0)

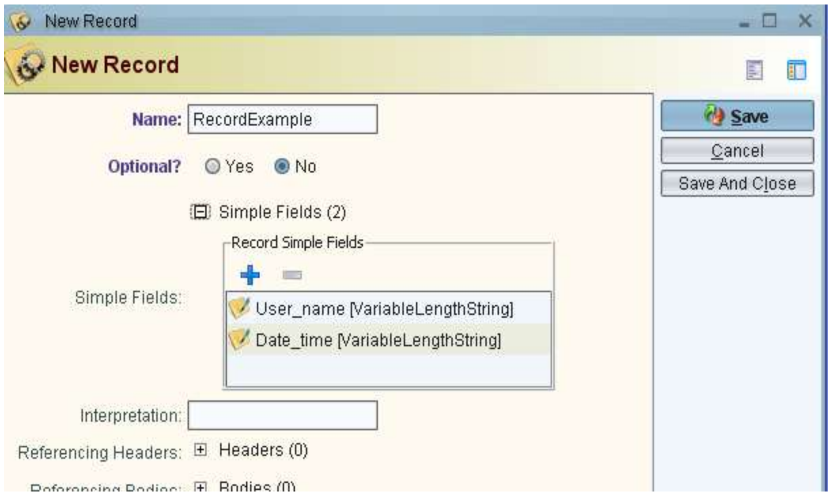
Referencing Records: Records (0)

Figure 63: New Array Populated Screenshot

10.3.2 Record Example

Continuing from the Array Example, we would now like to store this picture data in a record. However, we would also like to store when the picture was taken and by whom it was taken. This can be achieved by creating two Simple Fields of types Variable Length String that will hold the person's name and a date/time. We are also going to dictate that the order in which it is stored will be: Name, Date, Picture. This is purely by choice and not necessity.

First we create the two new Simple Fields:



Name: RecordExample

Optional? Yes [✓ No]

Simple Fields (2)

Simple Fields: Record Simple Fields

- User_name [VariableLengthString]
- Date_time [VariableLengthString]

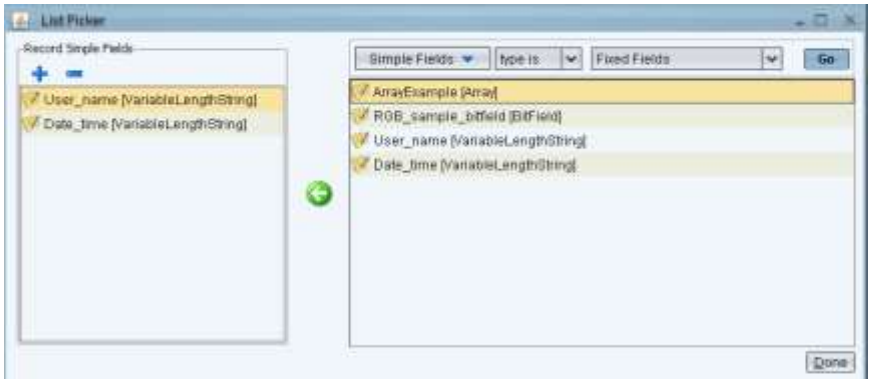
Interpretation:

Referencing Headers: Headers (0)

Referencing Bodies: Bodies (0)

Figure 64: New Record Simple Fields Screenshot

Then since we already created the array, we need to add the array by browsing to its definition and then adding it to the Record's simple fields



Record Simple Fields:

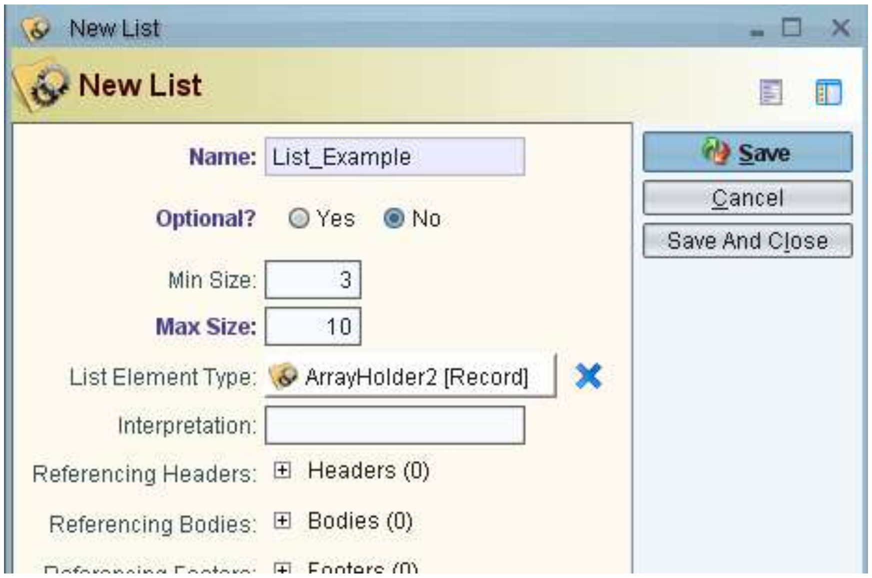
- User_name [VariableLengthString]
- Date_time [VariableLengthString]

- ArrayExample [Array]
- RGB_sample_bitfield [BitField]
- User_name [VariableLengthString]
- Date_time [VariableLengthString]

Figure 65: New Record Add Array Screenshot

10.3.3 List Example

Building upon the Record Example, we will now create a list of those picture records. We might need a list such as this if multiple pictures were being taken. For instance, if we wanted our list to be able to store anywhere between three and 10 records we would set the list as below. We also pick our previous example through the browse button for the "list element type".



Name: List_Example

Optional? Yes [✓ No]

Min Size: 3

Max Size: 10

List Element Type: ArrayHolder2 [Record]

Interpretation:

Referencing Headers: Headers (0)

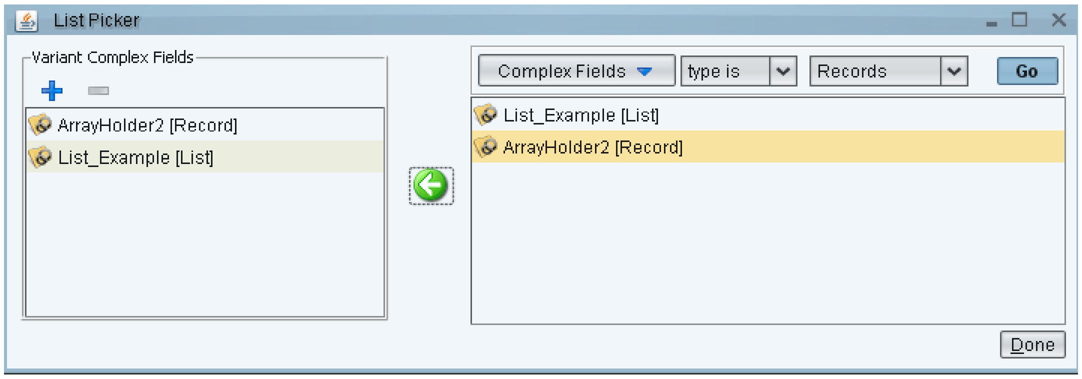
Referencing Bodies: Bodies (0)

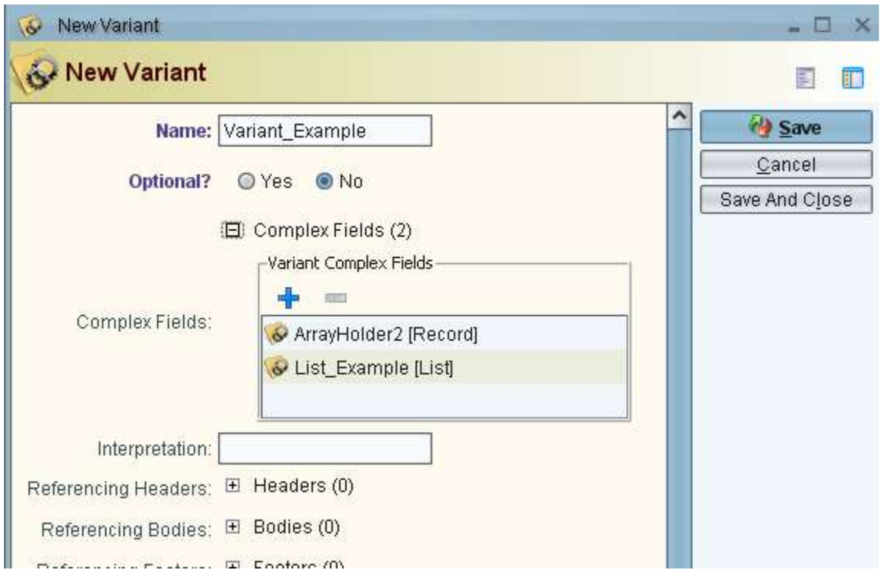
Referencing Footers: Footers (0)

Figure 66: List Example Screenshot

10.3.4 Variant Example

If we extend our Record Example and List Example, we can create a Variant that can hold either of these types depending on runtime declarations. This would be useful in a situation where we may only want to have a single picture record at runtime whereas other times we may want the list of pictures at runtime. Simply browse to the previously created items and pick them as member of the new variant.





Name: Variant_Example

Optional? Yes [✓ No]

Complex Fields: ArrayHolder2 [Record], List_Example [List]

Interpretation:

Referencing Headers: Headers (0)

Referencing Bodies: Bodies (0)

Figure 67: Variant Example Screenshot

10.3.5 Sequence Example

Creating sequences is the same as creating Variants. However, since they can take multiple values at runtime, they may be used to create complex data structures by holding multiple nested complex fields. We will create a simple example of a graph by representing its nodes and edges in a sequence

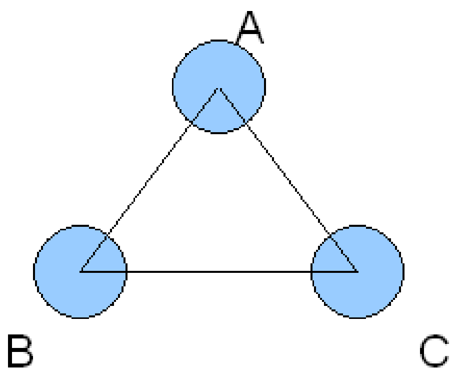
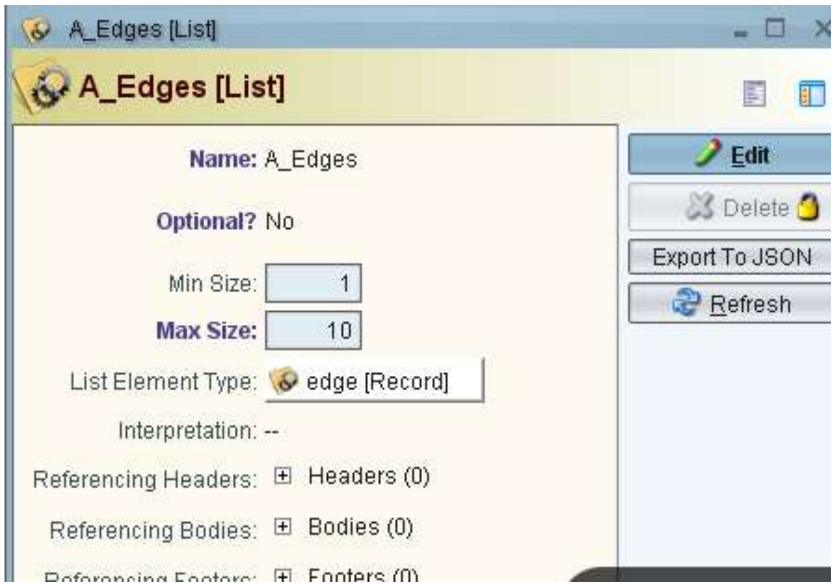


Figure 68: Sequence Example Graph

First we will need a list to store all of the edges for each node in the graph. Each element of the list will be a record with the edge name in it.



A_Edges [List]

Name: A_Edges

Optional? No

Min Size: 1

Max Size: 10

List Element Type: edge [Record]

Interpretation: --

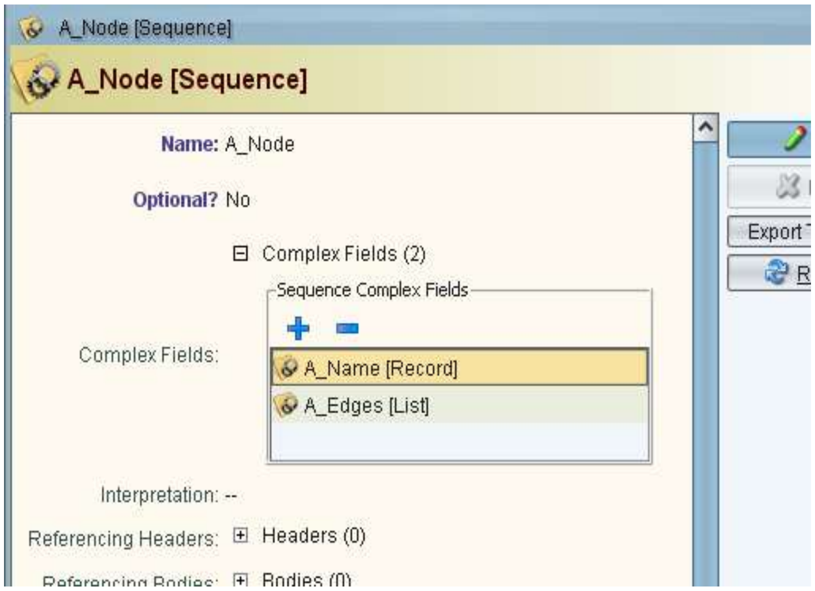
Referencing Headers: ⊞ Headers (0)

Referencing Bodies: ⊞ Bodies (0)

Referencing Footers: ⊞ Footers (0)

Figure 69: Sequence Example List Entry

Next we will need a sequence to store the node name and edges connected to each node:


A_Node [Sequence]

A_Node [Sequence]

Name: A_Node

Optional? No

Complex Fields (2)

Complex Fields: Sequence Complex Fields
- A_Name [Record]
- A_Edges [List]

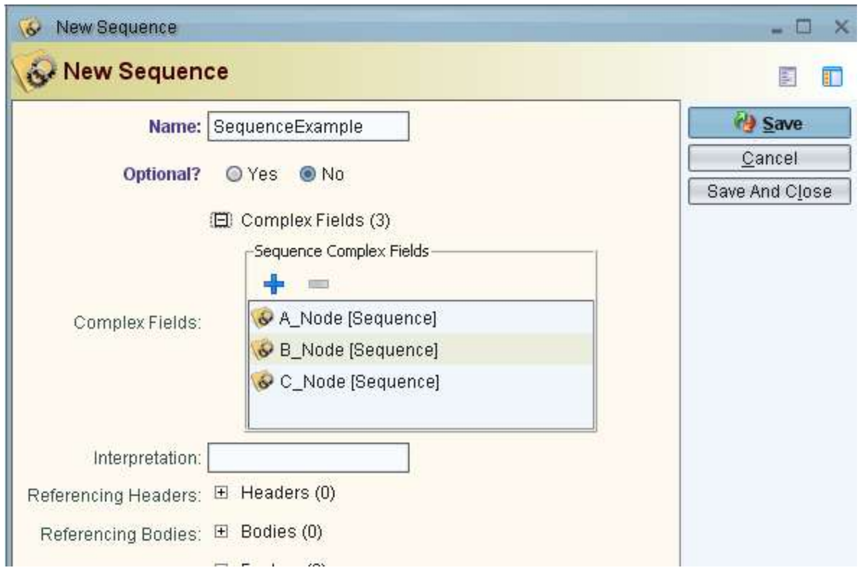
Interpretation: --

Referencing Headers: Headers (0)

Referencing Bodies: Bodies (0)

Figure 70: Sequence Example Node Entry

After replicating this procedure for all the other nodes of the graph, we will create a sequence with three sequences; one for each node of the graph. This will serve as a container for all the nodes of the graph.


New Sequence

New Sequence

Name: SequenceExample

Optional? ○ Yes [✓ No]

Complex Fields (3)

Complex Fields: Sequence Complex Fields
- A_Node [Sequence]
- B_Node [Sequence]
- C_Node [Sequence]

Interpretation:

Referencing Headers: Headers (0)

Referencing Bodies: Bodies (0)

Figure 71: Sequence Example Sequence Creation

Note that we are only defining the structure of the graph. Its contents will be described at runtime.

11 Defining Messages

In this section, the construction of message definitions is outlined with examples of their creation and functionality. Message Defs describe how data will be serialized so that it may be transferred over the network.

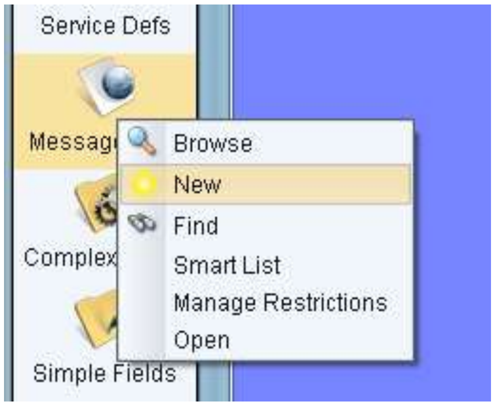
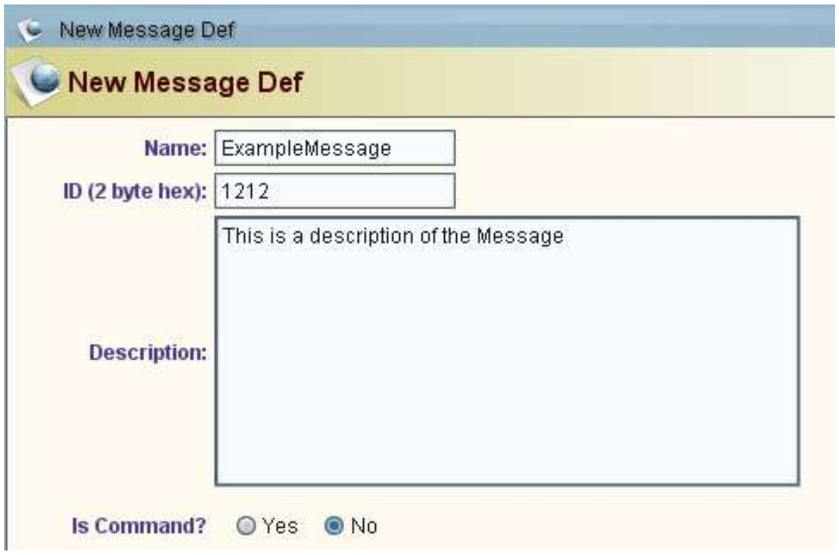


Figure 72: New Message Def Screenshot

- Message Defs need a name and a unique ID number which identifies the message. The name only needs to be unique within its parent context. The description box is optional and allows a user to describe the message for later reference.


New Message Def

New Message Def

Name: ExampleMessage

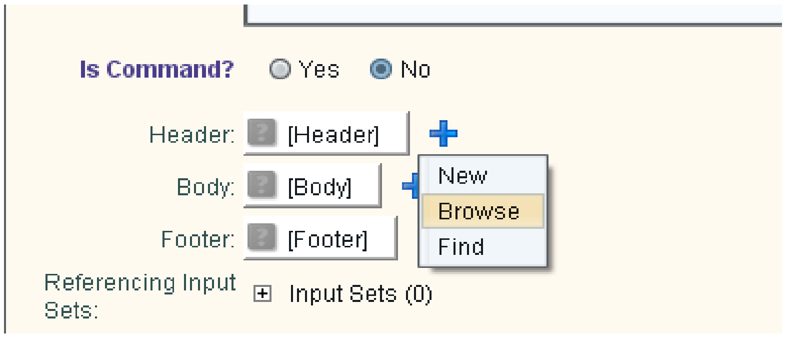
ID (2 byte hex): 1212

Description: This is a description of the Message

Is Command? Yes [✓ No]

Figure 73: New Message Def Populated Screenshot

- The next section of the Message Definition box is the header, body, and footer tabs. These you choose what information will be added to each section of the message. To add information, click on the plus sign and select New, Browse or Find.


Is Command? Yes [✓ No]

Header: [Header]

Body: [Body]

Footer: [Footer]

Referencing Input Sets: Input Sets (0)

Figure 74: New Message Def Add Attributes Screenshot

Select "Browse" (if a body already exists) or "new" to create a new body. An analogous approach can be taken for the message footer definitions.

Only complex fields may be placed in the header, body, and footer section of the message. If you only want to use a simple field, it must be wrapped in a complex field first. We will now continue on with a few examples of how to create some simple messages and then move to more complex messages.

11.1 Report Global Pose Message Example

//------ REPORT_GLOBAL_POSE_MESSAGE

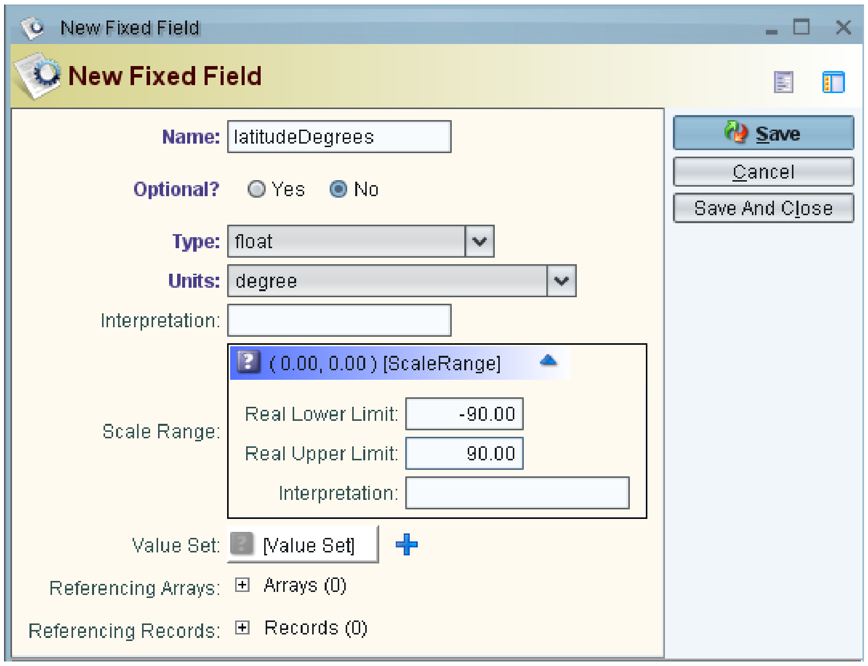
(basic)

The reportLocalPoseMessage is used by components to gather knowledge about the system's positions and orientation. It is composed from the following fields…

- latitudeDegrees - sent as a scaled int with range -90 to 90
- longitudeDegrees - sent as a scaled int with range -180 to 180
- elevationMeters - sent as a scaled int with range -10000 to 35000
- positionRmsMeters - sent as a scaled unsigned int with range 0 to 100
- rollRadians - sent as a scaled short with range -3.1415... to 3.1415...
- pitchRadians - sent as a scaled short with range -3.1415... to 3.1415...
- yawRadians - sent as a scaled short with range -3.1415... to 3.1415...
- attitudeRmsRadians - sent as a scaled short with range 0 to 3.1415...
- time - sent as a string

Note that several floating point values are represented as scaled integers. A scaled integer maps the integer range (for example, 0 – 65535 for an unsigned short) to the real data range. This allows for reduced message sizes during encoding, without scarifying precision. For additional information, please consult [AS5684].

We will first create new simple fields for each of these message elements. The most appropriate type of simple field to use is the fixed field. We choose a fixed field in this instance because we can attach units to the elements we need to define.



New Fixed Field

Name: latitudeDegrees

Optional? Yes [✓ No]

Type: float

Units: degree

Interpretation:

Scale Range: (0.00, 0.00) [ScaleRange]

Real Lower Limit: -90.00

Real Upper Limit: 90.00

Interpretation:

Value Set: [Value Set]

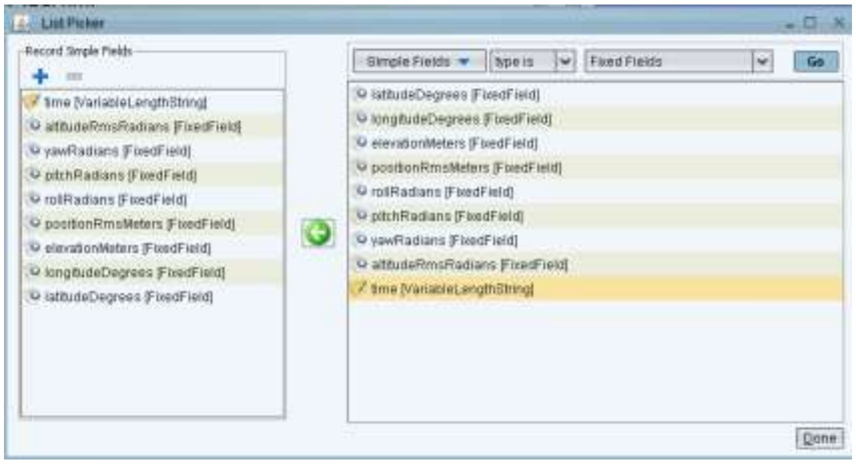
Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 75: Report Global Pose Field Example Screenshot

We create a new fixed field for each of the elements and set the range according to the referenced scale ranges for each element. We also create a variable length string for the time that the data was recorded.

After creating all of these simple fields, we will create a record that contains all of these elements. We must encapsulate the elements in a record because the message body must be a single complex field.



List Picker

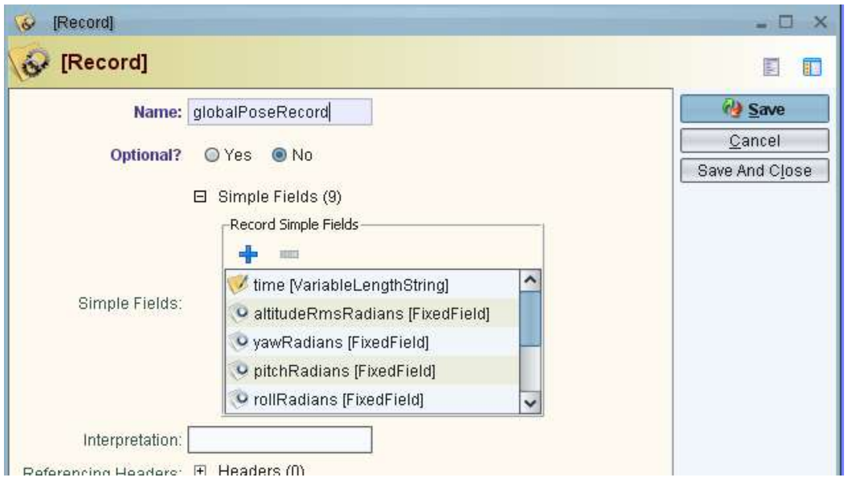
Record Simple Fields

time [VariableLengthString]
altitudeRmsRadians [FixedField]
yawRadians [FixedField]
pitchRadians [FixedField]
rollRadians [FixedField]
positionRmsMeters [FixedField]
elevationMeters [FixedField]
longitudeDegrees [FixedField]
latitudeDegrees [FixedField]

Simple Fields | type is | Fixed Fields

latitudeDegrees [FixedField]
longitudeDegrees [FixedField]
elevationMeters [FixedField]
positionRmsMeters [FixedField]
rollRadians [FixedField]
pitchRadians [FixedField]
yawRadians [FixedField]
altitudeRmsRadians [FixedField]
time [VariableLengthString]

Figure 76: Report Global Pose Record List Picker Example Screenshot



Name: globalPoseRecord

Optional? Yes [✓ No]

Simple Fields (9)

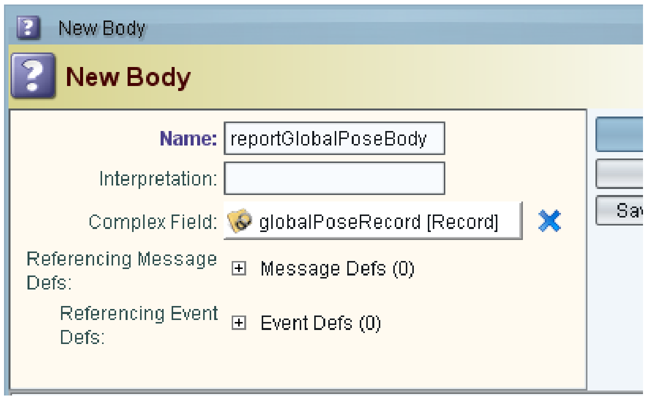
Simple Fields:
- time [VariableLengthString]
- altitudeRmsRadians [FixedField]
- yawRadians [FixedField]
- pitchRadians [FixedField]
- rollRadians [FixedField]

Interpretation:

Referencing Headers: Headers (0)

Figure 77: Report Global Pose Record Creation Example Screenshot

Now we can create the message with this record as its body



Name: reportGlobalPoseBody

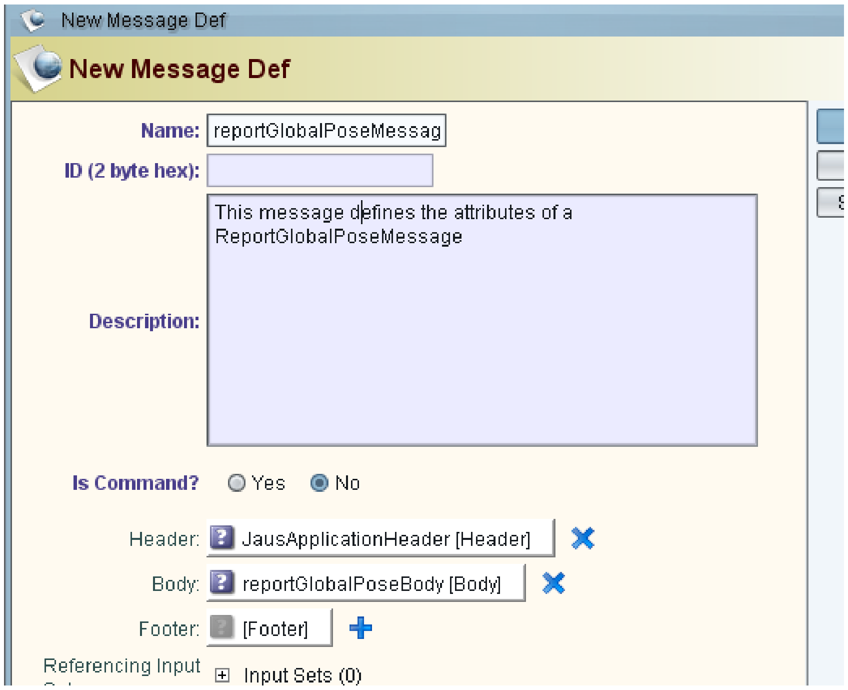
Interpretation:

Complex Field: globalPoseRecord [Record]

Referencing Message Defs: Message Defs (0)

Referencing Event Defs: Event Defs (0)

Figure 78: Report Global Pose New Body Screenshot


New Message Def

Name: reportGlobalPoseMessag

ID (2 byte hex):

Description: This message defines the attributes of a ReportGlobalPoseMessage

Is Command? Yes [✓ No]

Header: JausApplicationHeader [Header]

Body: reportGlobalPoseBody [Body]

Footer: [Footer]

Referencing Input Sets: Input Sets (0)

Figure 79: Report Global Pose Message Def Complete Screenshot

11.2 Report Image Message Example

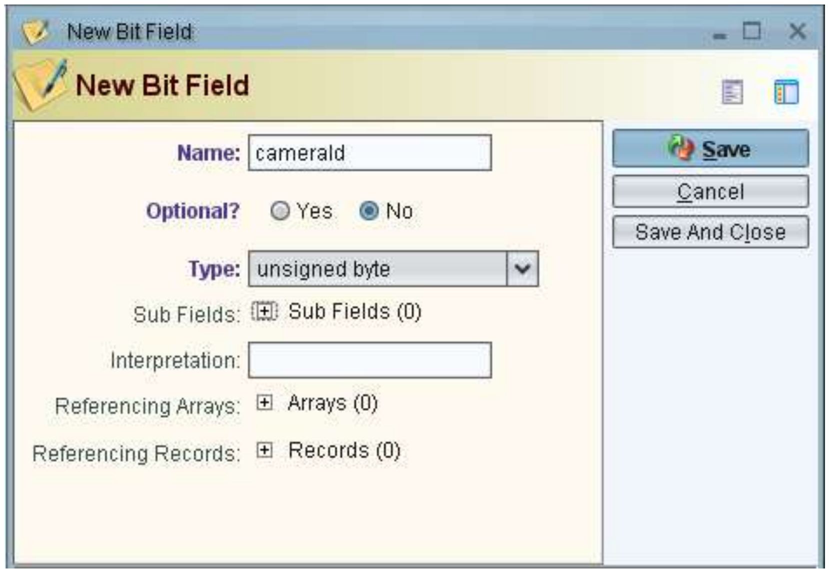
//----- REPORT_IMAGE_MESSAGE

(includes variable size array pointer)

The reportImageMessage is a JAUS message that allows for the transport of a variable sized image across the system. For this message, will we use several fixed fields to store message information and a variable length field to store the picture. The message needs to contain the following elements.

- CameraID: the unique id of the camera that took the picture
- videoFormat: the type of encoding that the image has
- data: the actual pointer in which the picture is stored
- bufferSizeBytes: the amount of bytes that the data pointer holds

We first create all the simple fields



Name: camerald

Optional? Yes [✓ No]

Type: unsigned byte

Sub Fields: Sub Fields (0)

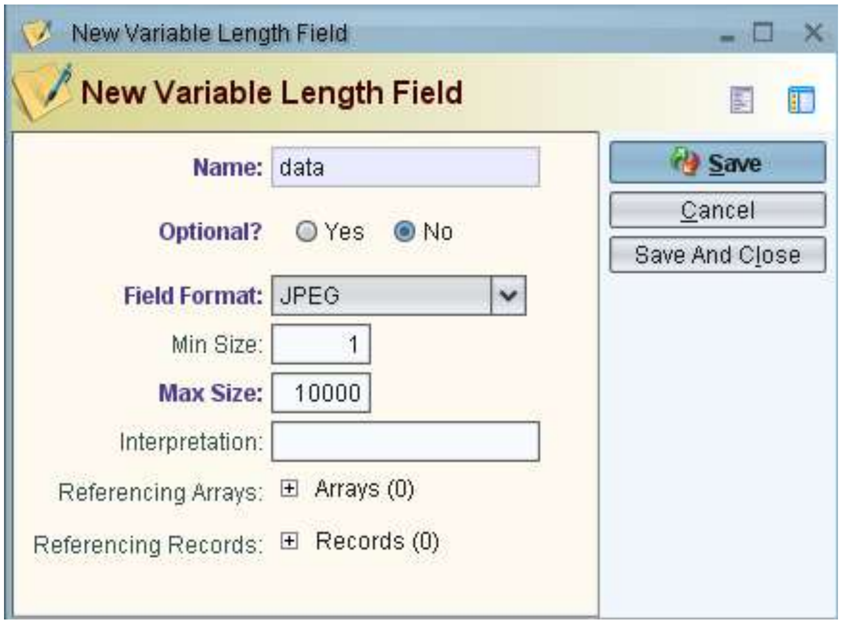
Interpretation:

Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 80: Report Image Example Fields Screenshot

We need to create the picture portion of the message. We will choose a variable length field to do this. The variable Length Field element contains the rest of the information that needs to be transferred in the message so we are now done defining all of the elements of our message.


New Variable Length Field

Name: data

Optional? Yes [✓ No]

Field Format: JPEG

Min Size: 1

Max Size: 10000

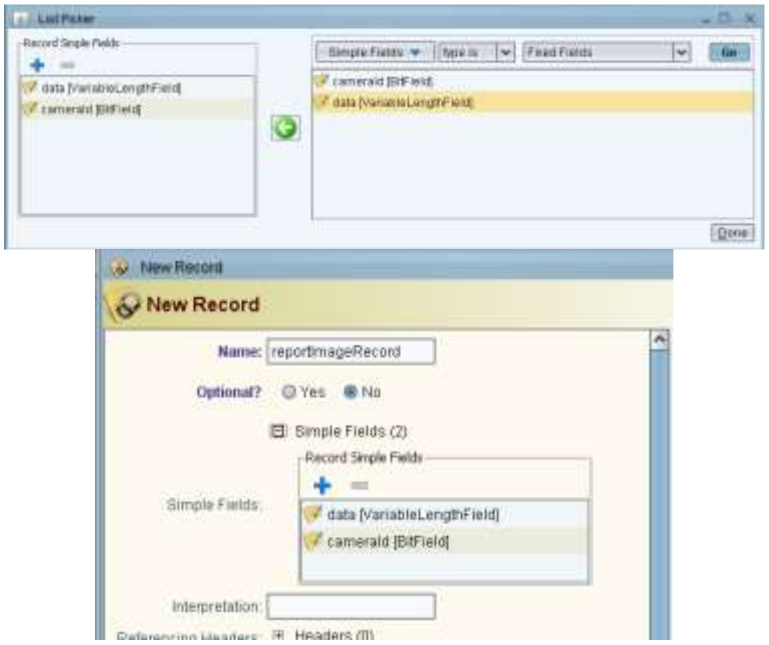
Interpretation:

Referencing Arrays: Arrays (0)

Referencing Records: Records (0)

Figure 81: Report Image Variable Length Field Screenshot

After creating all of these simple fields, we will create a record that contains all of these elements. We must encapsulate the elements in a record because the message body must be a single complex field.


List Picker

Record Simple Fields:
- data [VariableLengthField]
- cameraId [BitField]

Simple Fields | type is | Fixed Fields

- cameraId [BitField]
- data [VariableLengthField]

New Record

Name: reportImageRecord

Optional? Yes [✓ No]

Simple Fields (2)

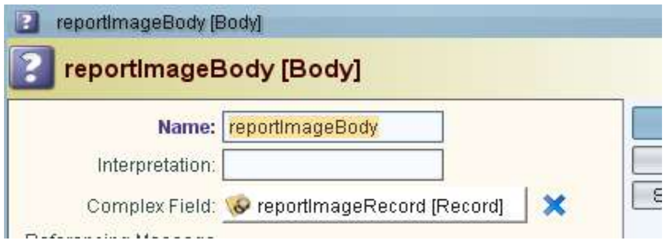
Simple Fields: Record Simple Fields
- data [VariableLengthField]
- cameraId [BitField]

Interpretation:

Referencing Headers: Headers (0)

Figure 82: Report Image Record Creation Screenshot

Now we can create the message with this record as its body



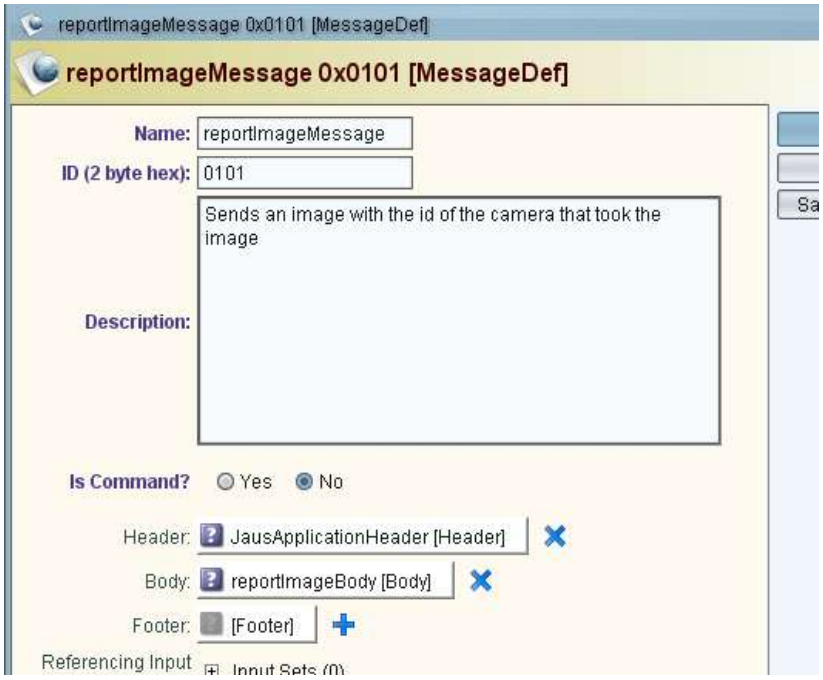
reportImageBody [Body]

reportImageBody [Body]

Name: reportImageBody

Interpretation:

Complex Field: reportImageRecord [Record]



reportImageMessage 0x0101 [MessageDef]

reportImageMessage 0x0101 [MessageDef]

Name: reportImageMessage

ID (2 byte hex): 0101

Description: Sends an image with the id of the camera that took the image

Is Command? Yes [✓ No]

Header: JausApplicationHeader [Header]

Body: reportImageBody [Body]

Footer: [Footer]

Referencing Input: Input Sets (0)

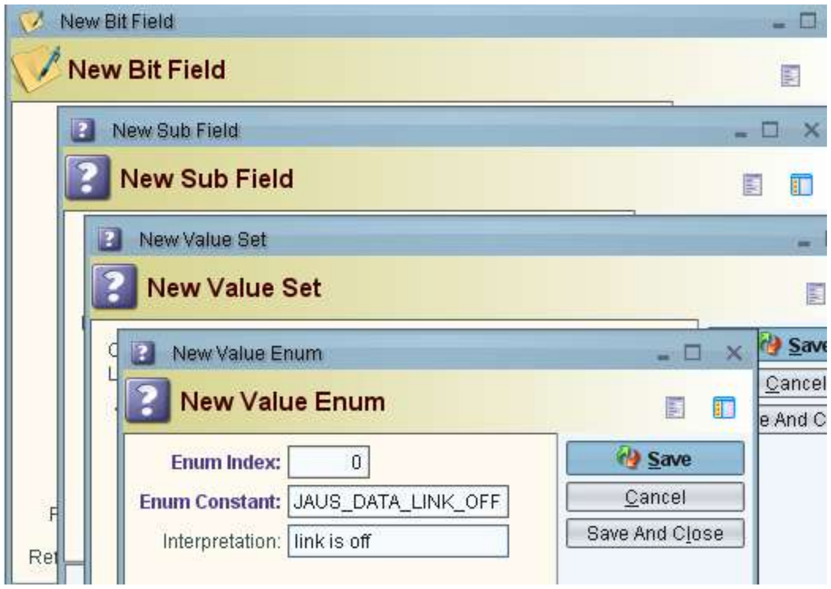
Figure 83: Report Image Message Creation Example Screenshot

11.3 Report Data Link Status Message Example

The reportDataLinkStatusMessage holds information about the status of a data link. The elements of the message are as follows

- dataLinkId: the unique id of the data link
- dataLinkState: the state of the link. The link can be either OFF, ON or STANDBY

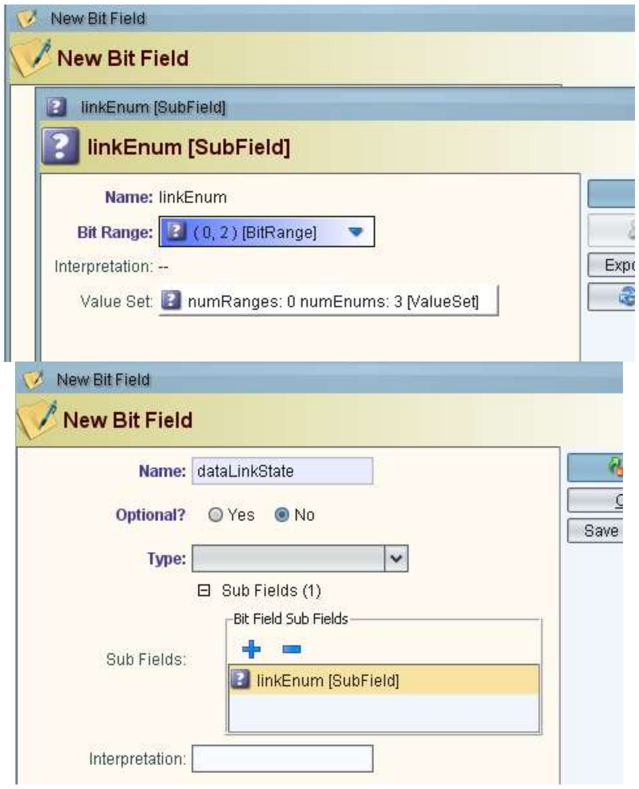
Since the dataLinkState can only hold one of three values, we will use a bit field with a value set that has 3 enumerated values.



Enum Index: 0
Enum Constant: JAUS_DATA_LINK_OFF
Interpretation: link is off

Figure 84: Report Data Link Bit Field Creation Screenshot

We need to create two more enumerated values for this set.

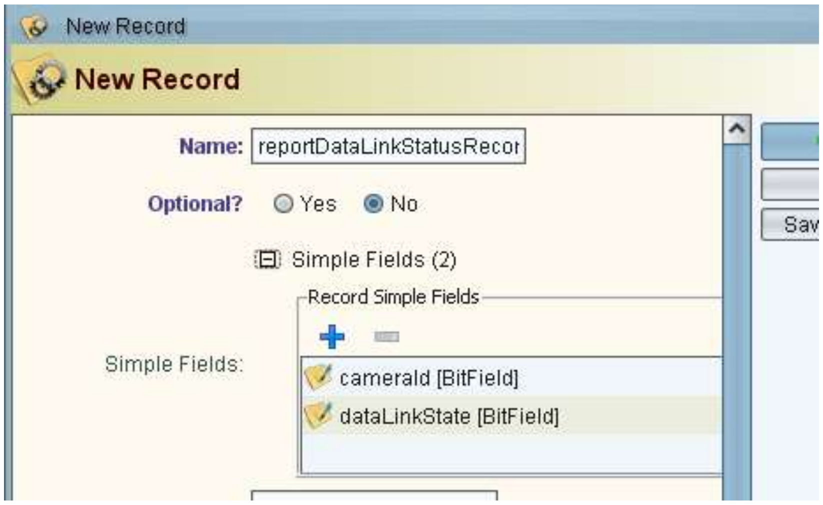


Name: linkEnum
Bit Range: (0, 2) [BitRange]
Interpretation: --
Value Set: numRanges: 0 numEnums: 3 [ValueSet]

Name: dataLinkState
Optional? Yes [✓ No]
Type:
Sub Fields: linkEnum [SubField]
Interpretation:

Figure 85: Report Data Link Enumerated Values Creation Screenshot

The dataLinkId can be defined as another bit field. We then need to encapsulate these simple fields into a record which can be set as the body of our message.


New Record

New Record

Name: reportDataLinkStatusRecor

Optional? Yes [✓ No]

Simple Fields (2)

Record Simple Fields

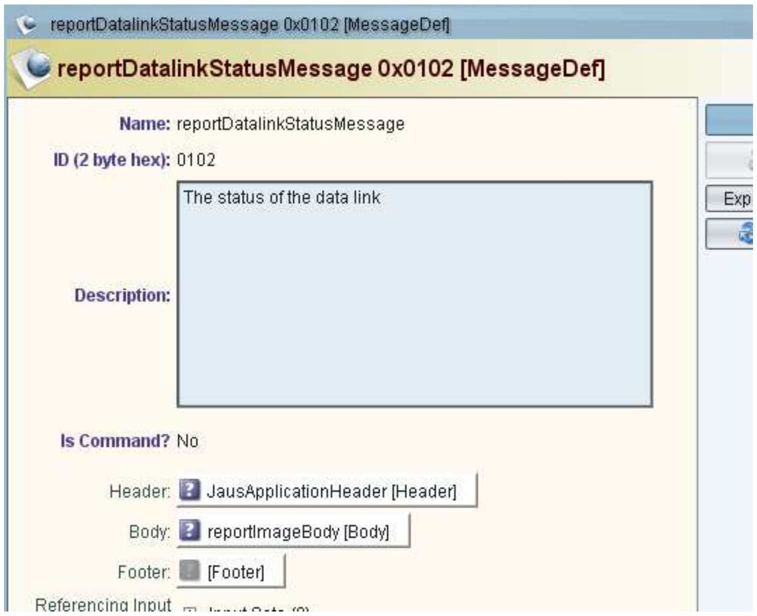
Simple Fields:

cameraId [BitField]

dataLinkState [BitField]

Figure 86: Report Data Link New Record Creation Screenshot

We can now create our message with this record as the message's body.


reportDatalinkStatusMessage 0x0102 [MessageDef]

reportDatalinkStatusMessage 0x0102 [MessageDef]

Name: reportDatalinkStatusMessage

ID (2 byte hex): 0102

Description: The status of the data link

Is Command? No

Header: JausApplicationHeader [Header]

Body: reportImageBody [Body]

Footer: [Footer]

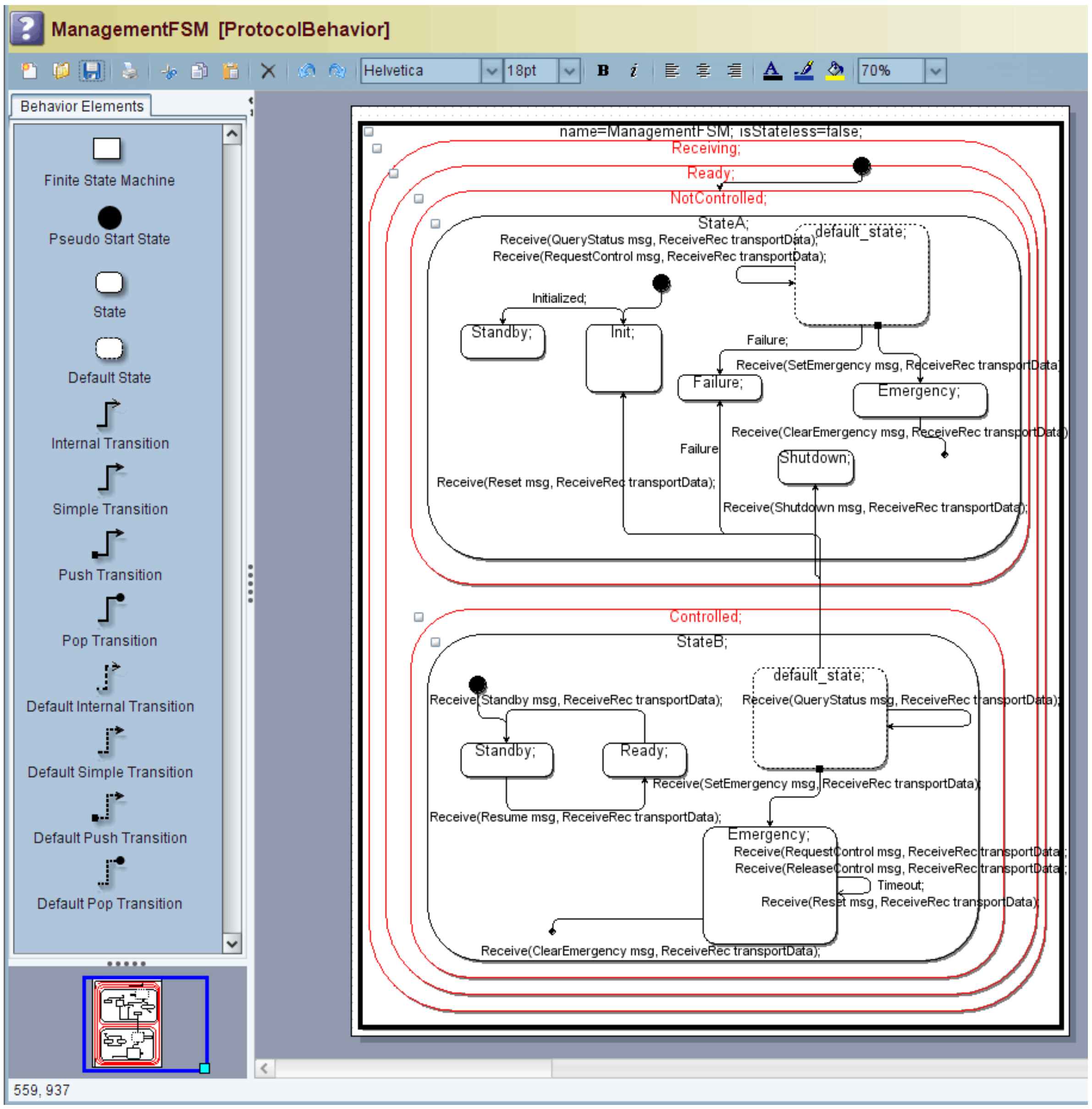
Figure 87: Report Data Link New Message Creation Screenshot

12 Defining Protocol Behavior

The protocol behavior of a service is defined using one or more concurrent finite state machines as shown below. JTS provides a GUI editor for state machines. This user interface may be opened by double clicking on an existing protocol behavior entry in the database, or by right clicking on the Protocol Behavior icon in the main application window (or Service Def window) and selecting “New” in the pop up menu.

It is recommended that the user read the AS-5684 section titled "Protocol Behavior".

The behavior elements are listed in the palette to the left. A single printable letter size page is provided for drawing out the state machine. The elements on the palette may be dragged and dropped onto the page when drawing out the state machines. Section 10.1 provides brief descriptions for each of these elements. Refer to the drawing tips in Section 10.2 and 10.4 to learn how to draw and format each element on the page. The states that are colored red are inherited states. Section 10.3 provides a brief description of state machine inheritance. The menu bar at the top provides basic menu items for creating, editing and saving the protocol behavior. The inset view at the bottom left may be used to zoom and pan across the page.



ManagementFSM [ProtocolBehavior]
Helvetica
18pt
70%
Behavior Elements
Finite State Machine
Pseudo Start State
State
Default State
Internal Transition
Simple Transition
Push Transition
Pop Transition
Default Internal Transition
Default Simple Transition
Default Push Transition
Default Pop Transition
559, 937
name=ManagementFSM; isStateless=false;
Receiving;
Ready;
NotControlled;
StateA;
default_state;
Receive(QueryStatus msg, ReceiveRec transportData);
Receive(RequestControl msg, ReceiveRec transportData);
Initialized;
Standby;
Init;
Failure;
Receive(SetEmergency msg, ReceiveRec transportData)
Failure;
Emergency;
Receive(ClearEmergency msg, ReceiveRec transportData)
Failure
Shutdown;
Receive(Reset msg, ReceiveRec transportData);
Receive(Shutdown msg, ReceiveRec transportData);
Controlled;
StateB;
default_state;
Receive(Standby msg, ReceiveRec transportData);
Receive(QueryStatus msg, ReceiveRec transportData);
Standby;
Ready;
Receive(SetEmergency msg, ReceiveRec transportData)
Receive(Resume msg, ReceiveRec transportData);
Emergency;
Receive(RequestControl msg, ReceiveRec transportData);
Receive(ReleaseControl msg, ReceiveRec transportData);
Timeout;
Receive(Reset msg, ReceiveRec transportData);
Receive(ClearEmergency msg, ReceiveRec transportData);

Figure 88: Protocol Behavior User Interface

12.1 Behavior Elements

- **Finite State Machine:** The finite state machine element represents a single state machine and is a simple container element that is used to draw one or more state machines. The attributes of this element are **name** and **isStateless**. The isStateless attribute must be set to true only if the service does not store data received from its clients during the course of its execution. The syntax for specifying these attributes is,

- name = *stateMachineName*; isStateless = *true OR false*;

```
example: name=management; isStateless=false;
```

- **Pseudo Start State:** A pseudo start state represents a default vertex that is the source for a simple transition to the initial state of a set of nested states. There must be one and only one pseudo start state per set of nested states. The transition used to point to the initial state must be a simple transition.
- **State:** The state element represents a single state whose name, entry and exit actions may be specified using the following syntax.
  - *name*; entry [OR exit]: *action1 (arg1,...);*...

```
example:

Init;
entry: initialize();
exit: beginTimer();
```

- **Default State:** Each state may specify at most one special nested state called the default_state. Like other states, the default state may be the source of transitions, but not a destination for transitions. If a state receives an event for which it has no transition defined, the state's sibling default state's transitions are evaluated. If the sibling default state specifies a transition for the event, then that transition is executed. If the transition is a self-transition, then the state machine re-enters the sibling state once the transition execution is complete. Entry and Exit actions cannot be defined for a default state.
- **Internal Transition:** An internal transition is always a self transition. The key difference between an internal transition and a simple transition is that the exit and entry actions of the source state of the internal transition are not executed when the internal transition is invoked and completes execution. In essence, the source state is never exited during transition execution. Parameters, guard conditions and actions may be defined for internal transitions using the syntax below. A single internal transition arrow may be used to represent internal transitions (separated by semi-colons) for one or more events (or triggers). The syntax for the transition can be brought up by invoking the transition's tooltip (place the mouse on the transition icon

in the Behavior Elements panel). New users may find this syntax hard to remember but the auto-completion guide (see Section 12.2) virtually eliminates the need to remember this syntax.

- *Trigger (paramType value, ...)[guard]/ action (arg1,...),...;...*

```
example:

AccessControl.Events.Transport.Receive( RequestControl msg, Re-
ceive.Body.ReceiveRec transportData)/sendConfirmControl('NOT_AVAILABLE',
transportData);
AccessControl.Events.Transport.Receive( ReleaseControl msg, Re-
ceive.Body.ReceiveRec transportData)/sendRejectControl('NOT_AVAILABLE',
transportData);
```

- **Simple Transition:** A simple transition is similar to an internal transition except that the source and destination states of the transition do not have to be the same, and the source state is exited when the transition is invoked. This implies that the source and destination states' exit and entry actions are called when the transition is invoked and its execution is completed respectively. The syntax for specifying the transition's trigger, parameters, guard conditions and actions is similar to that of internal transitions above.

- **Push Transition:** Push transitions are used to transition from the source state to the destination state such that the destination 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. The exit actions of the source state of the push transition are not executed when the transition is invoked. But, the entry actions of the destination state of the push transition are executed once the transition has completed its execution. The syntax for specifying a push transition is as shown below. A single push transition arrow may be used to represent push transitions (separated by semi-colons) for one or more events (or triggers). A push transition may have an additional end state defined within it. When specified, a simple transition is executed first to this end state, immediately after which the actual push transition is executed. 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 a different end state than the parent state of the push transition. The syntax for the transition can be brought up by invoking the transition’s tooltip (place the mouse on the transition icon in the Behavior Elements panel). New users may find this syntax hard to remember but the auto-completion guide (see Section 12.2) virtually eliminates the need to remember this syntax.

- *trigger (paramType value, ...)[guard]/ action (arg1,...),...{end_state};...*

-

```
example:

AccessControl.Events.Transport.Receive( SetEmergency msg ) / storeId(
msg ) ;
```

- **Pop Transition:** The pop transition is used to transition out of a state into which the state machine was pushed using a push transition. Pop transitions must be defined on states that are destination states of push transitions. The exit actions of the source state of the pop transition are executed when the transition is invoked. But, the entry actions of the destination state of the pop transition are not executed once the transition has completed its execution. The syntax of the pop transition is as defined below. A single pop transition arrow may be used to represent pop transitions (separated by semi-colons) for one or more events (or triggers). An optional secondary simple transition may be defined within a pop transition. This transition is executed once a pop transition has been issued for every corresponding push transition. In the syntax below, *end_transition* stands for the name of a secondary 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 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. The syntax for the transition can be brought up by invoking the transition’s tooltip (place the mouse on the transition icon in the Behavior Elements panel). New users may find this syntax hard to remember but the auto-completion guide (see Section 12.2) virtually eliminates the need to remember this syntax.

- *trigger(paramType value, ...)[guard]/action(arg1,...),...{end_transition(arg1,...};...*

```
example:
```

```
AccessControl.Events.Transport.Receive( ClearEmergency msg ) [isID-
Stored( msg )]/deleteID( msg );
```

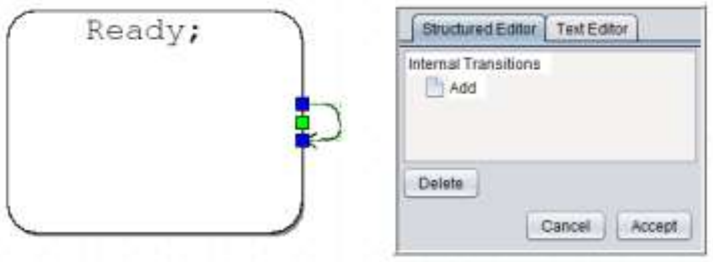
- **Default Internal Transition:** If a state receives an event for which it has no transition defined, the default internal transition is executed if it is defined. Since any transition can fall through to a default transition, the default internal transition has no trigger or parameter list specified. Aside from this difference, a default internal transition is like an internal transition. The syntax for the transition can be brought up by invoking the transition's tooltip (place the mouse on the transition icon in the Behavior Elements panel). New users may find this syntax hard to remember but the auto-completion guide (see Section 12.2) virtually eliminates the need to remember this syntax.
  - [*guard*]/ *action (arg1,...),...;...*
- **Default Simple Transition:** If a state receives an event for which it has no transition defined, the default simple transition is executed if it is defined. Since any transition can fall through to a default transition, the default simple transition has no trigger or parameter list specified. Aside from this difference, a default simple transition is like a simple transition.
- **Default Push Transition:** If a state receives an event for which it has no transition defined, the default push transition is executed if it is defined. Since any transition can fall through to a default transition, the default push transition has no trigger or parameter list specified. Aside from this difference, a default push transition is like a push transition. The syntax for the transition can be brought up by invoking the transition's tooltip (place the mouse on the transition icon in the Behavior Elements panel). New users may find this syntax hard to remember but the auto-completion guide (see Section 12.2) virtually eliminates the need to remember this syntax.
  - *[guard]/ action (arg1,...),...{end_state};...*
- **Default Pop Transition:** If a state receives an event for which it has no transition defined, the default pop transition is executed if it is defined. Since any transition can fall through to a default transition, the default pop transition has no trigger or parameter list specified. Aside from this difference, a default pop transition is like a pop transition. The syntax for the transition can be brought up by invoking the

transition's tooltip (place the mouse on the transition icon in the Behavior Elements panel). New users may find this syntax hard to remember but the auto-completion guide (see Section 12.2) virtually eliminates the need to remember this syntax.

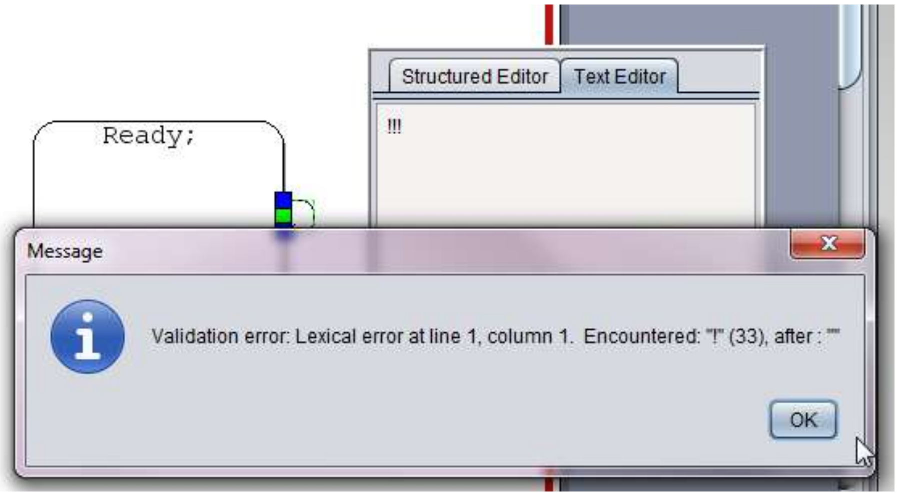
- *[guard]/action(arg1,...),...{end_transition(arg1,...};....*

12.2 Behavior Element Definition Editing

JTS allows protocol behavior element definitions to be changed in a text editor using a specific syntax or within a structured editor which maintains rigid constraints imposed by a tree structure representation of the definition. When an element is opened for editing, a panel will be displayed with the two editors along with an Accept and Cancel buttons.

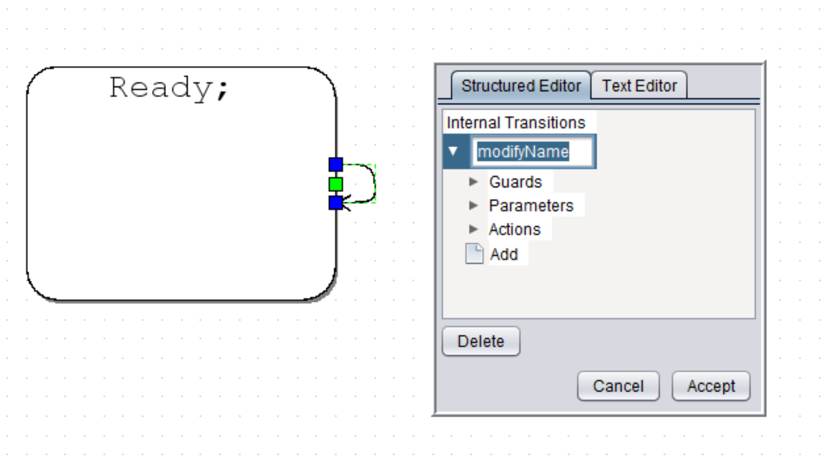


Pressing the accept button while focused on either the structured editor or the text editor will call validation methods before saving the definitions stored in the currently viewable editor. All definitions in the non-selected editor will be discarded after pressing the accept button. Pressing the cancel button will discard any changes made in both of the editors.

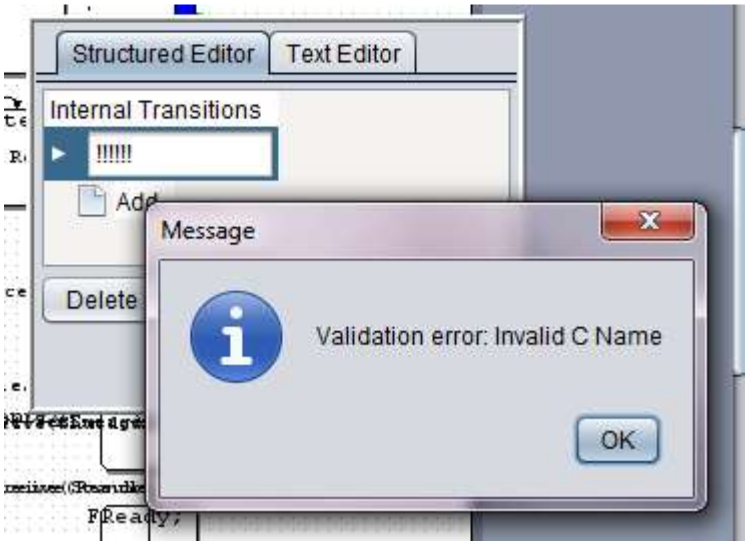


Validation error: Lexical error at line 1, column 1. Encountered: "!" (33), after : ""

To edit a value in the structured editor, highlight the item using the keyboard arrows or mouse and press F2. The tree item will then turn into a text field allowing the definition to be typed. To set the edited value in the tree, press enter. Note that setting the value in the tree does not yet save the definition beyond the scope of the structured editor.



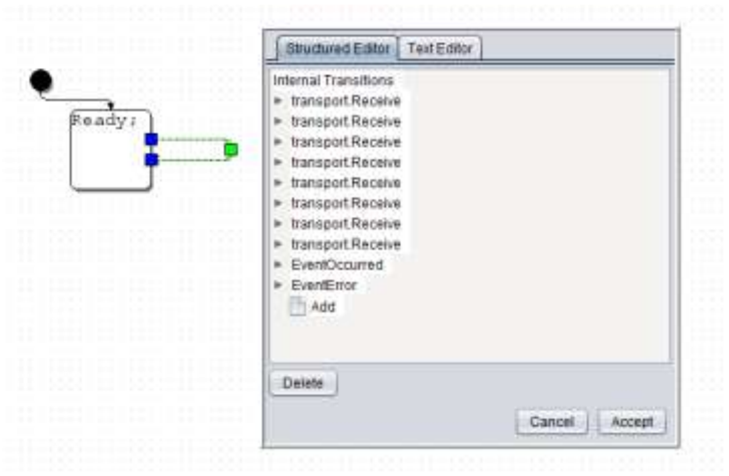
Within the structured editor, simplistic validation of field values is done before the field can be set. For example, a trigger name must be a valid C identifier so if pressing enter while editing a field with an invalid identifier, a popup will display the validation error for the current editing field. To cancel editing of a tree item, press the escape button.



Validation error: Invalid C Name

The structured editor also contains a delete button so that items from the tree can be deleted from the definition. Only tree items that can be removed from the tree are deleted when the delete button is pressed. To remove from the tree, select the item by navigating with the cursor or keyboard. When the desired item is highlighted use either the keyboard delete button or the button within the window to remove the highlight item from the tree. There is no undo feature but the cancel button will allow you to back out of editing without saving the definition so that it can be reopened in the event of any mistake.

Within the structured editor, some items of the tree contain an Add button. When this tree item is clicked with the cursor, or enter is pressed when highlighted, a new tree item is added corresponding to the list at the level of the Add button. This is how multiple triggers, actions, parameters etc are able to be added within the structured editor.



The structured editor and text editor may be moved within the page by dragging with the cursor. It may also be resized by dragging the corners of the window.

12.3 Auto-completion Guide

The Auto-completion Guide was designed to encourage reuse of previously defined definitions while designing new protocol behavior definitions. It will search through all service definitions linked to a protocol behavior and make appropriate suggestions about what can be included in a trigger definition while it is being edited.

To access the Auto-completion tool within the structured or text editor, simply press the Ctrl button once while editing a definition To turn off suggestions, press the Ctrl key again or the Esc key. The drop down will disappear after a selection is madebut will pop up again whenever the text area is refreshed by moving the position of the caret.

Figure 89: Auto-Complete Example after typing 'ev

Adrop down with the title of the section of the trigger being edited will appear followed by suggestions about what to add next. No validation of the edited trigger occurs until after the trigger is accepted so the Auto-completion will only suggest based on the information available and may make incorrect/invalid suggestions based on current information. The suggestions are case sensitive so typing 'Ev' instead of'ev' in the above example will yield no matching suggestions in the drop down.

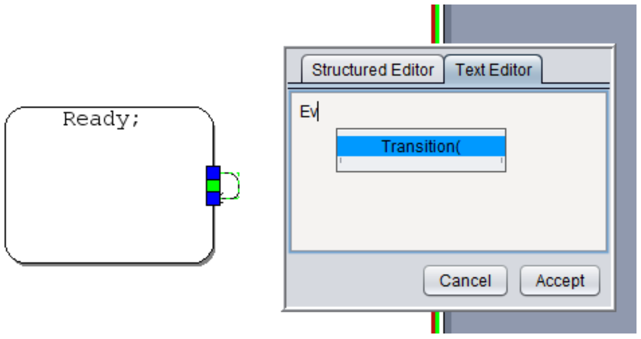


Figure 90: Case-Sensitive Auto-Completion

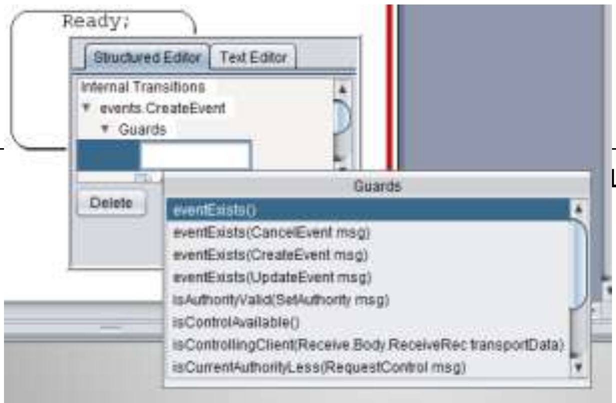
The Auto-completion tool will dynamically update suggestions based on what is typed while editing a trigger. To use one of the suggested items in the drop down, select the item with the cursor or use the up/down arrow and enter key when the desired item is highlighted.

Figure 91: Selection of Auto-Completion Suggestion

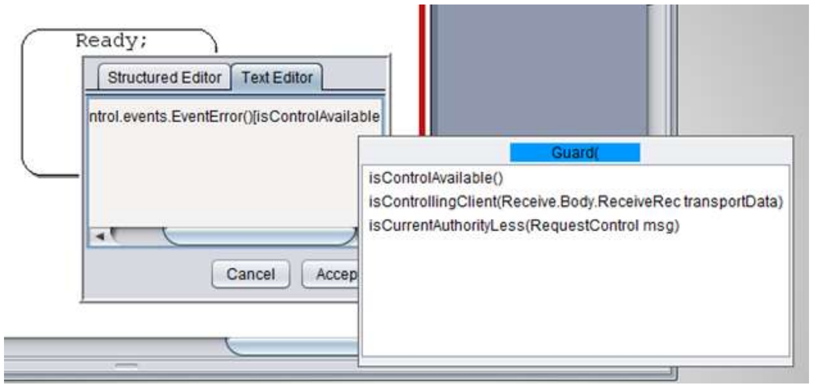
The five possible sections are Transition, Parameter, Argument, Guard and Action. The Argument type is included when editing both the Guard type and the Action type. Each of the items will show up after the appropriate separation character is added to the trigger definition.

Figure 92: Auto-Completion of a Guard Condition

The Auto-completion tool supports inline editing of text definitions which can be trigger by moving the caret to whichever portion of the trigger needs to be edited. Suggestions will be made based on a best guess depending on the information available and a replacement made for the estimated portion of the trigger being edited. Again, validation is only done after the


Ready;
Structured Editor
Text Editor
Internal Transitions
events.CreateEvent
Guards
Delete
Guards
eventExists()
eventExists(CancelEvent msg)
eventExists(CreateEvent msg)
eventExists(UpdateEvent msg)
isAuthorityValid(SetAuthority msg)
isControlAvailable()
isControllingClient(Receive.Body.ReceiveRec.transportData)
isCurrentAuthorityLess(RequestControl msg)

trigger is deselected so if there is a problem with the trigger syntax, guesses may be made within the Auto-completion which may have undesirable consequences.

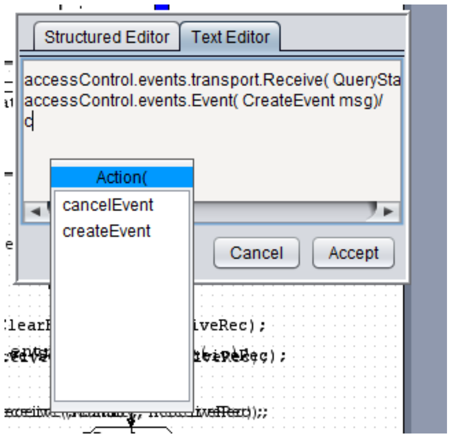


```
ntrol.events.EventError()[isControlAvailable
```

```
isControlAvailable()
isControllingClient(Receive.Body.ReceiveRec transportData)
isCurrentAuthorityLess(RequestControl msg)
```

Figure 93: Resolving a Guard Condition

New lines and tabs are supported. The Auto-completion drop down will follow the position of the caret in the textbox so as you type, the drop down position will continually change with each change in the position of the caret.

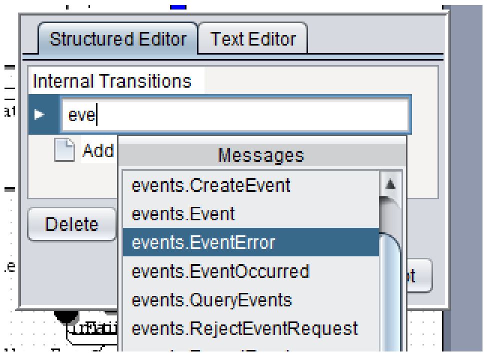


```
accessControl.events.transport.Receive( QuerySta
accessControl.events.Event( CreateEvent msg)/
c
```

```
cancelEvent
createEvent
```

Figure 94: Auto-Completion of an Action

Transition names are based on message and internal event names. If inheritance is used in the service definition, the messages will be namespaced by using their immediate parent name in the Auto-completion drop down. After the message name is chosen, Auto-completion will fill in the entire namespaced message name into the trigger text area.



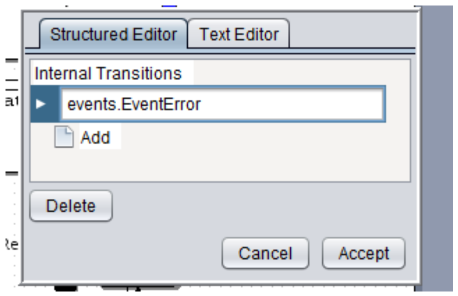
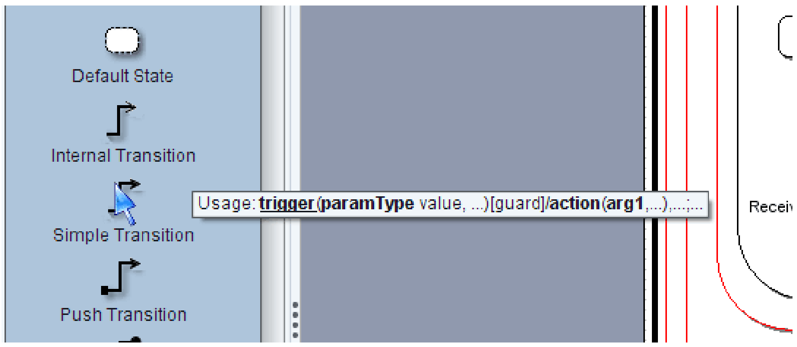


Figure 95: Auto-Completion Results

12.4 Drawing Tips

- **Page Size Constraint:** A single page has been provided for a reason. The constraint has been imposed to discourage the user from making the protocol behavior too complex. If complex behavior needs to be specified, it may be better to break up the behavior into sub-behaviors using inheritance, or by identifying orthogonal concerns in the complex behavior and separating these concerns into two or more interacting services.
- **Graph Movement:** The entire editing space can be moved by holding down the center mouse scroll wheel, Scrolling with the scroll wheel zooms in and out within the editing window.
- **Tooltips:** Since the syntax required to define state machines, states and transitions is hard to remember, tooltips have been provided to show the syntax. To view the syntax, place the mouse cursor over a behavior element in the palette or on the page for two seconds, and the tooltip will appear as shown in the figures below.

- 

Usage: trigger(paramType value, ...)[guard]/action(arg1,...),...;...

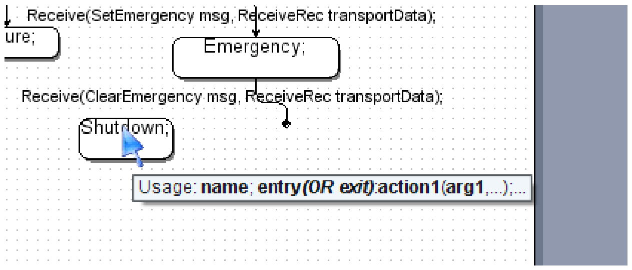
- 
Receive(SetEmergency msg, ReceiveRec transportData);
ure;
Emergency;
Receive(ClearEmergency msg, ReceiveRec transportData);
Shutdown;
Usage: name; entry(OR exit):action1(arg1,...);...

Figure 96: Tooltips

- **Moving and resizing:** The page view contains control features that allow the user to move and resize behavior objects on the page. To move a behavior object, place the mouse cursor on the object until a green outline forms around the object as shown in the figure below. Holding down the left or right mouse button will allow the objects location to move. Holding down the Ctrl button while moving the object will create a copy. Holding down the Alt button while dragging from a State, Default State or Pseudo Start State will create a simple transition to the state in which the mouse pointer is located when the mouse button is released. Internal Transitions can be created using the same routine by releasing the mouse button in the source state.

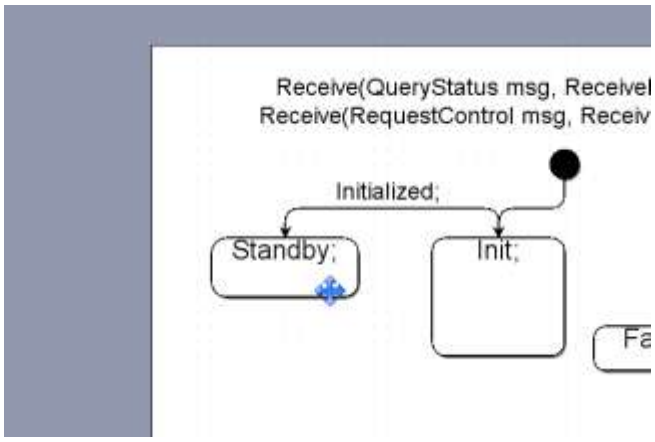

Receive(QueryStatus msg, Receive
Receive(RequestControl msg, Receiv
Initialized;
Standby;
Init;
Fa

Figure 97: Moving and Resizing Protocol Objects

- One or more behavior objects may be resized by selecting them with a mouse click or by stretching a rubber band around them. When this is done, green control boxes appear around the edges of the behavior objects. The objects can be resized by performing a mouse drag operation on one of these control boxes.
- For transition arrows, notice that the control box at the end of the transition turns blue when that end of the transition is connected to a state.
- The label of the transition may be moved by performing a mouse drag operation on the tiny yellow control box next to the transition label.

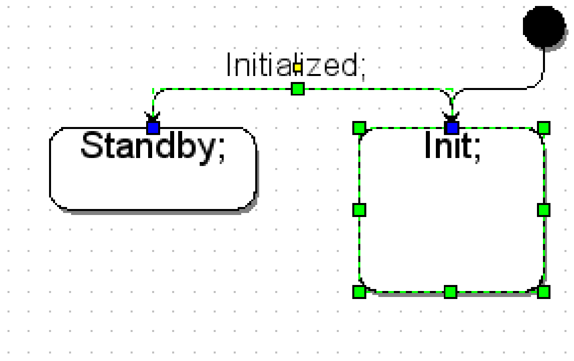


Figure 98: Moving Protocol Object Labels

- **Undo/Redo:** The user may undo or redo one or more operations by selecting the undo and redo buttons (shown in figure below) on the menu bar or by typing CTRL+Z or CTRL+Y.

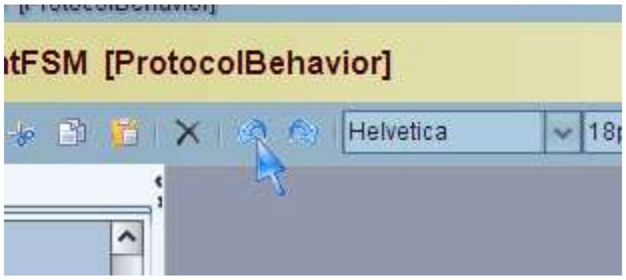


Figure 99: Protocol Editor Undo Icon

- **Abbreviated Transition Labels:** Since the signature of a transition can be rather long, the transition labels appear abbreviated when they are not being edited. When the user double clicks on a transition, the entire transition label appears in an editable text box as shown in the figures below. The abbreviated view only shows the transition trigger and parameter list without namespaces.

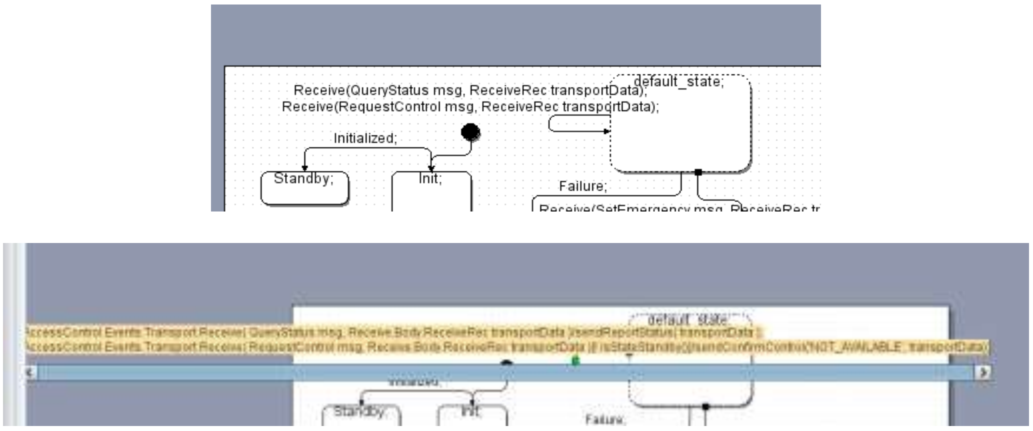

Receive(QueryStatus msg, ReceiveRec transportData);
Receive(RequestControl msg, ReceiveRec transportData);
default_state;
Initialized;
Standby;
Init;
Failure;

Figure 100: Protocol Trigger Abbreviated and Expanded Transition Labels

- **State Machine and State Nesting:** When drawing nested behavior objects[2], the user must first enter the parent shape by right clicking on the shape and then selecting "Shape" => "Enter Group" in the popup menu as shown in the figure below. Once the user has drawn the nested behavior objects, the user may exit the parent shape by selecting "Shape"=> "Exit Group" or "Shape" => "Home". This operation is necessary

[2] For example, when drawing states and transitions within a finite state machine object, or drawing nested states and transitions within a parent state object.

to ensure proper nesting of sub-behaviors. (Also see keyboard shortcuts in the last sub-section of this section)

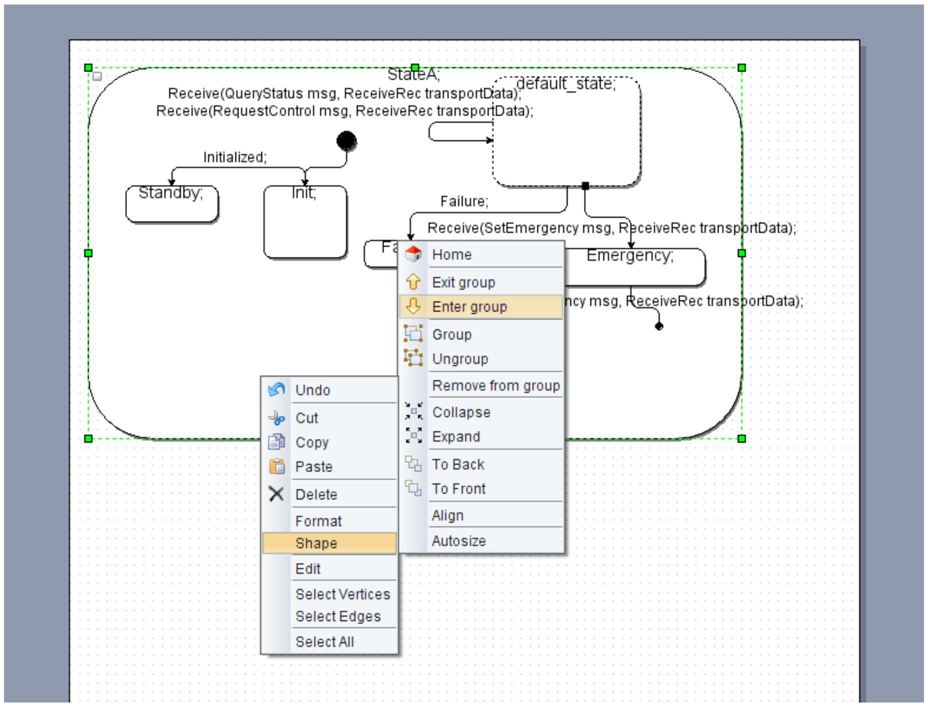
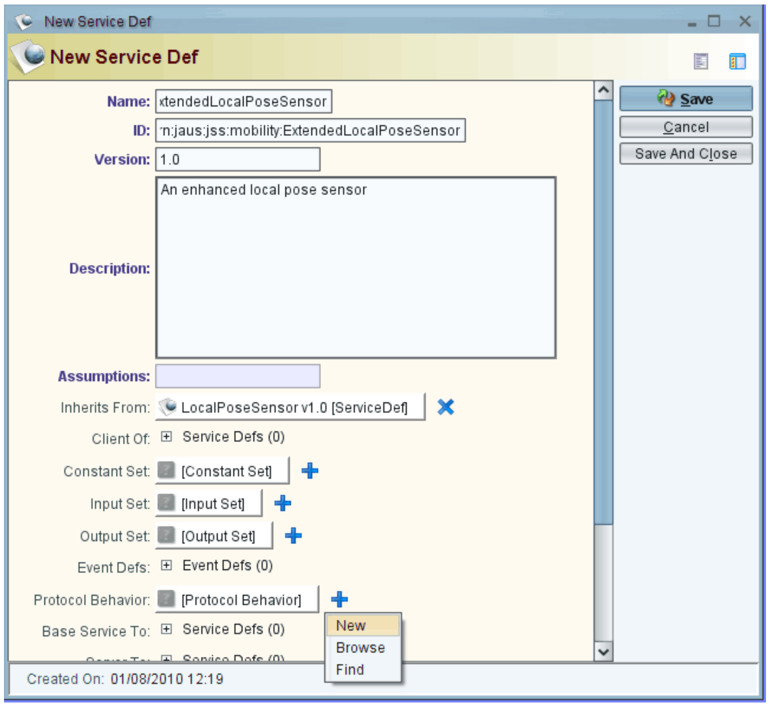


Figure 101: Entering a State Group

12.5 State Machine Inheritance

When the user specifies that a service definition inherits-from another service definition, a shell of the protocol behavior containing the base behavior states is automatically created for the user. This shell may be viewed by selecting a new protocol behavior after setting the inheritance relationship as shown in the figure below.



New Service Def

Name: xtendedLocalPoseSensor

ID: 'n:jaus:jss:mobility:ExtendedLocalPoseSensor

Version: 1.0

Description: An enhanced local pose sensor

Assumptions:

Inherits From: LocalPoseSensor v1.0 [ServiceDef]

Client Of: Service Defs (0)

Constant Set: [Constant Set]

Input Set: [Input Set]

Output Set: [Output Set]

Event Defs: Event Defs (0)

Protocol Behavior: [Protocol Behavior]

Base Service To: Service Defs (0)

Created On: 01/08/2010 12:19

Figure 102: Defining Inheritance

The shell protocol behavior opens in a new protocol behavior editing window as seen in the figure below. The red states imply read-only base behavior states. The layout and format of the states in the protocol behavior shell is as defined by the base service definition. While the labels are read-only, the states themselves may be moved or resized. This shell is a copy of the base behavior state machine. Modifying its layout does not affect the protocol behavior layout of the base service definition. The user can now extend this state machine by adding nested states and transition to the base states.

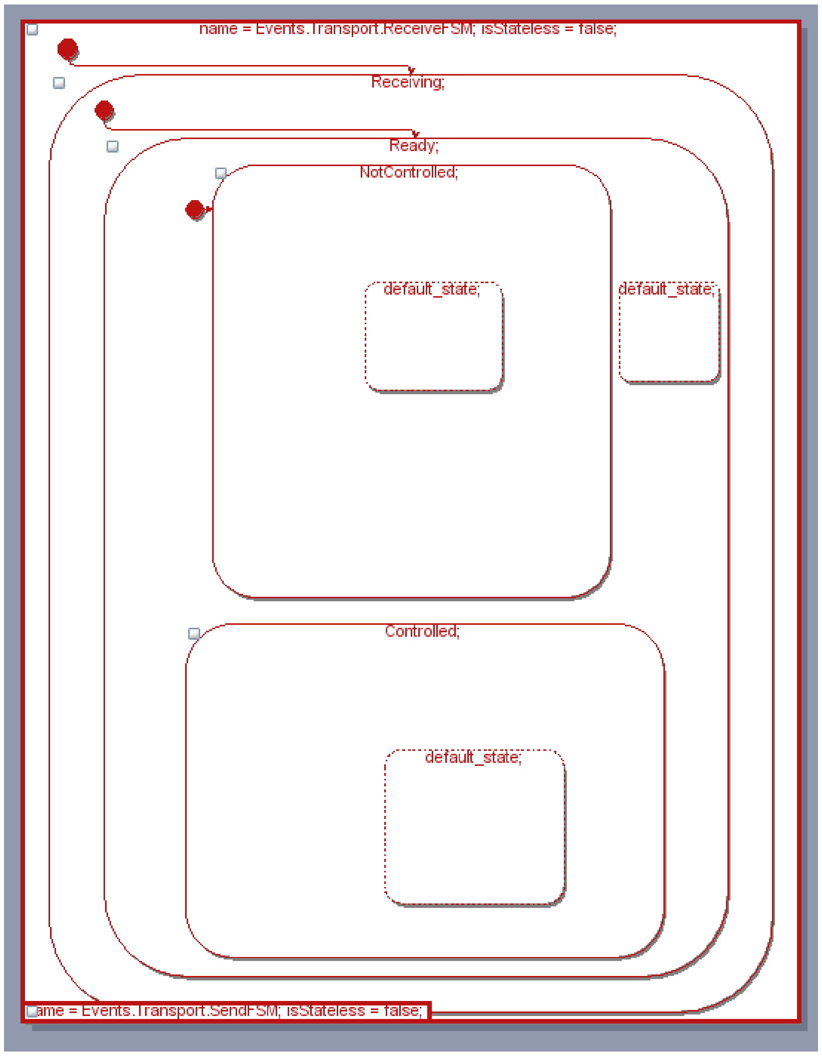

name = Events.Transport.ReceiveFSM; isStateless = false;
Receiving;
Ready;
NotControlled;
default_state;
default_state;
Controlled;
default_state;
ame = Events.Transport.SendFSM; isStateless = false;

Figure 103: Inherited States as Red Read-Only States

12.6 Redoing the Layout of Imported Protocol Behavior

Since JSIDLv1.0 does not specify layout information for state machines, protocol behavior that is imported from JSIDL compliant XML will need to be formatted for readability and clarity. JTS performs minimal formatting to help the user reformat the state machines. The figure below illustrates an imported copy of the Management Service protocol behavior[3]. JTS provides the simple step layout (highlighted in blue) where each step shows a level of nesting. Each column represents a set of sibling states. The parent state of the set is a step above and to the left of the

[3] Taken from AS5710 v1.0

column. For example, the sibling states NotControlled and Controlled are the nested states of Ready. When the set of sibling states contains a default state, the default state is drawn as the first state in the column and is placed adjacent to the parent state. The default states are marked with big blue check marks.

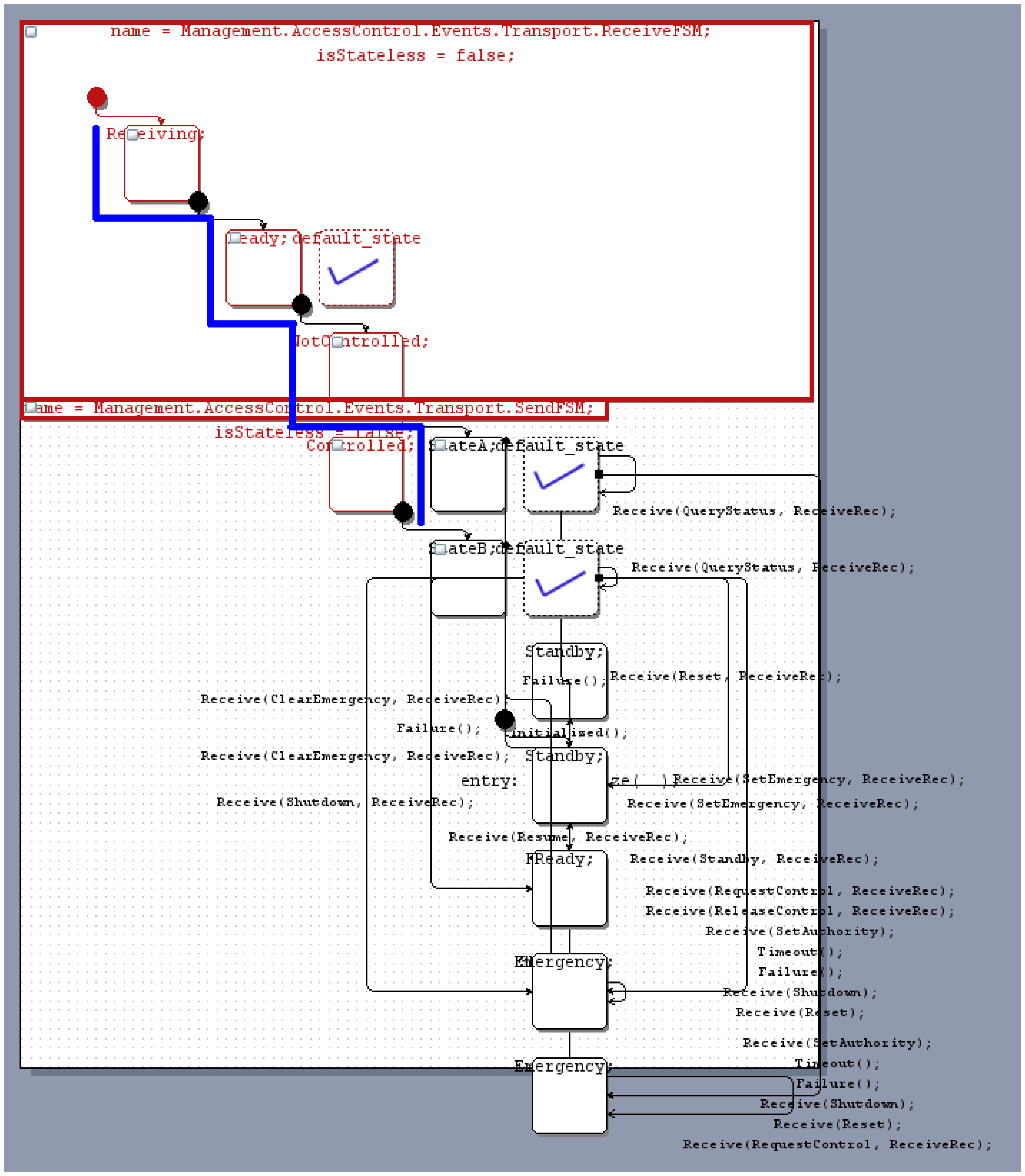

name = Management.AccessControl.Events.Transport.ReceiveFSM;
isStateless = false;
Receiving;
Ready;default_state
NotControlled;
name = Management.AccessControl.Events.Transport.SendFSM;
isStateless = false;
Controlled;
StateA;default_state
Receive(QueryStatus, ReceiveRec);
StateB;default_state
Receive(QueryStatus, ReceiveRec);
Standby;
Failure();
Receive(Reset, ReceiveRec);
Receive(ClearEmergency, ReceiveRec)
Failure();
Initialised();
Receive(ClearEmergency, ReceiveRec);
Standby;
entry:
Receive(SetEmergency, ReceiveRec);
Receive(Shutdown, ReceiveRec);
Receive(SetEmergency, ReceiveRec);
Receive(Resume, ReceiveRec);
Ready;
Receive(Standby, ReceiveRec);
Receive(RequestControl, ReceiveRec);
Receive(ReleaseControl, ReceiveRec);
Receive(SetAuthority);
Timeout();
Failure();
Emergency;
Receive(Shutdown);
Receive(Reset);
Receive(SetAuthority);
Timeout();
Emergency;
Failure();
Receive(Shutdown);
Receive(Reset);
Receive(RequestControl, ReceiveRec);

Figure 104: Default Layout After Service Import

Although this layout separates out the states and makes the inheritance hierarchy visible to the user, reformatting the state machine is still a fairly difficult task. A few tips are provided below to help make it easier for the user to reformat the state machine starting with this layout. (Also see keyboard shortcuts in the last sub-section of this section)

Since this process is not easy, it is recommended that the user save the formatted protocol behavior frequently. Should the layout get messy, the protocol behavior view simply needs to be reopened to get back to the last saved version.

- **Formatting Nested Behavior Objects**

1. Begin reformatting from the super state and move down each nested chain iteratively as described by steps 3-6 below. (Also see keyboard shortcuts in the last sub-section of this section)
2. First, enter the parent shape of the shape that needs to be formatted[4]. When starting from the top, the user must first enter the state machine object.
3. Next, collapse the nested states of the state being reformatted as shown in the figure below. In this case, it is the state called Receiving that is being reformatted. Collapse the nested states by clicking on the little white box on the top left corner of the state.

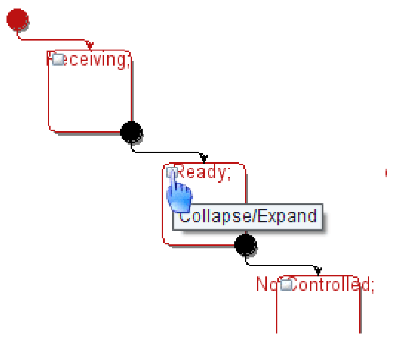


Figure 105: Collapsing and Expanding States

4. Once the parent shape has been entered (the state machine object in this case), the top left corner of the page coincides with the top left corner of the parent shape. So, in order to maximize the space utilization within the parent object, move the objects being reformatted to the top right corner of the page.

[4] See Section 10.2 on Drawing tips to learn how to enter and exit shapes.

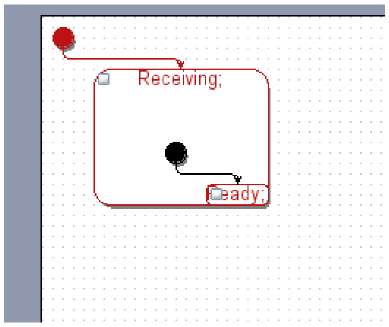
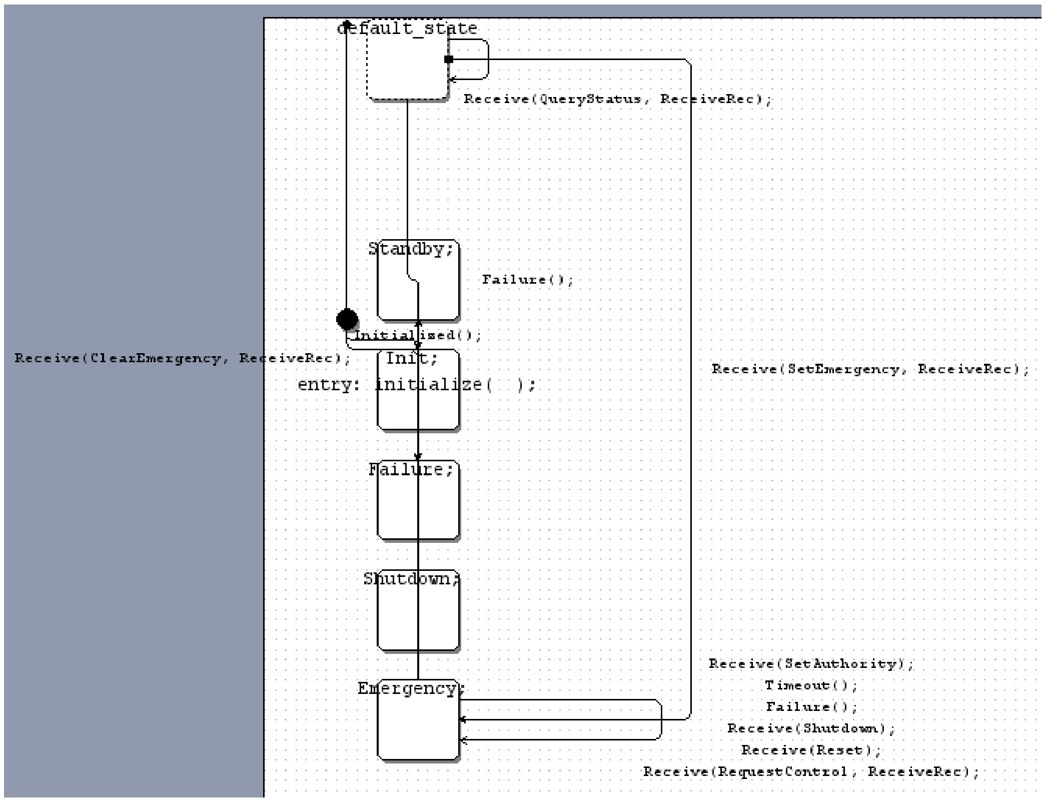


Figure 106: Moving States into Parent States

5. Perform steps 3-5 recursively until the deepest level of nested states in each chain of nesting has been reached.

6. Once the deepest level of nesting has been reached, format the behavior objects at this level in a manner that minimizes intersections. The figures below show the before and after views for the formatting performed on StateA’s nested state set. Refer to the next sub-section on Formatting Transitions to learn how the layout of crowded transitions can be reformatted.



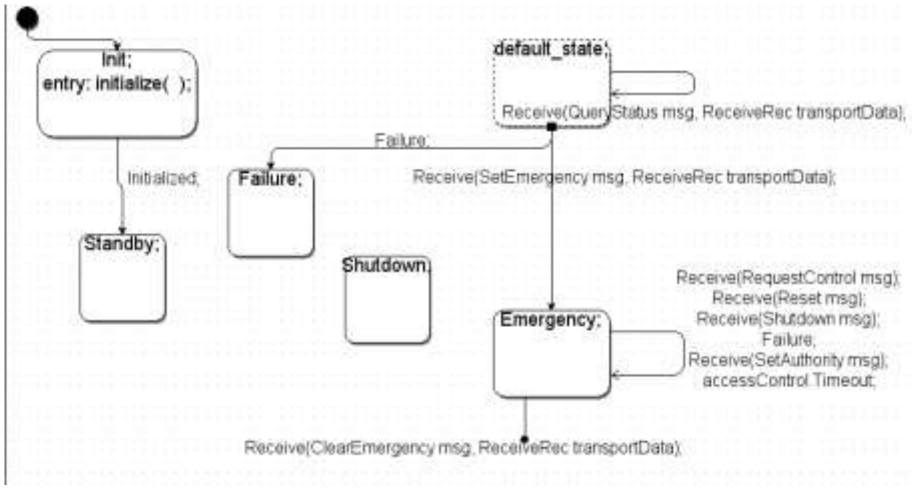

Init;
entry: initialize( );
default_state;
Receive(QueryStatus msg, ReceiveRec transportData);
Failure;
Initialized;
Failure;
Receive(SetEmergency msg, ReceiveRec transportData);
Standby;
Shutdown;
Receive(RequestControl msg);
Receive(Reset msg);
Receive(Shutdown msg);
Emergency;
Failure;
Receive(SetAuthority msg);
accessControl.Timeout;
Receive(ClearEmergency msg, ReceiveRec transportData);

Figure 107: Before and After Reorganizing States

7. Once all the deepest level nested states have been formatted, the user will need to exit out of all the shapes and then finalize the layout of the state machine by making slight adjustments to the shapes, sizes and locations of the behavior objects in order to ensure that the state machine fits on the page. The final layout for the Management state machine is shown below. It is recommended that the user enter a behavior object before

moving two or more of its nested behavior objects around. Alternately, the user may collapse the behavior object before moving the object and its nested objects simultaneously. This prevents behavior objects from entering (or snapping to) other objects to which they share no parent-child relationship.

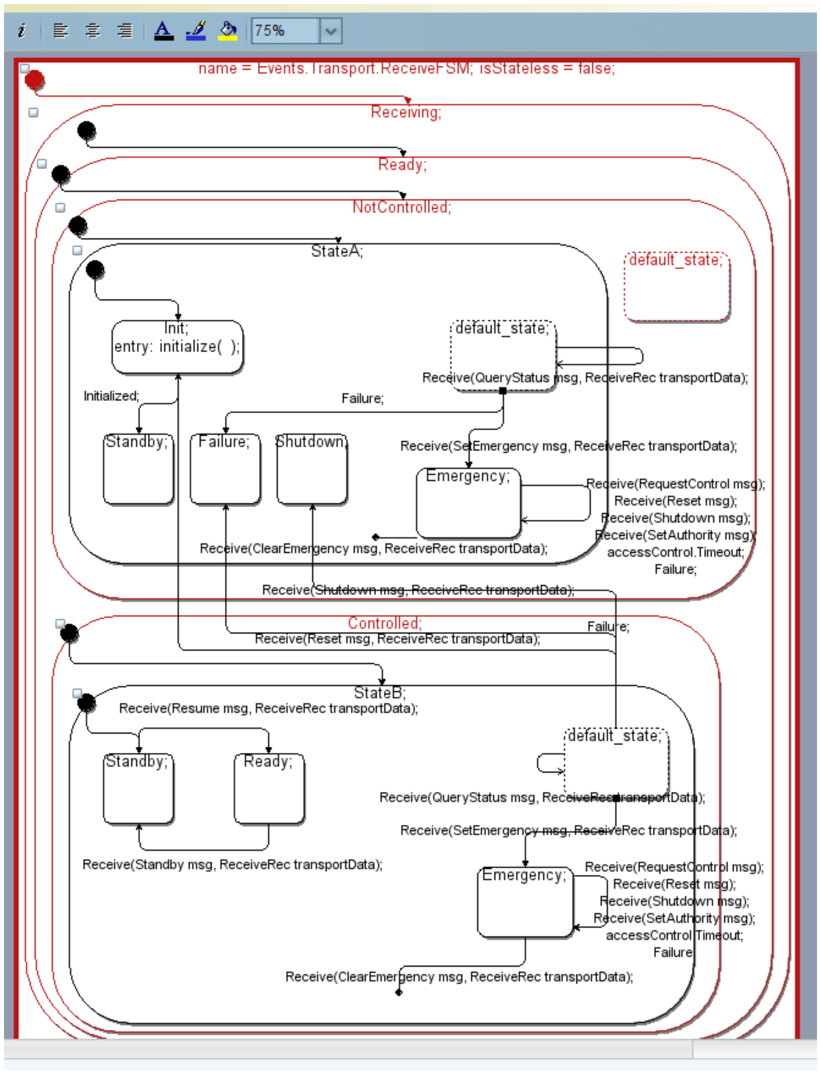

75%
name = Events.Transport.ReceiveFSM; isStateless = false;
Receiving;
Ready;
NotControlled;
StateA;
default_state;
Init;
entry: initialize( );
default_state;
Receive(QueryStatus msg, ReceiveRec transportData);
Initialized;
Failure;
Standby;
Failure;
Shutdown;
Receive(SetEmergency msg, ReceiveRec transportData);
Emergency;
Receive(RequestControl msg);
Receive(Reset msg);
Receive(Shutdown msg);
Receive(SetAuthority msg);
accessControl.Timeout;
Failure;
Receive(ClearEmergency msg, ReceiveRec transportData);
Receive(Shutdown msg, ReceiveRec transportData);
Controlled;
Failure;
Receive(Reset msg, ReceiveRec transportData);
StateB;
Receive(Resume msg, ReceiveRec transportData);
Standby;
Ready;
default_state;
Receive(QueryStatus msg, ReceiveRec transportData);
Receive(SetEmergency msg, ReceiveRec transportData);
Receive(Standby msg, ReceiveRec transportData);
Emergency;
Receive(RequestControl msg);
Receive(Reset msg);
Receive(Shutdown msg);
Receive(SetAuthority msg);
accessControl.Timeout;
Failure
Receive(ClearEmergency msg, ReceiveRec transportData);

Figure 108: Fully Re-Organized State Machine

12.7 Keyboard Shortcuts

The following keyboard shortcuts may be used in the protocol behavior UI.

```
"control S":"save"
```

```
"control shift S":"saveAs"

"control N":"new"

"control O":"open"

"control Z":"undo"

"control Y":"redo"

"control shift V": selectVertices

"control shift E":"selectEdges"

"F2":"edit"

"DELETE":"delete"

"UP":"selectParent"

"DOWN":"selectChild"

"RIGHT":"selectNext"

"LEFT":"selectPrevious"

"PAGE_DOWN":"enterGroup"

"PAGE_UP":"exitGroup"

"HOME":"home"

"ENTER":"expand"

"BACK_SPACE":"collapse"

"control A":"selectAll"

"control D":"selectNone"

"control X":"cut"

"control C":"copy"

"COPY":"copy"

"control V":"paste"

"control G":"group"

"control U":"ungroup"
```

```
"control ADD":"zoomIn"

"control SUBTRACT":"zoomOut"

- "DELETE":"delete"
```

13 Other Examples

Several other examples have been created to demonstrate additional capabilities of JTS or to address specific common implementation issues. These are not meant to be complete implementations, but rather they are designed to give enough detail to allow the developer to create their own implementation. Many of these example services inherit from the JSS core services. These definitions are located in the <JTS GUI dir>/resources/xml directory. Some core services may need to be imported before building up the component definitions within JTS for the following examples. Also included in the noted directory are various other published JSS services such as the mobility, manipulator and environmental sensing services.

13.1 Waypoint Driver Example

The Waypoint Driver example is based on several mobility services integrated into a single component to demonstrate waypoint following of a simulated vehicle. The simulator itself is very primitive, as the goal is not to demonstrate vehicle behavior but rather to show how multiple complex components can be integrated in a distributed system using published SAE JAUS services generated by JTS.

The Waypoint Driver component contains several critical services:

- Discovery: The Discovery Service provides a run-time directory of known services. At start-up, each service may register with Discovery to advertise its availability. When a client needs to make use of one or more of these services, it can query the Discovery Service for the JAUS ID of an available host. This component is based on the service published in SAE AS-5710.
- Vehicle Simulator: The Vehicle Simulator is a JTS-specific service that is not published by SAE JAUS. It provides basic vehicle simulation capabilities by updating the vehicle position based on wrench effort commands. The JSIDL representation for this Service is available in JAUSToolset/examples/xml/SkidSteerVehicleSim.xml.
- Pose Sensor: The Pose SensorService gets the vehicle position from the Vehicle Simulator and makes it available in a JAUS compliant service. This component is based on the Local Pose Sensor Service published in SAE AS-6009.
- Waypoint Driver: The Waypoint Driver component gets a list of waypoint command from the OCU and implements a primitive waypoint tracking scheme. The algorithm generates primitive wrench effort commands to the Vehicle Simulator based on the position currently reported by the Pose Sensor Component. This component is based on the Local Waypoint List Driver Service published in SAE AS-6009.

In addition to the Waypoint Driver Component, JTS includes a Java-based Example OCU. Based on the user's input, the OCU sends a list of waypoint commands to the Waypoint Driver and monitors the current (active) waypoint and displays the vehicle position on a simple 2D map. Once running, the slider bar can be used to alter the vehicle speed. This component is a JTS-specific service that is not published by SAE JAUS.

Since each service in the Waypoint Driver Component is generated as a separate library, we also need a mechanism for sharing data between the service implementations. For example, when the Skid Steer Vehicle Simulator updatesthe vehicle position, it needs to make that information available to the LocalPoseSensor service. This is achieved by a global SharedData object that isavailable to the state machines of each service in the component. Other data sharing data techniques are possible, such as a shared memory map or data file; this technique was selected only for its simplicity. However, since shared libraries (DLLs) on Microsoft Windows systems do not have a common memory space, the build system was modified to use static libraries rather than shared. This is achieved simple by changing the build line from "env.SharedLibrary(…)" to "env.Library(…)" in the Sconstruct file for each service.

Running the waypoint driver is simply a matter of building theWaypoint Driver and ExampleOCU components using the 'scons' build script, and running each resultant executable from the 'bin' directory. Since the two components are expected to communicate with each other, the Node Manager must be started prior to these Components, similar to the Addition Client/Server example.

13.2 Environmental Sensing Example

This example is found in the examples\StillImageServer_200 directory. It is intended to demonstrate how a camera can be used to allow the sending and receiving of still image data. However, in order to be agnostic of any underlying hardware, and for the purposes of demonstration only, the images are sourced from a set of JPEG files rather than live-video. These files are available in JAUSToolset/examples/StillImageServer_200/video and are required for correct operation of the example.

A simple Java-based Still Image Client has also been provided in JAUSToolset/examples/Java/StillImageClientComponent_220. This client simply queries a new image after the previous one is received, and displays the image in a small window. The JSIDL representation for the client is available in JAUSToolset/examples/xml/StillImageClient.xml.

Running the Environmental Sensing Server is simply a matter of building the components using the ‘scons’ build script, and running the executables from the ‘bin’ directory. Since the two components are expected to communicate with each other, the Node Manager must be started prior to these Components, similar to the Addition Client/Server example.

14 Search

The JTS user interface provides simple, yet powerful search capability that makes it very easy for the user to search the database for any JSIDL type. The various ways in which the user can perform a search are described below.

14.1 The Find Command

The Find command may be used to set up queries on any JSIDL type. The queries may be composed of multiple simple criteria in conjunction. To access all the types, select Types=>Types=>Browse from the Menu Bar.

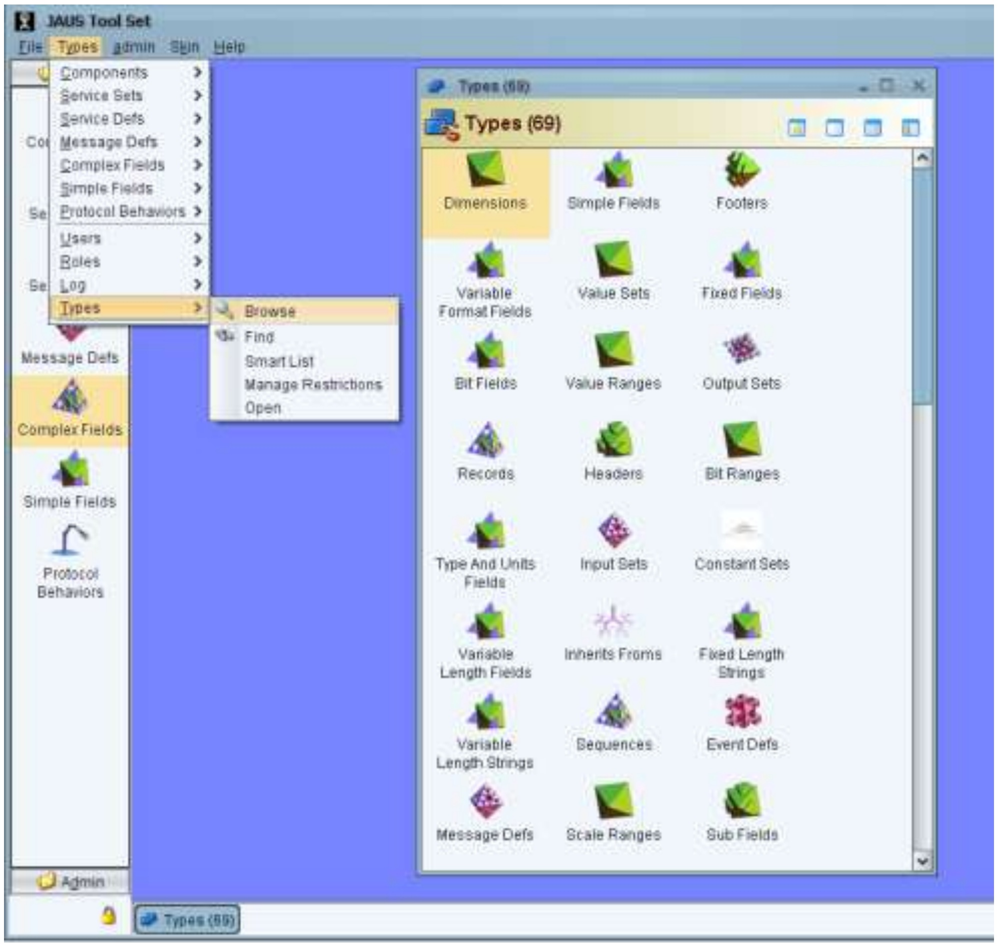
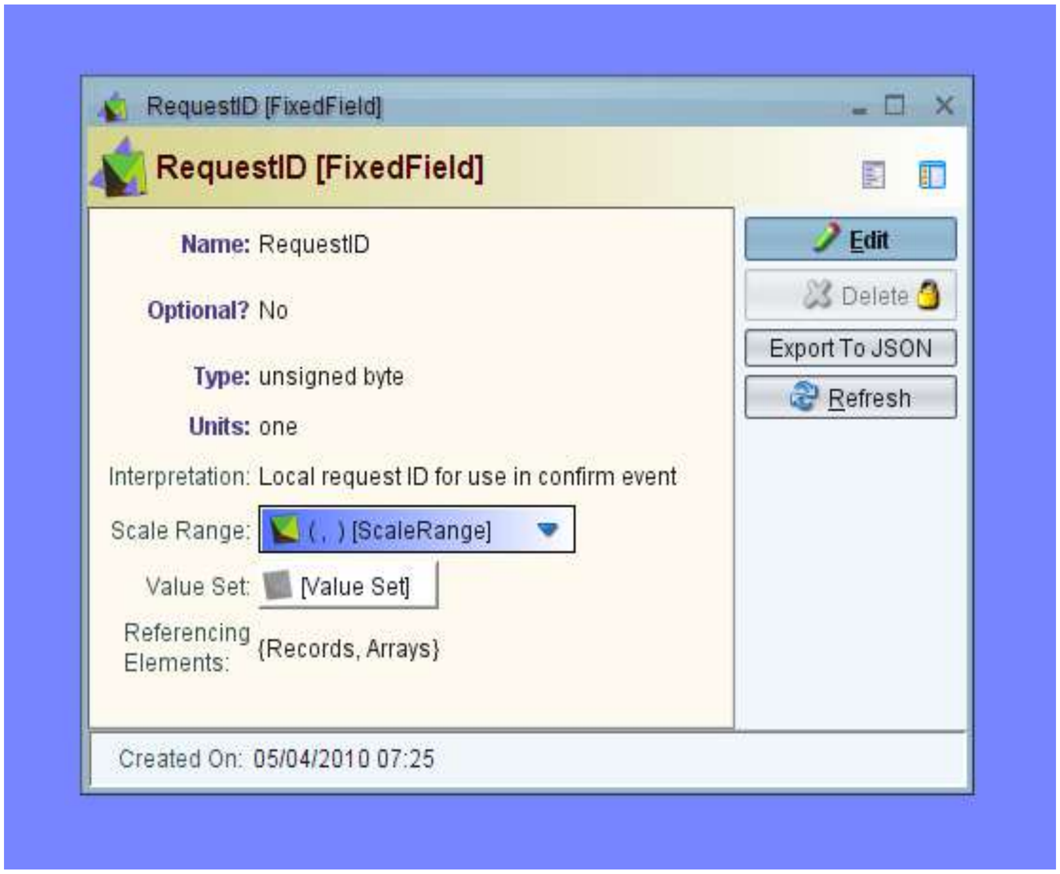


Figure 109: Find Command Screenshot

Often times, it may be necessary to find types that reference other types. For instance, let's suppose the user wishes to delete (or modify) a fixed field called RequestID. In order to perform such a change, the user will need to know if other types reference (or depend on the definition of) RequestID. This information can be found by first getting the set of types that can reference a fixed field. The "Referencing Elements" label gives this set (See figure below).



RequestID [FixedField]

Name: RequestID

Optional? No

Type: unsigned byte

Units: one

Interpretation: Local request ID for use in confirm event

Scale Range: (,) [ScaleRange]

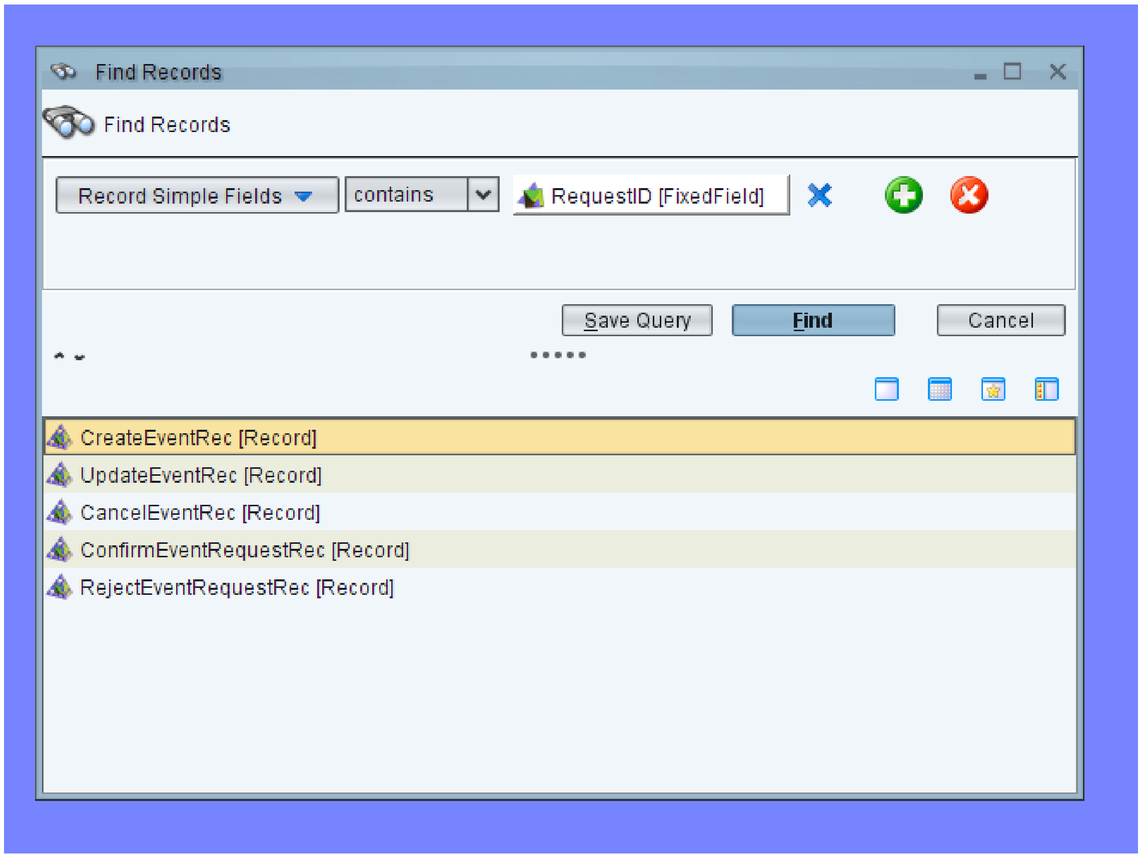
Value Set: [Value Set]

Referencing Elements: {Records, Arrays}

Created On: 05/04/2010 07:25

Figure 110: Referencing Elements Search Screenshot

Next, search the set of types for references to RequestID. To find all Records that reference RequestID, create a query on the Record type by right clicking on the Record type and selecting Find. The query is then set up as shown in the figure below. Note that this query can be further constrained by adding more conjunctive queries using the green "+" button. Also note that the query itself can be saved to the database for future use. Saved queries may be accessed through the Query type. And yes, Queries may be performed on stored queries!

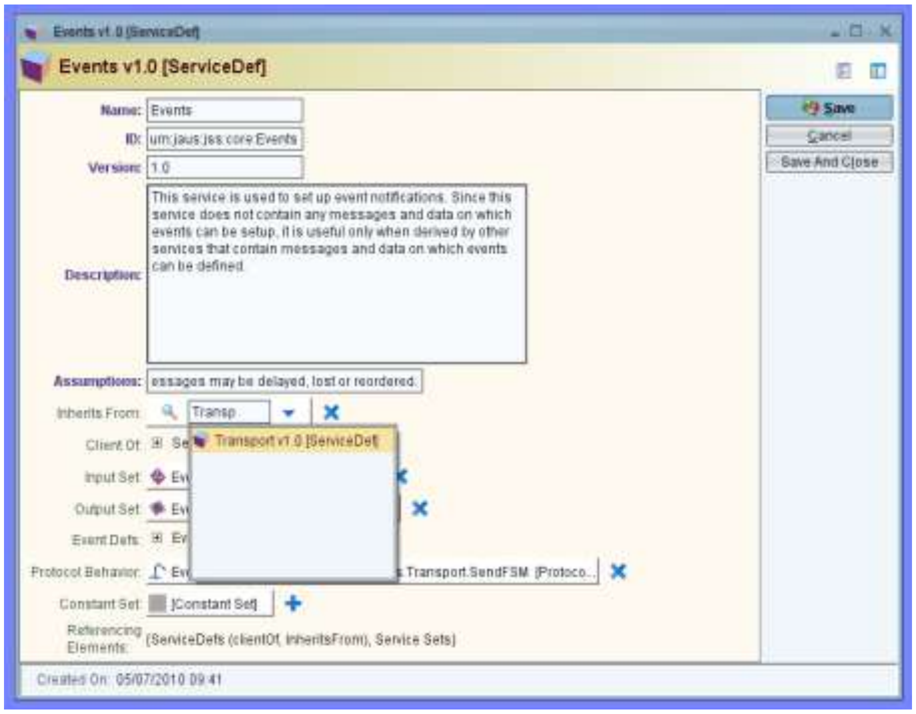


CreateEventRec [Record]
UpdateEventRec [Record]
CancelEventRec [Record]
ConfirmEventRequestRec [Record]
RejectEventRequestRec [Record]

Figure 111: Find Records Screenshot

14.2 Quick Search

Some associations between entities are displayed as buttons on the GUI. When the entity is in the edit mode, a quick search feature may be activated on these to-one associations by clicking once on the button. A single click brings up an auto-completion guide that filters existing database entries for that associated type as the user types the starting characters of the name of the association. The quick search can also be activated by tabbing through the internal frame's widgets (pressing the tab key to navigate through the widgets). The figure below shows an example of the quick search guide for the Inherits From association in a Service Definition internal frame.



Name: Events
ID: urn:jaus:jss:core:Events
Version: 1.0
Description: This service is used to set up event notifications. Since this service does not contain any messages and data on which events can be setup, it is useful only when derived by other services that contain messages and data on which events can be defined.
Assumptions: essages may be delayed, lost or reordered.
Inherits From: Transp
Transport v1.0 [ServiceDef]
Client Of: Se
Input Set: Ev
Output Set: Ev
Event Defs: Ev
Protocol Behavior: Ev s.Transport.SendFSM [Protoco...
Constant Set: [Constant Set]
Referencing Elements: (ServiceDefs (clientOf, inheritsFrom), Service Sets)
Created On: 05/07/2010 09:41

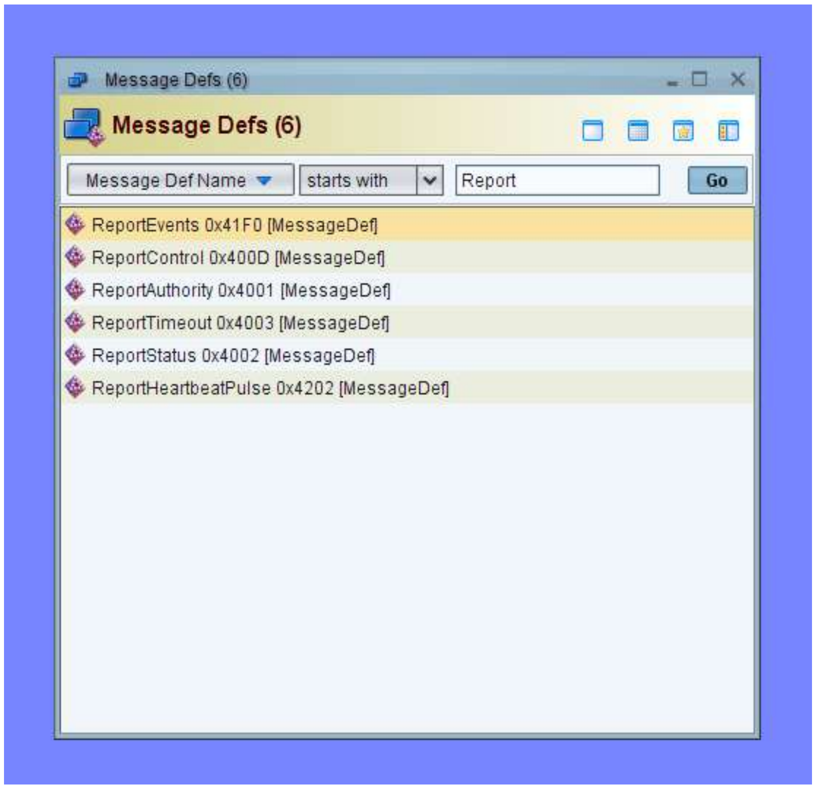
Figure 112: Quick Search with Auto-Completion Guide

14.3 Smart Lists

Smart Lists provide the ability to save a query performed using the Find command to the database.

14.4 Filtered Listings

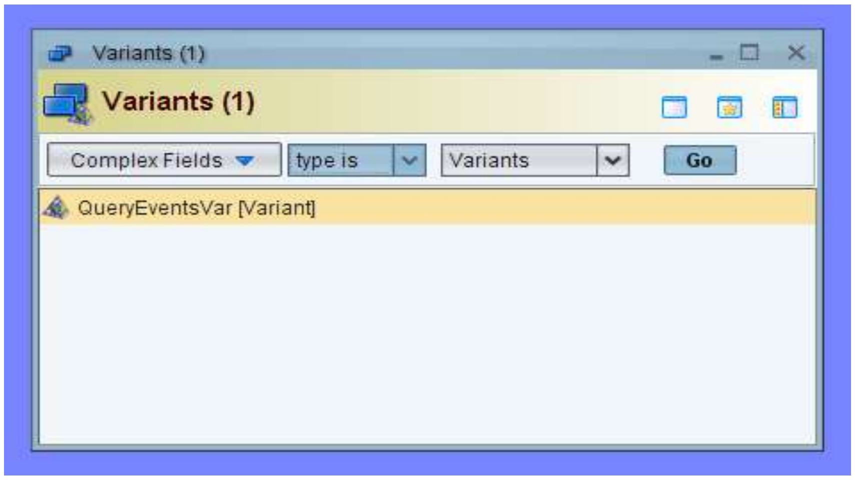
All list views are decorated with a simple query panel through which the lists may be filtered based on several search criteria that apply to the type of the list. For instance, the user may filter the list of message definitions to all messages that start with “Report” as shown in the figure below.



ReportEvents 0x41F0 [MessageDef]
ReportControl 0x400D [MessageDef]
ReportAuthority 0x4001 [MessageDef]
ReportTimeout 0x4003 [MessageDef]
ReportStatus 0x4002 [MessageDef]
ReportHeartbeatPulse 0x4202 [MessageDef]

Figure 113: Filtered Views

For lists that are polymorphic, as is the case for Simple Field and Complex Field lists, the query panel provides polymorphic query options as shown below.



QueryEventsVar [Variant]

Figure 114: Polymorphic Query Options

15 JSIDL Input / Output

XML Service Definitions written in JSIDL [5684] may be imported into the GUI by right clicking on the ServiceDef icon on the class list as shown below.

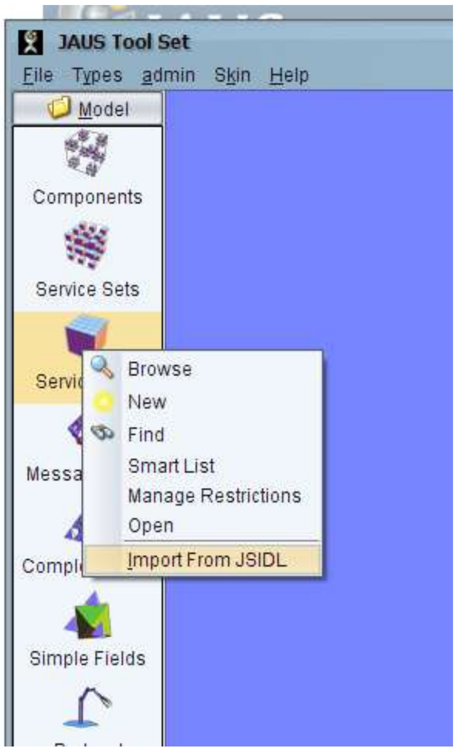
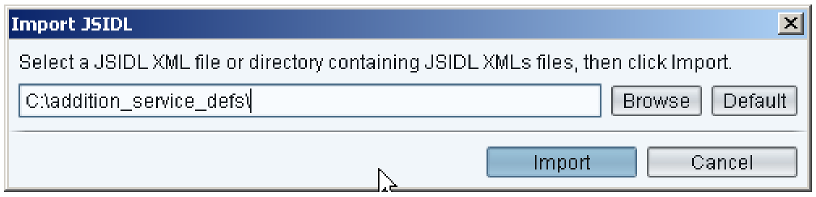


Figure 115: Importing a JSIDL Service Definition Written in XML

In the dialog that appears (Figure 116), enter the path to either a single JSIDL XML file, or a directory containing one or more JSIDL XML files. The Browse button allows a file or directory path to be selected via a browse dialog. The Default button will restore the last path used to perform an import or a default starting path if no imports have already been performed during the current JTS session.


Import JSIDL

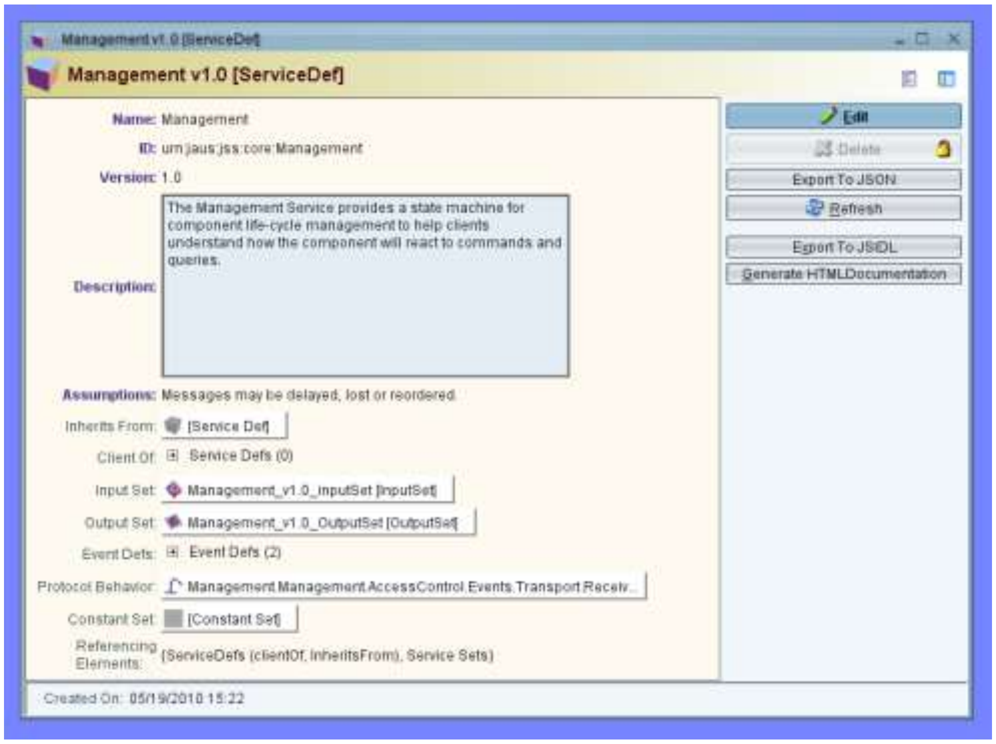
Select a JSIDL XML file or directory containing JSIDL XMLs files, then click Import.

C:\addition_service_defs\

Figure 116: Import Dialog

If the service definition being imported references other service definitions, declared type sets or declared constant sets, JTS will attempt to locate these referenced specifications within the directory and sub-directories of the parent directory of the service definition being imported. If these specifications are found, they are automatically imported. If all of the referenced specifications are not found, the import procedure will abort with an error.

Service definitions may also be exported to XML from the database. This is done by clicking on the "Export to JSIDL" button on the Service Definition internal frame and selecting a destination path as shown below.


Management v1.0 [ServiceDef]
Management v1.0 [ServiceDef]
Edit
Delete
Export To JSON
Refresh
Export To JSIDL
Generate HTMLDocumentation

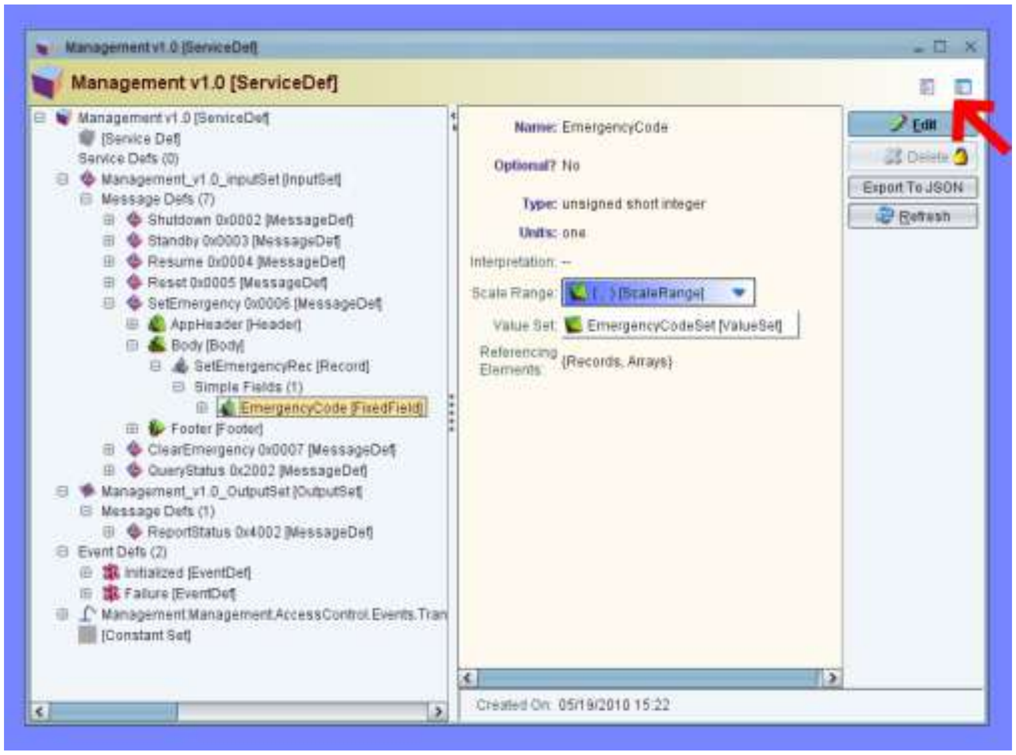
Name: Management
ID: urn:jaus:jss:core:Management
Version: 1.0
Description: The Management Service provides a state machine for component life-cycle management to help clients understand how the component will react to commands and queries.
Assumptions: Messages may be delayed, lost or reordered.
Inherits From: [Service Def]
Client Of: Service Defs (0)
Input Set: Management_v1.0_inputSet [InputSet]
Output Set: Management_v1.0_OutputSet [OutputSet]
Event Defs: Event Defs (2)
Protocol Behavior: Management.Management.AccessControl.Events.Transport.Receiv...
Constant Set: [Constant Set]
Referencing Elements: (ServiceDefs (clientOf, inheritsFrom), Service Sets)
Created On: 05/19/2010 15:22

Figure 117: Exporting a Service Definition to XML

Unfortunately, the exported XML is not fully compatible with the JSIDL schema defined in AS5684 [5684]. This is due to some additional tags that were added to the schema under the same namespace as JSIDLv1.0. Since the differences between the schema used by JTS and the original JSIDLv1.0 schema are only additions, removal of these additional tags is all that is required to arrive at an equivalent JSIDLv1.0 specification. Information about the additional tags is provided in the file JAUSToolset\GUI\resources\schema\JSIDL_Plus\jsidl_plus.rnc.

16 Tree View

A not so apparent but very useful feature in the JTS GUI is the tree view. Upon first use, the user is introduced to form based internal frames for reading or updating JSIDL objects. While convenient when working at a single level in the object hierarchy, the form based view is not very helpful when it comes to navigating through an object's hierarchy. JTS provides multiple views including a tree view via view icons on the top right corner of the internal frames. One of these views happens to be a tree view. The figure below shows the tree view for the Management Service. The red arrow points to the view icons in the top right corner. To the right of the tree view appears a form based view of the object that is selected in the tree view; the EmergencyCode fixed field in this case.


Management v1.0 [ServiceDef]

Management v1.0 [ServiceDef]

```
Management v1.0 [ServiceDef]
    [Service Def]
    Service Defs (0)
    Management_v1.0_inputSet [InputSet]
    Message Defs (7)
        Shutdown 0x0002 [MessageDef]
        Standby 0x0003 [MessageDef]
        Resume 0x0004 [MessageDef]
        Reset 0x0005 [MessageDef]
        SetEmergency 0x0006 [MessageDef]
            AppHeader [Header]
            Body [Body]
                SetEmergencyRec [Record]
                    Simple Fields (1)
                        EmergencyCode [FixedField]
            Footer [Footer]
        ClearEmergency 0x0007 [MessageDef]
        QueryStatus 0x2002 [MessageDef]
    Management_v1.0_OutputSet [OutputSet]
    Message Defs (1)
        ReportStatus 0x4002 [MessageDef]
    Event Defs (2)
        Initialized [EventDef]
        Failure [EventDef]
    Management.Management.AccessControl.Events.Tran
    [Constant Set]
```

Name: EmergencyCode

Optional? No

Type: unsigned short integer

Units: one

Interpretation: --

Scale Range: [ ) [ScaleRange]

Value Set: EmergencyCodeSet [ValueSet]

Referencing Elements: {Records, Arrays}

Created On: 05/19/2010 15:22

Figure 118: Tree View in a JTS Internal Frame

17 Software Framework

Note: Unless otherwise noted, the service definition files you will need for this section are located in: examples/ExampleOCU_200.

Note: All examples in this section are in C++; however, the code is nearly identical to the output of the Java and C# code. For a description of the differences between the generated languages, please refer to Section 16.2

As shown in previous examples, the JAUS Tool Set is capable of generating C++, Java, and C# source code that implements the service definitions for each component. This section will discuss key aspects of working with the generated code such as adding behaviour, sending and receiving messages, and configuring the run-time environment.

Throughout this section, a simple Operator Control Unit will be used as an example. The OCU is a primitive example that makes use of transitions, guards, and actions to achieve the protocol required to interact with an SAE JAUS Local Waypoint List Driver. Basic knowledge of the Waypoint Driver is convenient, but not required for this example. The protocol for this example is given in the following diagram and subsequent state transition tables.

Figure 119: Protocol Behavior for Example OCU

SERVICE STATE TRANSITION TABLE

Label	Trigger	Conditions	Actions
A	ReportIdentification		
B	ConfirmControl	! *isControlAccepted*	
C	ConfirmControl	*isControlAccepted*	
D	ReportStatus	! *inReadyState*	
E	ReportStatus	*inReadyState*	
F	ConfirmElementRequest		*sendExecuteList* *sendQueryActiveElement* *sendQueryLocalWaypoint*
G	ReportActiveElement		*printDebug*

			sendQueryActiveElement
	ReportLocalWaypoint		*saveTargetData* *sendQueryActiveElement*

SERVICE TRANSITION CONDITIONS

Condition	Interpretation
isControlAccepted	True if the ConfirmControl message indicates control is accepted.
inReadyState	True if ReportStatus message indicates the server is in the Ready state.

SERVICE TRANSITION ACTIONS

• Action	Interpretation
sendQueryIdentification	Send a Query Identification message to the Discovery component.
sendRequestControl	Send a Request Control message to the Waypoint Driver component.
sendQueryStatus	Send a Query Status message to the Waypoint Driver Component.
sendResume	Send a Resume message to the Waypoint Driver Component.
sendWaypoints	Send a list of 4 waypoints to the Waypoint Driver Component.
sendExecuteList	Send an Execute List message to the Waypoint Driver with a speed of 10.
sendQueryActiveElement	Send a Query Active Element message to the Waypoint Driver.
sendQueryLocalWaypoint	Send a Query Local Waypoint message to the Waypoint Driver component.
printDebug	Print debug information to the display.
saveTargetData	Save the current waypoint target information for display in printDebug().

17.1 Topology of the Generated Code

The JTS code generator outputs C++, Java, and C# source code to the specified component directory. A nested structure is used, such that code for the entire component occurs within the top-level directory while code for each defined service has its own sub-directory. In all cases, C++ header files are stored in the 'include' and source files are stored in 'src'. Header files and the include folder are only generated for C++, all generated Java and C# code is located in the /src directory. When code for a component is generated, there are five major elements.

- **Code for Component Control**: At the top level, the main.cpp/Main.java/Main.cs entry point is created along with header and source files for a class based on the component name. This component class handles the start-up of the component, which in turn instantiates each defined service. In general, these files will not require user modification (exceptions will be discussed in Section 17.9).
- **Common Code for Component Infrastructure**: The 'Common' library is used by all generated components, and provides default implementations for transport control, sending and receiving messages locally and remotely, and thread management. C# and Java have their own 'Common' libraries that provide access to the C++ Common library as well. These files will not require user modification.
- **SCons Build Scripts**: JTS uses SCons for its simplicity as a cross-platform build environment, The C# generated code utilizes an add-on located in the /site_scons subdirectory to compile These 'Sconstruct' scripts are automatically created for the component which in turn calls the scripts for each defined service. The C# add-on is included with JTS and does not need to be downloaded or installed by the user. These files will not require user modification.
- **Code for Message Classes**: Within each service definition, header and source files are created for internal events as well as input and output messages. These messages include functions for getting and setting the members, along with encoding (marshalling) and decoding (unmarshalling) functions for transmission in binary form. These files will not require user modification. Additional explanation on accessing member variables and functions is offered in Section 17.8.
- **Code for Handling Protocol Behavior**: Within each service definition, the protocol state machine is encapsulated within several files. The JTS uses the State Machine Compiler (SMC) which generates separate files for state management and user-defined stubs. As a result, all users will need to modify the src/<service urn>/

<service_name>_FSM.cpp(.java/.cs) file to populate the transition actions and guards. In some cases, the user will also modify src/<service urn>/ <service_name>Service.cpp (.java/.cs) to define values for primitive types used as transition parameters. This will be described extensively throughout the remainder of this section.

17.2 Inherited Services

The JAUS Toolset 2.0 release alters how inherited services are handled in generated code for all languages. In prior releases, inherited services were 'flattened' into a single generated service, and the behaviour of each inherited protocol definition was completely encapsulated by that resultant service. While this provided some benefit in components with relatively simple inheritance structures, it greatly restricted service re-use and prevented a component from housing multiple child services that inherited from a common parent.

In the latest release, this 'flattened' approach has been replaced with a 'separate-but-linked' implementation in which each service in the inheritance chain is generated as a separate class. However, because the protocol behaviour of inherited services may be impacted by the protocol of its parent and/or children services, these separately generated services include references to each parent. In the event that two or more child services share a common parent, then each child receives a shared reference to a single parent service, as required by [**AS5710**]. This ultimately allows a single component to house arbitrarily complex chains of inherited services, assuming such chains comply with [**AS5684**].

Because each of the generated services contains its own representation of the protocol behaviour state machine, some care is required to ensure consistency and synchronization between inherited services with linked state machines. For example, in some cases a transition defined by a child service may cause a transition in the parent service. In the new generated code structure, the child service must explicitly notify the parent that such a transition is required. *These notifications are established in the automatically generated* ***setupNotifications****() function and do not require any input from the end-user*.

There are two primary improvements with this change:

- A component can now host multiple services with common inheritance chains, as per the requirements in [**AS5710**]. An example of this is the Waypoint Driver component in JAUSToolset/examples.

- It is now easier to re-use and share implementations between components. For example the Waypoint Driver and Still Image components both inherit from Access Control. Because Access Control is now generated as a separate service library, we can simply copy the AccessControl_ReceiveFSM.cpp file from one component to the other and have a common implementation. Alternatively, the build system could be updated to use an Access Control library that is maintained separately from either component.

17.3 Differences Between Generated C++, Java, and C#

All three languages are designed to have a 1:1 mapping to the formal specifications. There should be little difference between languages integrating user code into a service.

For a detailed description of the differences between how Java, C#, and C++ handle encoding and decoding data, and other language specific syntax, please refer to the Developer's Guide.

17.4 Adding Protocol Behaviour: Guards and Actions

The source code generated by JTS is intended to build without user modification; however, to achieve any meaningful functionality, the protocol behaviour 'stubs' need to implemented. As previous examples have shown, these stub files are generated for each service, with names based on the service name appended with '_FSM.cpp', or '_FSM.java', or '_FSM.cs'.

A function stub is created for each action and guard defined by the service's state machine. The name of the function is based on the name of the action or guard; however, actions are appended with 'Action'. Furthermore, each stub contains the arguments defined by the transition giving the developer access to whatever information is required, such as (but not limited to) the message that triggered the transition and information on the message sender.

The implementation of each stub is at the complete discretion of the systems designer. The only restriction is that guards must return a Boolean true/false value, which in turn will drive the automatically generated state machine. Any guard that returns false will not complete the triggering transition.

We now consider several examples based on our simple OCU outlined at the start of this section. The function stubs are generated in ExampleOCU_200/src/urn_jts_example_ocu_1_0/ OCU_OcuFSM.cpp. Consider the "printDebug" action that occurs when a Report Active Element message is received in the Running state. This function accepts one argument, namely the triggering message, and prints debug information to the screen:

```
void OCU_OcuFSM::printDebugAction(ReportActiveElement msg)
{
     fprintf(stdout, "Current waypoint: %d Target: X=%.2f Y=%.2f \r",
           msg.getBody()->getActiveElementRec()->getElementUID(),
           target.getX(), target.getY());
     fflush(stdout);
}
```

A guard implementation is similar, but ultimately requires the function to return a Boolean. Consider a simple guard that checks to see if an incoming Report Status message specifies that the client is, in fact, in the 'Ready' state:

```
bool OCU_OcuFSM::inReadyState(ReportStatus msg)
{
     return (msg.getBody()->getReportStatusRec()->getStatus() == 1);
}
```

Once again the triggering message is passed as a parameter and the message data itself is used to evaluate the guard. More details on accessing message data is given in Section 17.8.

The examples used so far have both been relatively primitive, such that the service can completely resolve the action internally. In many cases, however, transitions are based on incoming messages which require a response. How can we use the Software Framework to send a message back to the requesting component?

17.5 Sending Messages

This section provides detail on sending messages to service interfaces co-located on the same component, or remotely on different components. Similar examples are used in Section 8. These components may reside on the same node (processor) or remote nodes when used with a properly configured IP-network. Generally, sending a message to a different component requires six steps:

1) Instantiate the desired message.

2) Populate any relevant data fields.

3) Encode the message into a byte array.

4) Instantiate a Send event and add the encoded message to it.

5) Set the destination JAUS ID of the intended recipient and the source JAUS ID of the sending component.

6) Invoke the 'send' function on the JausRouter to send the wrapped message.

In this procedure, we see a transport wrapper class is used for the encoded message. The Send event definition is contained in the 'Common' library, and is defined based on the Transport Service defined by [AS5710].

For simplicity, the JTS::StateMachine class from which all generated state machines inherit includes a default implementation for steps 3-6 above. Hence, the user need only instantiate and populate the message, and the sendJausMessage() function can be used to encode the message, add it to the appropriate transport wrapper, and send it using the Framework.

Consider an example from our OCU, in which the 'sendQueryStatus' action requires an implementation which sends a Query Status message to the Local Waypoint List Driver. It is assumed that the JAUS ID of the Waypoint Driver is known, and is referenced by the pointer 'waypoint_driver'.

```
void OCU_OcuFSM::sendQueryStatusAction()
{
     // Send QueryStatus message to Waypoint Driver Component
     QueryStatus query_status_msg;
     sendJausMessage( query_status_msg, *waypoint_driver );
}
```

In this particular case, the message itself supports no user configured data, so there is no action to populate the relevant data fields. Furthermore, the sendJausMessage() built-in function handles encoding and configuration of the transport wrapper before sending the data to the specified destination.

In our final example, we consider the sendQueryIdentificationAction(). In this case, we don't know the identifier of the destination, so we need to broadcast the message to all available components. Here we can control the scope of the message by using wildcard characters to represent the target subsystem, node, and component identifiers:

```
void OCU_OcuFSM::sendQueryIdentificationAction()
{
     /// Broadcast query_id message so we can find the
     /// Discovery component
     QueryIdentification query_id_msg;
     sendJausMessage( query_id_msg, JausAddress(0xFFFF, 0xFF, 0xFF) );
}
```

In the above examples, we need to communicate with external service interfaces, located either on the same or a remote component. In the case of communicating *between* Finite State Machines *within the same service*, however, internal events represent a simpler approach.

17.6 Sending Internal Events

In previous sections, we examined the implementation necessary to send a message to a service located locally or remotely. In both cases, the input message was part of the service's vocabulary. That is, the receiving service can support the message regardless of the source. In some cases, however, a service may respond to messages, or more accurately *events*, that are *strictly* internally generated. These events are not part of the services external interface (vocabulary) but rather represent triggers internal to the service such as failures, timeouts, etc.

Events are only defined for local (intra-service) signalling, and are invoked through the Internal Event Handler interface, as shown in the following example taken from SkidSteerVehicleSim:

```
     // After initialization, send the Initialized event to ourselves
     ieHandler->invoke(new Initialized());
```

Note that the event instance passed to the invoke method is used to signal the receiving service, and will be freed (de-allocated) upon receipt. Therefore, the memory used for the event must not be deleted from the stack by the caller. The following example will create a system crash when the calling function goes out of scope:

```
 void MyFunction()
 {
     // Never use memory allocated on the call stack for
     // an event.  This will cause a crash (core dump) when
     // the calling function goes out of scope.
     Initialized locallyAllocatedEvent;
     ieHandler->invoke( &locallyAllocatedEvent );  // no!
     return;  // bad things happen here…
 }
```

17.7 Triggering State Transitions

Previous sections in this chapter showed how actions and guards can be implemented with user defined code, including code that can instantiate, populate, and send messages to local or remote destinations. In this section, we'll focus on how those guards and actions are called by the generated code through the use of state transitions.

JTS generated code is inherently event-driven. Communications threads wait for messages or internal events. Once received, the framework passes the message or event to the <service_name>Service class, where the incoming trigger is mapped to the appropriate transition. In most cases, this mapping is created automatically by the code generator and there is no additional user input required. However, when the function contains basic types (unsigned byte, unsigned short, unsigned int, or unsigned long), the code generator cannot automatically determine the value for these arguments. By default, basic types will be uninitialized and will display a warning message at run-time. In order to pass the proper value to the function, the user must manually edit the automatically generated code in <service_name>Service class.

For example, if a transition parameter named 'sender' is given with a type of 'unsigned int', the following code will appear in the <service_name>Service class:

```
unsigned int sender;
printf("WARNING!  Using parameter 'sender' without initialization!\n");
```

Before executing the generated code, the uninitialized type should be replaced with the appropriate definition and the informational print statement removed. This modification to the generated code is only required when transition arguments use primitive types. When possible, use messages and events in transitions, rather than primitive types, as the code generator will automatically create all necessary source code to call these transitions effectively.

Finally, we note that a transition not permitted in a particular state will throw a `statemap::TransitionUndefinedException`. By default, the automatically generated code will catch and ignore this exception. In some instances, users may wish to add information or debug code to the exception handlers to help diagnose run-time issues.

17.8 Dealing with Message Data

The examples used so far have focused on relatively simple messages that have no or few data fields. In this section, we examine more complex message structures, such as working with scaled integers, presence vectors, and lists. The example code in this section is in C++; however, all generated Java and C# code contain the same structures and same functionality as the C++ code. As a result the syntax should be similar enough that it will be trivial to translate the given examples into C# or Java.

The simplest data field element is a fixed field. These represent common data types such as integers (1, 2, or 4 byte), floating point decimal values (4 or 8 byte), as well as strings (variable or fixed length). Fixed fields are used within arrays and records, and the JAUS Tool Set automatically creates get/set accessors for each field using the required data type. As an example, consider the Execute List message used by the OCU example to begin waypoint execution. This message contains two fixed fields, each within a single record which in turn is part of the message body. To set and retrieve the data elements before sending the message, we simply call the appropriate accessors based on the desired nesting level:

```
    ExecuteList execute_msg;
    // command speed 10 m/s
    execute_msg.getBody()->getExecuteListRec()->setSpeed(10.0);
    // start at waypoint 1
    execute_msg.getBody()->getExecuteListRec()->setElementUID(1);
    // debug
    printf(“Sending execute to start at waypoint %d with speed %g\n”,
      execute_msg.getBody()->getExecuteListRec()->getElementUID (),
      execute_msg.getBody()->getExecuteListRec()->getSpeed());
```

Note that even though ‘Speed’ is a scaled integer in this message, the data accessors (both get and set) make this calculation completely transparent. The data field is properly encoded as a four-byte integer for on-the-wire transmissions, but is treated as a double precision floating point in the calling code. To summarize, *there is no user action required to manage scaled integers.*

The ‘set’ method will also perform range checking on a fixed field, if a value_range is specified for it. Consider our previous speed example, which has a permitted range of [0, 327.67] inclusive. The ‘set’ function returns 0 (false) for all calls that exceed this range, and 1 (true) for all calls that succeed. For failures, the internally stored value is not updated, as shown in the following code:

```
    ExecuteList execute_msg;

    // verify success
    if (execute_msg.getBody()->getExecuteListRec()->setSpeed(10.0) == 1)
        printf("Successfully set value!\n");

    // demonstrate failure
    if (execute_msg.getBody()->getExecuteListRec()->setSpeed(1000.0) ==
0)
        printf("Set function returned failure.  Value not changed.\n");

    // debug.  This will print a speed value of 10.
    printf("Current speed setting is %g\n"
        execute_msg.getBody()->getExecuteListRec()->getSpeed();
```

For messages that support optional fields, the JTS automatically inserts a presence_vector with a large enough size to support the number of optional fields within the record. The presence vectors are fully compliant with JSIDL, and are encoded and decoded as part of the message. The use of optional fields and presence vectors is completely automatic for setting data into a message; however, the system designer must exercise caution before using an optional data field from a received message. Consider the Report Local Pose message used by the example Local Pose Sensor. This message reports the position of an unmanned system, but makes use of optional fields to eliminate unneeded fields and reduce bandwidth. Setting data into the message is straightforward:

```
      // Create a populate the ReportLocalPose message
      ReportLocalPose pose_msg;
      pose_msg.getBody()->getLocalPoseRec()->setX(50.0);
      pose_msg.getBody()->getLocalPoseRec()->setY(-50.0);
```

Hence, setting an optional field is identical to setting a required field. On the receive side, however, an optional field must be checked for existence before being accessed. Consider the following case which receives the ReportLocalPose message sent above:

```
      // Decoding the incoming message
      ReportLocalPose pose_msg;
      pose_msg.decode(cmptMsg->getData());

       // Access the fields
      printf("X = %g\n, pose_msg.getBody()->getLocalPoseRec()->getX());
// prints 50.0
```

```
     printf("Y = %g\n, pose_msg.getBody()->getLocalPoseRec()->getY());
// prints -50.0
     printf("Z = %g\n, pose_msg.getBody()->getLocalPoseRec()->getZ());
// prints ????
```

In this snippet, the code attempts to access the position for X, Y, and Z. However, only X and Y have been set. Since Z is an optional field and not set, the return value is undefined. To prevent these types of errors, the generated code automatically includes an is*Valid() method that returns a Boolean for each data element. If this function returns true, the optional field is, in fact, present within the message and can be safely accessed. A better solution to our Report Local Pose processing function therefore uses these functions before accessing any optional fields:

```
     // Decoding the incoming message
     ReportLocalPose pose_msg;
     pose_msg.decode(cmptMsg->getData());

      // Access the fields, checking for existence first
     if (pose_msg.getBody()->getLocalPoseRec()->isXValid())
        printf("X= %g\n, pose_msg.getBody()->getLocalPoseRec()->getX());
// prints 50.0

     if (pose_msg.getBody()->getLocalPoseRec()->isYValid())
        printf("Y= %g\n, pose_msg.getBody()->getLocalPoseRec()->getY());
// prints -50.0

     if (pose_msg.getBody()->getLocalPoseRec()->isZValid())
        printf("Z= %g\n, pose_msg.getBody()->getLocalPoseRec()->getZ());
// doesn't print
```

The JTS Code Generator is capable of supporting any of the complex fields, not just simple records as used in the above example. Of these types, List is the most common and may include records, sequences, or even other lists. Because of their wide ranging applicability, the next examples will cover messages containing lists extensively.

Unlike arrays which are fixed dimension lists of fixed fields, a list is variable length. As a result, the Code Generator creates the list within the message but does not initialize any members. In Java and C#, java.util.ArrayList<T> and System.Collections.List<T> are used to provide the variable size behaviour required. To put data into the list, we need to create the element and then add it using the generated list accessor functions. Consider the following example, in which a service populates a Register Services message to send to the Discovery component.

```
     // Create the register services message
     RegisterServices register_msg;

     // Each list element is a Service Rec.  Create one for each
     // service to be added
     // and set the name, major version, and minor version
     RegisterServices::RegisterServicesBody::ServiceList::ServiceRec
service1;
     service1.setMinorVersionNumber(0);
     service1.setMajorVersionNumber(1);
     service1.setURI("urn:jaus:jss:mobility:LocalWaypointListDriver");

     // … and create another one for an alternative service
     // being offered
     RegisterServices::RegisterServicesBody::ServiceList::ServiceRec
service2;
     service2.setMinorVersionNumber(0);
     service2.setMajorVersionNumber(1);
     service2.setURI("urn:jaus:jss:core:Management");

     // … and a third for good measure.
     RegisterServices::RegisterServicesBody::ServiceList::ServiceRec
service3;
     service3.setMinorVersionNumber(0);
     service3.setMajorVersionNumber(1);
     service3.setURI("urn:jaus:jss:core:Events");

     // Now we can add each Service Rec element to our list.
     // Using the addElement()
     // function, each new element is added at the end of the
    // existing list.
     register_msg.getRegisterServicesBody()->getServiceList()->
            addElement(service1);
     register_msg.getRegisterServicesBody()->getServiceList()->
            addElement(service2);
     register_msg.getRegisterServicesBody()->getServiceList()->
            addElement(service3);

     // encode and send normally.
     register_msg.encode(buffer);
```

Several additional functions help us extract data from a list. For example, getNumberOfElements() returns the current list size (count), while the getElement(int index) function will return a particular elements based on a zero-indexed array. The following code displays a list of all services contained in our Register Services message:

```
for (int i=0;
     i < register_msg.getRegisterServicesBody()->
```

```
         getServiceList()->getNumberOfElements();
      i++)
{
    // Get a pointer to the ServiceRec list element, using the
    // counter i as our index
    RegisterServices::RegisterServicesBody::ServiceList::ServiceRec*
service =
          register_msg.getRegisterServicesBody()->
             getServiceList()->getElement(i);

    // Once we have a record, we can pull data out using the
    // standard accessors
    printf("Found service: %s    version %d.%d\n",
         service->getURI().c_str(), service->getMajorVersion,
         service->getMinorVersion());
}
```

Finally, we examine a more complex case that uses multiple nesting levels, as well as lists containing sequences. In this scenario, the OCU uses a Set Element message to send a set of waypoints to the Local Waypoint List Driver. Each waypoint is first encoded in a Set Local Waypoint message, which in turn is added as a variable_format_field to the Set Element message. The implementation to encode a Set Local Waypoint message is straight-forward, and demonstrates previous lessons:

```
     SetLocalWaypoint waypoint;
     waypoint.getBody()->getLocalWaypointRec()->setX( 100.0 );
     waypoint.getBody()->getLocalWaypointRec()->setY( -100.0 );
     waypoint.getBody()->getLocalWaypointRec()->setWaypointTolerance(2);
     waypoint.encode(buffer);
```

This code sets a waypoint using local coordinates at (100, -100) and specifies a tolerance of 2 meters; that is, the unmanned system should get within 2 meters for the waypoint to be considered achieved. This encoded message now becomes the data stored in the variable_format_field of the ElementRec structure, which also includes the Element ID, the Previous ID and the Next ID, much like the entries in a doubly-linked list.

```
     SetElement::Body::SetElementSeq::ElementList::ElementRec element;
     element.setElementUID(1);  // first element in the list
     element.setPreviousUID(0); // previous element is zero (not de-
fined)
     element.setNextUID(2);     // next element is the second waypoint

     // To set the variable_format_field data, the accessor
     //  includes three parameters:
     // 1) The field type.  For this message, 0 = an
```

```
      // encoded JAUS message
      // 2) The size of the encoded message
      // 3) The byte stream for the encoded message
      element.getElementData()->set(0, waypoint.getSize(), buffer);
```

With our newly created element, we can now add it to the element list. Note that the list itself is stored within a sequence, and we must set the other values associated with the sequence; in this case, we set the Request ID:

```
    // Create the list request
    SetElement set_element_msg;

    // Access the RequestID stored in a record owned by the sequence
    set_element_msg.getBody()->getSetElementSeq()->getRequestIDRec()->
       setRequestID(1);

    // Finally, add the element created above to the list owned by the
sequence
    set_element_msg.getBody()->getSetElementSeq()->
     getElementList()->addElement(element);
```

While this sample only populates a single waypoint in the list, a fully realized implementation would simply continue to create more elements following the process above, and adding them to the list using addElement(). The OCU_OcuFSM.cpp shows this in sendWaypointsAction() function.

17.9 Configuring the Run-Time Environment

In previous sections, the emphasis was on adding functionality to the generated code in the form of triggered transitions, actions, and guards. This section will describe how the Software Framework can be configured such that running components can exchange messages either within the same node (processor) or in a distributed environment.

Note: The Software Framework currently included with the JAUS Tool Set supports node-to-node communication through the use of Jr Middleware. The middleware provides numerous configuration options to customize UDP, TCP, and/or serial performance; however, such configurations are beyond the scope of this document. For additional information, please consult www.jrmiddleware.org.

At start-up, each component is given a 4-byte identifier, called a JAUS Identifier or JAUS Address, which must be globally unique for all communicating components. This identifier is broken into three parts: the Subsystem ID (an unsigned short), the Node ID (an unsigned byte), and the Component ID (an unsigned byte). For convenience, this is represented as a series of three decimal values separated by colons, e.g. <subsystem>:<node>:<component>. Generally, anything with the same Subsystem ID should exist on the same platform; for example, the same robot, controller, or payload. Similarly, components with the same Node ID should exist on the same node, e.g. processor.

The JAUS Address is specified when a component is initialized in Main class of the generated code. By default, every component gets an address of 126:1:<component_id> where the component_id is set by the user when defining the component in the JTS UI. The main entry point for our example OCU is generated as follows:

```
#include <iostream>
#include "ExampleOCU.h"

using namespace std;

int main(int argc, char *argv[])
{
    // Instantiate the component and start it.
    ExampleOCU* cmpt = new ExampleOCU(126, 1, 200);

    // Catch exit signals
    signal( SIGINT, handle_exit_signal );
    signal( SIGTERM, handle_exit_signal );
    signal( SIGABRT, handle_exit_signal );

    // Start the component and the services
    cmpt->startComponent();

    // Wait until signaled to exit
    exit_signal.wait();

    // Shutdown the component and threads
    cmpt->shutdownComponent();

    // Give a little time for proper shutdown
    DeVivo::Junior::JrSleep(100);

    // Free the component
    delete cmpt;
}
```

The identifier for any component can be altered simply by changing the parameters to the component constructor. To change the OCU to use subsystem ID 105 and node 2:

```
ExampleOCU * cmpt = new ExampleOCU(105, 2, 200);
```

In order to help route messages both within a node and to external (remote) nodes, the JTS Software Framework includes a Node Manager (NM). The NM is responsible for all incoming and outgoing UDP traffic on the JAUS port, assigned by the IANA as 3794. The NM can be found in the JTS install, under the 'nodeManager' directory. Generally, the NM source code will not need to be modified.

All generated languages connect to the same NM, because they all use the C++ Software Framework to connect. As a result, the included C++ NM is run regardless of what language the generated code is in.

To run the Node Manager, simply build using 'scons' and run the executable from the bin directory:

```
$> cd trunk/JAUSToolset/nodeManager/

$> scons
scons: Reading SConscript files ...
scons: Building for CYGWIN...
scons: done reading SConscript files.
scons: Building targets ...
scons: `.' is up to date.
scons: done building targets.

$ cd bin

$ ./NodeManager nm.cfg
```

When the NM receives a message either from a component within the node or a component on a different node, it routes the message to the desired recipient, if known. To do this routing, the NM maintains a mapping of JAUS identifiers to source IP address and port that is updated for all incoming messages. When a local component is created, the framework automatically "registers" with the NM in order to update this mapping.

The default configuration file, nm.cfg, offers optimal settings for most environments using UDP communications. While a complete discussion of configuration options is beyond the scope of this document, we do note the existence of selective logging levels is LogMsgLevel. This output may be useful in trouble shooting communications problems.

17.10 Summary

In this Chapter, we examined the C++, Java, and C# code generated by the JTS. While the messages are completely implemented by the generator, the user must add the message handlers, actions, and guards that define the service's behaviour. The examples offered previously are selected for their simplicity in demonstrating common techniques and approaches. When working with services that rely on inheritance and/or nested states, the generated function stubs can become complex and confusing. However, the same guidance can be applied to these cases. For additional suggestions on dealing with complex behaviour, please see the JAUS Tool Set forum at http://forums.jaustoolset.org/index.php.

18 Document Generator

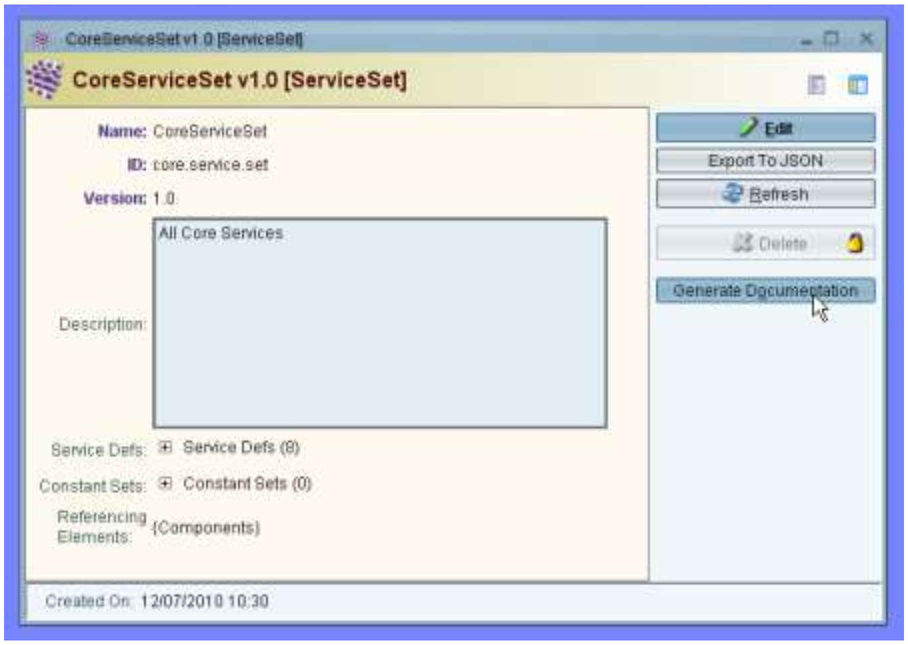
JTS provides a documentation tool that can output service specifications in several different types/formats:

1. Framed HTML, resembling Javadoc in structure
2. Linear HTML, a single long HTML page
3. Microsoft Office Open XML Document (Word .docx) format

Currently, documents may only be generated for Service Sets. The generated document details the Service Definitions in the selected Service Set.

18.1 Document Generation User Interface

To perform document generation, click the “Generate Documentation” button on a Service Set’s editor window (Figure 120), or click “Generate Documentation” on the context menu after right-clicking a Service Set’s icon in a list containing Service Sets (Figure 121).



CoreServiceSet v1.0 [ServiceSet]
Name: CoreServiceSet
ID: core.service.set
Version: 1.0
Description:
All Core Services
Service Defs: Service Defs (8)
Constant Sets: Constant Sets (0)
Referencing Elements: (Components)
Created On: 12/07/2010 10:30
Edit
Export To JSON
Refresh
Delete
Generate Documentation

Figure 120: Generate Documentation via ServiceSet Window

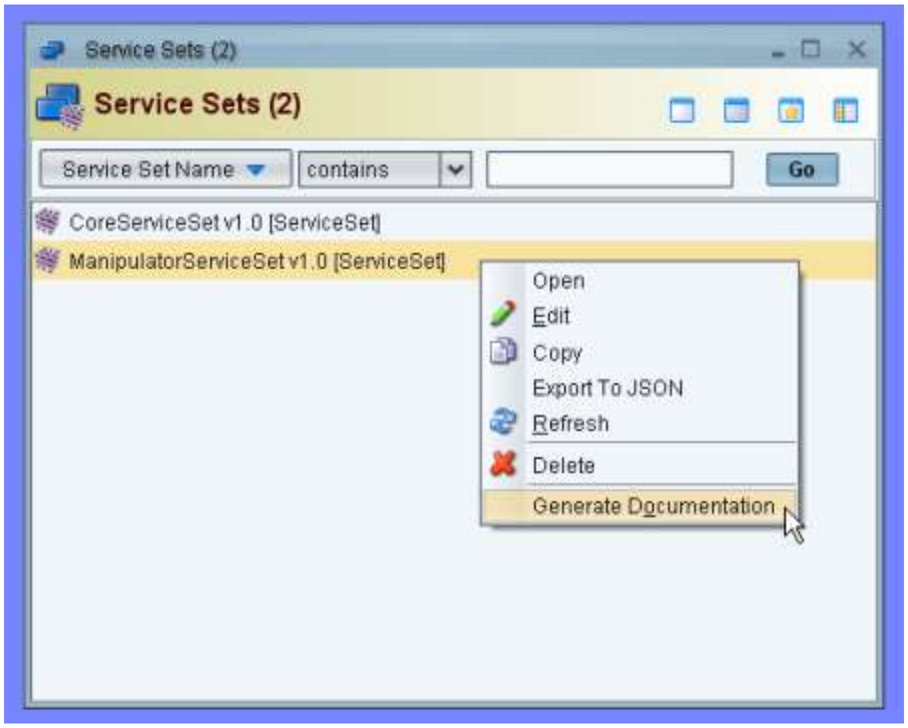
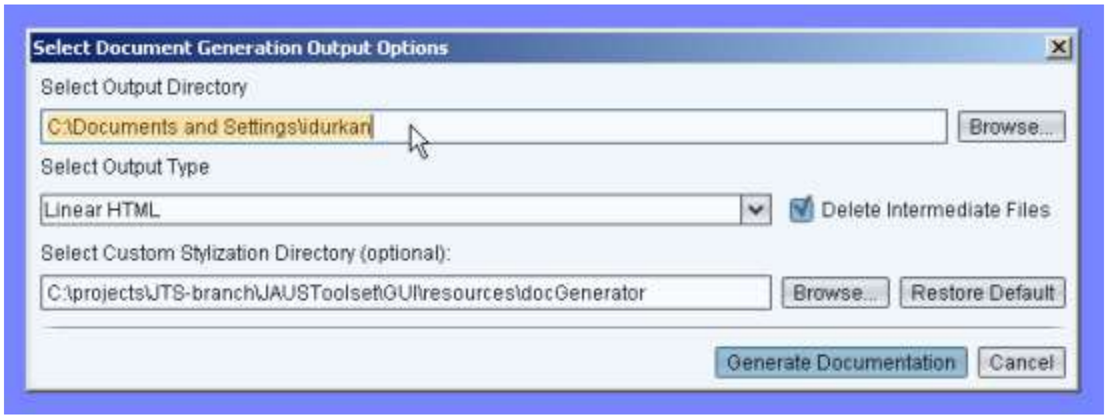


Figure 121: Generate Documentation via Context Menu

After clicking “Generate Documentation” the “Select Documentation Generation Output Options” dialog will appear. This dialog lets the user choose what type of documentation is generated, and the directory where document generation output will go. A custom stylization directory can also be selected. ***Section 18.2 provides more information about this directory and what it does.*** The “Delete Intermediate Files” checkbox controls whether various intermediate files created during document generation are removed after generation is complete. For a large service set, these files will be correspondingly large, so deletion saves space. See Figure 122 for an example of the dialog in action.



Select Output Directory: C:\Documents and Settings\durkan

Select Output Type: Linear HTML

✓ Delete Intermediate Files

Select Custom Stylization Directory (optional): C:\projects\JTS-branch\JAUSToolset\GUI\resources\docGenerator

Figure 122: Output Options Dialog

The Output Directory text field defaults to the current user’s home directory and the Custom Stylization Directory defaults to the location of JTS’ standard document generation stylization directory. The Output Type defaults to Linear HTML. Click “Restore Default” to reset the custom stylization directory to the standard one. The Browse button for each text field pops up a Select Directory browsing dialog to ease selecting a directory

Once satisfied with the output directory, output-type, and custom stylization directory, click “Generate Documentation”. Note both the Output Directory and Custom Stylization directory text fields must be populated with a directory that actually exists before document generation is allowed. If either directory does not exist, an error popup appears when clicking “Generate Documentation” and the user is allowed to fill in a valid directory.

18.2 Custom Stylization Directories

The Custom Stylization Directory selected via the Output Options Dialog contains all of the files used during the document generation process, including document templates, static image files, CSS stylesheets for styling HTML, and the XSLT stylesheets that transform different representations of Service Definitions into their final output form. The Document Generation Stylization Customization Guide, a separate document, gives instructions regarding the creation and development of Custom Stylization Directories to customize document generation output. **Consult the Document Generation Stylization Customization Guide for more information, including customization examples.**

18.3 Documentation Generation-General Behavior

After clicking “Generate Documentation”, all required static files are copied to the output directory, and the document generation process is performed. Although subdirectories will be created, the main documentation file is located at the top level of the output directory. The table indicates how the output is named. <service_set_name> represents the name of the Service Set selected for document generation.

Table 5: Output File Names

Output Type	**Main Output File Name**
Linear HTML	• `<service_set_name>_output.html`
• Framed HTML	• `index.html`
• Word .docx	• `<service_set_name>_output.docx`

Both of the HTML-type outputs make assumptions about the location of included files relative to the output document, so when packaging documentation for distribution, ensure the directory structure is maintained. The table below summarizes which files must be kept in the same directory as the main output file, for each output format:

Table 6: Files to Keep with Output

Output Type	**Keep what files in position?**
Linear HTML	• images/ directory linearStyleSheet.css
• Framed HTML	• Everything
• Word .docx	• Nothing - .docx is self-contained

For the documentation to be truly useful, descriptions for services should be filled out, as well as interpretations for the various message components.

18.4 XMLMind XFC Dependency

In order to generate Word .docx format documentation, the document generator requires XMLMind's XSL-FO Converter (XFC). XFC is a closed-source product, but a free Personal Edition is available. Note the Personal Edition will add a watermark in the footer of the pages it generates.

JTS provides a "dummy" version of the XFC Java library inside the `lib/runtime` directory below the directory where JTS.jar is located. The jarfile is named `xfc-stubs.jar`. The XFC download should provide a file `xfc.jar`; this file contains the XFC Java library. To begin using the real version of XFC, remove `xfc-stubs.jar` from the `lib/runtime` directory and copy `xfc.jar` to the same location. ***If a real version of xfc.jar is not added to lib/runtime, or some other location on the CLASSPATH when running JTS, and the dummy is not***

removed, no Word output will be generated. Console output will be displayed indicating the dummy code was run.

18.5 Behavior Diagrams in Generated Documentation

The document generator adds diagrams illustrating protocol behavior for each service documented. However, the protocol behavior diagram needs to be formatted elsewhere in JTS for each service, or else only a placeholder protocol behavior diagram image will be output for the service in question.

To format a service's protocol behavior diagram, open the Service Definition in the Service Set being documented; open its Protocol Behavior, then save the behavior diagram to file. The file can be saved anywhere, with any name. Section 8.3 "Describe the Protocol" describes how to access the Protocol Behavior for a Service Definition; in the protocol behavior editor, the Save button is found in the toolbar. After the save is performed, the diagram is properly formatted and will not be replaced by a placeholder in generated documentation.

18.6 Navigating Linear HTML and Word Documentation

The Linear HTML and Word documentation types have a table of contents that provides links to locations in the rest of the document. When using the standard document generation stylization directory's template documents, generated Linear HTML and Word documents have the following structure:

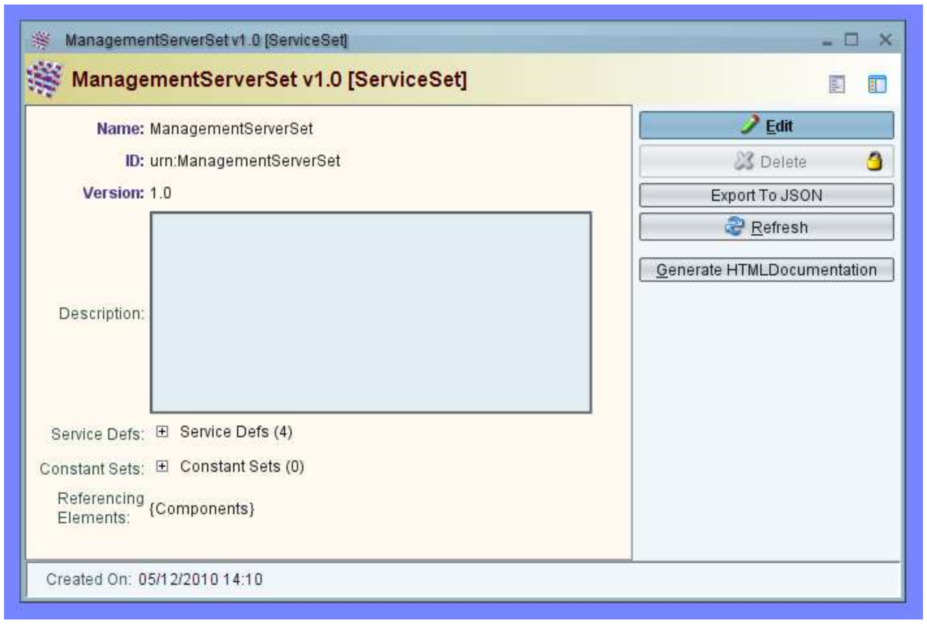
- Title Page & Introduction *(Static text from template document)*
- Table of Contents, Table of Figures, Table of Tables
- Section 1: Scope: *(Static text from template document)*
- Section 2: References: *(Static text from template document)*
- Section 3: Common Conventions: *(Static text from template document)*
- Section 4: Service Definitions: Contains information about each service definition, including message sets and protocol behavior

- Section 5: Notes: *(Static text from template document)*
- Appendix A: JSIDL code listings for each Service Definition.

The ***Document Generation Custom Stylization Guide*** describes how to customize the template documents and alter the output document's structure for Linear HTML-type and Word-type document generation.

18.7 Navigating Framed HTML Documentation

The Framed HTML documentation type is more complex in appearance than Linear HTML and Word output, and works similarly to Javadoc documentation. Framed HTML JTS provides a HTML documentation tool that outputs service specifications in a presentable HTML format. In release 1.0 of the tool, the HTML document generator can only be executed from a Service Set internal frame. In the figure below the button called "Generate HTML Documentation" will generate the output documentation at a user specified path.


ManagementServerSet v1.0 [ServiceSet]

ManagementServerSet v1.0 [ServiceSet]

Name: ManagementServerSet

ID: urn:ManagementServerSet

Version: 1.0

Description:

Service Defs: ⊞ Service Defs (4)

Constant Sets: ⊞ Constant Sets (0)

Referencing Elements: {Components}

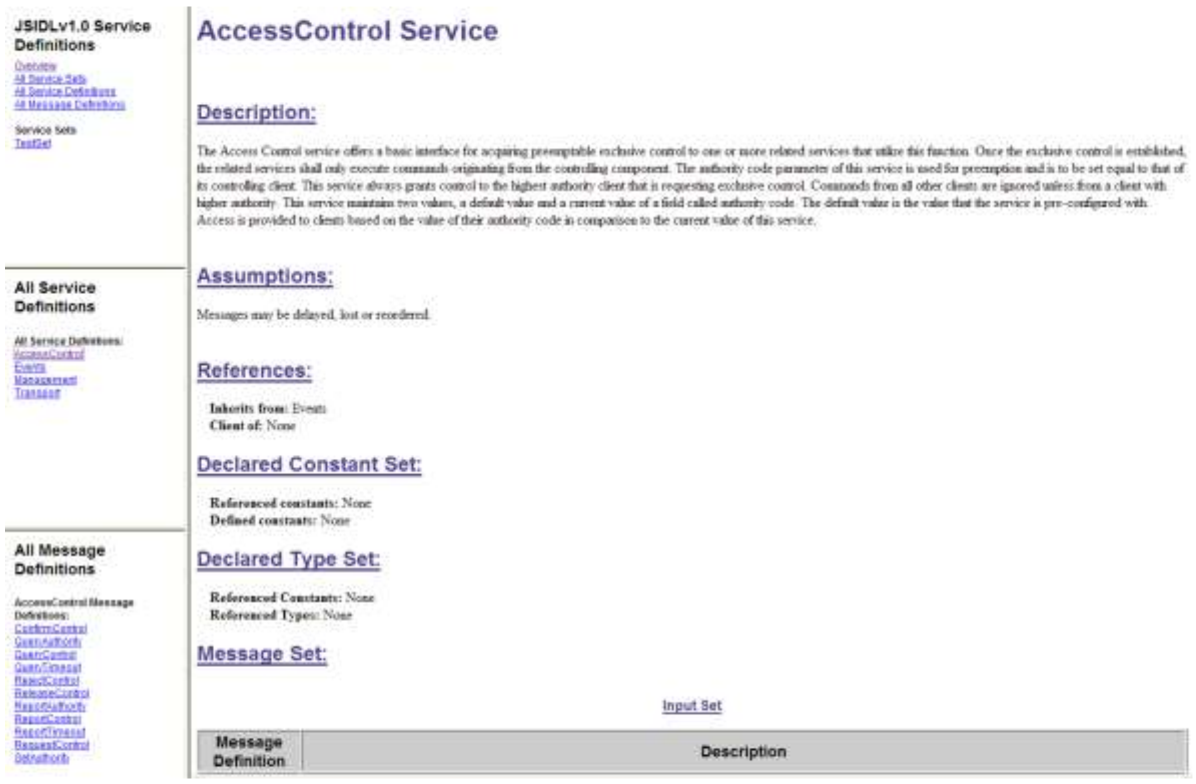
Created On: 05/12/2010 14:10

Figure 123: HTML Documentation Generation

The output contains four frames. The frames on the left list menus for navigating content:

- Upper left: Navigate service set
- Center left: Navigate service definitions
- Bottom left: Navigate message definitions

The large frame to the right is used to display content, such as service sets, message sets, service definitions, and message definitions. Figure 124 provides an example.



JSIDLv1.0 Service Definitions

Overview
All Service Sets
All Service Definitions
All Message Definitions

Service Sets
TestSet

All Service Definitions

All Service Definitions:
AccessControl
Events
Management
Transport

All Message Definitions

AccessControl Message Definitions:
ConfirmControl
QueryAuthority
QueryControl
QueryTimeout
RejectControl
ReleaseControl
ReportAuthority
ReportControl
ReportTimeout
RequestControl
SetAuthority

AccessControl Service

Description:

The Access Control service offers a basic interface for acquiring preemptable exclusive control to one or more related services that utilize this function. Once the exclusive control is established, the related services shall only execute commands originating from the controlling component. The authority code parameter of this service is used for preemption and is to be set equal to that of its controlling client. This service always grants control to the highest authority client that is requesting exclusive control. Commands from all other clients are ignored unless from a client with higher authority. This service maintains two values, a default value and a current value of a field called authority code. The default value is the value that the service is pre-configured with. Access is provided to clients based on the value of their authority code in comparison to the current value of this service.

Assumptions:

Messages may be delayed, lost or reordered.

References:

Inherits from: Events
Client of: None

Declared Constant Set:

Referenced constants: None
Defined constants: None

Declared Type Set:

Referenced Constants: None
Referenced Types: None

Message Set:

Input Set

Message Definition	Description

Figure 124: HTML Documentation Output

19 Wireshark Plugin

The Wireshark plug-in was created to allow users to use the popular network analyzer tool to monitor JAUS message traffic in real time. This plug-in captures JAUS messages and provides detailed information about their data content in the format defined by the JSIDL message specification. The following sub-sections describe how the plug-in is to be built and executed.

19.1 Installing JAUS Plug-in

19.1.1 Download and Install Wireshark

Download Wireshark from http://www.wireshark.org/download/win32/wireshark-win32-1.4.0.exe.

Note: NOTE: Libraries included in the JTS distribution are only tested with Wireshark 1.4.0. Building from source to work with a different version is beyond the scope of this Guide. Please reference the Wireshark development guide at http://www.wireshark.org/docs/wsdg_html_chunked/.

19.1.2 Copy and Move files

Copy the following files:

- jaus.dll to <Wireshark installation target directory>\plugins\[version number]\
- libxml2.dll, iconv.dll, and zlib1.dll to the <Wireshark installation target directory>

Make a directory or copy the folders if supplied to <Wireshark installation target directory>

- packet-jaus-support\jausxml\

The dissector is assuming that the xml files of jaus messages are in the packet-jaus-support\jausxml\ folder, copy the xml files to here.

19.2 Using JAUS dissector

19.2.1 Run Wireshark.exe

Note: If running Windows 7, you may need to run wireshark in administrative mode because the plugin needs write access to the program files directory.

Wireshark will load the plug-in on start-up.

Check to see if it is loaded by typing “jaus” in the filter box which should then turn green.

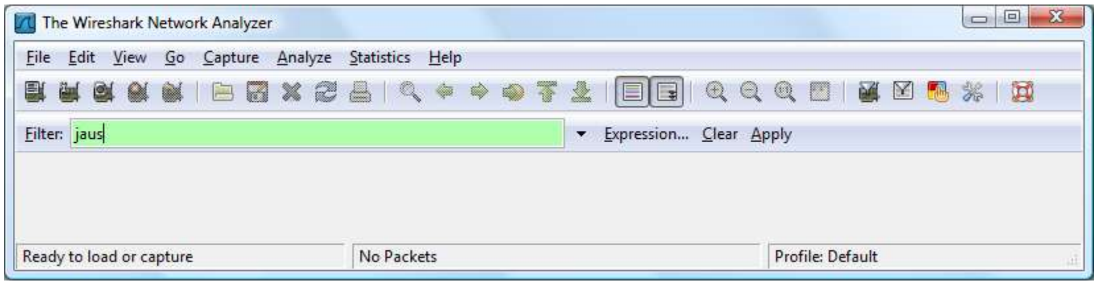

The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Help
Filter: jaus
Expression... Clear Apply
Ready to load or capture
No Packets
Profile: Default

Figure 125: Wireshark Network Analyzer

The example capture file JAUS_port_example_run.pcap can be loaded into wireshark (File>open) to show the dissector working. This example already has all other packets that were not jaus filtered out but using the jaus filter will do this on any capture.

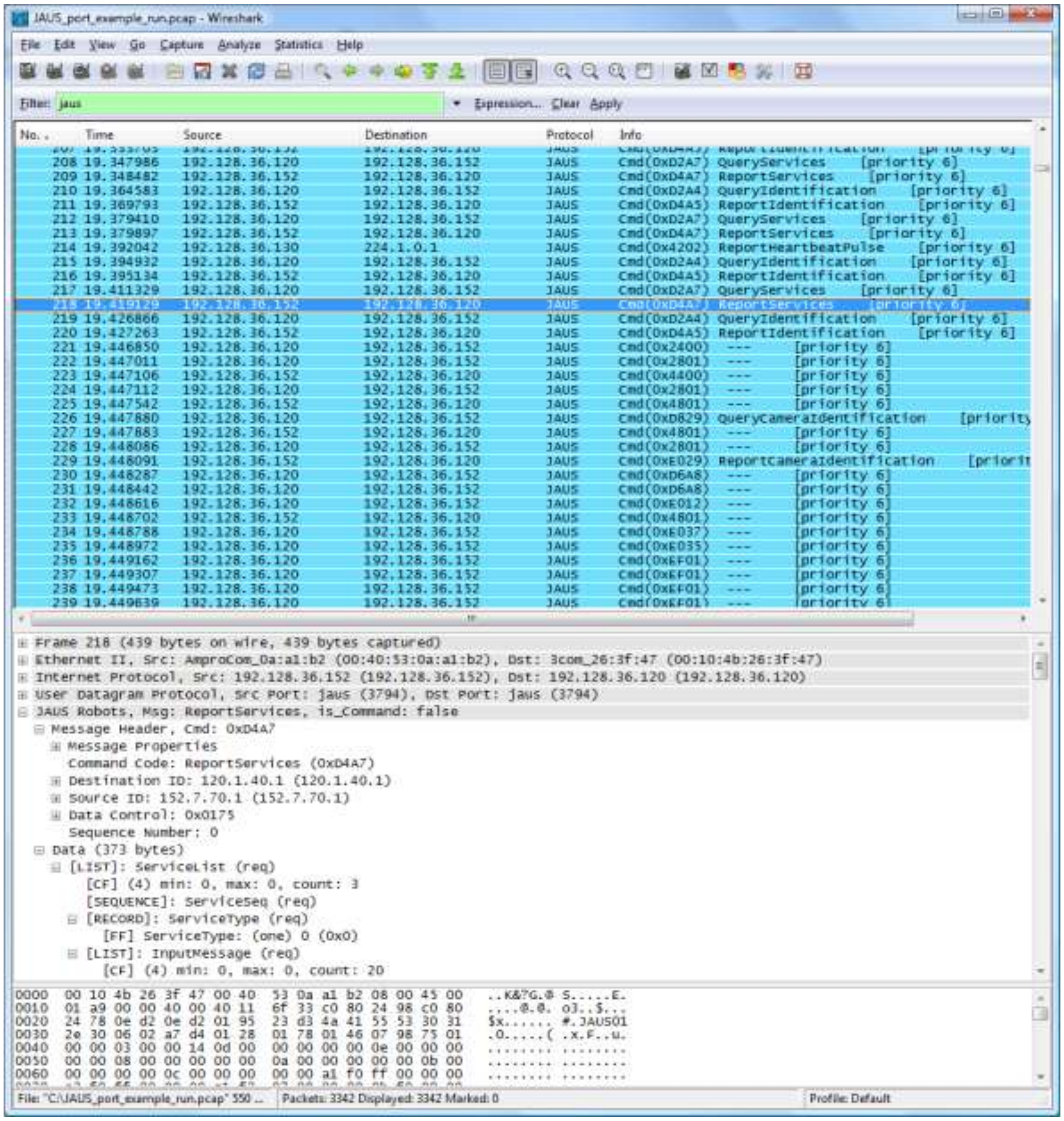
The ports on which the jaus filter will register with Wireshark can be changed in the preferences (Edit > Preferences – Find the JAUS protocol after expanding the protocols list). The two options are to change the standard JAUS port for TCP/UDP from the default of 3794 and the UDP port on which intra-node messages will be on when intra-node forwarding is enabled in Junior Node Manager.

Enable intra-node message forwarding in the node manager configuration file using the following as an example:

```
<UDP_Loopback_Configuration
    EnableLoopback   = "1"
    UseOPC2.75_Header = "0"
    UDP_Address      = "169.255.0.100"
    UDP_Port         = "55555"
    MulticastTTL     = "1"
    MulticastAddr    = "239.255.0.100"
    MaxBufferSize    = "70000"
/>
```

This will enable the loopback using the JUDP header sending messages to an unused network IP of 169.255.0.100 on the port 55555.

The address can be another computer on the network that you want to see the messages with or the loopback address of 127.0.0.1 that will be viewable in Wireshark on Linux, not Windows. If you want to use a Multicast Address, set EnableLoopback to “2”.


JAUS_port_example_run.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: jaus
Expression... Clear Apply

```
No. .  Time       Source            Destination       Protocol  Info
  208 19.347986  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD2A7) QueryServices      [priority 6]
  209 19.348482  192.128.36.152    192.128.36.120    JAUS      Cmd(0xD4A7) ReportServices      [priority 6]
  210 19.364583  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD2A4) QueryIdentification     [priority 6]
  211 19.369793  192.128.36.152    192.128.36.120    JAUS      Cmd(0xD4A5) ReportIdentification     [priority 6]
  212 19.379410  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD2A7) QueryServices      [priority 6]
  213 19.379897  192.128.36.152    192.128.36.120    JAUS      Cmd(0xD4A7) ReportServices      [priority 6]
  214 19.392042  192.128.36.130    224.1.0.1         JAUS      Cmd(0x4202) ReportHeartbeatPulse     [priority 6]
  215 19.394932  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD2A4) QueryIdentification     [priority 6]
  216 19.395134  192.128.36.152    192.128.36.120    JAUS      Cmd(0xD4A5) ReportIdentification     [priority 6]
  217 19.411329  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD2A7) QueryServices      [priority 6]
  218 19.419129  192.128.36.152    192.128.36.120    JAUS      Cmd(0xD4A7) ReportServices      [priority 6]
  219 19.426866  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD2A4) QueryIdentification     [priority 6]
  220 19.427263  192.128.36.152    192.128.36.120    JAUS      Cmd(0xD4A5) ReportIdentification     [priority 6]
  221 19.446850  192.128.36.120    192.128.36.152    JAUS      Cmd(0x2400)  ---      [priority 6]
  222 19.447011  192.128.36.120    192.128.36.152    JAUS      Cmd(0x2801)  ---      [priority 6]
  223 19.447106  192.128.36.152    192.128.36.120    JAUS      Cmd(0x4400)  ---      [priority 6]
  224 19.447112  192.128.36.120    192.128.36.152    JAUS      Cmd(0x2801)  ---      [priority 6]
  225 19.447542  192.128.36.152    192.128.36.120    JAUS      Cmd(0x4801)  ---      [priority 6]
  226 19.447880  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD829) QueryCameraIdentification     [priority
  227 19.447883  192.128.36.152    192.128.36.120    JAUS      Cmd(0x4801)  ---      [priority 6]
  228 19.448086  192.128.36.120    192.128.36.152    JAUS      Cmd(0x2801)  ---      [priority 6]
  229 19.448091  192.128.36.152    192.128.36.120    JAUS      Cmd(0xE029) ReportCameraIdentification     [priorit
  230 19.448287  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD6A8)  ---      [priority 6]
  231 19.448442  192.128.36.120    192.128.36.152    JAUS      Cmd(0xD6A8)  ---      [priority 6]
  232 19.448616  192.128.36.120    192.128.36.152    JAUS      Cmd(0xE012)  ---      [priority 6]
  233 19.448702  192.128.36.152    192.128.36.120    JAUS      Cmd(0x4801)  ---      [priority 6]
  234 19.448788  192.128.36.120    192.128.36.152    JAUS      Cmd(0xE037)  ---      [priority 6]
  235 19.448972  192.128.36.120    192.128.36.152    JAUS      Cmd(0xE035)  ---      [priority 6]
  236 19.449162  192.128.36.120    192.128.36.152    JAUS      Cmd(0xEF01)  ---      [priority 6]
  237 19.449307  192.128.36.120    192.128.36.152    JAUS      Cmd(0xEF01)  ---      [priority 6]
  238 19.449473  192.128.36.120    192.128.36.152    JAUS      Cmd(0xEF01)  ---      [priority 6]
  239 19.449839  192.128.36.120    192.128.36.152    JAUS      Cmd(0xEF01)  ---      [priority 6]
```

```
⊞ Frame 218 (439 bytes on wire, 439 bytes captured)
⊞ Ethernet II, Src: AmproCom_0a:a1:b2 (00:40:53:0a:a1:b2), Dst: 3com_26:3f:47 (00:10:4b:26:3f:47)
⊞ Internet Protocol, Src: 192.128.36.152 (192.128.36.152), Dst: 192.128.36.120 (192.128.36.120)
⊞ User Datagram Protocol, Src Port: jaus (3794), Dst Port: jaus (3794)
⊟ JAUS Robots, Msg: ReportServices, is_Command: false
  ⊟ Message Header, Cmd: 0xD4A7
    ⊞ Message Properties
      Command Code: ReportServices (0xD4A7)
    ⊞ Destination ID: 120.1.40.1 (120.1.40.1)
    ⊞ Source ID: 152.7.70.1 (152.7.70.1)
    ⊞ Data Control: 0x0175
      Sequence Number: 0
  ⊟ Data (373 bytes)
    ⊟ [LIST]: ServiceList (req)
        [CF] (4) min: 0, max: 0, count: 3
        [SEQUENCE]: ServiceSeq (req)
      ⊟ [RECORD]: ServiceType (req)
          [FF] ServiceType: (one) 0 (0x0)
      ⊟ [LIST]: InputMessage (req)
          [CF] (4) min: 0, max: 0, count: 20
```

```
0000  00 10 4b 26 3f 47 00 40  53 0a a1 b2 08 00 45 00   ..K&?G.@ S.....E.
0010  01 a9 00 00 40 00 40 11  6f 33 c0 80 24 98 c0 80   ....@.@. o3..$...
0020  24 78 0e d2 0e d2 01 95  23 d3 4a 41 55 53 30 31   $x...... #.JAUS01
0030  2e 30 06 02 a7 d4 01 28  01 78 01 46 07 98 75 01   .0.....( .x.F..u.
0040  00 00 03 00 00 14 0d 00  00 00 00 00 0e 00 00 00   ........ ........
0050  00 00 08 00 00 00 00 00  0a 00 00 00 00 00 0b 00   ........ ........
0060  00 00 00 00 0c 00 00 00  00 00 a1 f0 ff 00 00 00   ........ ........
```

File: "C:\JAUS_port_example_run.pcap" 550 ... Packets: 3342 Displayed: 3342 Marked: 0 Profile: Default

Figure 126: Wireshark Protocol Traces with Packet Dissection

19.3 Things to Know

The header is dissected whether or not the message is in the xml. If the message is not in the xml, a --- is reported as the name in the list and NotFoundInXML in the tree and the data is left alone. A file is created in the packet-jaus-support/ folder named Messages_Found.txt to show/check that the xml parser found messages.

The filter can be used to filter out certain messages or IDs by right clicking on the item in the tree and click “Apply as Filter” or type in a filter in the box (names at bottom left when a tree item is selected). Examples:

```
jaus.command == 0x4202

jaus.dest == 152.7.70.1
```

20 Protocol Validation

PROMELA (Protocol Meta Language) source code can be generated and used for validation of the protocols defined for services.

20.1 Overview

Formal validation of a software model saves time that could be wasted implementing a model that does not work properly. Validation begins by generating PROMELA code that is used to describe a protocol or state machine. The PROMELA code is then interpreted by SPIN, which allows for validation of the protocol. In addition SPIN can be used to run simulations of the model to ensure that it behaves as expected.

20.2 Environment

The list of supported systems is provided below.

	PROMELA Generation	SPIN Validation via
Windows	XP	XP via JSpin v.5.0 (SPIN v.6.0)
Linux	Ubuntu 10.04	Ubuntu 10.04 via JSpin v.4.7(SPIN v.5.2.5)
Cygwin	Not Tested	Not Tested
OS X	Not Tested	Not Tested

There are no additional packages required for generating PROMELA source code. See section 3.2 for all JTS required packages.

20.3 Tools

SPIN Model Checker: The main tool for validation is SPIN, a popular open-source software tool used for formal verification of distributed software systems. The SPIN Model Checker can be found at http://spinroot.com

jSpin: Throughout this manual, jSpin will be used in all the examples. jSpin is a Java GUI that is wrapped around SPIN. The results you get from SPIN and jSpin should be identical as long as all the options selected are the same.

20.4 Workflow

The service needs to be defined using the methods described in the previous sections. Once the JSIDL for the service has been produced or the service set has been defined in JTS, there are three steps to perform: generate PROMELA code, modify the generated code, and use jSpin to validate the model.

20.5 Installing and Getting Started

20.5.1 Obtaining and Installing the PROMELA Code Generator

The PROMELA code generator is installed as part of JTS. See section 4 for the details.

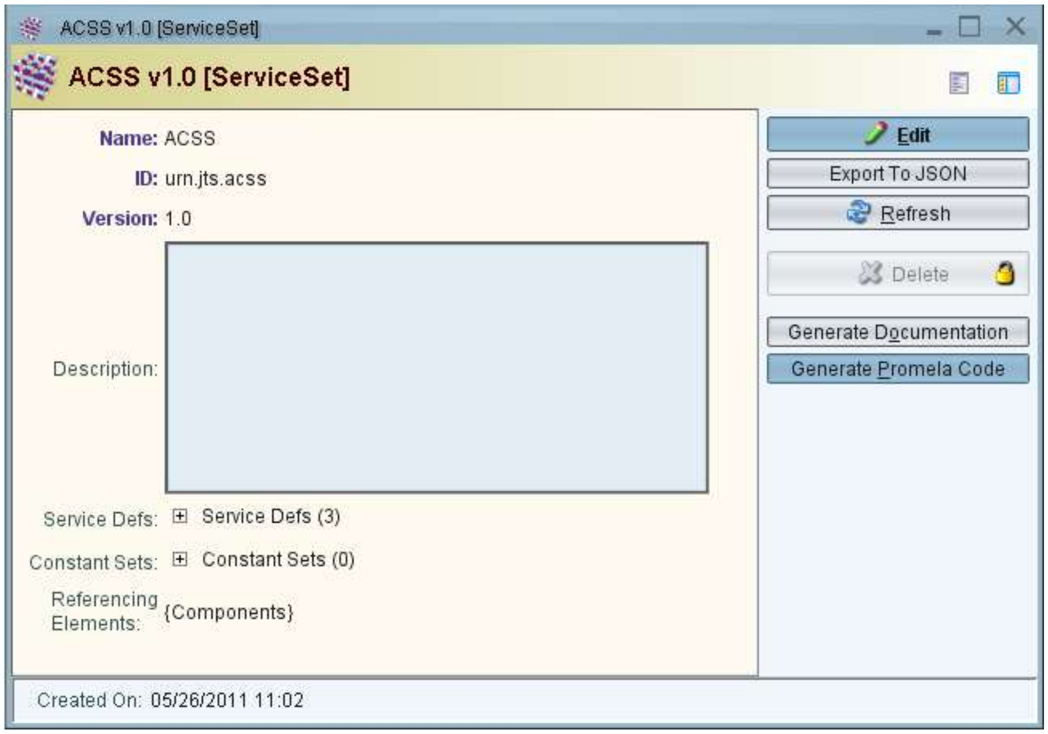
20.5.2 Ant Targets

The targets for the PROMELA code generator can be found in the table from section 4.3.

20.5.3 Generating PROMELA Source Code

Generating the source code can be done two different ways.

First, the user can generate source code from the JTS GUI. This is accomplished by opening the component or service set and clicking the “Generate Promela Code” button.


ACSS v1.0 [ServiceSet]

Name: ACSS

ID: urn.jts.acss

Version: 1.0

Description:

Service Defs: ⊞ Service Defs (3)

Constant Sets: ⊞ Constant Sets (0)

Referencing Elements: {Components}

Created On: 05/26/2011 11:02

Figure 127 Generate PROMELA Source Code

This can be done for a Component as well. Clicking the button causes a dialog to pop up allowing the user to select the location of the output files. In the event that files have already been output to this location, the code generator automatically renames the files with conflicting names to prevent data loss.

The second way to generate PROMELA source code is to run the generator in a stand-alone mode. This is done by running a command formatted as follows: “java –jar PromelaCodeGenerator.jar <main JSIDL file path> <JSIDL dependency path> <JSIDL schema path> <optional: output path>”. The argument descriptions are found inside the angle brackets. For Windows, make sure that any arguments containing spaces are contained within double quotes. If the last argument is not specified, the system property “user.home” is used for the output location but "/JSIDL_Validation/PromelaOutput/" is appended to the end to avoid cluttering up the directory.

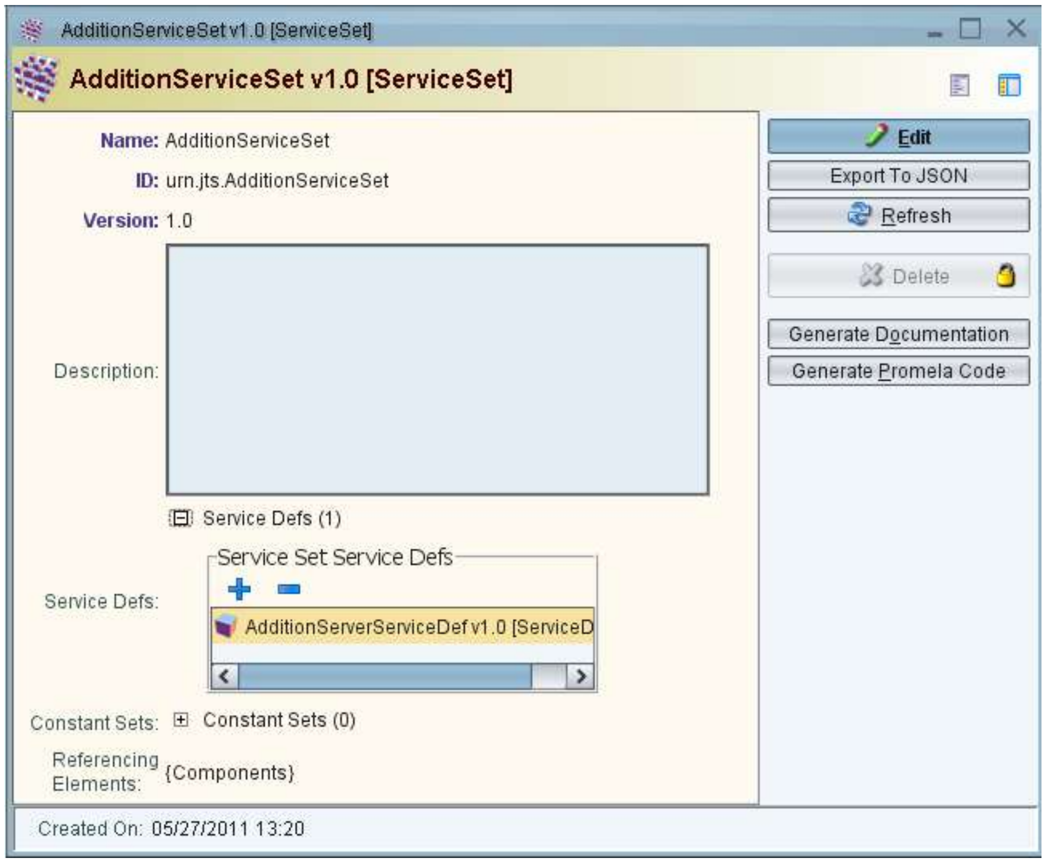
Due to the resolution of references that is done by JTS, the output of the two methods above may not be the same. For instance, if the code is generated for the core service AccessControl, the stand-alone mode will result in several more files. The extra files hold definitions for data that is referenced in the rest of the model. The JTS GUI generated code contains the same definitions, but they are instead inline with the service definitions and can be found in the files corresponding to those service definitions.

20.6 A Practical PROMELA Example

This example will be using the JTS GUI to generate the code for validation. As a simple example, the Addition service will be used, that is described in section 9.

20.6.1 Getting Ready to Generate Code

Follow the directions in sections 9.2 through 9.4 to create the service. When you finish the service set should look something like this:



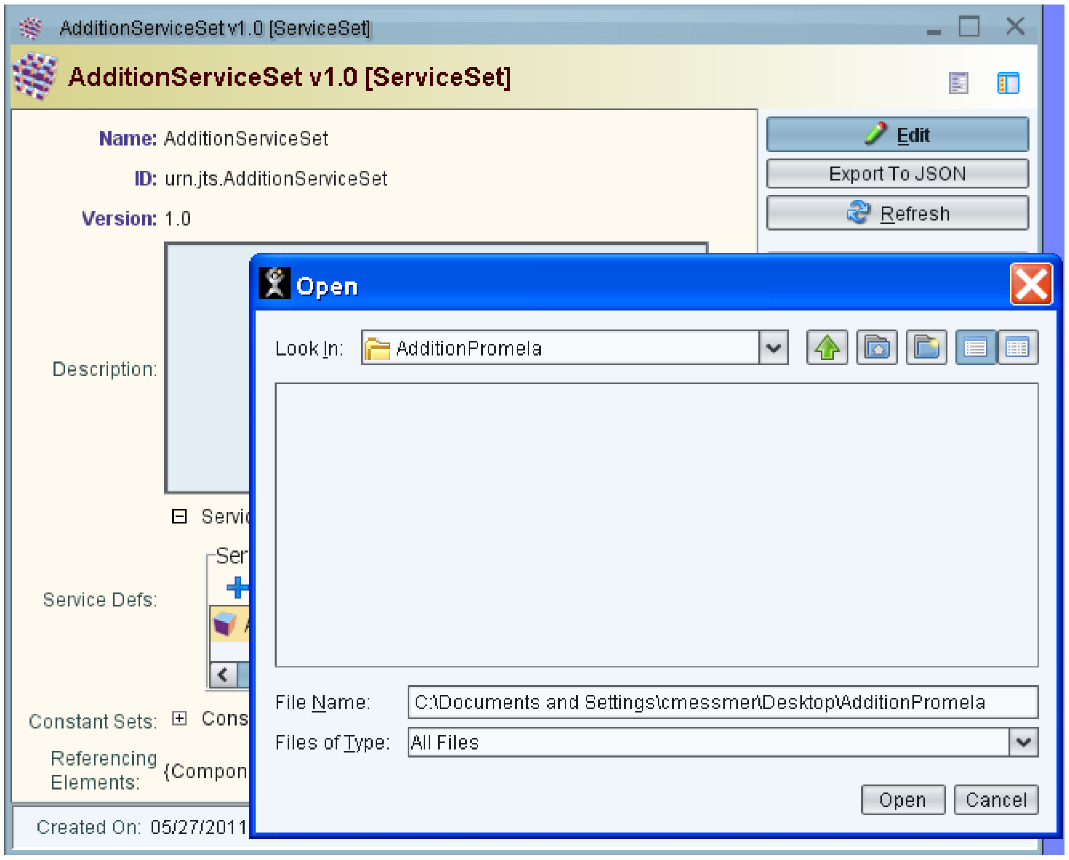
Name: AdditionServiceSet
ID: urn.jts.AdditionServiceSet
Version: 1.0
Description:
Service Defs (1)
Service Defs: AdditionServerServiceDef v1.0 [ServiceD
Constant Sets: Constant Sets (0)
Referencing Elements: {Components}
Created On: 05/27/2011 13:20

Figure 128 Addition Service Set Created

For this example the AdditionClient will not be generated.

20.6.2 Generate Source Code

To generate the PROMELA source code, open the service set that was created for the AdditionServer. Next, click on the "Generate Promela Code" button and select the location for the output files.


AdditionServiceSet v1.0 [ServiceSet]

Name: AdditionServiceSet
ID: urn.jts.AdditionServiceSet
Version: 1.0
Created On: 05/27/2011

Open

Look In: AdditionPromela
File Name: C:\Documents and Settings\cmessmer\Desktop\AdditionPromela
Files of Type: All Files

Figure 129 Select Location for Generated Code

20.6.2.1 Files Generated for Every Service

The following files will always be generated at the location selected:

main.pml – This file should never be manually edited or the protocol will not match what is being validated. This file contains the service process, which contains the state machine that was defined for the service.

channels.pml – This file should never be manually edited, but should not affect the model. This file contains "channel" definitions that are used by the service. Channels are used to send and receive specific data and cannot be used to send data other than what is defined.

userEditableClients.pml – This file contains a rudimentary implementation for a client process. All it does is randomly select a message to send to the service. The message data is not initialized or meant to convey any real meaning. This file should be edited by the user to activate specific behavior within the service.

userEditableConfig.pml – This file contains configuration data for the service and client implementation. The purpose is to centralize the location for all configurable data. The default configuration includes three settings: CLIENTS, QUEUE_SIZE, and CLIENT_CHANNELS. CLIENTS is the number of client processes to be used in the validation. CLIENT_CHANNELS is the number of channels that are required to send messages from the service to a specific client. Most of the time CLIENTS and CLIENT_CHANNELS are the same, but can be different if the user adds another process into the model. An example of this is when the user creates a process to send system events.

userEditableEvents.pml – This file is used to store event data definitions and as a location for event implementations. The file contains directions within the file comments for implementing events.

userEditableGuardsAndActions.pml – This file is where the majority of the user editing will occur. This file contains definitions for all the guards and actions contained within the model. These definitions need to be modified since, as generated, all guards evaluate to 'true' and all actions just print what the action function name is.

20.6.2.2 Files Specifically Generated for this Service

The following file was generated for the defined service; the number of files and their names depend on the service set definition:

urn_jaus_example_addition_server_static.pml – This file's name is constructed from the ID of the definition, since the ID is guaranteed to be unique within the service. The file contains all the data defined within the service definition whose ID = "urn_jaus_example_addition_server". All files used by the service that contain definitions will cause a corresponding unique file to be generated.

20.6.3 Modify Generated Service Code

20.6.3.1 userEditableConfig.pml

In this example the configuration will not be modified from the default values, except for the CLIENT_CHANNELS value will be increased to 2. This is because, in the next section, a new process will be added to the implementation in order to start the initialization of the server.

20.6.3.2 userEditableEvents.pml

There is a single event that moves the service from its initial state into a ready state, so the code that is added to the file is:

```
active proctype fireEvents(){

        InitToReadyEvent ! _pid;

}
```

This code creates a process that sends the InitToReadyEvent message to the server. The parameter “_pid” is a PROMELA system parameter that is equal to the process ID of the process that the parameter is found within. In this case it is the process ID of the fireEvents process.

20.6.3.3 userEditableGuardsAndActions.pml

This file will contain three actions and no guards. The actions are inline functions that are each a direct replacement for the call which is made from the main service. For this example, the default implementation is fine for the first two actions since they are informative:

```
inline Action_serverInitialized(){

 //Replace this print statement with a code line ending with a ;

// Ending a line with a back-slash allows the definition to continue on the next line.

// This can be repeated as many times as necessary.  All 3 of these lines would make up the
definition.

printf("executing action: Action_serverInitialized();");

}

inline Action_fsmStarted(){

 //Replace this print statement with a code line ending with a ;

// Ending a line with a back-slash allows the definition to continue on the next line.

// This can be repeated as many times as necessary.  All 3 of these lines would make up the
definition.

printf("executing action: Action_fsmStarted();");
}
```

The last action is intended to send a response to the client process. This type of action has a long name that is a concatenation of the message type that was received and the action name, with "_Action_" placed in between. The default action content will be removed and replaced so the action then looks like this:

```
inline urn_jaus_example_addition_server_QueryAddition_Action_sendReportAddition(inputMessage)
{
        // get data out of the inputMessage
        int inputdataA1;
        inputdataA1 = inputMessage.AdditionInput.A1;
        int inputdataA2;
        inputdataA2 = inputMessage.AdditionInput.A2;
        // declare instance of output message
        urn_jaus_example_addition_server_ReportAddition outputMessage;
        outputMessage.AdditionOutput.AdditionResult = inputdataA1 + inputdataA2;
        // send the message to the client
        ReportAddition[incoming_pid] ! _pid, outputMessage;
}
```

Notice the "incoming_pid". All messages that are sent or received include the process ID, and for messages received by the service, the pid for the sender is always stored in a variable called "incoming_pid". Since the inline function has access to all the caller's data, we can use the "incoming_pid" to ensure that the response message goes out on a channel that is being checked by the client application with the same pid.

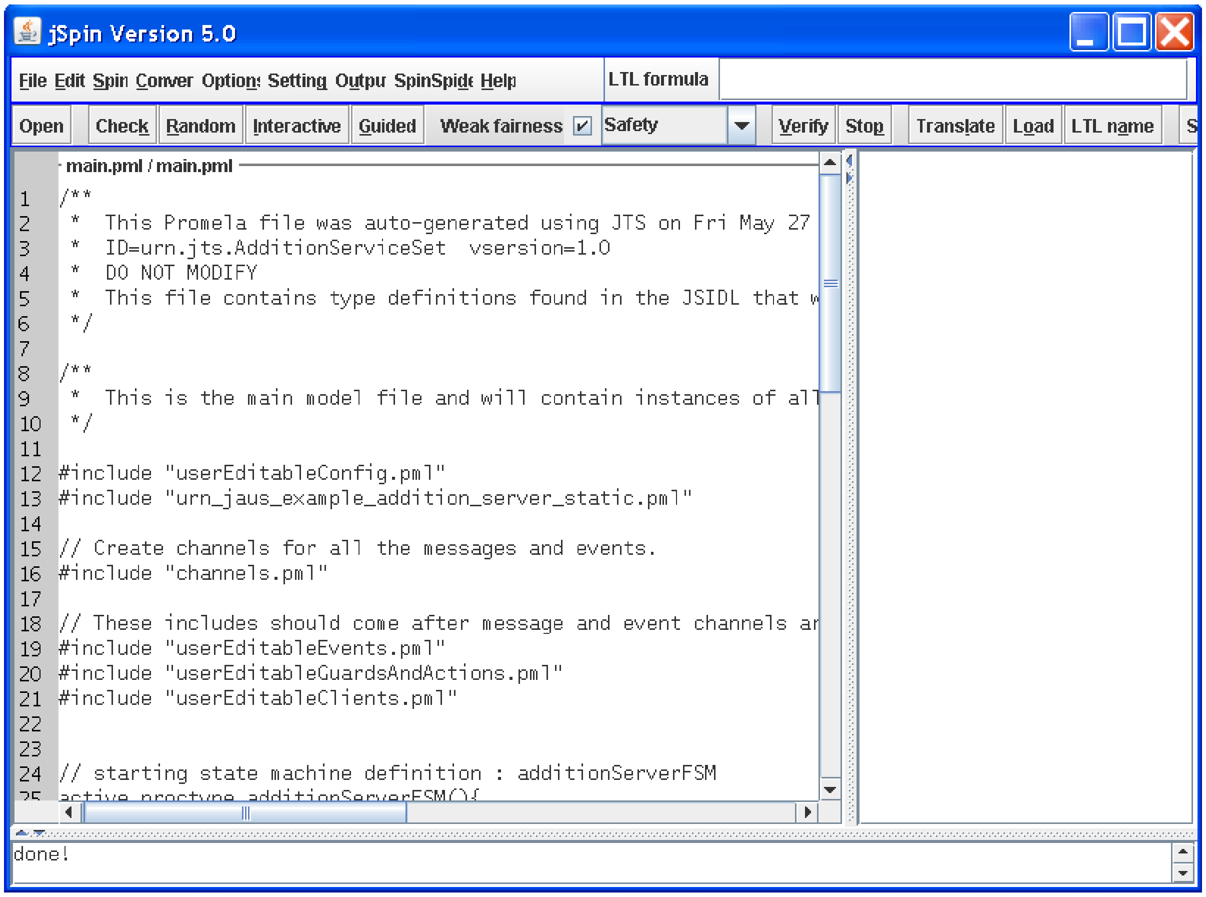
20.6.3.4 userEditableClients.pml

The default client implementation is a simple loop that is a one-state state machine with an internal transition that is implemented like a default_transition, meaning there is no trigger defined and there is no exit from the state. This state will be modified to include receiving the response message from the service. To make it even simpler, after the QueryAddition message is sent, a simple transition will occur taking the client to a state where it will just wait for the response. Once the response is received, the result will be printed and the client will exit. Now the client implementation looks like this:

```
// Client implementation
active [CLIENTS] proctype clientProcess(){
        // declare data for the messages
        urn_jaus_example_addition_server_QueryAddition QueryAddition_impl;
        urn_jaus_example_addition_server_ReportAddition responseMessage;
        // put some values into the declared data here
        int counter = 0;
        pid service_pid;
CLIENT_START_STATE:
        do
                ::  true ->
                printf("send messages\n");
                        counter = counter + 3;
                        //set the data and modify it so we get so variation in our results
                        QueryAddition_impl.AdditionInput.A1 = 5 + counter * 2;
                        QueryAddition_impl.AdditionInput.A2 = 31 + counter;
                         //This message sends two numbers to be added
                        QueryAddition ! _pid, QueryAddition_impl;
                        goto WAIT_FOR_RESPONSE_STATE;
         od;
WAIT_FOR_RESPONSE_STATE:
        do
                //receive the message on the channel corresponding to our pid
                :: ReportAddition[_pid] ? service_pid, responseMessage ->
                        printf("Received a response with value = %d",
responseMessage.AdditionOutput.AdditionResult);
                        //goto CLIENT_START_STATE;
                        break;
        od;
}
// End of client implementation
```

20.6.4 Validate the Generated Model

There are two main ways to validate the model using SPIN. The first method is to run the executable pan.exe, which is generated, compiled and run when the user selects “Verify” from the jSpin toolbar. The second method is to run a simulation by clicking on the “Random” button from the toolbar. A simulation is sometimes necessary if the state machine interactions are complex enough to cause state space explosion. State space explosion can even happen with our simple example, if the user comments out the “break” statement at the end of the previous section and uncomments the “goto CLIENT_START_STATE;” To avoid state space explosion, always allow the client process to finish. For this example, all of the validation will be done using jSpin.


jSpin Version 5.0
File Edit Spin Conver Option Setting Outpu SpinSpide Help
LTL formula
Open Check Random Interactive Guided Weak fairness ✓ Safety Verify Stop Translate Load LTL name
main.pml / main.pml

```
/**
 *  This Promela file was auto-generated using JTS on Fri May 27
 *  ID=urn.jts.AdditionServiceSet  vsersion=1.0
 *  DO NOT MODIFY
 *  This file contains type definitions found in the JSIDL that w
 */

/**
 *  This is the main model file and will contain instances of all
 */

#include "userEditableConfig.pml"
#include "urn_jaus_example_addition_server_static.pml"

// Create channels for all the messages and events.
#include "channels.pml"

// These includes should come after message and event channels ar
#include "userEditableEvents.pml"
#include "userEditableGuardsAndActions.pml"
#include "userEditableClients.pml"


// starting state machine definition : additionServerFSM
active proctype additionServerFSM(){
```

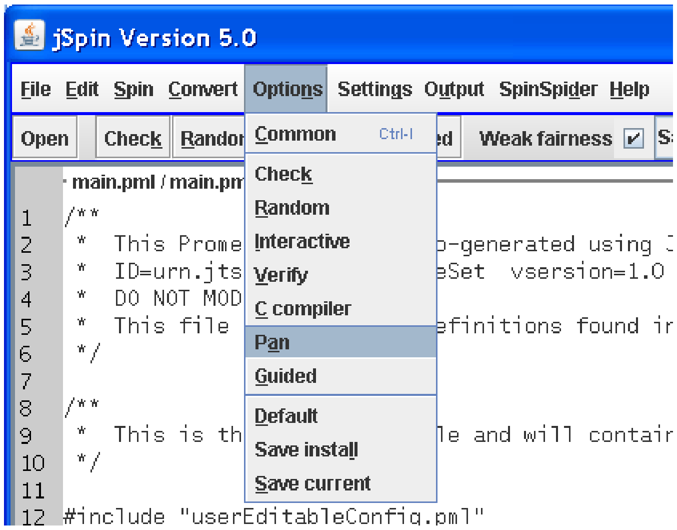
```
done!
```

Figure 130 jSpin 5.0

20.6.4.1 jSpin Settings

There are a few settings that should be configured before attempting to validate a model.

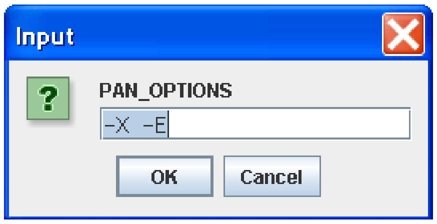
First, the check for an end state should be disabled since JSIDL doesn’t have an end state.



```
/**
 *  This Prome                 o-generated using J
 *  ID=urn.jts                 eSet  vsersion=1.0
 *  DO NOT MOD
 *  This file                  efinitions found ir
 */

/**
 *  This is th                 le and will contair
 */

#include "userEditableConfig.pml"
```

Figure 131 Turning Off End State Checking

This is done by selecting the “Options” menu and clicking on “Pan”.

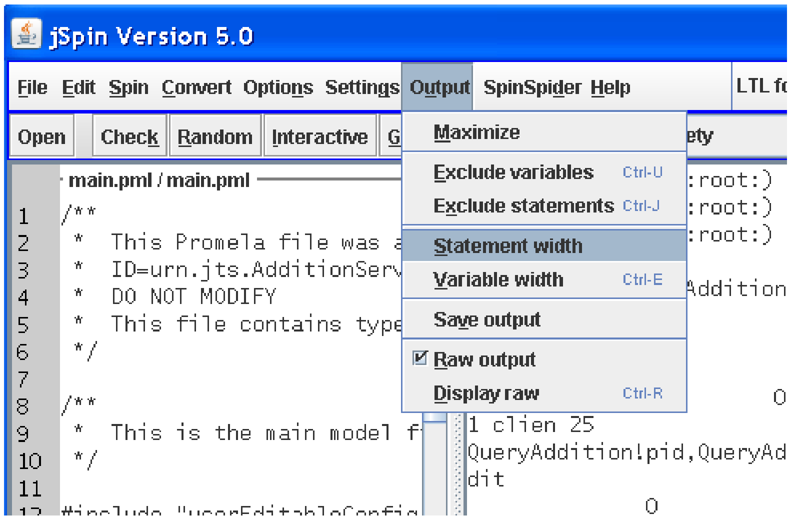


PAN_OPTIONS: -X -E

Figure 132 Setting Pan Options

Adding the “-E” to the existing options, tells SPIN to not check for end states.

The other setting that should be changed is the statement width.

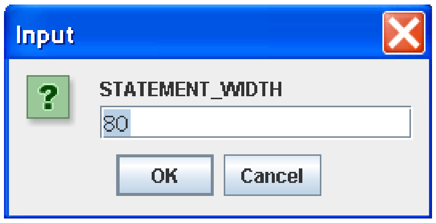


```
/**
 *  This Promela file was a
 *  ID=urn.jts.AdditionServ
 *  DO NOT MODIFY
 *  This file contains type
 */

/**
 *  This is the main model f
 */
```

```
:root:)
:root:)
:root:)

Addition

                 0
1 clien 25
QueryAddition!pid,QueryAd
dit
           0
```

Figure 133 Setting the Statement Width for jSpin

Select the “Output” menu and click the “Statement width” menu item.



STATEMENT_WIDTH: 80

Figure 134 Setting Statement Width Dialog

Once the dialog appears, set the value to something that should be able to hold an entire line of code. If the width is too small, it is difficult to tell what the statement says in jSpin. In this example the width is set to 80.

20.6.4.2 jSpin Syntax Checking

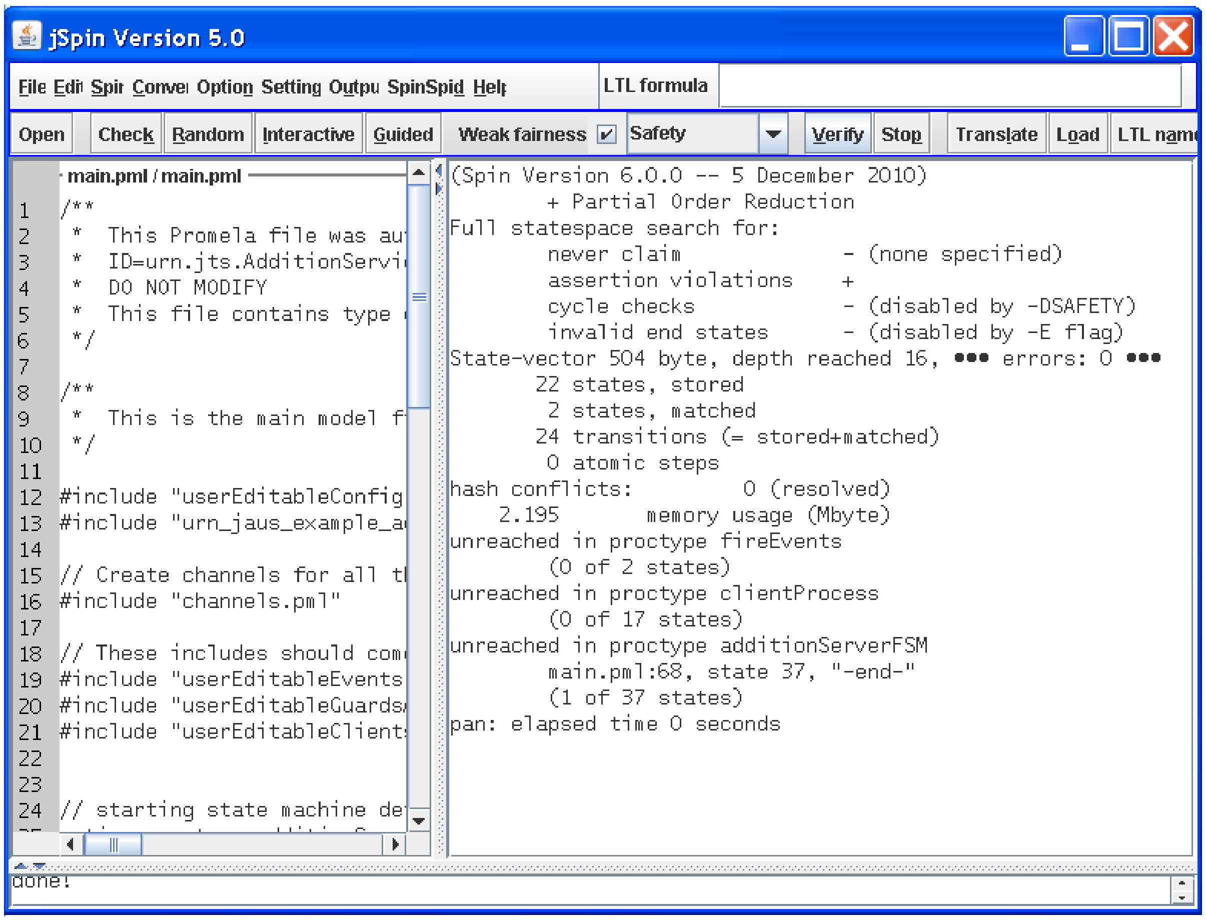
After editing the user editable code, always perform a syntax check by clicking on the “Check” button at the top of the dialog. Error messages will be printed in the right hand pane.

A known defect in SPIN is that errors often reference the wrong line number or file. If you are sure that the referenced line is correct, start looking elsewhere for the bug. Some suggestions for where to look are: the same line number but different file, immediately before or after the line, or the previous process definition.

When all else fails, start commenting out code till the error goes away. Start with commenting out entire processes before looking elsewhere as this should narrow down the area.

20.6.4.3 jSpin Verify

Only code that has been syntax checked can be verified.


jSpin Version 5.0
File Edit Spin Conver Option Setting Outpu SpinSpid Help
LTL formula
Open Check Random Interactive Guided Weak fairness ✓ Safety Verify Stop Translate Load LTL name
main.pml / main.pml
done!

```
/**
 *  This Promela file was au
 *  ID=urn.jts.AdditionServi
 *  DO NOT MODIFY
 *  This file contains type
 */

/**
 *  This is the main model f
 */

#include "userEditableConfig
#include "urn_jaus_example_a

// Create channels for all t
#include "channels.pml"

// These includes should com
#include "userEditableEvents
#include "userEditableGuards
#include "userEditableClient


// starting state machine de
```

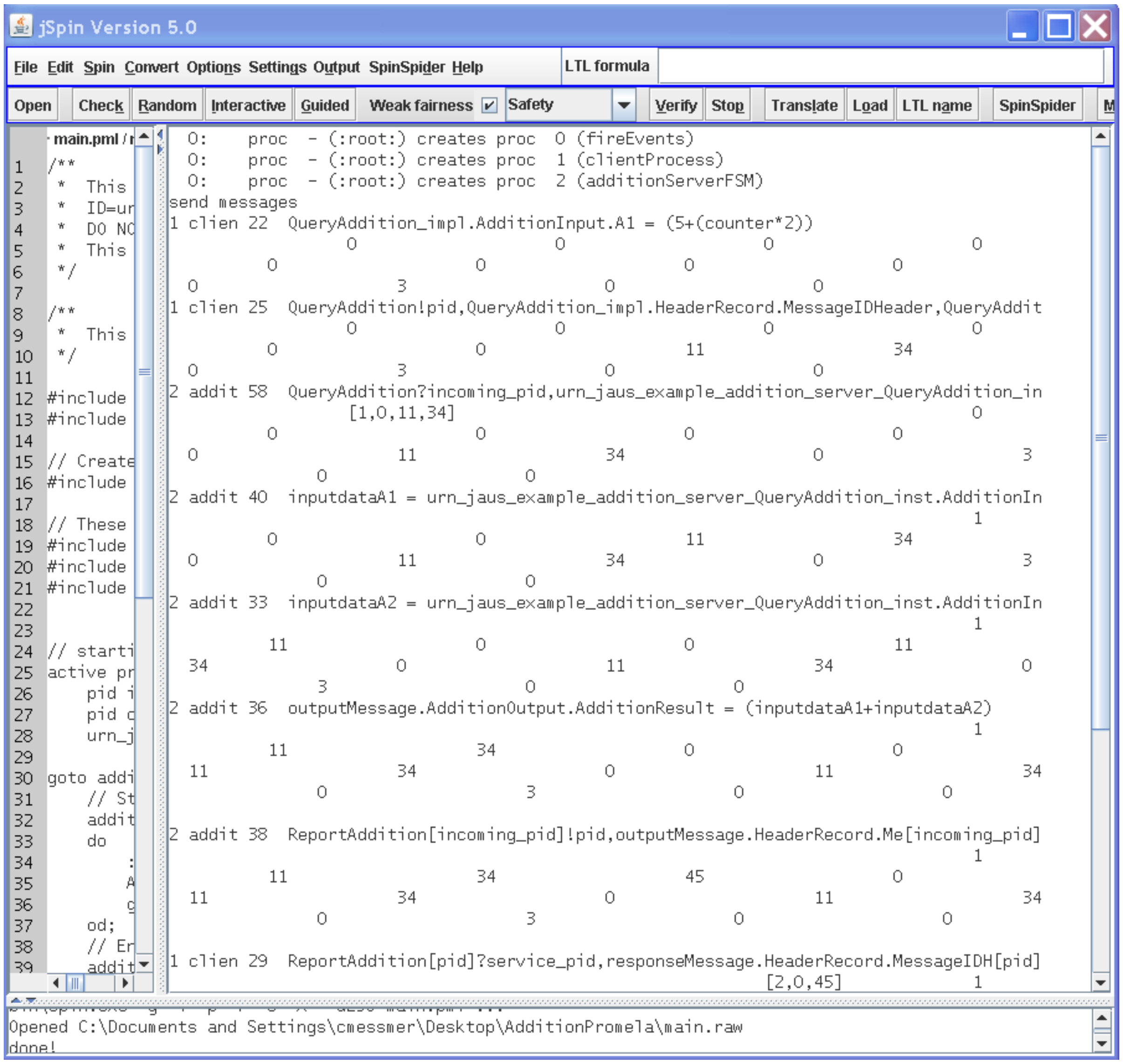
```
(Spin Version 6.0.0 -- 5 December 2010)
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      - (disabled by -E flag)
State-vector 504 byte, depth reached 16, ••• errors: 0 •••
       22 states, stored
        2 states, matched
       24 transitions (= stored+matched)
        0 atomic steps
hash conflicts:          0 (resolved)
    2.195       memory usage (Mbyte)
unreached in proctype fireEvents
        (0 of 2 states)
unreached in proctype clientProcess
        (0 of 17 states)
unreached in proctype additionServerFSM
        main.pml:68, state 37, "-end-"
        (1 of 37 states)
pan: elapsed time 0 seconds
```

Figure 135 Run Verify on Addition Example

Selecting the "Verify" button at the top of the dialog should result in text output describing the model. First notice the line about a third of the way down that shows "errors: 0". If you forget to turn off end state checking, the number of errors would be 1. The other thing to notice is the three sections that start with "unreached in". This shows how many states the model never got to. Notice that the end state is included here. Since the example specifically exits from fireEvents and clientProcess, there are zero unreached states. The service contains one unreached state, because it never reaches an end state.

20.6.4.4 jSpin Simulation

While “Verify” gives some insight into the validity of the model, it does not guarantee that the model acts like it was designed to. To ensure that the model reacts properly to input from the client, a simulation can be performed. Click the “Random” button on the toolbar.


jSpin Version 5.0
File Edit Spin Convert Options Settings Output SpinSpider Help
LTL formula
Open Check Random Interactive Guided Weak fairness Safety Verify Stop Translate Load LTL name SpinSpider
main.pml

```
/**
 *  This
 *  ID=ur
 *  DO NO
 *  This
 */

/**
 *  This
 */

#include
#include

// Create
#include

// These
#include
#include
#include


// starti
active pr
    pid i
    pid c
    urn_j

goto addi
    // St
    addit
    do
        :
          A
          g
    od;
    // Er
    addit
```

```
  0:    proc  - (:root:) creates proc  0 (fireEvents)
  0:    proc  - (:root:) creates proc  1 (clientProcess)
  0:    proc  - (:root:) creates proc  2 (additionServerFSM)
send messages
1 clien 22  QueryAddition_impl.AdditionInput.A1 = (5+(counter*2))
                  0                 0                 0                 0
          0                 0                 0                 0
  0                 3                 0                 0
1 clien 25  QueryAddition!pid,QueryAddition_impl.HeaderRecord.MessageIDHeader,QueryAddit
                  0                 0                 0                 0
          0                 0                11                34
  0                 3                 0                 0
2 addit 58  QueryAddition?incoming_pid,urn_jaus_example_addition_server_QueryAddition_in
                  [1,0,11,34]                                           0
          0                 0                 0                 0
  0                11                34                 0                 3
      0                 0
2 addit 40  inputdataA1 = urn_jaus_example_addition_server_QueryAddition_inst.AdditionIn
                                                                        1
          0                 0                11                34
  0                11                34                 0                 3
      0                 0
2 addit 33  inputdataA2 = urn_jaus_example_addition_server_QueryAddition_inst.AdditionIn
                                                                        1
         11                 0                 0                11
 34                 0                11                34                 0
            3                 0                 0
2 addit 36  outputMessage.AdditionOutput.AdditionResult = (inputdataA1+inputdataA2)
                                                                        1
         11                34                 0                 0
 11                34                 0                11                34
      0                 3                 0                 0

2 addit 38  ReportAddition[incoming_pid]!pid,outputMessage.HeaderRecord.Me[incoming_pid]
                                                                        1
         11                34                45                 0
 11                34                 0                11                34
      0                 3                 0                 0

1 clien 29  ReportAddition[pid]?service_pid,responseMessage.HeaderRecord.MessageIDH[pid]
                                                           [2,0,45]     1
```

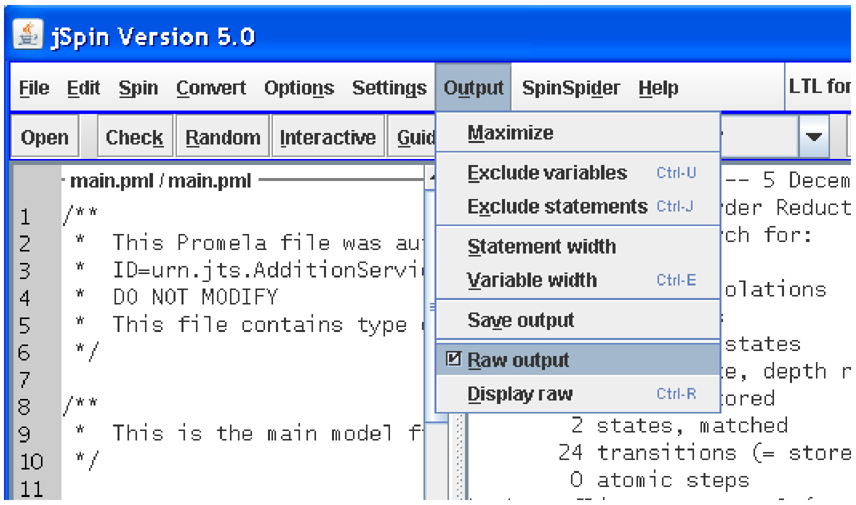
```
Opened C:\Documents and Settings\cmessmer\Desktop\AdditionPromela\main.raw
done!
```

Figure 136 Output from Random Simulation

The random simulation allows the user to see what line is being executed in the model, and even what the data values are. Within the output screen, the first few lines show what processes are created and what the process IDs are. The lines that follow wrap around several times, making the content more difficult to decipher. Each line starts with the pid followed by the name of the process. The field width is often too short to see the full process name. The next section of the line is the code that is being executed by the process. In the first two lines of execution, the client clearly sets values and then sends a QueryAddition message. The first line starting with a "2" is the service receiving the QueryAddition message. Notice that directly below that line is a set of square brackets containing the values of the data, "[1, 0, 11, 34]". In this example 11 and 34 are the two numbers that should be added. The "1" is the pid for the client application. The "0" is the MessageIDHeader value found in the QueryAddition Message header. This value is not being populated in our implementation.

Skipping most of the intervening lines, the last line shows the client process receiving the response with a value of "45".

In most cases, a random simulation is sufficient. If more information is desired, turn on "raw data" before running the random simulation.



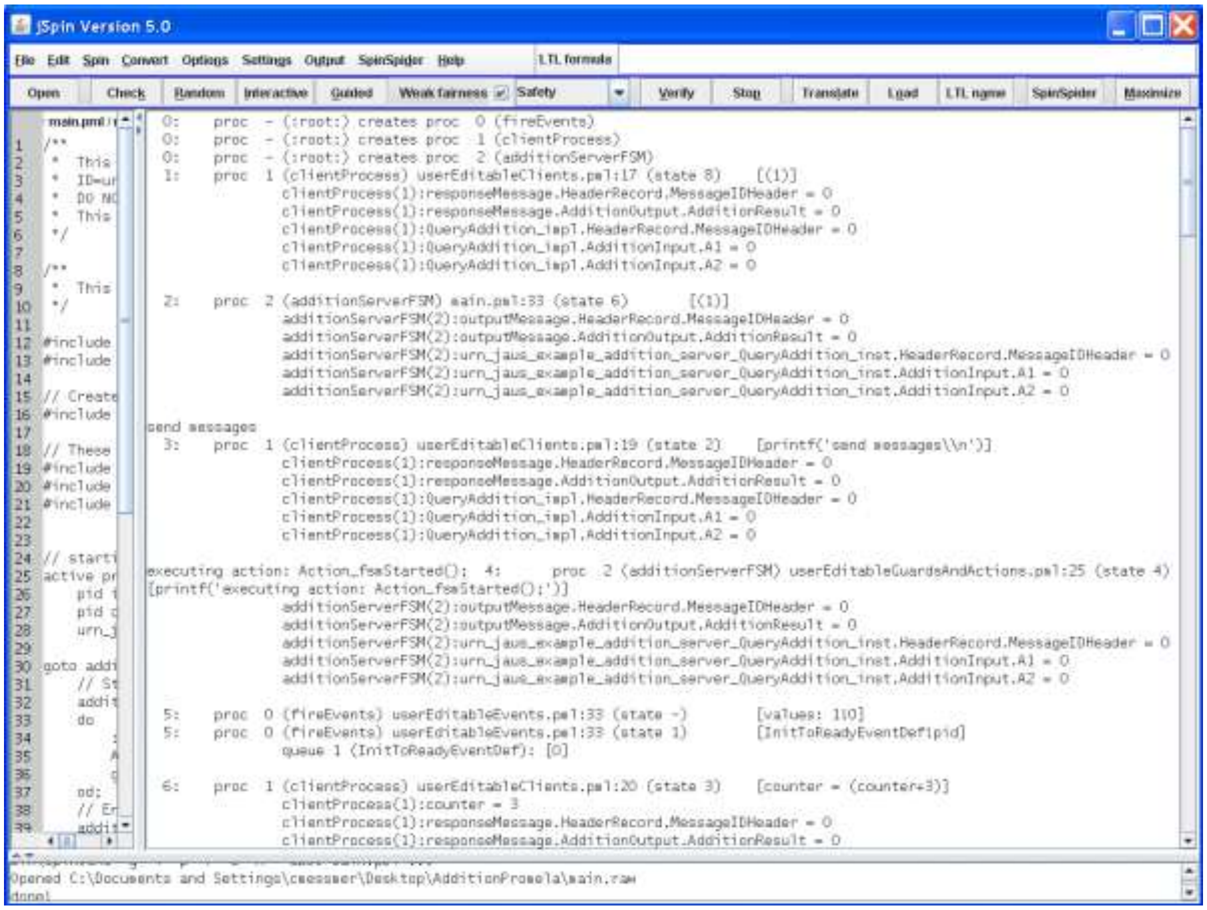
```
/**
 *  This Promela file was au
 *  ID=urn.jts.AdditionServi
 *  DO NOT MODIFY
 *  This file contains type
 */

/**
 *  This is the main model f
 */
```

```
-- 5 Decem
der Reduct
ch for:

olations

states
e, depth r
ored
   2 states, matched
  24 transitions (= store
   0 atomic steps
```

Figure 137 Capturing Raw Output

Select the "Output" menu and make sure the "Raw output" box is checked.

Now, click the "Random" button to perform a random simulation.


jSpin Version 5.0
File Edit Spin Convert Options Settings Output SpinSpider Help
LTL formula
Open Check Random Interactive Guided Weak fairness Safety Verify Stop Translate Load LTL name SpinSpider Maximize
main.pml

```
/**
 *  This
 *  ID=ur
 *  DO NO
 *  This
 */

/**
 *  This
 */

#include
#include

// Create
#include

// These
#include
#include
#include


// starti
active pr
    pid t
    pid c
    urn_j

goto addi
    // St
    addit
    do
        :
          A
          g
    od;
    // En
    addit
```

```
  0:    proc  - (:root:) creates proc  0 (fireEvents)
  0:    proc  - (:root:) creates proc  1 (clientProcess)
  0:    proc  - (:root:) creates proc  2 (additionServerFSM)
  1:    proc  1 (clientProcess) userEditableClients.pml:17 (state 8)    [(1)]
                clientProcess(1):responseMessage.HeaderRecord.MessageIDHeader = 0
                clientProcess(1):responseMessage.AdditionOutput.AdditionResult = 0
                clientProcess(1):QueryAddition_impl.HeaderRecord.MessageIDHeader = 0
                clientProcess(1):QueryAddition_impl.AdditionInput.A1 = 0
                clientProcess(1):QueryAddition_impl.AdditionInput.A2 = 0

  2:    proc  2 (additionServerFSM) main.pml:33 (state 6)       [(1)]
                additionServerFSM(2):outputMessage.HeaderRecord.MessageIDHeader = 0
                additionServerFSM(2):outputMessage.AdditionOutput.AdditionResult = 0
                additionServerFSM(2):urn_jaus_example_addition_server_QueryAddition_inst.HeaderRecord.MessageIDHeader = 0
                additionServerFSM(2):urn_jaus_example_addition_server_QueryAddition_inst.AdditionInput.A1 = 0
                additionServerFSM(2):urn_jaus_example_addition_server_QueryAddition_inst.AdditionInput.A2 = 0

send messages
  3:    proc  1 (clientProcess) userEditableClients.pml:19 (state 2)    [printf('send messages\\n')]
                clientProcess(1):responseMessage.HeaderRecord.MessageIDHeader = 0
                clientProcess(1):responseMessage.AdditionOutput.AdditionResult = 0
                clientProcess(1):QueryAddition_impl.HeaderRecord.MessageIDHeader = 0
                clientProcess(1):QueryAddition_impl.AdditionInput.A1 = 0
                clientProcess(1):QueryAddition_impl.AdditionInput.A2 = 0

executing action: Action_fsmStarted();  4:      proc  2 (additionServerFSM) userEditableGuardsAndActions.pml:25 (state 4)
[printf('executing action: Action_fsmStarted();')]
                additionServerFSM(2):outputMessage.HeaderRecord.MessageIDHeader = 0
                additionServerFSM(2):outputMessage.AdditionOutput.AdditionResult = 0
                additionServerFSM(2):urn_jaus_example_addition_server_QueryAddition_inst.HeaderRecord.MessageIDHeader = 0
                additionServerFSM(2):urn_jaus_example_addition_server_QueryAddition_inst.AdditionInput.A1 = 0
                additionServerFSM(2):urn_jaus_example_addition_server_QueryAddition_inst.AdditionInput.A2 = 0

  5:    proc  0 (fireEvents) userEditableEvents.pml:33 (state -)        [values: 1!0]
  5:    proc  0 (fireEvents) userEditableEvents.pml:33 (state 1)        [InitToReadyEventDef!pid]
                queue 1 (InitToReadyEventDef): [0]

  6:    proc  1 (clientProcess) userEditableClients.pml:20 (state 3)    [counter = (counter+3)]
                clientProcess(1):counter = 3
                clientProcess(1):responseMessage.HeaderRecord.MessageIDHeader = 0
                clientProcess(1):responseMessage.AdditionOutput.AdditionResult = 0
```

```
Opened C:\Documents and Settings\cmesser\Desktop\AdditionPromela\main.raw
done!
```

Figure 138 Raw Data Output

Typing Ctrl-R or selecting “Display Raw” from the “Output” menu gives the above result. The key difference is that everything is displayed in the raw data. As each statement is displayed, a complete list of all data within scope is listed. It is still possible to distinguish between processes, but nothing is hidden.

Both the normal and raw output methods allow the user to perform a manual analysis of the functioning service to determine defects. In this example, the analysis can be boiled down to the content of the ReportAddition message that the client process receives.

```
[ReportAddition[pid]?service_pid,responseMessage.HeaderRecord.MessageIDHeader,responseMessage
.AdditionOutput.AdditionResult]

                queue 4 (ReportAddition[1]):

                clientProcess(1):service_pid = 2
                clientProcess(1):responseMessage.HeaderRecord.MessageIDHeader = 0

                clientProcess(1):responseMessage.AdditionOutput.AdditionResult = 45

                clientProcess(1):QueryAddition_impl.HeaderRecord.MessageIDHeader = 0

                clientProcess(1):QueryAddition_impl.AdditionInput.A1 = 11

                clientProcess(1):QueryAddition_impl.AdditionInput.A2 = 34
```

The inputs are 11 and 34, which gives the AdditionResult of 45.

20.7 Things to Know

There are a lot of important things to understand before attempting to validate a protocol: PROMELA patterns, PROMELA data mapping to JSIDL, and special implementation topics. Understanding this section is critical to understanding how the model works and how to debug it.

20.7.1 PROMELA Patterns

There are several patterns used in the code to produce the output model. State machines, states, transitions, triggers, guards, actions, and naming conventions are all part of the model generated from the main ServiceDef in the ServiceSet.

20.7.1.1 State Machines

State machines are implemented as a separate process. In PROMELA it looks like this:

```
active proctype serviceNameFSM(){

...

}
```

20.7.1.2 States

States are only found within a state machine implementation. The state implementation looks like this:

```
THE_STATE_NAME:
do
...

od;
```

20.7.1.3 Transitions

Transitions, found within states are denoted with a double colon and are followed by the trigger.

```
:: AnEventTrigger() ->

        if

        :: GuardFunction(param1, etc) ->

                Action1();

                Action2();

        fi;
```

or the trigger could be an incoming message as shown below.

```
:: SomeMessage ? param1, param2, ..., parmx ->
```

The trigger is always followed by the guards and actions (left out for brevity). Guards are always found inside the “if” section and preceded by a double colon. Guard definitions must always evaluate to true or false. The actions are function calls, implemented by the user.

20.7.1.4 Guards

The implementation of guards is stubbed out in the userEditableGuardsAndActions.pml file. Guards are implemented using a #define, which will contain an expression that is evaluated at runtime. Guard definitions can accept parameters and use any data that is within scope of the call to the guard. An example guard implementation has been taken from the AccessControl core service model and is shown below.

```
#define isControllingClient(incoming_pid) (incoming_pid == current_client_pid)
```

Notice that a parameter is used and evaluated against another variable. “current_client_pid” is a global value that has been defined previously. Make sure that the definition is always included in parentheses since logical operators can be applied to any guard function.

20.7.1.5 Actions

The implementation for actions is stubbed out in the userEditableGuardsAndActions.pml file. Each action is implemented using an inline function that should contain the code executed for the action. An example action implementation has been taken from the AccessControl core service model and is shown below.

```
/**
*  Send a Reject Control message to controlling client
*/
inline Action_Send_RejectControl(sender_pid, param1){
        urn_jaus_jss_core_MessageSet_CommandClass_RejectControl rej;
        rej.RejectControlRec.ResponseCode = param1;
        RejectControl[current_client_pid] ! sender_pid, rej;
}
```

In this case, a message needed to be constructed and sent to the client process that was in control of the service.

20.7.1.6 Naming Conventions

Data types are all prepended with the ID of the service definition or declared type set to which they belong. For instance, the BasicTypes.xml included in the core services contains a fixed_field called AuthorityCode. The name of the AuthorityCode type is converted to urn_jaus_jss_core_MessageSet_BasicTypes_AuthorityCode because the ID for the declared type set is “urn:jaus:jss:core:MessageSet:BasicTypes”.

State names begin with the state machine name followed by the parent state’s name and end with their own name. The code found in section 20.7.3.1 shows a useful example of some nested states that would belong to a state machine named ReceiveFSM.

20.7.2 PROMELA Data Mapping to JSIDL

JSIDL data types and structures are mapped to PROMELA data types and structures as part of the conversion process.

Table 7 Data Type Mapping to PROMELA

JSIDL Types	PROMELA Type	Notes
byte, short integer, integer, long integer, unsigned byte, unsigned short integer, unsigned integer, unsigned long integer	int	Since the values of the datatypes are what is important and the unsigned types for PROMELA are not easily implemented, a signed integer is used for all integral datatypes except for enums.
enum	mtype	JSIDL enums are string values that are associated with integers. These strings can include special characters and whitespace, which makes it difficult to convert directly. The implementation removes all special characters and whitespace to make it compatible with the

		mtype.
fixed length string, variable length string,		No corresponding PROMELA type was found.
float, long float variable length field		No corresponding PROMELA type was found.
bit field		No corresponding PROMELA type was found.
arrays	[ ]	Limited support exists for arrays. Most of the data used to validate the protocol exists within a message. Messages are defined using typedef and PROMELA prohibits using an array within a typedef. For most cases, this means arrays cannot be used.

As you can see there are several types that don't have a corresponding PROMELA type. Where possible, substitutions have been made that should not impair the function of the model. In several cases, no substitution is possible and the data is not converted. If the protocol is using a string, variant, or a variable length data type as the trigger for a transition or the data evaluated in a guard, the protocol will need some other data defined and evaluated to take its place. In this case a global variable defined in one of the user editable files should be used. This method doesn't require any changes to the protocol or PROMELA model, which is always preferred.

20.7.3 Special Implementation Topics

20.7.3.1 Inherited State Machines

JSIDL allows for state machines to inherit from other state machines. For the PROMELA code generator, the inheritance chain is followed to the most base service definition. All protocols are incorporated into existing state machines with the same name. Duplicate state definitions are merged into a single state, so that inheritance is preserved. Transitions are kept in order and in separate categories, so that the final state machine can evaluate the transitions in the order mandated by the AS5684.

20.7.3.2 Nested States

The JSIDL allows for nested states, and the JTS core services use nested states extensively. The PROMELA documentation doesn't mention or seem to indicate that PROMELA can support nested states.

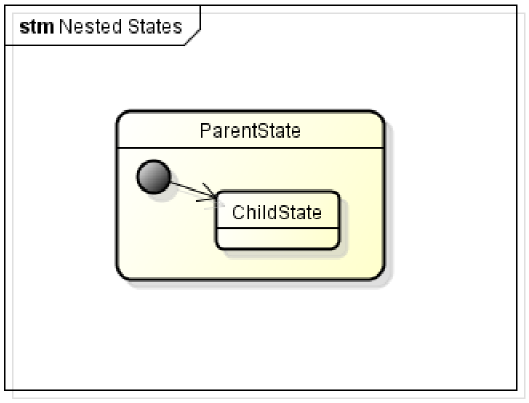


Figure 139 JSIDL Nested States

In order for the PROMELA code to compile, it is necessary to organize nested state code like this:

```
        // Entry Action
ReceiveFSM_Receiving:
        do
        :: true ->
              goto ReceiveFSM_Receiving_IMPL;
        od;
        // End Entry Actions
        ReceiveFSM_Receiving_IMPL:
        do
        ::
              // Start Entry Actions
              ReceiveFSM_Receiving_Ready:
              do
              :: true ->
                    someEntryAction();
                    goto ReceiveFSM_Receiving_Ready_IMPL;
              od;
              // End Entry Actions
              ReceiveFSM_Receiving_Ready_IMPL:
              do
              :: someTrigger() ->
                    if
                    :: someGuard() ->
                          someAction();
```

```
                        goto ANOTHER_STATE;
                  fi;
            od;
      od;
```

Note that the entry action code has been left in this example and will be discussed in the next section. The important thing to recognize is that the highlighted double colon must only occur once within the body of the state immediately preceding the definitions of all sub-states. This is true because PROMELA is nondeterministic by nature and the double colon means that the following lines are an optional path. Our deterministic implementation requires that there be only one valid path to take.

Probably the most important thing to note about nested states is that transitions are only implemented for states that don't have any child states. If a parent state has transitions defined, then those transitions are just passed on to the children. Let's consider the case where this simple state machine is used to generate PROMELA code:

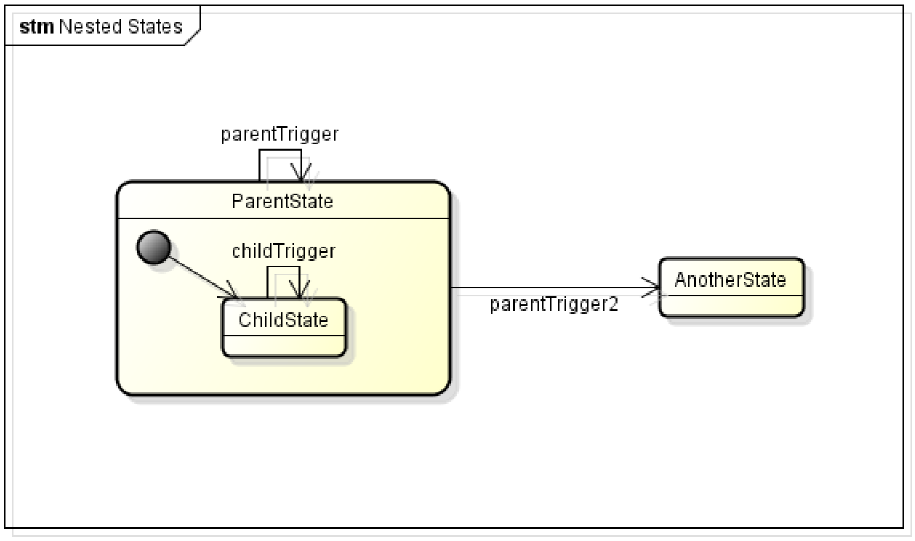


Figure 140 Simple State Machine with Nested States

The parent state contains two transitions and the child contains one. The actual function of this state machine could be rewritten to exclude the parent state altogether like this:

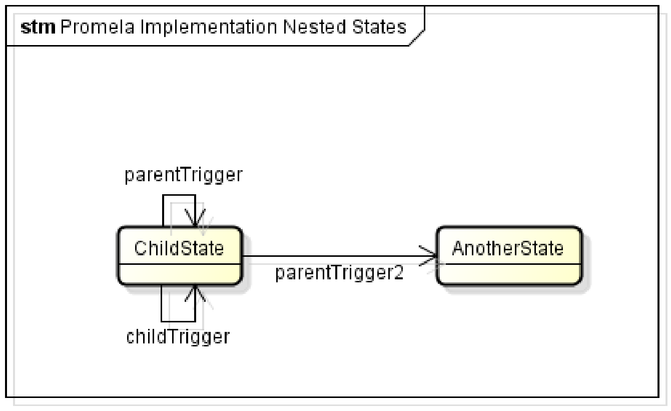


Figure 142 PROMELA Simplification of Nested States

The PROMELA implementation simplifies the state machine in this way. The PROMELA code for this is shown below

```
ParentState:
do
:: true->
        // could add some entry actions here
        goto ParentState_IMPL;
od;
ParentState_IMPL:
do
::
// add child states here
        ParentState_ChildState:
        do
        :: true->
                // could add some entry actions here
                goto ParentState_ChildState_IMPL;
        od;
        ParentState_ChildState_IMPL:
        do
        :: parentTrigger() ->
                // unguarded transition
                if
                :: true->
                        // no actions defined
                fi;
        :: parentTrigger2() ->
                // unguarded transition
                if
                :: true->
```

```
                        // no actions defined
                        goto AnotherState;
                fi;
        :: childTrigger() ->
                // unguarded transition
                if
                :: true->
                        // no actions defined
                fi;
        od;
od;

AnotherState:
do
// do some stuff here
od;
```

Notice that the parent state is still represented in our implementation, but the only thing it really does is contain the child states.

20.7.3.3 Entry Actions

In the example from the previous section, the entry actions that were generated are implemented as separate states. This was necessary because it is syntactically invalid to insert calls to action functions immediately following the highlighted double colon, which is where they would need to be inserted.

20.7.3.4 Exit Actions

The exit actions for a state are simply added to the list of actions immediately preceding any goto statement. If we add exit action doSomethingCool() to the parent or child state from the example in section 20.7.3.1, the child state would look like this:

```
ReceiveFSM_Receiving_Ready_IMPL:
do
:: someTrigger() ->
      if
      :: someGuard() ->
            someAction();
            doSomethingCool();
            goto ANOTHER_STATE;
      fi;
od;
```

The call to doSomethingCool() would be repeated for every instance where a goto is used to exit the state. While other implementations were considered, this was the simplest to implement and evaluate.

20.7.3.5 Default Transitions

Default transitions are evaluated after the triggers for all other transitions have failed. The only real difference between a default transition and the other transitions is that the default transition doesn't have a trigger. Using the state in the previous example, if we add a default transition to the state it would look like this:

```
ReceiveFSM_Receiving_Ready_IMPL:
do
:: someTrigger() ->
      if
      :: someGuard() ->
            someAction();
            doSomethingCool();
            goto ANOTHER_STATE;
      fi;

      :: true ->
            if
            :: guardForDefaultTransition() ->
                defaultTransitionAction();
            fi;
od;
```

20.7.3.6 Default States

The default state for JSIDL is a collection of transitions that are evaluated if none of the transitions from the state evaluate to true and are taken. This allows for a set of states to have a single definition for common transitions, if they share a default state.

Default states are implemented by copying the transitions from the default state into every state to which the default state applies. Great care is taken to ensure that default state transitions are only evaluated after those of the state. Section 20.7.3.9 contains a complete description of how the transition order is maintained.

20.7.3.7 Push and Pop Transitions

Push and pop transitions are not supported in PROMELA. The implementation is a workaround that allows for the same push and pop functionality, even though the concept isn't the same.

The goal is to be able to transition to a separate state and back again from multiple locations within a state machine. Many different solutions were investigated, but in the end only one was a viable option for automatic code generation.

The basic idea is to make a state specific copy of the state we are pushing to and popping from. This will allow every state to transition to and come back from the push/pop state without needing to specify a dynamic state to pop back to. An example implementation is below where CALLING_STATE1 and CALLING_STATE2 both push to a PUSH_POP_STATE. Separate copies of the PUSH_POP_STATE are created.

JSIDL Push/Pop transition:

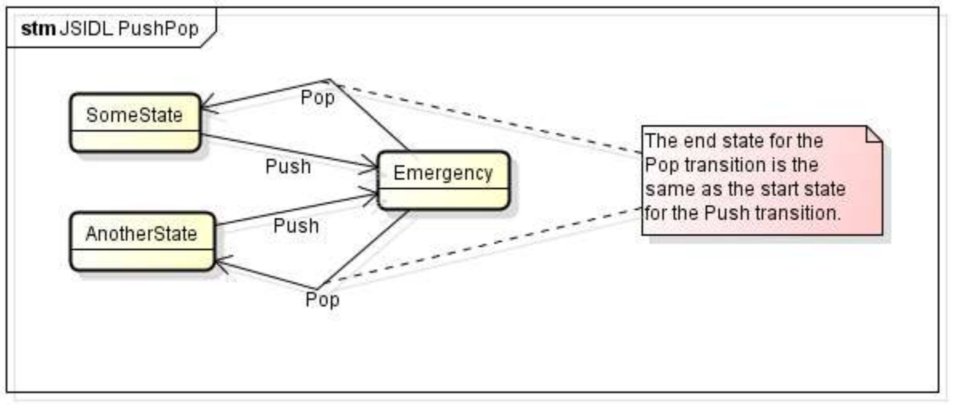

stm JSIDL PushPop
SomeState
Pop
Push
Emergency
AnotherState
Push
Pop
The end state for the Pop transition is the same as the start state for the Push transition.

Figure 143 JSIDL Push and Pop Transitions

Modified Push/Pop transitions to support Promela modeling:

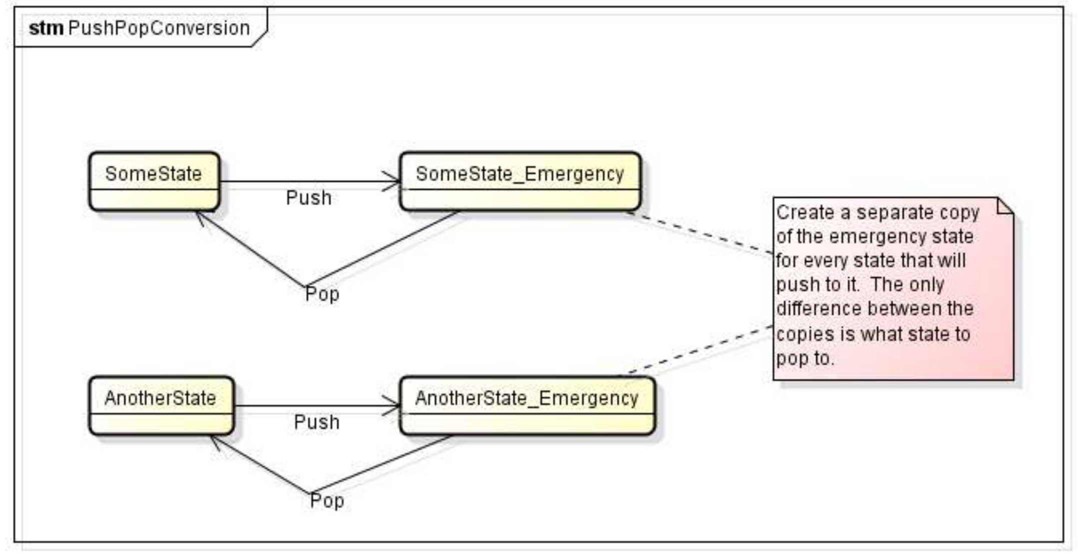

stm PushPopConversion
SomeState
Push
SomeState_Emergency
Pop
Create a separate copy of the emergency state for every state that will push to it. The only difference between the copies is what state to pop to.
AnotherState
Push
AnotherState_Emergency
Pop

Figure 145 PROMELA Simplification of Push and Pop Transitions

Push/Pop transition implementation in PROMELA:

```
CALLING_STATE1:
do
/* do some stuff here */
:: do_push_transition == true ->
      goto CALLING_STATE1_PUSH_POP_STATE;
od;
CALLING_STATE2:
do
/* do some stuff here */
:: do_push_transition == true ->
      goto CALLING_STATE2_PUSH_POP_STATE;
od;
CALLING_STATE1_PUSH_POP_STATE:
do
/* do some stuff here */
:: do_pop_transition == true ->
      goto CALLING_STATE1;
od;
CALLING_STATE2_PUSH_POP_STATE:
do
/* do some stuff here */
:: do_pop_transition == true ->
      goto CALLING_STATE2;
od;
```

20.7.3.8 Guards

Guards are implemented using #define. While a similar capability exists by using an inline function, the inline function has no return value so the guard could not be evaluated. JSIDL allows for guards to be composed of multiple functions connected with logical operators || (or) and && (and). In addition, the ! (not) operator may be applied. JSIDL does not make any attempt to separate parts of the expression, but forces it to be stored as a single string. The string is parsed and the functions are separated so they can be implemented. The function definitions are each stubbed out in the userEditableGuardsAndActions.pml file.

20.7.3.9 Transitions

According to the AS5684, transitions need to be evaluated in this order:

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

The SPIN model checker which evaluates the PROMELA code is inherently nondeterministic. This means that when multiple transition conditions would evaluate to 'true', then those are all possible. JSIDL says that only the first one in the above evaluation order is possible. In an effort to conform to the AS5684 transition evaluation order, a workaround needed to be devised. PROMELA has an "unless" keyword that allows us to nest the evaluations and control the order of evaluation.

```
:: CreateEvent ? incoming_pid ->
        if
        /**
        *  True if parameters are not supported.
        */
        ::  ! isSupported(incoming_pid) ->
                //  Send Reject Event Request message
                RejectEventRequest(_pid, incoming_pid);
        ...
        fi
        unless {
        if
        /**
        *  True if parameters are supported and the event already exists.
        */
        ::  isSupported(incoming_pid) && eventExists(incoming_pid) ->
                //  update the event
                updateEvent();
        ...
        fi
        unless {
        if
        /**
        *  True if parameters are supported and the event does not already exist.
        */
        ::  isSupported(incoming_pid) && ! eventExists(incoming_pid) ->
                //  create the event
                createEvent();
        ...
        fi
        }}
```

The above code was modified for this document but was taken from the AccessControl service. Notice that there are three transitions and that all three are triggered by receiving the CreateEvent message. The order of evaluation for the transitions is from the bottom up. Basically, it says "do the first transition unless you can do the second transition unless you can do the third transition." So the third transition must be evaluated first because of the "unless". This ensures that transitions are evaluated in a controlled order. Each state maintains lists of transitions based on the JSIDL transition type and the code is generated from a combined transition list that follows the JSIDL spec.

20.7.3.10 Naming Overlap

Because it is almost guaranteed that there will be naming overlap between states, transitions, actions, guards, etc., there is a need to maintain some semblance of scope when converting to PROMELA. PROMELA code has no real concept of scope, and especially not at the level that JSIDL uses. For instance a JSIDL Message could define a new type called Foo and then use it internally. If all we did was to say:

```
typedef Foo{
    ...
}
```

and the next message also defined Foo internally, for PROMELA both would be global definitions. PROMELA is very much like C when it comes to scope. You have global scope and functional scope. C is a bit more than that, but PROMELA is not.

In order to implement these different definitions, the parent's type is prepended to the names. This creates unique names for all definitions, even if they do overlap.

For instance, within the core services the type "ResponseCode" is defined more than once. In the CommandClass it is defined within the "RejectElementRequest" message and also within the "ConfirmControl" message. This is not a problem with the JSIDL since the definitions are internal to the messages. The same thing is not possible for PROMELA, so the two types are renamed to urn_jaus_jss_core_MessageSet_CommandClass_RejectElementRequest_ResponseCode and urn_jaus_jss_core_MessageSet_CommandClass_ConfirmControl_ResponseCode, respectively. Notice that the message itself contains the ID for the DeclaredTypeSet to which it belongs. This eliminates the potential for naming overlap between type sets.

20.7.3.11 Action Function Overloading

PROMELA doesn't allow for function overloading but the JSIDL does. Many instances were found where a single function was given multiple types of parameters depending on the situation. In order to bypass this issue, the trigger for the transition is prepended to the action name. In addition actions where the Transport service is used to send data includes the string "Send_Action", while regular actions only have "Action". These modifications allow functions to be overloaded because the function names are no longer identical, but are unique.

21 Compact JSIDL and Eclipse Plug-in

Compact JSIDL (CJSIDL) is a grammar that provides C-like syntax for a user developing JAUS compliant service interfaces. The CJSIDL Eclipse Plug-in provides an editor for CJSIDL, access to JTS code and document generators, and conversion to and from JSIDL.

TABLE OF CONTENTS FOR CJSIDL

21.1 Overview

This section will provide a reference for CJSIDL syntax and a guide to using the CJSIDL Eclipse Plug-in.

21.2 Environment

For information on the environment for the source code, please reference the JTS Developer's Manual. When running the distributed binary of the plugin, all that is required is Eclipse 3.6 with the Xtext 1.0.2[5]. The JTS plugin has not been tested with versions of Eclipse later than 3.6.

21.2.1 Installation

To install the plugin, download the CjsidlEclipsePlugin.zip archive, and extract the files into the eclipse/ directory of your Eclipse installation. All of the necessary plugin files will be installed in the eclipse/plugins/ directory automatically.

After extracting the files, the JTS_COMMON_PATH system variable needs to be set up. Follow the instructions in section 4.1.8. with the Commons folder being located in eclipse/plugins/ org.jts.eclipse.data_1.0/templates/Common/.

Running the generated code from the Eclipse plug-in requires the same tools as JTS – please see section 4.1.2 for installation instructions for Java, Python, and SCons.

21.3 CJSIDL Grammar

The CJSIDL grammar follows the JSIDL 1.1 specification.

In this section regular expressions are used to explain the syntax of the grammar. With one marked exception, all capitalized text between angle brackets is text that is defined by the user. An explanation of what that text should contain will follow the example. Any other text and punctuation are keywords and expected markups.

A CJSIDL file is identified by the extension .csd or .cjsidl. Each file contains exactly one service definition, a declared type set, or a declared constant set. A service set may contain type and constant sets, but the file may only have one URI.

Unless otherwise noted, assume that all structures are *order dependent*.

Throughout the rest of this section, the following syntax will be used to describe the CJSIDL grammar elements:

[5] http://www.eclipse.org/Xtext/download.html

< > - used to denote a user defined element

Bold formatting – used for keywords, and required format characters in the grammar

* - indicates that there can be zero or more elements.

+ - indicates that there can be one or more elements.

? - indicates that there can be zero or one instances of an element.

21.3.1 Interpretations and Comments

In the following examples there will be references to optional <INTERPRETATIONS>. These are to be distinguished from regular single and multi line comments and are attached to the specific JSIDL object represented on the line immediately below the interpretation. Table 8 is a list of available comment syntax and coloring. Keep in mind that while CJSIDL supports comments, the comments are not included when converting to JSIDL, but interpretations are.

Table 8 CJSIDL Comments

Syntax	Description	Syntax highlighting
//	Single line comment	Green
/* … */	Multi line comment	Green
## ... ##	Interpretation	Blue

21.3.2 Declared Constant Set

The basic syntax for a declared const set is:

```
constants <NAME> (id = <URI>, version = <VERSION>){
<DECLARED CONSTANT SET REF>*
<CONSTANTS>*
}
```

where the ID is the name of the declared constant set, the URI is a valid URI such as “urn.jts.Ping”, and the version number is a decimal. Having a declared constant set reference is optional, and as many can be imported as necessary, but they must all be at the top of the file.

Defining a constant is also optional, and as many can be defined in the set as needed.

For the syntax of a constant set reference, see section 21.3.4.2

21.3.2.1 Constant Definition

A constant is a simple numeric type with an ID, a value, and a unit type. The syntax is:

```
<INTERPRETATION>?<numeric type> <NAME> = <value> <units>;
```

A list of numeric types and units can be found in Appendix B – Available Types and Units.

The value must be an integer or decimal value.

21.3.3 Declared Type Set

The basic syntax for a declared type set is:

```
typeset <NAME> (id = <URI>, version = <VERSION>){
<DECLARED CONST SET REFS>*
<DECLARED TYPE SET REFS>*
<TYPE DEFS>*
<TYPE REFERENCE>*
}
```

where the ID is the name of the declared constant set, the URI is a valid URI such as "urn.jts.Ping", and the version number is a decimal. A declared type set may have any number of references to declared constant sets or other declared type sets within the project. There can also be any number of type definitions and references to existing types as well.

The CJSIDL language is order dependent, with all declared constant set references needing to be defined first, followed by all declared type set references, followed by all type definitions, and ending with all type references.

For the syntax of a constant set reference, see section 21.3.4.2.

For the syntax of a type set reference, see section 21.3.4.3.

21.3.3.1 Type Definition

A type definition is the declaration of a message, array, record, list, variant, sequence, fixed field, variable field, bitfield, fixed length string, variable length string, variable length field, variable format field, header, body, or footer. Each of these has a unique declaration as defined in the following sections.

21.3.3.1.1 Optional Types

To define a type as being optional, use the keyword ‘optional’ at the start of the type definition, and after the interpretation.

For example:

```
## Comments for recExmpl##
optional record recExmpl{…};
```

```
optional field uint8 fieldExmpl one <0, 7> round;
```

Types that can use the optional keyword are:

Fixed length string	Variable length field	Variant
Variable length string	Variable format field	Sequence
Fixed field	Bitfield	Record
Variable field	List	Array

21.3.3.1.2 Message Definition

The syntax for a message is:

```
message <MESSAGE CODE> <NAME>{
description “STRING”;
<HEADER>?
<BODY>
<FOOTER>?
}
```

The message code is a hexadecimal value starting with “0x” and must be 4 hex digits in length, and the NAME is the name of the message.

The description is a string describing the intended use of the message.

The description, header, body, and footer definitions are order independent and only the body is required. The header and footer are optional.

For details on the syntax of the header, body, and footer see sections 21.3.3.1.15, 21.3.3.1.16, and 21.3.3.1.17 respectively.

Additionally, the keyword ‘command’ may be used at the start of the definition to indicate the message is a command:

```
command message <MESSAGE CODE> <NAME>{…}
```

21.3.3.1.3 Array Definition

The syntax for an array is:

```
<INTERPRETATION>?
<TYPE REFERENCE> <NAME>[SIZE];
```

The type reference is the name of an existing fixed field, variable length string, fixed length string, bitfield, variable format field, variable field, or variable length field that has already been defined. A reference to any other type will create an error.

The NAME is the name of the array, and the size is defined much like Java. For example:

```
## create a 5 element array of type ExampleStringType ##
ExampleStringType stringArray[5];
```

21.3.3.1.4 Record Definition

The syntax for a record is:

```
<INTERPRETATION>?
record <NAME>{
<TYPE DEFINITION or REFERENCE>+
}
```

A record may also be marked as optional by adding the keyword ‘optional’ in front of the definition.

A record must contain a minimum of one type definition or reference. The type can only be an array, fixed field, variable field, bitfield, fixed length string, variable length string, variable length field, or variable format field. The use of any other type inside a record will cause an error.

For details on the syntax of a type reference, see section 21.3.3.2.

21.3.3.1.5 List Definition

The syntax for a list is:

```
<INTERPRETATION>?
list <NAME> <COUNT INTERPRETATION>? [<MIN>, <MAX>]{
<CONTAINER TYPE>
}
```

where NAME is the name of the list, and INTERPRETATION is an optional comment. MIN and MAX are either references to constants or a numeric literal and bound the size of the list.

The CONTAINER TYPE is a reference or definition of a record, list, variant, or sequence.

21.3.3.1.6 Variant Definition

The syntax for a variant is:

```
<INTERPRETATION>?
variant <NAME> <COUNT INTERPRETATION>? [<MIN>, <MAX>]{
<VTAGS>*
}
```

where a VTAG is defined as:

```
vtag: <CONTAINER TYPE>
```

The CONTAINER TYPE is a either a reference or definition of a record, list, variant, or sequence.

21.3.3.1.7 Sequence Definition

The syntax for a sequence is:

```
<INTERPRETATION>?
sequence <NAME> {
<CONTAINER TYPE>+
}
```

The CONTAINER TYPE is a either a reference or definition of a record, list, variant, or sequence. This type is used to store an ordered list of container types. The order is determined by the order of definition within the sequence.

21.3.3.1.8 Fixed Field Definition

The syntax for a fixed field is:

```
<INTERPRETATION>?
field <NUMERIC TYPE> <NAME> <UNIT> <RANGE>?;
```

A list of NUMERIC TYPEs and UNITs can be found in Appendix B – Available Types and Units.

RANGE is optional, and is either a scaled range definition, value set definition, or value range.

21.3.3.1.8.1 Scaled range:

```
<INTERPRETATION>?
< <LOWER LIMIT>, <UPPER LIMIT> > <FUNCTION>
```

FUNCTION is ‘floor’, ‘ceiling’, or ‘round’. Note that for this example, the outer set of angular brackets is part of the language.

For example:

```
## a scaled range ##
<0, 10> floor
```

21.3.3.1.8.2 Value set definition

```
{<VALUE>+;} offset?
```

where VALUE is a value spec or value range, and at least one is defined. If using both value spec’s and value range’s, they are order independent in the list. The optional “offset” keyword determines if the offset_to_lower_limit flag is set in the corresponding JSIDL.

21.3.3.1.8.3 Value spec

```
<INTERPRETATION>?
<STRING> = <INT>
```

where STRING is any string and INT is an integer.

For example:

```
## a value found in a fixed field definition ##
“SOME STRING VALUE” = 5;
```

21.3.3.1.8.4 Value range

```
<INTERPRETATION>?
( <LOWER LIMIT>, <UPPER LIMIT> )
```

For a value range, either parentheses or square brackets can be used in place of the parentheses above, where a rounded parenthesis is exclusive, and a square bracket is inclusive. For example:

```
## A range from 0 to 5, excluding 0 and 5##
( 0, 5 )
```

```
## A range from 0 to 5, excluding 0 and including 5##
( 0, 5 ]
```

```
## A range from 0 to 5, including 0 and excluding 5##
[ 0, 5 )
```

```
## A range from 0 to 5, including 0 and 5 ##
[ 0, 5 ]
```

21.3.3.1.9 Variable Field Definition

The syntax for a variable field is:

```
<INTERPRETATION>?
variable_field <NAME> {
<TAGS>+
};
```

A variable field contains at least one tag, which is defined as

```
tag <INT> : <NAME> <NUMERIC TYPE> <UNIT> <VALUE SET/SCALED RANGE>?;
```

The INT can be an integer or a reference to a defined constant. A list of numeric types and units can be found in Appendix B – Available Types and Units.

Additionally, the tag can include an optional value set definition or scaled range definition after UNIT. The syntax for these can be found in sections 21.3.3.1.8.2 and 21.3.3.1.8.3, respectively.

21.3.3.1.10Bitfield Definition

The syntax for a bitfield is:

```
<INTERPRETATION>?
bit_field <TYPE> <NAME>{
<SUBFIELD>*
}
```

In a bitfield, the TYPE must be a uint8, uint16, uint32, or uint64.

The subfield is defined as:

```
<INTERPRETATION>?
<NAME> [ <INT>: <INT>] <VALUE RANGE> ;
```

The INT values are the start and end bit positions that this subfield occupies within the bitfield. For the syntax of the VALUE RANGE see section 21.3.3.1.8.4.

21.3.3.1.11 Fixed Length String Definition

The syntax for a fixed length string is:

```
<INTERPRETATION>?
string <NAME> [ <INT> ];
```

The INT value determines the size of the string.

21.3.3.1.12 Variable Length String Definition

The syntax for a variable length string is:

```
<INTERPRETATION>?
vstring <NAME> [ <INT>, <INT> ];
```

The INT values determine the minimum and maximum size that the string will have.

21.3.3.1.13 Variable Length Field Definition

The syntax for a variable length field is:

```
<INTERPRETATION>?
```

```
vfield <FIELD FORMAT> <NAME>
<COUNT INTERPRETATION>? [<MIN>, <MAX>]? ;
```

The count and its interpretation are optional. If a minimum count is specified, the maximum count remains optional.

21.3.3.1.14 Variable Format Field Definition

The syntax for a variable format field is:

```
<INTERPRETATION>?
variable_format_field <NAME> {
<COUNT INTERPRETATION>?
<INT TYPE> tag <VALUE RANGE>;
<FORMAT ENUM> +
}
```

where INT TYPE is uint8, uint16, or uint32, and the syntax for a VALUE RANGE can be found in section 21.3.3.1.8.4. The range must fit within the INT TYPE that is specified. In addition, the range must encompass all the tagged enum values that follow in the definition.

FORMAT ENUM can be defined as:

```
tag <INT> : <FIELD FORMAT or STRING>
```

INT can also be a constant reference, and a list of FIELD FORMATs can be found in Appendix B – Available Types and Units.

21.3.3.1.15 Header Definition and Reference

The syntax for a header is:

```
<INTERPRETATION>?
header{
    <CONTAINER TYPE>?
}
```

The CONTAINER TYPE is a either a reference or definition of a record, list, variant, or sequence

A reference to an existing header is defined as:

```
<INTERPRETATION>?
```

```
header <REF> <NAME>;
```

REF is the name of a header that has already been defined, and the NAME is the identifier of the header reference.

21.3.3.1.16 Body Definition

The syntax for a body is:

```
<INTERPRETATION>?
body{
    <CONTAINER TYPE>?
}
```

The CONTAINER TYPE is a either a reference or definition of a record, list, variant, or sequence

A reference to an existing body is defined as:

```
<INTERPRETATION>?
body <REF> <NAME>;
```

REF is the name of a body that has already been defined, and the NAME is the identifier of the body reference.

21.3.3.1.17 Footer Definition

The syntax for a footer is:

```
<INTERPRETATION>?
footer{
    <CONTAINER TYPE>?
}
```

The CONTAINER TYPE is a either a reference or definition of a record, list, variant, or sequence

A reference to an existing footer is defined as:

```
<INTERPRETATION>?
footer <REF> <NAME>;
```

REF is the name of a header that has already been defined, and the NAME is the identifier of the footer reference.

21.3.3.2 Type Reference

A type reference is simply an instance of an already defined type. The syntax is

```
<INTERPRETATION>?
<TYPE NAME> <NAME>;
```

where TYPE NAME is the name of the type being referenced, and NAME is the name of the new type reference.

21.3.4 Service Definition

The basic structure for a service definition is:

```
service <NAME> (id = <URI>, version = <VERSION>){
    description = "STRING";
    assumptions = "STRING";
    <REFERENCES>?
    <CONSTANT SET>?
    <TYPE SET>?
    <INTERPRETATION>?
    messages{
        <INTERPRETATION>?
        input{
            <MESSAGE>*
        }
        <INTERPRETATION>?
        output{
            <MESSAGE>*
        }
    }
    <INTERPRETATION>?
    eventset{
        <EVENT>*
    }
    <INTERPRETATION>?
    protocol {
        <STATE MACHINES>+
    }
}
```

The description and assumptions must be enclosed in double quotes and followed by a semicolon.

Inside the message set, no MESSAGE has to be defined, however CJSIDL requires that the messages {…} structure exist. For details on the syntax for a MESSAGE, see section 21.3.3.1.1.

The event set structure is also required, even if no EVENT is defined. For details on the syntax for an EVENT, see section 21.3.4.4.

21.3.4.1 References

Service definitions can reference other existing service definitions within the project.

```
references{
    inherits_from <INTERPRETATION>? import <URI> as <NAME>;
    client_of <INTERPRETATION>? import <URI> as <NAME>;
}
```

Both inherits_from and client_of are optional, and there can be as many clients as necessary. Only one service can be inherited.

For both imports, the URI must be the URI of a service definition located within the existing project. The NAME is the id that will be used to reference objects within the service definition throughout the file.

21.3.4.2 Declared Constant Set References

The declared constant set reference is essentially the import of a constant set located in the project. They syntax is:

```
<INTERPRETATION>?
using constants <URI> as <NAME>;
```

where <URI> is the URI of an existing constant set within the project. If the URI belongs to a type set, a service set, or is otherwise invalid the editor will mark the line with an error.

21.3.4.3 Declared Type Set References

The declared type set reference is essentially the import of a type set located in the project. They syntax is very similar to the declared constant set reference:

```
<INTERPRETATION>?
using typeset <URI> as <NAME>;
```

where <URI> is the URI of an existing type set within the project. If the URI belongs to a constant set, a service set, or is otherwise invalid the editor will mark the line with an error.

21.3.4.4 Events

The syntax for an event is very similar to that of a message.

```
<INTERPRETATION>?
event <NAME> {
    description = “STRING”;
    <HEADER>?
    <BODY>
    <FOOTER>?
}
```

HEADER, BODY, and FOOTER are order independent, and only the BODY is required. They can be references or definitions, and the syntax for them can be found in sections

For details on the syntax of the header, body, and footer see sections 21.3.3.1.15, 21.3.3.1.16, and 21.3.3.1.17 respectively.

To reference an existing event inside the eventset, use the following syntax:

```
<INTERPRETATION>?
eventset <REFERENCE> <NAME>;
```

REFERENCE is the name of an existing event, or a scoped name of an event that exists in an inherited service definition (ex: serviceDefA.serviceDefB.eventName). NAME is the id of this reference.

21.3.4.5 Protocol Behavior

The protocol behavior in CJSIDL is a collection of state machines and states built with transitions, guards, and actions.

The overall syntax for the protocol behavior is:

```
<INTERPRETATION>?
<STATELESS>? protocol{
<STATE MACHINE>+
}
```

where STATELESS is the optional keyword ‘stateless’, and STATE MACHINE is the definition of at least one state machine as described in section 21.3.4.5.6.

21.3.4.5.1 Actions

An action is defined as:

```
<INTERPRETATION>?
<ID | URI> (<PARAMETER>*);
```

where an ID or a URI can be used to identify the action, and zero or more PARAMETERs are used. Parameters are separated by commas and are either a reference to an available transition parameter, or a string.

21.3.4.5.2 Guards

A guard is defined as:

```
<INTERPRETATION>?
guard : <FUNCTION> <EQUIV> <FUNCTION>;
```

EQUIV is “=” or “!=”. The functions are user defined and can contain parameters that were passed into the transition that contains this guard or the parameter can be a string.

A guard can also be defined as:

```
<INTERPRETATION>?
guard : <FUNCTION> <<LOGICAL OPERATOR> <FUNCTION>>*;
```

where LOGICAL OPERATOR>is && or ||.

In both cases, `FUNCTION` is defined as

```
<NAME> (<PARAM>+)
```

where PARAMs are separated by commas, and a ‘!’ can be added to the front of the `FUNCTION` as logical negation.

For example:

```
guard : exampleFunc("string") = anotherFunc("string2");
```

or

```
guard : exampleFunc("string") || !anotherFunc("string2");
```

21.3.4.5.3 Transitions

There are four kinds of transitions available: internal, simple, push, and pop. They all have similar syntax in CJSIDL.

```
<INTERPRETATION>?
<TYPE> transition <NAME> (<PARAM>* ){
    <GUARD>?
    <ACTIONS>
    <SEND ACTIONS>?
    <DESTINATION>
}
```

The TYPE is the keyword internal, simple, push, or pop. The syntax for a GUARD can be found in section 21.3.4.5.2.

SEND ACTIONS and ACTIONS are order independent. Both are lists of actions, as defined in section 21.3.4.5.1. The syntax for a SEND ACTION list is

```
send actions{
    <ACTION>*
}
```

The ACTIONS syntax is:

```
actions{
    <ACTION>*
}
```

Multiple PARAMs are separated by commas. The PARAM can be a reference to a defined type, or an integer with an ID. Valid integers are uint8, uint16, uint32, and uint64.

For example:

```
(recExample rec, ## ref comment ## strExample str, uint8 num)
```

DESTINATIONS vary based on the type of transition. See Table 9 CJSIDL Transition Destination Syntax for details on the correct syntax for each transition.

Table 9 CJSIDL Transition Destination Syntax

Transition Type	DESTINATION syntax
Internal	`<INTERPRETATION>?`

Simple	`<INTERPRETATION>?`
	`next` `<STATE REF>;`

Push	`<INTERPRETATION>?`
	`popto next` `<STATE REF>;`

Pop	`<INTERPRETATION>?`
	`secondary` `<TRANSITION REF> (<PARAM>+);`

Note that the keyword ‘next’ is interchangeable with the keyword ‘->’. PARAM is defined identically to the PARAM in the transition definition above.

STATE REF is a reference to another state in the state machine. It will be scoped to ensure that the correct state is called (ex: StateA.StateB.StateC, where States A, B, and C are nested).

A TRANSITION REF is a reference to another transition in the state that contains the push transition that originally entered the state. For full details on the functionality of push and pop transitions, see [5684] AS5684.

21.3.4.5.4 State

The syntax for a state is:

```
<INTERPRETATION>?
state <NAME>{
<ENTRY ACTIONS>?
<EXIT ACTIONS>?
<TRANSITION>*
<DEFAULT TRANSITIONS>*
<DEFAULT STATE>?
<SUB STATE>*
}
```

21.3.4.5.5 Default State

The DEFAULT STATE is defined as:

```
<INTERPRETATION>?
state default{
    <TRANSITION>*
    <DEFAULT TRANSITION>*
}
```

The DEFAULT TRANSITION is defined exactly like a normal transition, except with the keyword ‘default’ between the interpretation and the transition type. For example:

```
## Simple transition example ##
default simple transition {
    actions{}
    -> NextStateName.SubStateName;
}
```

21.3.4.5.6 State Machine

The syntax for a state machine is:

```
<INTERPRETATION>?
state_machine <NAME> (<START STATE>){
<DEFAULT STATE>?
<STATE>*
}
```

The START STATE is the reference to the first state the state machine will be in. This reference will be scoped if the state is nested (ex: StateA.StateB.StateC).

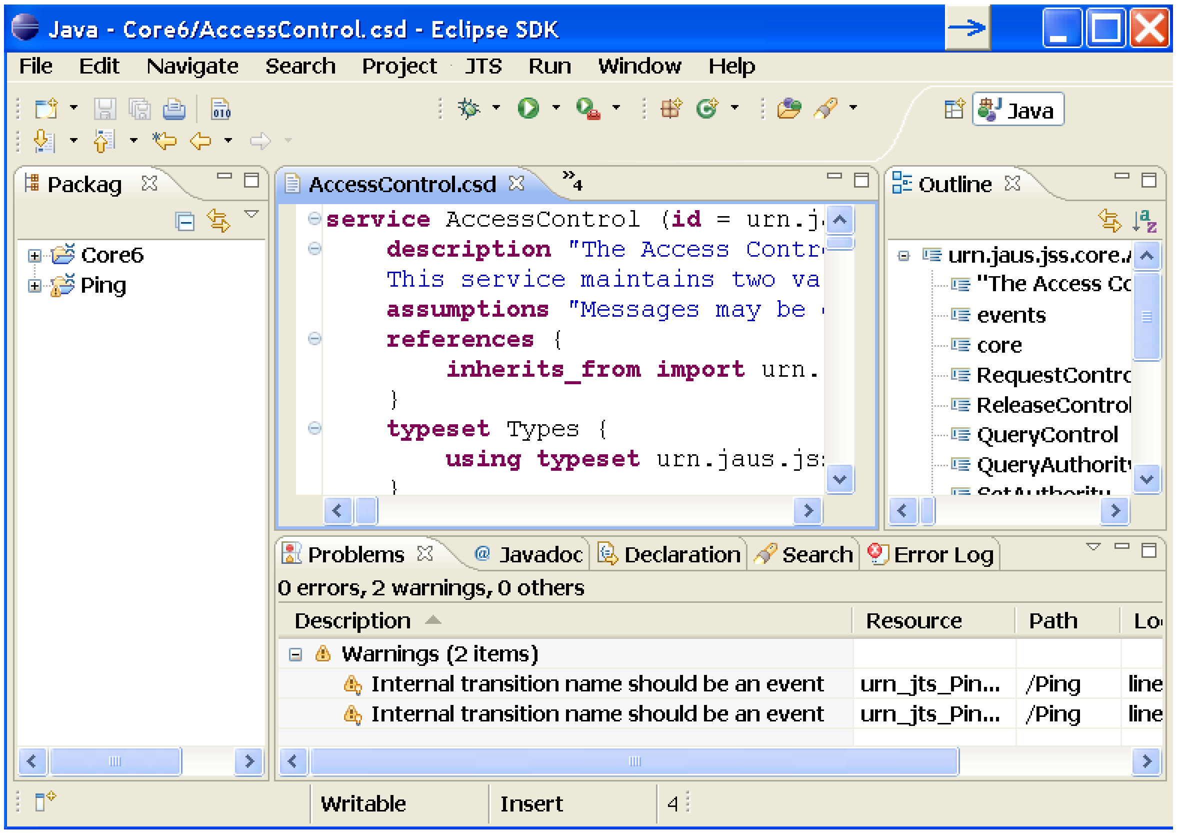
The syntax for a STATE can be found in section 21.3.4.5.4.

The syntax for a DEFAULT STATE can be found in section 21.3.4.5.5.

21.4 Eclipse Plug-in

The CJSIDL Eclipse plug-in can be broken down into three components – the JTS menu, the new project and file wizard, and the editor. The editor is the core of the plug-in, the JTS menu provides the JTS functionality and the ability to export CJSIDL to JSIDL, and the wizards provide a convenient GUI to create new CJSIDL projects and files, or import a project from JSIDL.

Figure 146 shows an instance of Eclipse running the CJSIDL plug-in. In the top bar, the JTS menu is visible, and the basic syntax highlighting of the .csd file is visible. On the left side of the window is the Package Explorer, which shows the projects that are active in the workspace.


Java - Core6/AccessControl.csd - Eclipse SDK
File Edit Navigate Search Project JTS Run Window Help
Java
Packag
Core6
Ping
AccessControl.csd
Outline
Problems
Javadoc
Declaration
Search
Error Log

```
service AccessControl (id = urn.j
    description "The Access Contr
    This service maintains two va
    assumptions "Messages may be 
    references {
        inherits_from import urn.
    }
    typeset Types {
        using typeset urn.jaus.js
    }
```

```
urn.jaus.jss.core.A
    "The Access Co
    events
    core
    RequestContro
    ReleaseControl
    QueryControl
    QueryAuthorit
```

0 errors, 2 warnings, 0 others

Description	Resource	Path	Lo
Warnings (2 items)			
Internal transition name should be an event	urn_jts_Pin...	/Ping	line
Internal transition name should be an event	urn_jts_Pin...	/Ping	line

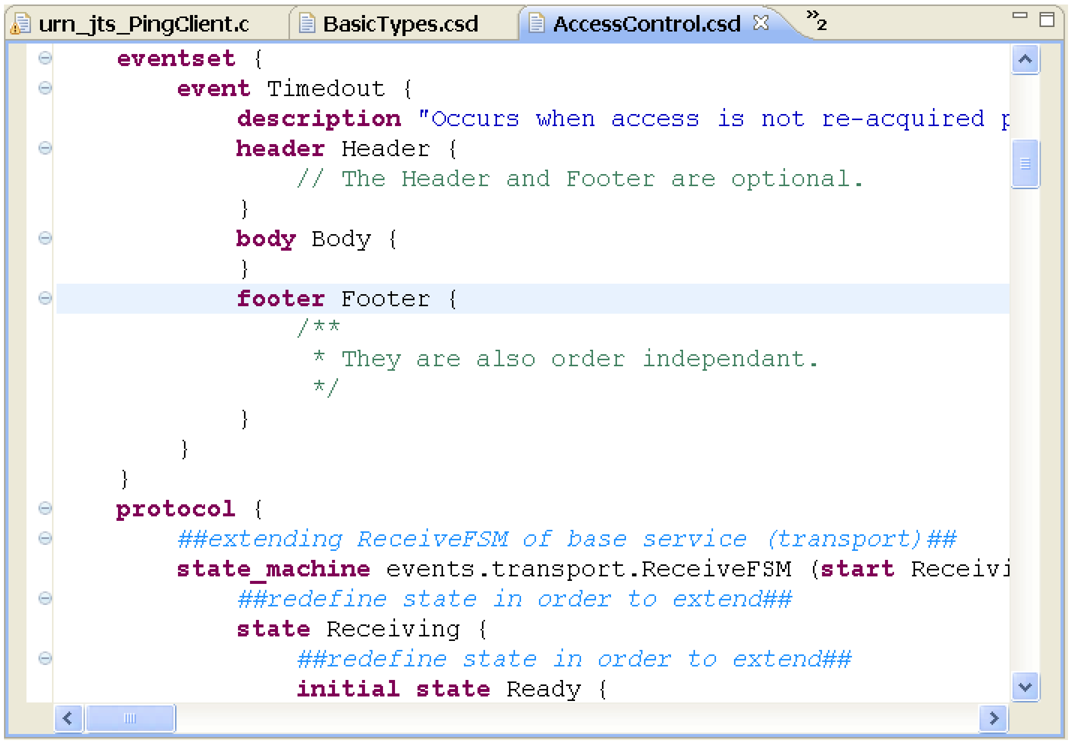
Writable | Insert | 4

Figure 146: CJSIDL Eclipse Plug-in

21.4.1 Editor

The CJSIDL is a full editor that has been generated from the CJSIDL grammar through XText. It provides syntax highlighting, formatting, auto completion, and validation. Figure 147 shows an instance of the editor that displays all of the comment styles and syntax highlighting.

Also note that the editor is outlined with a light blue bar – this indicates that the editor is selected. If the Package Explorer, Outline, or another window is selected within Eclipse, it will be outlined in the blue bar. When using the JTS menu it is important that either the editor or a project inside the Package Explorer is selected. That is how the menu knows which project to convert to JSIDL.



```
    eventset {
        event Timedout {
            description "Occurs when access is not re-acquired p
            header Header {
                // The Header and Footer are optional.
            }
            body Body {
            }
            footer Footer {
                /**
                 * They are also order independant.
                 */
            }
        }
    }
    protocol {
        ##extending ReceiveFSM of base service (transport)##
        state_machine events.transport.ReceiveFSM (start Receivi
            ##redefine state in order to extend##
            state Receiving {
                ##redefine state in order to extend##
                initial state Ready {
```

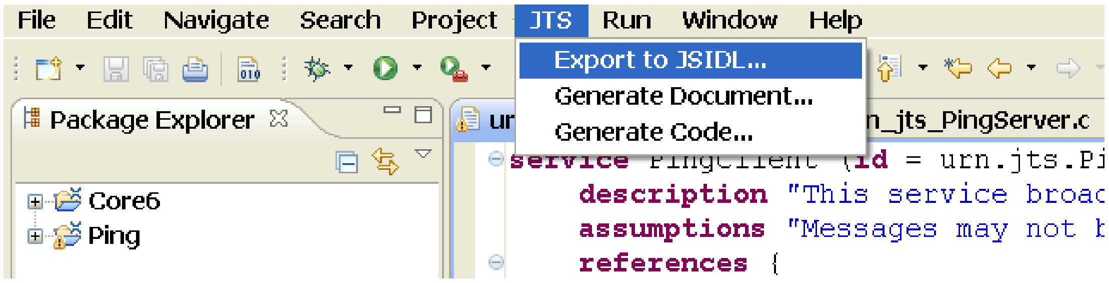
Figure 147 CJSIDL Editor Window

To access the auto complete feature, press ctrl + space while in the editor. This will pull up a list of objects that can be referenced.

To activate the white space formatting, either right click in the editor and select “Format”, or while the editor is selected press ctrl+shift+s.

21.4.2 JTS Menu

The JTS Menu located at the in the toolbar along with File, Edit, Source, etc. is the location of all the JTS functionality included with the plug-in. The JTS code generator, document generator, and the ability to export to JSIDL are all found in this menu, as shown in Figure 148.



```
service PingClient (id = urn.jts.Pi
    description "This service broac
    assumptions "Messages may not b
    references {
```

Figure 148: Eclipse Plug-in JTS Menu

The code and document generators behave exactly as in JTS – this section provides information on the Eclipse plug-in GUI provided. For detailed information on utilizing generated code and the functionality of the generators, please see sections 4, 17, and 18.

Please note that all three functionalities will use all of the files located in the selected project, as the plug-in is project based. If a file should be excluded from the task, it should first be removed from the project.

21.4.2.1 Generate Code

The code generator included in the CJSIDL plug-in is the same generator used for JTS, and produces the same code. For details on how to modify the generated code and set up the environment, please refer to sections 4.1.8, 4.1.9, 7.3, and 9 (excluding information on using the JTS GUI).

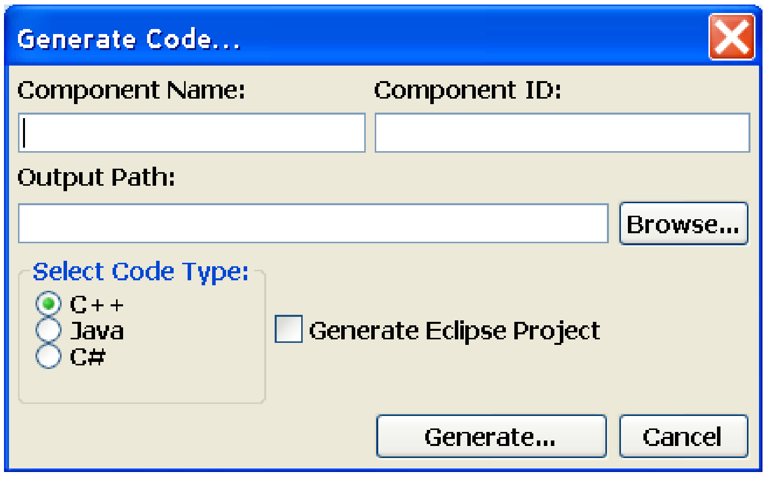


Figure 149: CJSIDL Plug-in Code Generator

When choosing a component name, use a name that will make a valid C++ or Java class name. For the component ID, an integer between 1 and 256 must be used.

The output path must be a valid folder that exists. Because of limitations in a third party tool used by the generator, the output path cannot contain any spaces.

If running Eclipse on Windows, code in C++, Java, or C# may be generated.

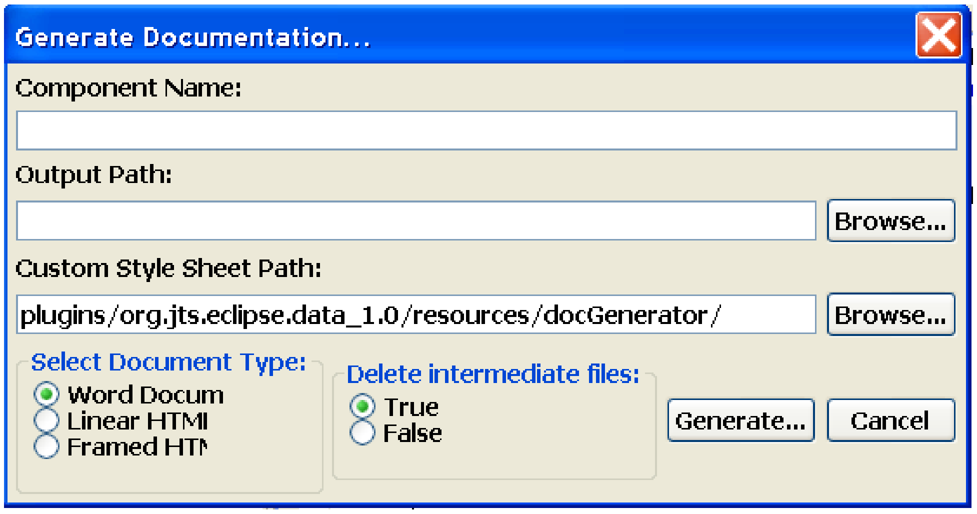
If running Eclipse on a Linux system, code in C++ or Java may be generated. C# code may be generated as well, but is untested.

For any generated language, selecting Generate Eclipse Project will compile the generated code and import the generated project as an Eclipse project. If the project is in Java, then Eclipse will apply the appropriate Java functionality to the project and generate an Ant script to call SCons. Otherwise, it is up to the user to ensure that the correct Eclipse plug-in is available for development in C++ or C#.

In order to compile the generated code Eclipse needs to be able to find the SCons program. The first time an Eclipse project is generated a pop-up will appear asking for the location of the SCons program. By default it is located in the same directory as the Python installation, in the folder Scripts/. On Windows the file needed is scons.bat.

21.4.2.2 Generate Documents

Like the code generator, the document generator is the same generator used by JTS. For full details of the output and use of the generated files, refer to section 18. This includes details on the Custom Style Sheet path.


Generate Documentation...

Component Name:

Output Path:

Custom Style Sheet Path: plugins/org.jts.eclipse.data_1.0/resources/docGenerator/

Select Document Type: [✓ Word Docum] Linear HTMl Framed HTM

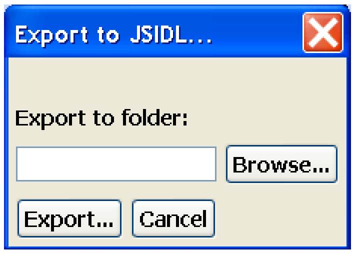
Delete intermediate files: [✓ True] False

Figure 150: CJSIDL Plug-in Document Generator

The document generator requires the name of the component, and the output path for the files. The name should be the same name that would be used for the code generator. The output path must be a folder and must exist. Given the number of files generated, it is recommended that the output path be a dedicated documentation folder.

21.4.2.3 Export to JSIDL

The window for exporting to JSIDL is very simple. All that is required is a valid output path, and for a valid project to be selected. A valid output path is one that exists and contains no spaces.


Export to JSIDL...

Export to folder:

Figure 151: CJSIDL Plug-in Export to JSIDL

21.4.3 New Project and File Wizards

In order to create a new file or project for CJSIDL, either select File -> New -> Other… from the top menu of Eclipse or press ctrl+n. In the window that appears, there will be a folder labeled CJSIDL as shown in Figure 152. Select the desired wizard, and click next.

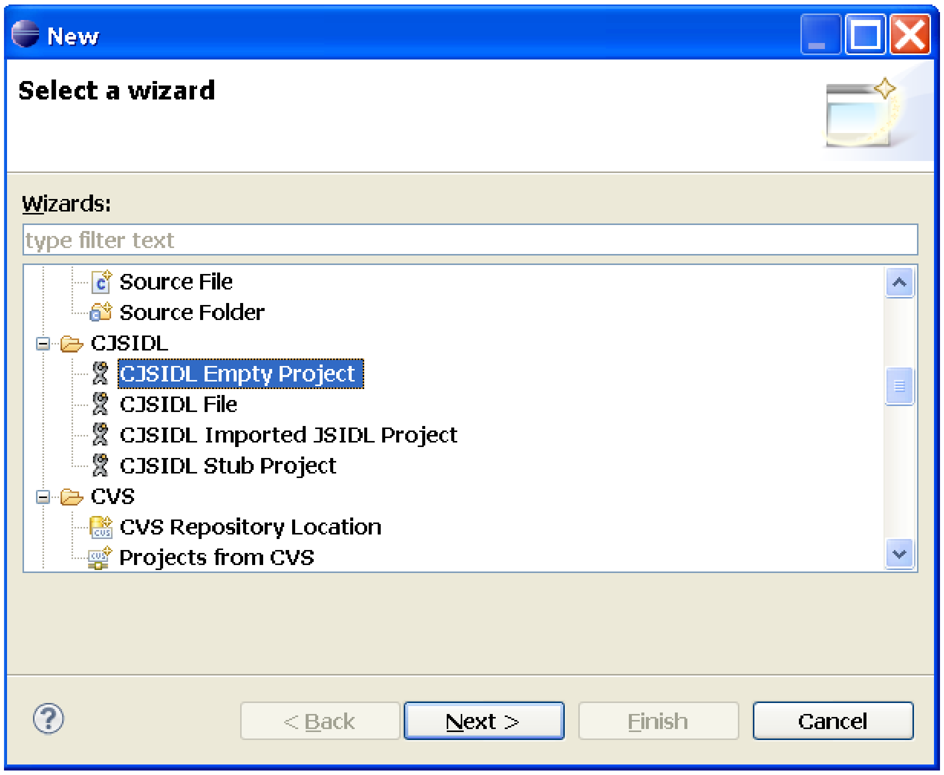
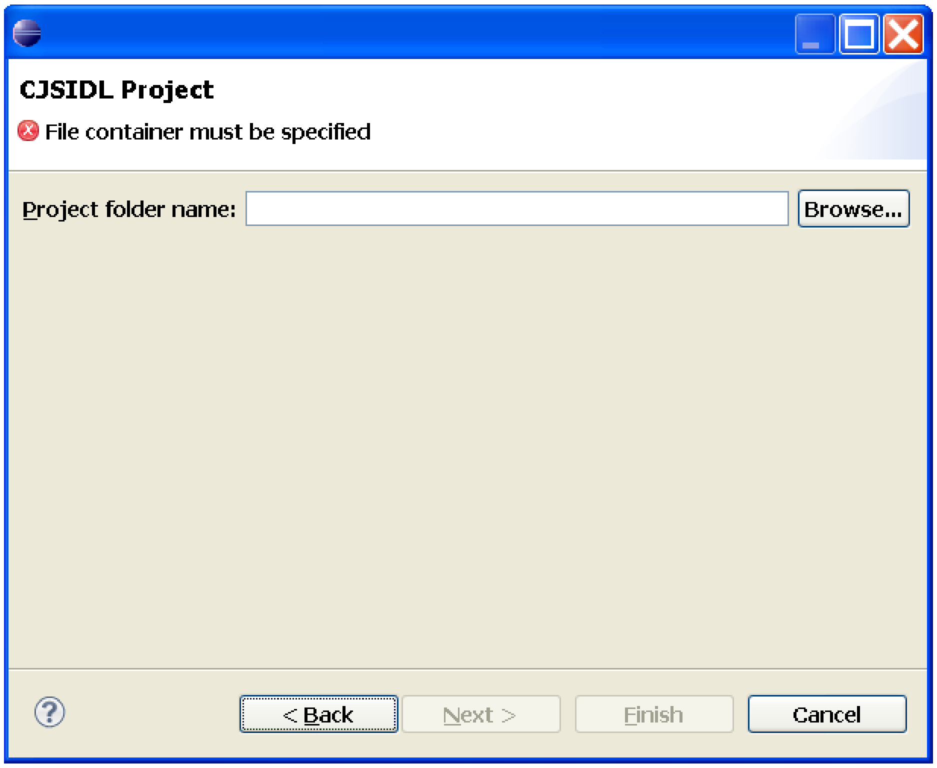


Figure 152: CJSIDL New Window

21.4.3.1 New Empty Project

When creating a new empty project, only the name of the new project is required. The browse button is available to review the existing project names, but the new project name must be unique. Click Finish to create the project.



CJSIDL Project

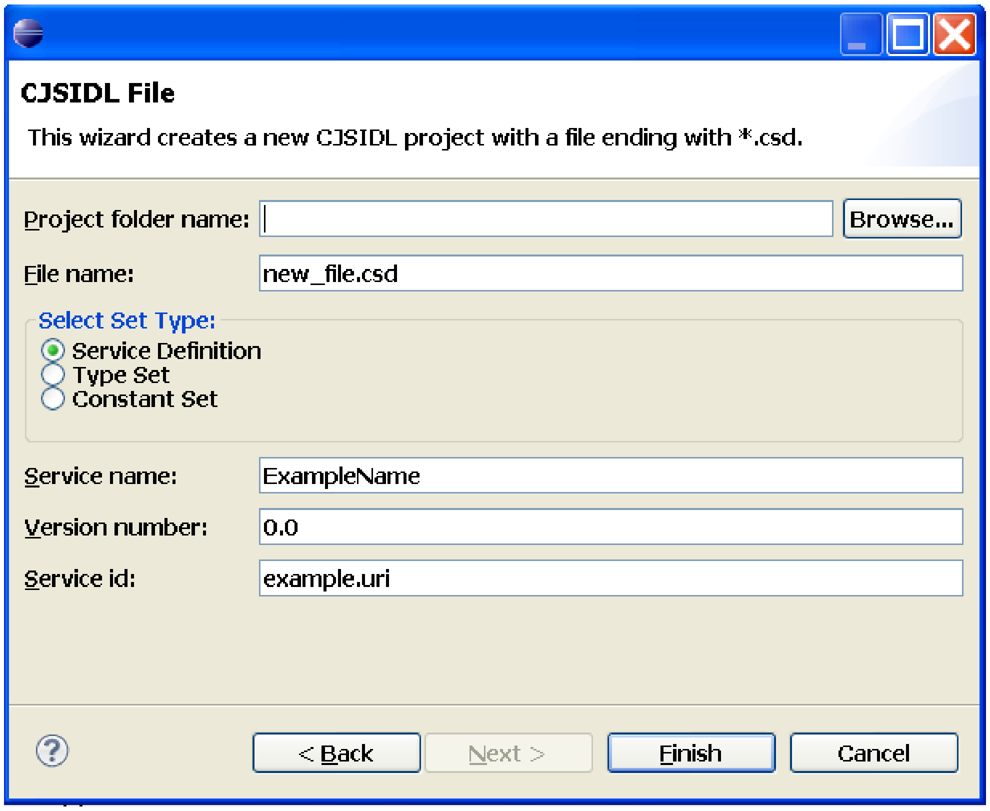
File container must be specified

Project folder name:

Figure 153: CJSIDL New Empty Project Wizard

21.4.3.2 New Stub Project

A new stub project will contain one bare-bones file with the given information. A new project name must be provided, and a file name. As shown in Figure 154 a service definition, constant set, or type set may be created. Clicking Finish will create the new project and file, then open the file in the editor.



CJSIDL File

This wizard creates a new CJSIDL project with a file ending with *.csd.

Project folder name:

File name: new_file.csd

Select Set Type:
[✓ Service Definition]
Type Set
Constant Set

Service name: ExampleName

Version number: 0.0

Service id: example.uri

Figure 154: CJSIDL New Stub Project Wizard

21.4.3.3 New Imported Project

The only way to import existing JSIDL is to create a new project. The project name must be unique. The imported JSIDL must be a folder, and all JSIDL inside the folder and subfolders will be imported into the project.

Clicking Finish will create the new project, convert the JSIDL to CJSIDL and open the CJSIDL files in the editor.

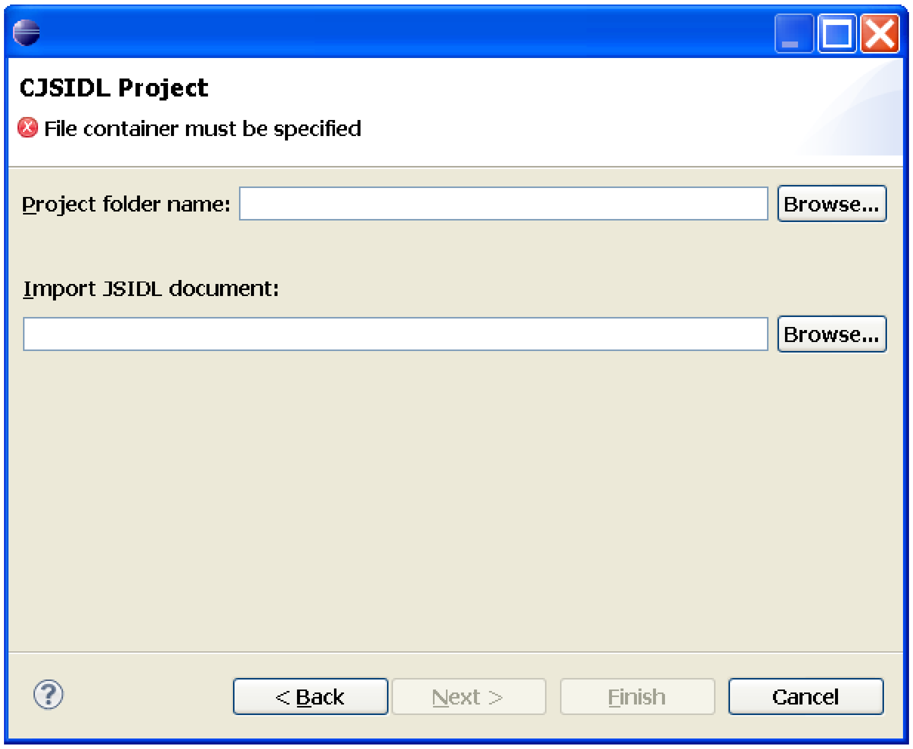


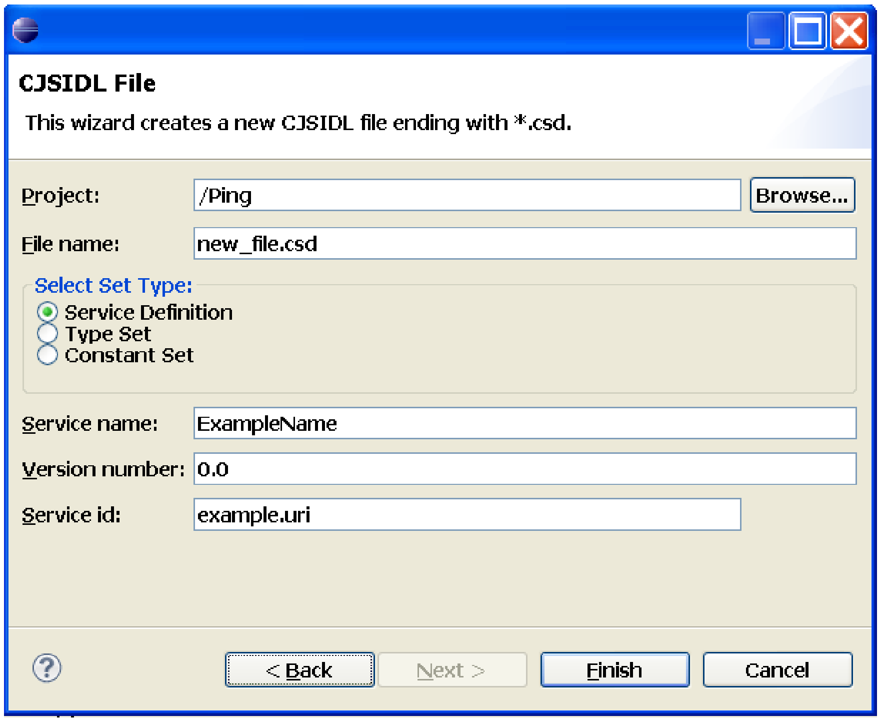
Figure 155: CJSIDL New Imported Project Wizard

Note: There is a known issue when importing JSIDL that includes cross-file references (such as using inherits_from and client_of references, or links to constant or reference sets). The editor will not be able to immediately resolve those links, rendering the files difficult to edit. The work-around it to close Eclipse and restart it. This will initialize the files correctly.

To confirm that this is the problem – open the Error Log view (Window -> Show View -> Error Log) and the most recent error should be “org.eclipse.xtext.build.erimpl.XtextBuilder – Passed org.eclipse.xtext.resource.IResourceDescriptions is not of type org.eclipse.xtext.resource.impl.ResourceSetBasedResourceDescriptions”

21.4.3.4 New File

The new file wizard will create a new stub file, exactly like the one created by the new stub project wizard. The stub file is a bare-bones file filled out with the given information. The project name is automatically filled out with the current selected project, but can be changed by selecting Browse and choosing a different project. As shown in Figure 156 a service definition, constant set, or type set may be created. Clicking Finish will create the new project and file, then open the file in the editor.



CJSIDL File

This wizard creates a new CJSIDL file ending with *.csd.

Project: /Ping

File name: new_file.csd

Select Set Type:
[✓ Service Definition] Type Set Constant Set

Service name: ExampleName

Version number: 0.0

Service id: example.uri

Figure 156: CJSIDL New File Wizard

21.4.4 Troubleshooting

There are a few known issues in the plug in. These issues are related to the conversion of JSIDL and CJSIDL, and accessing the code and document generators.

21.4.4.1 Link Resolution

If the plug in is not resolving a link (to an external file or internally to the file) there are a few steps that can be taken:

1. Ensure that the link is valid. Does a file with that URI exist in the project? Does that type exist in the correct scope? Note that the editor can not resolve links to files outside of a project.
2. If the link appears to be valid, open the error log:
   - Select the drop down menu “Window”
   - Select “Show View”
   - Click “Error Log”
   - If the error log contains the following error, restart Eclipse:

```
org.eclipse.xtext.builder.impl.XtextBuilder
- Passed org.eclipse.xtext.resource.IResourceDescriptions not of type
org.eclipse.xtext.resource.impl.ResourceSetBasedResourceDescriptions
```

21.4.4.2 Errors within the editor

If there has been an error with converting CJSIDL to or from JSIDL or with the code or document generators, it is possible that the error has disrupted the workspace and affected the functionality of the editor.

To resolve the issue, there are two steps. In most cases the first should resolve the problem.

1. Clean the project
2. Restart Eclipse

21.4.4.3 Errors converting JSIDL

When importing or exporting JSIDL, it is necessary to restart Eclipse. CJSIDL to JSIDL conversion is used in all functions in the JST menu, and a restart is required after running them. For full details, please refer to the Developer’s Manual.

If Eclipse is not restarted, Eclipse will begin logging errors as described in section 21.4.4.1.

If the plug-in is unable to import the JSIDL, confirm that the JSIDL is 1.1 and not 1.0, as 1.0 will cause errors when being unmarshalled.

If the JSIDL being imported has several layers of states that have duplicate names, it is necessary to rename them to have unique IDs. Otherwise there will be problems with the parser being able to differentiate between identically named states.

21.4.4.4 Errors with generating code, documents, or JSIDL

When performing any of the functions under the JTS menu, the converter is called to create JSIDL bindings. This uses the same workaround discussed in the Developer’s Manual that requires a restart of Eclipse.

If there are any errors other than the pop up alert regarding the conversion, please confirm that the data in the project is valid.

21.5 CJSIDL Integration with JTS

In order to simplify the workflow between JTS and the Eclipse plug-in, the ability to import and export CJSIDL has been integrated into the JTS GUI.

21.5.1 Import

CJSIDL can be imported into JTS by right clicking on the Service Defs menu as shown in Figure 157. This figure also shows the popup menu.

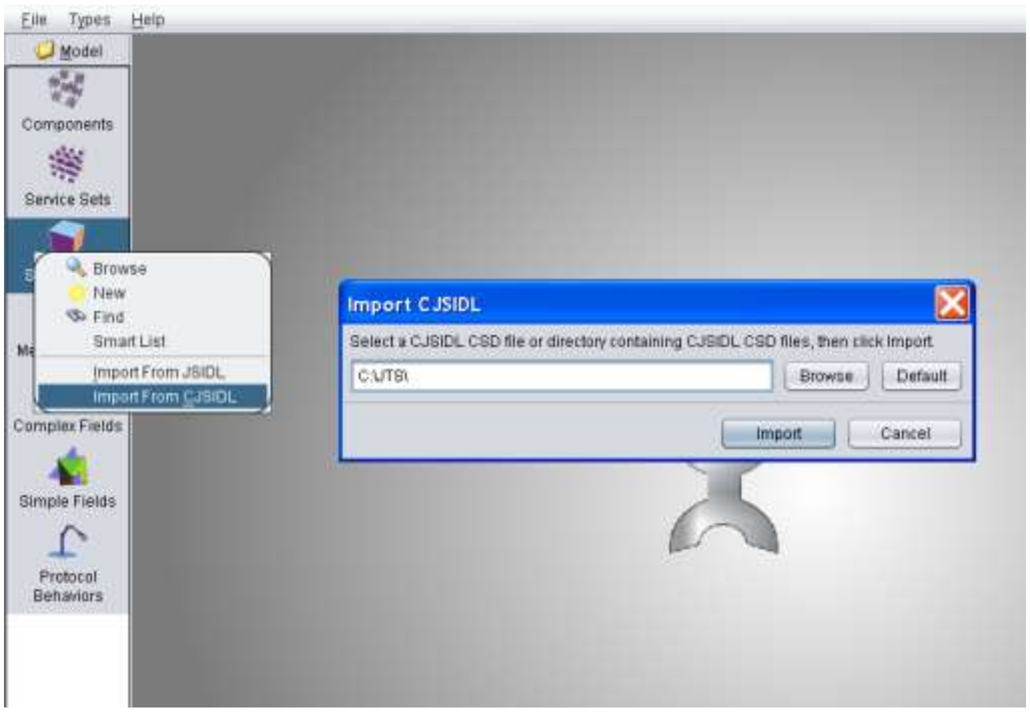
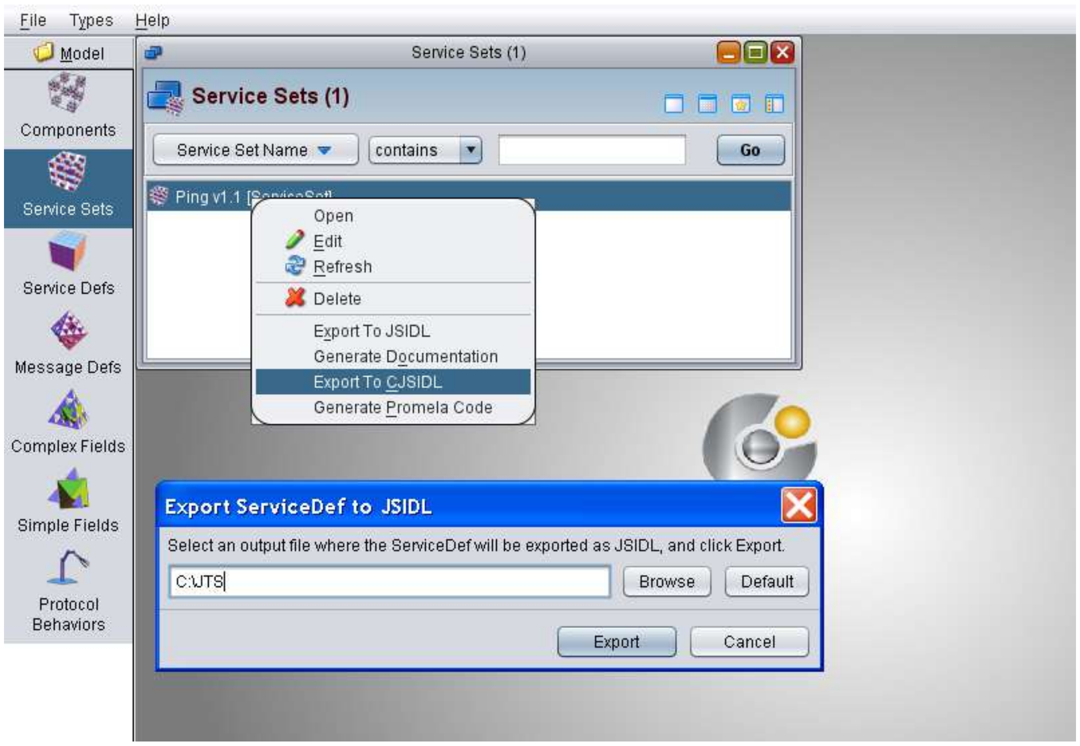


Figure 157: Importing CJSIDL to JTS

Invalid CJSIDL will cause an error to be displayed and the import to fail.

21.5.2 Export

JTS can export JSIDL as CJSIDL for use in the Eclipse plug-in by right clicking on a service set and selecting the menu option “Export to CJSIDL”. Figure 158 shows the location of the Export to CJSIDL option and the pop up dialog.



Select an output file where the ServiceDef will be exported as JSIDL, and click Export.

C:\JTS

Figure 158: Exporting CJSIDL from JTS

22 Appendix A – Command Line Input

Some modules in JTS can be used independently of the GUI through the command line. This section breaks down the command line options for each module.

22.1 Code Generator

To call the code generator from the command line, use the following command from GUI/:

Command line options for the code generator include:

Table 10 Code Generator Command Line Options

Compiler option	Description	Default value
--input, - i	The path to the service set	
--outdir, -o	The path to the generated output	creates a folder titled generatedOutput in GUI/
--name, -n	The name of the component	JAUSComponent
--id	The component ID	1
--c++	Generates C++ code	
--java	Generates Java code	
--cs	Generates C# code	

So an example use would be:

```
ant run-code-generator –i examples/foo.xml –o /test/4_11_11/ -n TestCmpt –
id 200 –java
```

This will unmarshal the JSIDL foo.xml, the generated code will be placed in GUI/test/4_11_11/TestCmpt_200, and the generated code will be in Java.

22.2 Document Generator

There are two ways to launch the document generator – through the command `ant run-doc-generator` and directly through the Python script located in GUI/scripts. This script will properly call the more complex Ant task. These instructions are for the Python script, which is the recommended means of invoking the document generator.

To launch the python command, run `cd GUI/scripts` to enter the scripts directory and run the following command: `python DocGeneratorCLI.py –o [DIR] [optional] --[lhtml/fhtml/word] [service defs] .`

Note that [service defs] must be the path of at least one service definition JSIDL. Multiple files may be listed, but attempting to use a service set or any other kind of JSIDL XML will cause errors.

The following table is a list of all available command line options for the document generator:

22.2.1.1.1.1.1 Table 11 Document Generator Command Line Options

Table 9 – Document Generator command line options

Compiler option	Description	Default value	Required
-o, --output [DIR]	Specifies the output director for the generated document.	n/a	Yes
--styleCust [DIR]	The directory of a customized style sheet for the document. More information can be found in the JTS User's Guide.	n/a	No
--keepIntermeds	A flag to specify if intermediate forms of generated documents used to create the final document should be kept or deleted.	False	No
--lhtml	Generates linear HTML output.	n/a	Yes
--fhtml	Generatess framed HTML output.	n/a	Yes
--word	Generates an MS Word document.	n/a	Yes

Please note that --lhtml, --fhtml, and –word are mutual exclusive. One must be selected, but only one can be selected.

So, an example use in Windows would be:

```
cd GUI\scripts

python    DocGeneratorCLI.py    –o    "C:\document_output"    --styleCust
"C:\document_style_cuts"      --word      "C:\service_defs\service1.xml"
```

```
"C:\service_defs\service2.xml"
```

Another exampel use in Linux would be:

```
cd GUI/scripts

python     DocGeneratorCLI.py     -o     ~/document_output     --styleCust
~/document_style_cuts          --word          ~/service_defs/service1.xml
~/service_defs/service2.xml
```

22.3 Automated Testing Framework

The Automated Testing Framework (ATF) can be launched two ways, through the Ant build script or directly through Python. A full explanation and list of command line options can be found in section 5.6 of the Developer's Guide.

23 Appendix B – Available Types and Units

This appendix contains a list of types and units that are available in the CJSIDL grammar.

23.1 Simple Numeric Type

uint8	int8	float
uint16	int16	double
uint32	int32	
uint64	int64	

23.2 Field Formats

AU	MP3	RNG
BMP	MP4	User_defined
JAUS_MESSAGE	MPEG-1	WAV
JPG	MPEG-2	XML
MJPEG	RAW	XSD
MP2	RNC	

23.3 Units

ampere	becquerel	coulomb_per_square_meter
ampere_per_meter	bel	cubic_meter
ampere_per_square_meter	candela	cubic_meter_per_kilogram
angstrom	candela_per_square_meter	curie
are	colulomb_per_cubic_meter	day
bar	coulomb	degree
barn	coulomb_per_kilogram	degree_Celsius

farad	liter	radian
fared_per_meter	lumen	radian_per_second
gray_per_second	lux	radian_per_second_squared
hectare	meter	reciprocal_meter
henry	meter_per_second	rem
henry_per_meter	meter_per_second_squared	roentgen
hertz	metric_ton	second
hour	minute	siemens
joule	mole	sievert
joule_per_cubic_meter	mole_per_cubic_meter	square_meter
joule_per_kelvin	nautical_mile	steradian
joule_per_kilogram	neper	tesla
joule_per_mole	newton	volt_per_meter
joule_per_mole_kelvin	newton_meter	volt
katal	newton_per_meter	watt
katal_per_cubic_meter	ohm	watt_per_meter_kelvin
kelvin	one	watt_per_square_meter
kilogram	pascal	watt_per_square_meter_steradian
kilogram_per_cubic_meter	pascal_second	weber
knot	rad	

24 People and Copyrights

The alphabetically ordered list below contains names of developers and organizations that have supported the development of JTS from its inception to release 2.0.

Alex Evans (WINTEC, Inc)

Arfath Pasha (WINTEC, Inc)

Chuck Messmer (Progeny Systems Corp)

Danny Kent (DTI)

Dave Martin (DeVivo AST)

Drew Lucas (University of Florida)

Eric Thorn (University of Florida)

Gina Nearing (Progeny Systems Corp)

Ian Durkan (Progeny Systems Corp)

Jaehoon Lee (WINTEC, Inc)

Jean Francois Kamath (University of Florida)

Jim Albers (Fastpilot, Inc.)

Mark Bofill (DeVivo AST)

Nicholas Johnson (University of Florida)

Parag Batavia (Neya Systems)

Rich Ernst (OSD)

Tom Galluzzo

Support has been provided by the Office Under Secretary of Defense for Acquisition, Technology & Logistics / Unmanned Warfare (OUSD (AT&L)/PSA-UW), Navy Program Executive Officer Littoral and Mine Warfare (PEO LMW), PEO Unmanned Aviation and Strike Weapons (U&W), Office Naval Research (ONR), and Air Force Research Lab (AFRL).

XMLmind XSL-FO Converter Copyright © 2002-2009 Pixware SARL