MODELING THE MESSAGING AND COMPONENT INTERFACES OF AUTONOMOUS SYSTEMS

by

Sean Whitsitt

A Thesis Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE

In the Graduate College
THE UNIVERSITY OF ARIZONA

August 5, 2011
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: ________________________________

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

__________________________________________  Date
Dr. Jonathan Sprinkle
Assistant Professor
# Table of Contents

**List of Figures** .............................................. iv  
**List of Tables** ............................................ vii  
**Abstract** .................................................. 1  

**Chapter 1. Introduction** ................................. 2  
1.1. Potential Impact ........................................ 3  
1.2. Contribution of the Work .............................. 4  

**Chapter 2. Background** ................................. 5  
2.1. Related Work ........................................... 5  
2.2. JAUS Standard .......................................... 7  
2.2.1. OpenJAUS SDK ...................................... 11  
2.2.2. Alternative JAUS Implementations .................. 11  
2.2.3. JAUS versus OpenJAUS .............................. 12  
2.2.4. OpenJAUS Projects ................................. 12  
2.3. Related Topics ......................................... 13  
2.3.1. Model Based Design ................................. 13  
2.3.2. Component-Based Frameworks ....................... 15  

**Chapter 3. Methods** ..................................... 21  
3.1. Metamodelling .......................................... 21  
3.1.1. Messages .......................................... 25  
3.1.2. Components ........................................ 32  
3.2. Interpretation ......................................... 35  
3.2.1. Message Interpreter ............................... 41  
3.2.2. Component Interpreter ............................. 44  

**Chapter 4. Results** .................................... 46  
4.1. Artifacts ............................................. 46  
4.1.1. Messages ........................................... 46  
4.1.2. Skeleton Code ...................................... 48  
4.1.3. Compression ........................................ 49  
4.1.4. Components ......................................... 51  
4.2. In-depth Example ...................................... 51  
4.2.1. Test Environment .................................. 53  
4.2.2. Example Models .................................... 54
# Table of Contents—Continued

4.3. Large Test Case: Modeling OpenJAUS ........................................... 57

Chapter 5. Future Work ................................................................. 64
  5.1. Modeling Environment Enhancements ................................. 64
  5.2. Code Generator Enhancements ........................................... 66

Chapter 6. Acknowledgements ...................................................... 68

References ....................................................................................... 69

Appendix A. Example JAUSMessage and Struct code. ................. 75
  A.1. PredictionMessage model .................................................. 75
  A.2. PredictionMessage code ..................................................... 76
  A.3. State code ....................................................................... 85
LIST OF FIGURES

Figure 2.1. The process of metamodeling with MIC[1].................................15
Figure 2.2. An example component diagram. This diagram shows a very  simple structure for the navigation of an autonomous ground vehicle [2].  This diagram was drawn to describe a vehicle developed in the Spring 2011 Semester ENGR550 class at the University of Arizona.................16
Figure 3.1. Metamodel for the JAUS Message paradigm.........................22
Figure 3.2. Metamodel for the JAUS Component paradigm.....................25
Figure 3.3. Example of a model being constructed in the Messaging paradigm.  The red text and dashed arrows show items of note in the example...29
Figure 3.4. Example of a model being constructed in the Component paradigm.  The red text and dashed arrows show items of note in the example...29
Figure 3.5. Pseudocode for the main interpreter that then runs the Message  and Component interpreters.........................................................36
Figure 3.6. Pseudocode for the Message interpreter. This shows how the  Message interpreter visits each JausMessage and generates four types of  artifact.................................................................36
Figure 3.7. Pseudocode for JausMessage visitor function. Note that the  JausType visitor functions are only given as a general concept of code generation inside of the foreach loop.........................................................37
Figure 3.8. Pseudocode for Struct visitor function. Note that the JausType  visitor functions are only given as a general concept of code generation inside of the foreach loop.........................................................37
Figure 3.9. Pseudocode for the Component interpreter. This shows how the  Component interpreter visits each MessagingModel..........................38
Figure 3.10. Pseudocode for the MessagingModel visitor function. This shows  how the MessagingModel visitor function visits each Component............38
Figure 3.11. Pseudocode for Component visitor function. This shows how the  Components are generated and how the Receivers and Senders contribute  to the generated Component.........................................................39
Figure 4.1. Output from the automatically generated JAUS Message tests  for the iPad/Car example used throughout this thesis..........................50
Figure 4.2. Skeleton code for the common message header. Illustrating the  code markers where generated code is inserted into the skeleton file.......51
Figure 4.3. Filled code for the common message header for the drive-by-  iPad case study. Note that all markers between hash marks have been  eliminated from figure 4.2..........................................................52
Figure 4.4. The case study is a ByWire XGV...........................................54
Figure 4.5. The UML Sequence Diagram for the drive-by-iPad example. Note that the two iPad and two carNavigation sequences are running concurrently in each component. That is, while the iPad is idle it is processing VelodyneDataMessages from the carNavigation component, and while the carNavigation component is idle it is gathering sensor data to send to the iPad.

Figure 4.6. A PredictionMessage is a message that contains the predicted path that the car will take. This message is sent to the user so he or she may confirm the calculated path. Note that the PredictionMessage contains a JausArray which is an array of State Structs. States are the expected state the vehicle will be in at a given point along the predicted path.

Figure 4.7. A VerifyPathMessage is a message that only contains a JausTime object that ought to match the JausTime that was sent with a PredictionMessage. This is used as the confirmation message from the user. This JausTime object is the same object that gets created when the user first picks a waypoint to send the vehicle to.

Figure 4.8. A WayPointRequestMessage is a message that contains a WayPoint and a time stamp. This is a user sent message that requests a path from the car’s computer. Note that this message contains only a WayPoint Struct which contains information on the WayPoint to which the user wishes to move the vehicle.

Figure 4.9. A VelodyneDataMessage consists of an array of VelodyneData points. This makes up the view around the vehicle according to the Velodyne LIDAR unit. Note that this message contains a JausArray that is an array of VelodyneData Structs. These structs contain a single point of 3d data from the Velodyne sensor.

Figure 4.10. The azCarModel MessagingModel shows the visual description of the drive-by-iPad example JAUS Components. Note that this model contains two Components, each Component contains several Ports, and each Port contains a reference to the JausMessage that it will send or receive. Red triangles that point to the right are Senders and blue triangles that point to the left are Receivers.

Figure 4.11. This is partial output from the tests of the large scale test case. There are a total of 158 messages that are tested in the large scale test. All of the tests report “SUCCESS.”

Figure 5.1. Possible metamodel for the JAUS Component paradigm of the JAUS modeling project’s next iteration. Compare this Figure with Figure 3.2.
List of Figures—Continued

Figure A.1. This is the PredictionMessage model that the code in Sections A.2 and A.3 was generated from. 75
LIST OF TABLES

Table 2.1. The JAUS Standard is actually a set of standards. These are just a few examples of the standards that make up JAUS.  

Table 2.2. This is the byte layout for a JAUS Message header. The data specific to that message would follow directly after this for as many bytes as are specified in the Data Size field. Note that this header is 128 bits long, and the Data Size field is 12 bits long meaning a JAUS Message may contain up to 4096 bytes of data before needing to be split into multiple messages.  

Table 3.1. These are the text formats that will be used to indicate what type of object is being referenced in the text. For example, a JAUS Message refers to the JAUS concept of a JAUS Message, while a JausMessage refers to the JausMessage type in the JAUS Messaging paradigm, and a PredictionMessage refers to a specific message created with the JAUS Modeling Language that this thesis describes.  

Table 3.2. Notes on the types in the JAUS Message Paradigm. These are the types that can be added to JausMessages or Structs.  

Table 3.3. Notes on the types in the JAUS Message Paradigm. These are the types that cannot be added to JausMessages or Structs. Except for the JausMessage and Struct types, these are abstract types that other objects extend.  

Table 3.4. Constraints in the JAUS Message Paradigm. These requirements must be met by a user creating JAUS Messages with the JAUS Messaging metamodel (Figure 3.1).  

Table 3.5. Constraints in the Component Paradigm. These requirements must be met by a user creating JAUS Components with the JAUS Component metamodel (Figure 3.2).  

Table 4.1. The general artifacts generated by both of the interpreters. More than one of each of these artifacts can be generated as noted by the Multiples column. Note that one Project Messages directory and one Project Tests directory is made per JausMessageTypeTemplate while one Project Component directory is made per Component.  

Table 4.2. The number of artifacts, lines of code, and size in bytes of each example project. The generated code and generated tests have been split into two categories for each project to show the scope of the tests. Note that this table does not include tests for the hand written example projects.
The Joint Architecture for Unmanned Systems (JAUS) is a standard originally developed by the United States Department of Defense as an open architecture for unmanned systems. The JAUS standard was designed to describe sensing, control, and computational communications of components within unmanned systems. This thesis presents a modeling environment capable of generating a JAUS-compatible prototype of the software necessary for inter-computer communications. A metamodel is used to specify the domain-specific modeling language to model the messages used in JAUS, the interfaces between components, and some of the functionality of the components that transmit and receive messages. Thus it is shown that by creating a modeling language capable of representing and generating working code for the messages and components in an autonomous system, the infrastructure and messages for sensing, communication, and control for specific autonomous vehicles can be generated. The case study and test environment for the software generated by this project is an autonomous ground vehicle, modeled on a Ford Escape Hybrid that is used in laboratory experiments. Future contributions to this work are described as additions to the functionality in generated components, allowing for more flexibility and complexity of the generated artifacts.
Chapter 1

INTRODUCTION

The complexity of the design, implementation, and testing of autonomous vehicle systems requires the integration of software components that manage sensing, control, actuation, and logging. Due to the variety of tasks that must be addressed and the difficulty of debugging a multithreaded design [3], a single executable file that “runs” the entire system is impractical. Alternatively, a component-based design permits a functional decomposition of the system into atomic tasks, and allows those tasks to communicate via message passing—abstracting even whether a set of processes are running on the same machine or over the network.

Many different technologies address this particular issue of middleware. One architecture put forth by the US Department of Defense is the Joint Architecture for Unmanned Systems (now a standard of the Society of Automotive Engineers). JAUS (Joint Architecture for Unmanned Systems) is a set of standards that govern the construction of and interactions between computers and components in these systems. Systems built with JAUS are meant to be composed of individual nodes that have no need to understand the deployment details of inter and intra system communication. This is accomplished by having a generic format for transmitted messages and the data contained in those messages. These message subclasses are highly formulaic and creating them by hand or through cloning is tedious and prone to human error. An approach that automatically generates JAUS messages after a user creates a model of the desired message reduces the time necessary to create new messages as well as permitting rapid rollout of updates to message implementations if alterations are needed.

Also, it is highly likely that any software built for an autonomous vehicle will be
running on embedded hardware. Using embedded hardware means that new versions of the messaging software may need to be deployed if the hardware specifications for the deployment system change. Using metamodeling to design the message and component structure means that alterations need to be made only to the code generator to bring the software generated from the models within compliance of the new hardware specifications. Editing a code generator is not necessarily a trivial task, but it is much simpler to edit in one place and regenerate the deployed code than it is to edit in many scattered files for an entire JAUS system.

The components in a JAUS system that transmit and receive JAUS Messages are more complex than the messages, but contain similarly formulaic parts and often benefit from integration with high-level simulation and data capture frameworks such as MATLAB/Simulink or LabView. As such, there are two main goals in building a JAUS modeling system:

1. Construct a modeling paradigm for JAUS Messages.
2. Construct a modeling paradigm for the specification and integration of JAUS Components.

This thesis seeks to achieve both of these goals. However, it should be noted that the JAUS Components produced in this thesis are functional skeletons that can be executed, but contain no higher level functionality. In the future the system will be capable of generating all of the software for an unmanned system, and include integrative simulation and testing functionality, all from a model-based specification.

1.1 Potential Impact

In the National Defense Authorization Act for Fiscal Year 2001 [4] (S. 2549, Sec. 217) the United States Congress mandated that by 2015 one-third of all ground combat vehicles used in the Armed Forces should be unmanned. In response to this mandate,
the Department of Defense created the original JAUS standard that was later handed off to the Society for Automotive Engineers. Thus, it is expected that the majority (if not all) of the unmanned vehicles that will be used in the United States military will implement the JAUS standard. Because of the considerable size of the U.S. military’s fleet of vehicles, there is a large incentive to improve upon the development process of JAUS systems.

1.2 Contribution of the Work

The contribution of this thesis is the simplification of the software development process for messages for JAUS-based autonomous systems, achieved by raising the level of abstraction for the process. Specifically, this thesis simplifies the message and component design process, required for autonomous systems development. Many current systems are developed at the code level and require many hours of typing out, testing, and validating code. Delegating these responsibilities to an automatic code generator reduces the time necessary for producing, testing, and validating software. This also allows developers to discover flaws in the design more quickly during testing, fix those flaws, and produce a new version of the software. Similarly, a developer can also generate future versions of their software from these models simply by updating the models and regenerating the software. This means that by creating a modeling language capable of representing and generating working code for the structure of messages and components in an autonomous system, the infrastructure and messages for sensing, communication and control for specific autonomous vehicles can be generated.
Chapter 2

Background

There are a few background concepts with which the reader should be familiar before delving into the main purpose of this thesis. Background will be given on some related research; JAUS, OpenJAUS, and some projects using OpenJAUS; and Related topics like Model Based Development and Component Based Systems.

2.1 Related Work

There are several projects that relate to the research described in this thesis. One such research project is for synthesizing executable simulations from structural models of component-based systems [5]. This research attempts to address the issue that many robotics experts face in the integration of simulation tools with robotics software. The modeling language developed by Schuster and Sprinkle [5] can be used in the design and construction of robotics systems and in synthesizing experiments from the models of these systems. That paper discussed a running autonomous ground vehicle, similar to the one used in the case study for this thesis. Their example demonstrates the use of a modeling language on running simulations with the hundreds of configuration parameters for existing components and middleware message structure.

Another related research project is the Component Synthesis with Model Integrated Computing (CoSMIC) modeling environment [6, 7], which addresses challenges in the domain of Distributed Real-time and Embedded (DRE) application development. CoSMIC is a collection of Domain Specific Modeling Languages (DSMLs) and interpreters that use the principles of MIC to provide Quality of Service (QoS)-enabled component middleware to aid in the software deployment and configuration process. CoSMIC handles the issue of product deployment in a component oriented
way, similar to the component oriented design behind JAUS.

The Microsoft Robotics Developer Studio (MS Robotics) [8] is a development platform similar to the modeling language described in this thesis. MS Robotics has a complete development environment for creating robotics applications, including a Visual Programming Language (i.e. a modeling language) that can generate C# code. It also contains a built in simulation environment for testing robotics applications designed with it. However, MS Robotics cannot be used to create JAUS driven platforms.

The Orca Robotics Project [9] is an open source framework for Component-Based robotics projects. Orca itself is similar to JAUS (see Section 2.3.2 for more details), and its main goal is to address the growing complexity of robotics research by promoting software reuse. As such, they have created a community repository where developers are encouraged to use the preexisting software components in the repository and are encouraged to add their own newly developed components.

Some projects that use Orca are as follows. In [10] an autonomous vehicle is implemented using the Orca infrastructure (with ICE (see Section 2.3.2) as the middleware). In [11] Orca is used as the framework to build a software architecture for a human-robot team. In [12] Orca is used in a robot designed to navigate using the Partially Observable Markov Devision Process. In [13] a robot is constructed to execute a task by combining its perception of an environment with a human operator.

GearBox is a collection of software tools that can be integrated into different systems. GearBox has been designed not as a framework itself, but to instead be incorporated into other frameworks and projects [14]. Similar to the Orca Robotics Project, one of the main goals of GearBox is software reusability. In fact, GearBox is listed as a dependency for the Orca software installation on the Orca website [15].

Some projects that use GearBox are as follows. Gearbox is a dependency of the Orca Robotics Project [16]. In the Robot Operating System [17], GearBox is one of many software packages available for use [18].
Open Control Platform (OCP) is designed to provide a layer of abstraction between software and the system on which they operate [19, 20, 21]. This middleware abstraction layer is meant as a way to simplify development of high-level algorithms by providing a single platform for development. OCP can then be implemented on many different systems (embedded systems, Linux, Windows, Unix, etc.) instead of having to retool the higher level logic for the different hardware dependencies of those systems. One of the most notable projects that OCP has been used with is the DARPA Software Enabled Control [22, 23] (SEC) project. SEC’s goal was to provide real-time control systems for, increase the automation used in, improve the robustness, and provide reusable software for unmanned and manned vehicles. SEC used OCP to accomplish that last goal and to provide a testing platform.

Two other projects that use OCP are as follows. In [24] the OCP is the development platform for an API developed for a receding horizon controller for an autonomous UAV. In [25] a Non-linear Model Predictive Controller is designed for evasive maneuvers in a UAV with the help of the OCP.

2.2 JAUS Standard

This thesis is using the SAE AS-4 (Version 3.3) JAUS Standard [26]. JAUS is composed of several different standards (See Table 2.1) that the SAE publishes separately. One such standard is the JAUS Transport Specification (AS5669A) which defines specifications for data transmission over UDP, TCP, and Serial connections. Another is the JAUS Service Interface Definition Language (AS5684) which defines data structures as an XML schema for services, messages, and protocols. Also, the JAUS Core Service Set (AS5710A) which defines low level services such as those at the transport layer of network transmission. Another is the JAUS Mobility Service Set (AS6009) which defines common services like GPS and vehicle control. Another is the JAUS Human Machine Interface Service Set (AS6040) which defines services
Table 2.1: The JAUS Standard is actually a set of standards. These are just a few examples of the standards that make up JAUS.

for standard human computer interfaces like pointing devices, keyboards, and other controls. Also, there is the JAUS Manipulation Service Set (AS6057) which defines services meant to control robotic manipulators. All of these standards seek to eliminate ambiguity in the domain of unmanned and autonomous vehicles. Many of these standards also build upon one another. For instance, the JAUS Mobility Service Set defines a standard message structure for positioning data that then gets encoded into the byte format defined in the JAUS Transport specification for transmission over a network.

JAUS Messages Overview

From a modeling perspective the most interesting aspect of a JAUS message is the structure of the data contained in the message. The rest of the message body is header information that determines where the message will be sent and where the message originated. For these messages to be robust there are three key concepts that must be addressed. First, a message must be able to contain any standard primitive type as well as collections of primitive types (i.e. arrays and other common data structures). Second, there must be a way to dictate the order in which data items are inserted and removed from the generic data buffer. Lastly, messages must be capable of compressing larger data types (e.g. floating point numbers) into smaller data types (e.g. integers and bytes) in order to reduce message size for sensor data. This thesis addresses these issues.

In general, the JAUS Message data structure contains information pertaining to
Table 2.2: This is the byte layout for a JAUS Message header. The data specific to that message would follow directly after this for as many bytes as are specified in the Data Size field. Note that this header is 128 bits long, and the Data Size field is 12 bits long meaning a JAUS Message may contain up to 4096 bytes of data before needing to be split into multiple messages.

The code, whether hand-written or generated, for these messages includes all of the functionality to include the data types in the user generated models as well as the specifics for how the data should be packaged. OpenJAUS handles all of the low-level details for placing byte data into a buffer that will be transmitted to another JAUS Component.
JAUS Components Overview  All JAUS Components are state based. JAUS Components must have a ready state that allows it to accept data, a standby state where the component is processing data (standby in this case means that the Component is not ready for new data, not that the Component is standing by waiting for data), and an initialize state that sets the component up and performs any necessary operational checks. All three of these states must be capable of checking the status of the component and performing actions to either take the component offline in event of a failure, or moving the component toward its ready state so that it can accept new messages.

A JAUS Component is a class that has the ability to create, transmit, and/or receive JAUS Messages. The messages that they receive can originate either from another Component or user generated input (e.g. instructing the vehicle to move to a certain point). These classes are also capable of processing the data they receive from messages as well as data they receive from any external sensors. For instance, in a vehicle a GPS unit might be constantly updating with new coordinates. These coordinates would then be observed and processed by a GPS JAUS Component into localized X, Y, and Z coordinates and forwarded via a JAUS Message to another JAUS Component. This next Component could then make decisions about how the vehicle should move to accomplish a task, package those decisions as commands into a JAUS Message and send them to yet another Component. This final Component would then be able to execute the commands it has been given to move the vehicle.

The implementation details for individual Components are specific to those Components’ intended behavior. This thesis considers Components to be skeletons of functional JAUS Components that will be filled in later by other means. What the Component actually does with the received messages is left for a user to fill in later. See Chapters 3 and 5 of this thesis for more information on Components.

JAUS Components have to interact with some basic software components that handle the details of managing a JAUS system. A Node Manager keeps track of
what JAUS Components are running on a system, and it is the first line of contact on a particular computer running JAUS Components for incoming messages. The Node Manager is responsible for making sure JAUS Components that register with it receive the messages that are intended for them. This permits Components to operate in an Object Oriented way without the need for polling on a port.

2.2.1 OpenJAUS SDK

This thesis uses OpenJAUS version 3.3. The OpenJAUS SDK is an open source C and C++ implementation of the JAUS standards that provides all of the necessary source code to create an unmanned system [27, 28]. This SDK provides the abstraction between the artifacts described by this thesis and the underlying mechanisms which the JAUS standards lay out.

2.2.2 Alternative JAUS Implementations

OpenJAUS is a tool for research and development on unmanned and autonomous systems, though it is not the only such tool that implements the JAUS standards. Other open source implementations of the JAUS Standard are The JAUS Tool Set [29], JAUS++ [30], RI-JAUS [31], and a LabVIEW Toolkit for JAUS [32]. All of these tools are directly comparable to OpenJAUS as valid alternative JAUS implementations. The decision to use OpenJAUS for this thesis was arbitrary. Though, it is important to note that the interpreters that will be described for this thesis could have been for any of these JAUS implementations. In fact, a part of the future work for this thesis could be to create new interpreters that can generate software for each of these JAUS implementations.
2.2.3 JAUS versus OpenJAUS

It is important to understand that JAUS and OpenJAUS are related but not the same. JAUS is a standard and OpenJAUS is an implementation. An accurate analogy would be to say that JAUS is to OpenJAUS what ANSI C [33] is to GCC [34]. ANSI C defines how the C programming language ought to behave and how its syntax should be interpreted in much the same way that JAUS defines the rules and regulations that a JAUS System should abide. Basically, OpenJAUS is an implementation of JAUS and there can be many other implementations of JAUS. It should also be noted that OpenJAUS is not under the same management group as JAUS, and neither group is formally affiliated with the other. Lastly, the SAE (which governs JAUS standards) does not provide a reference implementation of JAUS, only a reference document.

2.2.4 OpenJAUS Projects

A few examples of projects making use of OpenJAUS are: AGNAS, the Aviation Ground Navigation System by MountainTop Technologies [35]; teams competing in the JAUS-based portion of the Intelligent Ground Vehicle Competition (IGVC) [36]; Team CIMAR [37, 38] in the DARPA Grand Challenge; and Team Victor Tango [40] in the DARPA Urban Challenge.

AGNAS is an on-going project similar to the one described in this thesis. Similar to this modeling project, the researchers on AGNAS are attempting to develop and deploy technologies to autonomously (or semi-autonomously) complete tasks currently manned tasks at airports (such as towing and docking aircraft). Though, it should be noted that the goals of the AGNAS project are much more direct than the goals of this thesis as their main objective is a specific problem in autonomous systems. On the other hand, this thesis seeks to streamline parts of the development process (message and component design and implementation) for autonomous systems like AGNAS.
The IGVC is an annual competition where teams compete by constructing and running autonomous ground vehicles. A subset of this competition is to construct a vehicle that implements the JAUS protocol. Technically the JAUS Challenge portion of the IGVC only judges the teams’ implementation of the JAUS protocol. This means that a vehicle that ranks highly in the JAUS Challenge need not have placed well in other challenges as based on the rules of the IGVC.

The vehicle produced by Team CIMAR for the 2005 DARPA Grand Challenge (NaviGATOR) [39] was a research platform similar in concept to the modified Hybrid Ford Escape (See Section 4.2.1) used in this thesis. Both are autonomous vehicles designed to be used with the JAUS standard. Both vehicles were being used for academic research. However there are a few key differences. First, NaviGATOR was designed from the ground up while the modified Hybrid Escape is a stock commercial vehicle designed more for an urban environment with hardware added on to interface with the on-board computers and sense the environment. Second, the software for Team CIMAR’s NaviGATOR was in no way built using MDB, the majority of the software for this thesis has been automatically generated from models.

For the 2007 DARPA Urban Challenge Team Victor Tango developed a vehicle using OpenJAUS with a Hybrid Ford Escape [40] (Odin). Because of Team Victor Tango’s vehicle choice Odin is even more similar to the test platform used in this thesis than Team CIMAR’s NaviGATOR. However, like NaviGATOR, Odin was not developed using MDB (see section 2.3 for information about MBD).

2.3 Related Topics

2.3.1 Model Based Design

This JAUS modeling project is being constructed on the principles of Model Based Design (MBD), metamodeling, and Model Integrated Computing (MIC). MBD is the development of software or other artifacts for Computer Based Systems (CBSs)
through the use of models. MIC is a process which uses the concept of metamodeling to construct metamodels that formally describe the language in which domain-specific models can be built [41]. Metamodeling then, is a process by which models that formally describe other models are constructed [42]. Similarly, metamodels are models that formally describe a modeling language.

Aside from designing metamodels, there are two concepts involved in creating a modeling environment: interpretation and constrains. Interpretation is the process of distilling artifacts out of models depending on the meaning that has been associated with those models. Models can have many different interpretations. For instance, a Component Diagram (e.g. Figure 2.2) could be interpreted into software written in C, Java, Ruby, Perl, or any other language that can handle Component Based Systems. Constraints are limitations placed on the modeling language that are meant to enforce whatever rules the developer has designed. Constraints typically do not affect the interpretation of models, only the construction of models during the modeling process. For example, a developer might design a modeling language that connects objects of varying types together and he or she may implement a constraint that prevents certain combinations of model from being connected together.

**GME** This thesis uses the Generic Modeling Environment’s (GME) MetaGME language to design a domain-specific modeling language [42]. This language (JausML) is also set up to run through GME as any language created with MetaGME must. GME is a powerful environment and tool that enables a developer or engineer to rapidly create domain-specific modeling solutions for generating artifacts for CBSs [41]. In any useful GME domain-specific solution there are two main components that need to be developed: the modeling environment or metamodels (which includes the constraints on the metamodels), and the interpreter or interpreters that translate models into final application domain artifacts. However, it is important to note that these models are little more than pretty pictures without some sort of interpretation
behind them. Also, MetaGME uses the Object Constraint Language [43] (OCL) to implement constraints on modeling languages.

2.3.2 Component-Based Frameworks

A general categorization of JAUS classifies it as a component-based framework. This has already been hinted previously in discussion of JAUS Messages and JAUS Components. Component-based systems are systems composed of smaller components that fit within a framework that defines these components and how they interact [44]. Figure 2.2 shows an example diagram of a component-based system. The system in the diagram is an autonomous vehicle that uses GPS coordinates and no other external input to guide itself along a given path. The larger boxes with dotted lines indicate that four of the components are running on one computer while the other three are running on the vehicle’s onboard computer. This example could be implemented in JAUS or a number of other component-based frameworks.

One thing to keep in mind with this thesis is that it is extensible to any of the following (and many more) component-driven architectures. This thesis will demon-
Figure 2.2: An example component diagram. This diagram shows a very simple structure for the navigation of an autonomous ground vehicle [2]. This diagram was drawn to describe a vehicle developed in the Spring 2011 Semester ENGR550 class at the University of Arizona.
strate a modeling language capable of generating software useable in an OpenJAUS project, but the concepts herein are relevant to message-passing interfaces, and not limited to OpenJAUS and JAUS. The decision to use JAUS as the standard for this research is because the hardware available operates on the JAUS standard. More importantly than the OpenJAUS software being generated is the concept of generating a component based software package from a model of that package and drastically reducing development time.

There are many component-based frameworks that are similar to JAUS at the most basic levels such as Microsoft’s Component Object Model (COM), JavaBeans, the Internet Communication Engine (ICE) [45], the Common Object Request Broker Architecture (CORBA) [46], and the Orca robotics project [9]. Also, it is important to note that the component based system described in this thesis is considered middleware. This means that they have defined rules for connecting software components of one or more systems together. The reader should keep in mind that middleware is language agnostic, meaning that its interfaces can be implemented programming languages.

**Microsoft COM** The Microsoft Component Object Model [47] is a standard for creating software components in the Microsoft Windows operating systems that was developed in 1993. COM is used to enable communication between processes, enable communication over networks, and dynamic object creation in the range of software development languages compatible with Microsoft’s operation systems. The use of components and the fact that COM components can communicate between processes and over networks makes COM similar to JAUS. However, COM was designed for the domain of Microsoft’s operating systems while JAUS was developed for the domain of unmanned and autonomous systems.

**JavaBeans** JavaBeans [48] are Components in the Java programming language that can be reused and manipulated by developers. The JavaBeans convention specifies
method naming conventions, class constructor conventions, and class behavior conventions. These components are meant to be designed separately, then assembled together to make more sophisticated programs. If a developer creates classes that conform to these specifications, he or she can then use JavaBean tools to connect them together. JavaBeans are similar to the components in JAUS systems in how they connect to one another, but it should be noted that JavaBeans typically access methods of other JavaBeans instead of sending messages back and forth as JAUS Components do.

**CORBA** CORBA is a standard that enables software distributed on multiple computers written in several different programming languages to work together. It can be used to create a simple plug-and-play interface for method calls between objects residing on the same system or on a remote system. This interface between applications is an interface definition language (IDL) which determines the interface that a local object presents to other applications. A CORBA implementation defines the mappings from a local host application and its implementation language to the IDL and then back again to all other applications and implementation languages. CORBA requires a developer to write the IDL code that provides the interface between his or her objects and the IDL.

Even though CORBA was developed for more generic domains than JAUS, the interactions between the IDL and developer created objects are very similar to the JAUS Messaging interface between JAUS Components. This is because the IDL creates a standardized format for passing information back and forth between objects, which is exactly the concept behind JAUS Messages. However, the format for passing this data is somewhat different since a JAUS system is sending messages back and forth while a CORBA system is making method calls between objects. However, the difference between those two information passing concepts is mostly semantic.

The following are a few examples of CORBA implementations and projects. In
a CORBA-based controls solution is implemented for UAVs. The ADAPTIVE Communication Environment (ACE) implementation provides a framework for concurrent communication software. An example of ACE shows how ACE frameworks helps transition between native operating system APIs and high level abstract middleware. The TAO implementation is an Object Request Broker for ACE. In TAO, TAO is used as a way to improve real-time Quality of Service in a CORBA implementation.

ICE was originally developed by several of the developers that worked on CORBA (including Michi Henning), hence ICE’s similarity to CORBA as a platform-independent framework. However, ICE is not governed by a standards body (whereas CORBA is governed by the Object Management Group (OMG)). ICE was developed to reduce the complexity of developing component-based systems, and (although open-source) is generally managed by a few key developers.

ICE provides method calls between multiple implementation languages and platforms and works under a publish/subscribe system. Similarly to CORBA, ICE is a middleware layer that allows applications and objects to communicate and use each other’s functionality. ICE was developed for a specific domain: internet communications. Thus, ICE is by design less complex and much smaller. However, this also means that it is less able to handle corner cases.

ICE is similar to and different from JAUS in the same sort of ways that CORBA is similar to JAUS, but it is arguable that the lower complexity of ICE makes it a bit more similar to JAUS. Both JAUS and ICE define a component based framework that operates within a specific domain.

Orca Finally, the Orca robotics project is largely analogous to JAUS. JAUS defines a domain of software components and the interface in which they connect, as does Orca. JAUS was developed for a domain of unmanned and autonomous vehicles, which could be considered a subset of the domain of robotics for which Orca was
developed. The largest differences between JAUS and Orca are the existence of a styleguide for callbacks for JAUS, which provides default method names for the receipt of certain messages, and a standard design in JAUS to scale floating-point values to reduce message size.

Another key difference between JAUS and Orca is that JAUS is an architecture standard and not an implemented framework (OpenJAUS is an implementation of JAUS) whereas Orca both defines a standard and the implementation of that standard. Because of this the Orca robotics project can maintain other goals than simply developing a standard or writing software to a standard (e.g. encouraging software reuse in robotics development). This coupled with the open-source nature of Orca also more easily enables it to become a community project with many contributors while JAUS remains a standard maintained by a closed group.
Chapter 3

METHODS

3.1 Metamodeling

First, note that Table 3.1 lists the textual formats that will be used in this chapter to represent different kinds of objects.

There are two distinct metamodels that were produced as a result of this research. The more complicated of the two is the JAUS Message Paradigm. As the name indicates, this metamodel (Figure 3.1), its interpreter (see Section 3.2), and all of its constraints (Table 3.4) make up the modeling language for building JAUS Messages. This modeling language allows any kind of JAUS Message to be created within the limits of the OpenJAUS SDK. The second and less complicated of the two is the JAUS Component Paradigm. This paradigm (Figure 3.2), its constraints (Table 3.5), and its interpreter (again see Section 3.2) compose the modeling language that allows for the creation of JAUS Component skeleton files. These files are the basic set up of a JAUS Component, but lack any functionality that would manipulate data. Also, Figure 3.3 and Figure 3.4 each show an example of a model being constructed in the Message paradigm and Component paradigm respectively.

<table>
<thead>
<tr>
<th>Context</th>
<th>Text format</th>
</tr>
</thead>
<tbody>
<tr>
<td>GME</td>
<td>GME formatted text</td>
</tr>
<tr>
<td>JAUS</td>
<td>JAUS and OpenJAUS formatted text</td>
</tr>
<tr>
<td>JAUS Modeling Language</td>
<td>JAUS MODELING LANGUAGE FORMATTED TEXT</td>
</tr>
</tbody>
</table>

Table 3.1: These are the text formats that will be used to indicate what type of object is being referenced in the text. For example, a JAUS Message refers to the JAUS concept of a JAUS Message, while a JausMessage refers to the JausMessage type in the JAUS Messaging paradigm, and a PredictionMessage refers to a specific message created with the JAUS Modeling Language that this thesis describes.
Figure 3.1: Metamodel for the JAUS Message paradigm.
<table>
<thead>
<tr>
<th>Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>StructReference</td>
<td>May reference existing Structs.</td>
</tr>
<tr>
<td>JausArray</td>
<td>May contain one reference to a JausType other than JausArray.</td>
</tr>
<tr>
<td>JausTime</td>
<td>Is a unique struct</td>
</tr>
<tr>
<td>JausEventLimit</td>
<td>Is a JausStruct</td>
</tr>
<tr>
<td>JausMissionCommand</td>
<td>Is a JausStruct</td>
</tr>
<tr>
<td>JausMissionTask</td>
<td>Is a JausStruct</td>
</tr>
<tr>
<td>JausGeometryPointLLA</td>
<td>Is a JausStruct</td>
</tr>
<tr>
<td>JausGeometryPointXYZ</td>
<td>Is a JausStruct</td>
</tr>
<tr>
<td>JausWorldModelVectorObject</td>
<td>Is a JausStruct</td>
</tr>
<tr>
<td>JausWorldModelFeatureClass</td>
<td>Is a JausStruct</td>
</tr>
<tr>
<td>JausInteger</td>
<td>Is a Primitive</td>
</tr>
<tr>
<td>JausUnsignedInteger</td>
<td>Is a Primitive</td>
</tr>
<tr>
<td>JausShort</td>
<td>Is a Primitive</td>
</tr>
<tr>
<td>JausUnsignedShort</td>
<td>Is a Primitive</td>
</tr>
<tr>
<td>JausLong</td>
<td>Is a Primitive</td>
</tr>
<tr>
<td>JausUnsignedLong</td>
<td>Is a Primitive</td>
</tr>
<tr>
<td>JausFloat</td>
<td>Is a Primitive</td>
</tr>
<tr>
<td>JausDouble</td>
<td>Is a Primitive</td>
</tr>
<tr>
<td>JausBoolean</td>
<td>Is a Primitive</td>
</tr>
</tbody>
</table>

Table 3.2: Notes on the types in the JAUS Message Paradigm. These are the types that can be added to JausMessages or Structs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>JausMessage</td>
<td>A base container for other data objects.</td>
</tr>
<tr>
<td>Struct</td>
<td>A base container for other data objects.</td>
</tr>
<tr>
<td>JausType</td>
<td>The base abstract class that defines data objects for JausMessages and Structs.</td>
</tr>
<tr>
<td>JausPrimitiveType</td>
<td>Primitive variables allowable in OpenJAUS.</td>
</tr>
<tr>
<td>JausStruct</td>
<td>Struct types that are included in OpenJAUS.</td>
</tr>
</tbody>
</table>

Table 3.3: Notes on the types in the JAUS Message Paradigm. These are the types that cannot be added to JausMessages or Structs. Except for the JausMessage and Struct types, these are abstract types that other objects extend.
<table>
<thead>
<tr>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singular Containment</td>
<td>A JausArray may only contain one type of object.</td>
</tr>
<tr>
<td>Forced Containment</td>
<td>A JausArray must contain at least one type of object.</td>
</tr>
<tr>
<td>No Recursive Containment</td>
<td>A JausArray may not contain a JausArray.</td>
</tr>
<tr>
<td>Non-null Message References</td>
<td>A JausMessageReference must refer to an existing JausMessage.</td>
</tr>
<tr>
<td>Non-null Struct References</td>
<td>A StructReference must refer to an existing Struct.</td>
</tr>
<tr>
<td>Forced Values</td>
<td>A Constant must have a value.</td>
</tr>
<tr>
<td>Struct Object Ordering</td>
<td>Objects in a Struct must be connected with an ordering unless it has one object (Precedence connections exempt).</td>
</tr>
<tr>
<td>JausMessage Object Ordering</td>
<td>Objects in a JausMessage must be connected with an ordering unless it has one object (Precedence connections exempt).</td>
</tr>
<tr>
<td>Proper Ordering</td>
<td>Objects that are ordered may only be a source or destination once each.</td>
</tr>
<tr>
<td>Struct Single Strand</td>
<td>A Struct must have two objects with a single connection to them unless it only has one object.</td>
</tr>
<tr>
<td>JausMessage Single Strand</td>
<td>A JausMessage must have two objects with a single connection to them unless it only has one object.</td>
</tr>
<tr>
<td>Different Command Codes</td>
<td>Unique JausMessage objects must have unique command code values.</td>
</tr>
</tbody>
</table>

Table 3.4: Constraints in the JAUS Message Paradigm. These requirements must be met by a user creating JAUS Messages with the JAUS Messaging metamodel (Figure 3.1).
3.1.1 Messages

*JausMessageTemplates* In the Message domain metamodel (figure Figure 3.1), the highest level of the paradigm contains the *JausMessageTemplates* folder which in turn contains all of the *JausMessages*, *Structs*, and *Constants* that a user will create for his or her message models. Any given project may contain many different *JausMessageTemplate* folders. Each of these folders is meant to indicate a different messaging system, independent of the other systems. Note, it is still possible to include references to *JausMessages* and *Structs* between the different messaging systems. This was done because the metamodel cannot make the assumption that any given message system being created with this metamodel is full and complete. This means simply that any given message system may include third party OpenJAUS *Messages* and *Structures*, where the user does not have access to the original source code for those *Messages* and *Structs*. However, it is expected that the user will have access to an API for the third party software (otherwise the metamodel makes the assumption that a user may be reverse engineering third party software and that is outside of the
scope of this thesis). In this case, an additional \texttt{JausMessageTemplate} folder may be created by the user where the API for these third party messages and structs can be implemented and referenced. It is then up to the user to manually edit the Makefiles generated by the interpreter to include the proper directories and files for the third party software.

\textit{Constants} The three items that may be contained within a \texttt{JausMessageTemplate} folder are \texttt{JausMessages}, \texttt{Structs}, and \texttt{Constants}. The simplest of these three items are \texttt{Constants}. \texttt{Constants} are directly analogous to \texttt{#define} values in C and C++ (as will be demonstrated later this is how they are interpreted into code). The values in a \texttt{Constant} can be anything that C will accept as a \texttt{#define} value, and the constraint on \texttt{Constants} in this modeling language reenforces this point. The constraint on \texttt{Constants} (called Forced Values), requires that a \texttt{Constant} have a non-null and non-blank value as its value. Aside from the standard use of \texttt{#define} values where they can be used in data manipulation to avoid repeated hard coded values that are time consuming to redefine, \texttt{Constants} are useful in defining the upper and lower bounds on certain primitive types that get compressed for message transmission.

\textit{JausMessages and Structs} The second two items that can be contained in a \texttt{JausMessageTemplate} folder (\texttt{JausMessages} and \texttt{Structs}) are rather similar to one another. Both can contain the same kinds of objects and references and both have the same constraint. Though, the constraint on \texttt{JausMessages} and \texttt{Structs} is actually on their respective reference classes. Note first that the object that a \texttt{JausMessageReference} or a \texttt{StructReference} refers to will determine the type of the variable that is generated by the interpreter to represent that reference (in C it will become a pointer to a \texttt{JausMessage} or \texttt{Struct} of the given type). The constraint that must be met is: any \texttt{JausMessageReference} or \texttt{StructReference} must refer to an existing \texttt{JausMessage} or existing \texttt{Struct} respectively. This is due to the strongly typed nature of the C language.
This means that a reference that does not have a strict type would cause the software generated from that model to fail to compile.

This compiling issue might be avoidable by using generic JausMessage and Struct structures. Such structures could be composed of function pointers and a generic buffer of a given length. However, this ignores the fact that a JausMessageReference or StructReference would then generate a generic JausMessage or Struct with null function pointers, or at best function pointers that pointed to invalid functions. In other words, there is no way to insure that a generic JausMessage or Struct type would be valid without the constraint forcing their references to point to existing objects. Note that there is a JausMessage type available in OpenJAUS that represents a basic message, but aside from a few predefined OpenJAUS structures this modeling language avoids using this generic type for the reasons given above. The predefined structures that use the generic type have functionality built in to handle the issues given above.

There is one major difference between JausMessages and Structs. This is the difference in how each object is generated, and what it means to be a JausMessage versus a Struct. A JausMessage is the highest layer (see Figure 4.6 for an example of a composed JausMessage) of a message that will be transmitted in a JAUS system, while Structs are simply data containers. Both object types require functions that can package them into a buffer for transmission, but JausMessages include header information that is important to the sub-application layers of TCP/IP transmission (and is both hidden and unnecessary for the user creating the models). Basically, while JausMessages may be contained in other JausMessages, they are the objects that get transmitted whereas Structs are contained in a transmitted JausMessage. This is why JausMessage models have a CommandCode associated with them. This is the unique identifier that a system built with OpenJAUS will use to determine which type of message has been received by a Component. The generation of these objects will be discussed in Section 3.2.
Both JausMessages and Structs can contain any object that is a JausType. However, the abstract JausType itself is not a valid object that can be included in a JausMessage or Struct. Including a generic JausType is disallowed for the same reasons that a generic JausMessageReference or Struct is disallowed. This generic class is extended by many other models (as well as two references and a connection), which both simplifies the Messaging metamodel and makes it easier to update the metamodel to allow JausMessages and Structs to contain more kinds of objects. JausTypes are also required to be connected together by a Precedence connection that enforces an ordering in the generated code. This is so that third party message types can be accurately modeled and generated. Without this Precedence, it is possible that a user may run into a scenario where he or she has generated code from an API of a third party message which expects data to be packaged in one order, while the third party implementation of the message packages data in another order. This will result in a conflict between third party JAUS Components that use the third party implementation of the message, and JAUS Components that use the user’s generated implementation of the message.

Also, note from the metamodel that a Precedence is a JausType. This was done as a visually simple way of including the connection in the JausMessage and Struct models. Precedence connections are exempt from the constraint that forces JausTypes to be connected to other JausTypes with a Precedence. This is because it would be silly and redundant otherwise.

There are four models that directly extend the JausType abstract type: JausPrimitiveType, JausStruct, JausArray, and JausTime. Note that this does not include JausMessageReference, StructReference, or Precedence. These three are absent from the list as they are two references and a connection and not of the model metatype. Only two of the four direct descendants of the JausType FCO are not abstract models that are then further extended. The first of these is JausArray, and the second is JausTime.
Figure 3.3: Example of a model being constructed in the Messaging paradigm. The red text and dashed arrows show items of note in the example.

Figure 3.4: Example of a model being constructed in the Component paradigm. The red text and dashed arrows show items of note in the example.
JausArrays A JausArray is the basic and only array class that OpenJAUS offers. In the modeling context, a JausArray may contain one other non-JausArray JausType. A JausArray is expected to contain an indeterminate number of objects of the type contained in the JausArray model. The generated software tracks the number of objects in the JausArray, but that is hidden to the user while both writing code and creating models. This limitation is due to how OpenJAUS implemented its proprietary array class (JausArray). Specifically the implementation of the jausArrayDestroy(*) function in C/C++ requires a pointer to the destroy(*) function for the object type it contains. However, the jausArrayDestroy(*) function is incompatible with the pointer it requires. This also makes it infeasible to have a JausArray contain more than one type of object. Future iterations of this JAUS modeling project may attempt to address this issue, but it was deemed outside of the scope of this thesis. This is due to the fact that the case study for this research topic would not benefit from storing any of its data as an array of arrays.

JausTimes A JausTime is a special structure defined by OpenJAUS that acts as a time stamp. It is its own class of JausType as opposed to being considered a JausStruct because it contains a few extra functions for entering and extracting date and time data that make it unique from any other JausStruct. However, in the actual modeling environment this is hidden from the user. While constructing models the JausTime struct appears as any other predefined JausStruct. This is because the extra functionality of the JausTime type is unimportant until the user is determining how data is manipulated which is outside of the scope of this thesis.

JausStructs OpenJAUS defines a few structures that may be useful to a user building a JAUS system and these have been included in the Messaging metamodel under the abstract class of JausStruct. There are a total of seven of these predefined structures. All of these structures are immutable in the modeling environment and they define a
common type of data that might need to be transmitted in a system designed for an autonomous vehicle.

- The EventLimit type defines a struct that contains a value that can be any primitive type or an RGB value which is meant to represent when a certain event should end. It also contains a type value that determines which kind of value is in use. However, the particular meaning of the EventLimit value is up to the user.

- The GeometryPointLLA is a representation of position data that uses latitude, longitude and altitude values as double precision floating point primitives.

- The GeometryPointXYZ represents the same kind of location data, but instead uses local X, Y, and Z coordinates.

- The MissionCommand structure packages a JausMessage that represents the command, a type value for that JausMessage so it can be packaged and unpackaged, and a flag that indicates whether or not the command should be blocked on.

- The MissionTask structure is composed of a list of MissionCommands and a list of other MissionTasks, allowing it to represent a complex task ordering so that commands can be compiled, sent, and executed in a specific order. The order in which the tree structure of a MissionTask is executed is up to the user implementing the solution.

- The WorldModelFeatureClass contains the same data primitives as the EventLimit structure. It is meant to represent a feature in a user defined environment (e.g. rocks, buildings, other world objects). The interpretation of the data is up to the user.
• The WorldModelVectorObject contains a list of WorldModelFeatureClasses and a list of GeometryPointLLAs. Using these two lists a WorldModelVectorObject can represent the environment in which an autonomous vehicle is operating.

JausPrimitiveTypes OpenJAUS also defines a set of all primitive types that it understands how to package into a buffer for transmission between JAUS Components that have been included in the Messaging metamodel. This list includes ten different primitives: JausUnsignedLong, JausLong, JausUnsignedShort, JausShort, JausUnsignedInteger, JausInteger, JausFloat, JausDouble, JausByte, and JausBoolean. All of these primitive types have a field that they inherit from the JausPrimitiveType abstract model that allows them to be initialized with a default value. When marshaling data into a buffer, OpenJAUS can place each of these values into the buffer in its original state. However, OpenJAUS also includes some functionality to compress some of the larger data types to smaller data types. This can be done in three different ways, and in all three ways the data is packaged between a maximum allowable value and a minimum. The first of the three ways is as a JausUnsignedInteger, the second is as a JausUnsignedShort, and the third is as an arc tangent. In this metamodel the only data type that is using this compression functionality is the JausDouble data type. This was done because it is expected that for long integer types, a user will either need the full range of the long, or will use a smaller data type. All of the other data types have been deemed to be small enough on their own to require compression.

3.1.2 Components

MessagingModelTemplates and Messaging Models The Component metamodel constructed for this thesis represents the minimum components necessary to construct skeleton JAUS Components. At the top level of the paradigm there is the MessagingModelTemplates folder object. MessagingModelTemplates folders are where the templates for each individual system are created. These templates are called Messaging-
<table>
<thead>
<tr>
<th>Constraints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ports Connected</td>
<td>A Port must be connected to another Port object.</td>
</tr>
<tr>
<td>Forced Type</td>
<td>A Port must have at least one JausMessage type.</td>
</tr>
<tr>
<td>Singular Type</td>
<td>A Port may only have one JausMessage type.</td>
</tr>
<tr>
<td>Same Types</td>
<td>Connected Ports must have the same JausMessage type.</td>
</tr>
<tr>
<td>Connection Direction</td>
<td>Connected Ports may only be connected from Sender to Receiver.</td>
</tr>
</tbody>
</table>

Table 3.5: Constraints in the Component Paradigm. These requirements must be met by a user creating JAUS Components with the JAUS Component metamodel (Figure 3.2).

Models in the Component metamodel. Each of those MessagingModels represents an individual JAUS system. Any given user created project can contain many different MessagingModels. This allows users to design multiple JAUS systems for each set of JausMessages that they create. MessagingModels are entirely unique and independent from one another, meaning that no connections can be made between MessagingModels and Components cannot be shared between MessagingModels.

Components and Ports Inside of the MessagingModel, individual Components are created and connected. Each Component created inside of a MessagingModel may contain any number of Ports. These Ports are how the Components can be connected to other Components. Components themselves cannot actually be connected together, it is the Ports which get connected to each other instead. This allows the connection points between Components to be visible to users. Components may only be connected to other Components in the same MessagingModel. All of the constraints in the Component paradigm define and limit how Ports can connect together (see Table 3.5).

1. Any existing Port must be connected to another Port. This is meant to prevent Components from being created with unused functionality. It is assumed that if
a user creates a Port that he or she must want that Port to be able to forward a message to another Component.

2. Each Port object must have at least one JausMessage type associated with it. The metamodel handles this association by having the Port object contain a JausMessage. The reader may be curious as to why this method was chosen instead of simply including a field that the user could fill out for the JausMessage type to associate with the Port. This was done so that JausMessages that the user has made in a JausMessageTemplates folder can be copied into a Port. This minimizes the chance for human error in the process of creating Ports and permits scoping (i.e. two different templates may have the same name). It should also be noted that the name of the JausMessage object that is added to the Port is the only portion of the object that is used to define the JausMessage type associated with the Port. This means that the type can be changed without deleting the current object and adding a new object. Lastly, it should be noted that a Port is not limited to the types of JausMessage that the user has already created. As long as the Port contains a JausMessage with a valid name, the metamodel will accept it. This allows the user to create Ports that use third party messages without having to create the message from an API.

3. As an addendum to the third constraint, a Port may only have one JausMessage type associated with it. This is meant to prevent confusion between messages and to force the user to create one Port object per message he or she needs to transmit or receive. This allows the visual representation of the model to be more comprehensive.

4. Connected Ports must have the same JausMessage type. While the generated Components are not created with the functionality to actually send or receive messages, this constraint prevents the user from creating designs that will not
function. When a JausMessage is received it is unpacked using the functions provided by the JausMessage class associated with the Port it was received on. These functions will fail, or provide invalid data if the proper JausMessage is not used.

5. The last constraint requires an understanding of the two Port subclasses: Sender and Receiver. The names of these two objects describe them perfectly. Sender is a Port object that sends messages, while Receiver is a Port object that receives messages. These two types of model allow the user to design a component diagram that shows exactly what Components will send JausMessages, and what Components will receive JausMessages. The constraint on these two Port types is that a JausMessagePath connection must be made from a Sender to a Receiver.

3.2 Interpretation

From the user’s perspective in this project there is just one interpreter that runs and interprets all of the models that exist in his or her project. This thesis combines a Messaging interpreter and a Component interpreter for the separate Messaging and Component paradigms.

The OpenJAUS SDK is larger than what is typically expected as output from either interpreter for a simple message model. Also, since both modeling paradigms have been designed to use the OpenJAUS SDK, both of them have to include the OpenJAUS library files as an artifact of their interpretation. This ensures that any time a user interprets his or her models, the necessary OpenJAUS libraries are included in the source code that is generated. This is another reason that both interpreters are run at the same time (with no option for the user to run them individually). It prevents the OpenJAUS library from having to be copied more than once.
Interpreter(project) {
    Project Name = Generate Project Name
    Copy OpenJAUS libraries to the project folder
    foreach (folder in project) {
        if (folder is messageFolder) {
            Message_Interpreter(folder)
        } else {
            Component_Interpreter(folder)
        }
    }
}

Figure 3.5: Pseudocode for the main interpreter that then runs the Message and Component interpreters.

Message_Interpreter(JausMessageTemplate) {
    // Read in the skeleton files
    CMake = Create CMake File
    Common = Create Common Header
    UnitTest Source = Create UnitTest Source
    UnitTest Header = Create UnitTest Header
    Folder Name = Create Folder Name
    Test Folder Name = Create Test Folder Name
    foreach(JausMessage in JausMessageTemplate) {
        Command Codes += Generate CommandCodes
        Includes += Generate Includes
        CMake Includes += Generate CMake Include Directories
        Test Functions += Generate Tests
        Visit_Message(JausMessage)
    }
    foreach(Struct in JausMessageTemplate) {
        Includes += Generate Includes
        CMake Includes += Generate CMake Include Directories
        Visit_Struct(Struct)
    }
    Insert generated code into their respective skeletons.
    Write skeletons to files.
}

Figure 3.6: Pseudocode for the Message interpreter. This shows how the Message interpreter visits each JausMessage and generates four types of artifact.
Visit_Message(JausMessage) {
    // read in the skeleton files
    Source = Create JausMessage Source
    Header = Create JausMessage Header
    Test Source = Create JausMessage Test Source
    Test Header = Create JausMessage Test Header

    Object Name = Create Object Name
    Function Name = Create Function Name

    Order JausTypes
    foreach(JausType in JausMessage) {
        Functions += Generate Code for Functions
        Tests += Generate Code for Tests
    }

    Insert generated code into their respective skeletons.
    Insert Object and Function Names into skeletons.
    Write Main, Header, and Class to files.
}

Figure 3.7: Pseudocode for JausMessage visitor function. Note that the JausType visitor functions are only given as a general concept of code generation inside of the foreach loop.

Visit_Struct(Struct) {
    // read in the skeleton files
    Source = Create Struct Source
    Header = Create Struct Header

    Object Name = Create Object Name
    Function Name = Create Function Name

    Order JausTypes
    foreach(JausType in Struct) {
        Functions += Generate Code for Functions
    }

    Insert generated code into their respective skeletons.
    Insert Object and Function Names into skeletons.
    Write Main, Header, and Class to files.
}

Figure 3.8: Pseudocode for Struct visitor function. Note that the JausType visitor functions are only given as a general concept of code generation inside of the foreach loop.
ComponentInterpreter(MessagingModelTemplate) {
    foreach(MessagingModel in MessagingModelTemplate) {
        Visit_MessagingModel(MessagingModel)
    }
}

Figure 3.9: Pseudocode for the Component interpreter. This shows how the Component interpreter visits each MessagingModel.

Visit_MessagingModel(MessagingModel) {
    foreach(Component in MessagingModel) {
        Visit_Component(Component)
    }
}

Figure 3.10: Pseudocode for the MessagingModel visitor function. This shows how the MessagingModel visitor function visits each Component.

Figures 3.5 to 3.11 show the pseudocode of the interpreters. This pseudocode is meant to give the reader a general idea of how the interpreters go about running through all of the objects in a given project. The following several paragraphs expand on the general concepts found in the pseudocode.

While the Messaging and Component interpreters have been merged into one interpreter, there is still some separation between them hidden behind the scenes. The High-Level interpreter that the user uses to interpret a project simply takes the JausMessageTemplate folders and the MessagingModel folders, separates them and sends them off to the Messaging interpreter and Component interpreter respectively.

Both the Message interpreter and the Component interpreter have been designed to use a combination of a Visitor design pattern and a Template design pattern [54]. The details of each artifact will be discussed in Section 4.1, but in general each artifact type has a skeleton artifact associated with it that gets filled out by one of the two interpreters. Each skeleton artifact has at least one, and typically many, sections of code that get generated by the interpreters inserted into key points in the artifact. Each of these sections of code is generated by one of the interpreters visiting each
Visit_Component(\text{Component}) \{" 
  // read in the skeleton files 
  \text{Main} = \text{Create Main Executable Source} 
  \text{Header} = \text{Create Header} 
  \text{Class} = \text{Create Class Source} 
  \text{Component\_Name} = \text{Generate Component Name} 
  \text{Object\_Name} = \text{Generate Object Name} 
  \text{Register} += \text{Generate Node Manager Register Code} 

  \text{foreach}(\text{Receiver in Component}) \{ 
    \text{Includes} += \text{Generate Include} 
    \text{Message Register} += \text{Generate Message Register Code} 
    \text{Receive\_Messages} += \text{Generate Receivable Code} 
  \} 

  \text{foreach}(\text{Sender in Component}) \{ 
    \text{Message Register} += \text{Generate Message Register Code} 
    \text{Includes} += \text{Generate Include} 
  \} 

  \text{Insert generated code into their respective skeletons.} 
  \text{Insert Object Name into skeletons.} 
  \text{Write Main, Header, and Class to files.} 
\} 

\text{Figure 3.11: Pseudocode for Component visitor function. This shows how the Components are generated and how the Receivers and Senders contribute to the generated Component.}
model that affects the particular artifact being generated and producing the necessary code for that section.

This fusion of two design patterns was used for several reasons. First, the Visitor design pattern makes sense because each of the high level models in a given project (i.e. \texttt{JausMessages}, \texttt{Components}, and \texttt{Structs}) have at least one artifact that is generated as a direct product of that object. This means that \texttt{JausMessages} and \texttt{Structs} directly translate to their own class, and \texttt{Components} directly translate into their own executable source file. This makes the generation of each of those artifacts simple when the Visitor pattern moves to the individual models in the project that is being interpreted.

Second, the Template pattern was used to reduce the complexity of the interpreters. In all of the artifacts that are generated by the interpreters, there are many common lines of code. For example, the \texttt{JausMessage} classes that get created from \texttt{JausMessage} models there are functions that allocate space for and place data into the message buffer that will be transmitted over a network from one Component to another. These functions are the same no matter what the user has done in creating a specific \texttt{JausMessage}. As such it makes it simpler for the interpreter to ignore these “cookie-cutter” sections of code.

Third, the Template pattern allows for the end artifacts to be altered without changing the modeling paradigms or the interpreters. For instance, if the OpenJAUS SDK is ever updated and the methods used by it to package data into a buffer change, the skeleton artifacts that the interpreter uses to generate end artifacts can be altered to reflect these changes. As a caution to the reader, this will not prevent the interpreters and modeling paradigms from being affected by all of the possible changes to either the JAUS standard or the OpenJAUS SDK, but it will protect them against minor changes (in output style, header inclusions, etc.).

Lastly, with the Visitor pattern, the traversal of the models in a project is decoupled from the code that is generated as a result of the traversal. This allows for the
models to be visited by the interpreters multiple times, each time extracting different information for a different section of code. As these sections are constructed they can then be inserted into the skeleton (i.e. the Template in the Template pattern) at the appropriate location. This also allows the interpreters to visit the models in a different order each time the interpreters iterates through them.

3.2.1 Message Interpreter

The Message interpreter is responsible for generating all of the code for the JAUS Message paradigm. This includes code for \texttt{JausMessage} classes, \texttt{Struct} classes, and the common header information required for these classes. The Message interpreter begins it’s process by visiting each folder in the list of \texttt{JausMessageTemplates} folders given to it by the High-Level interpreter. For the \texttt{Constants} that are a part of a \texttt{JausMessageTemplate} folder, these are interpreted as a part of the visitor function of the \texttt{JausMessageTemplate} and are incorporated into a common header file for the folder. The Message interpreter then separates the \texttt{JausMessages} and \texttt{Structs} into two groups and visits each in turn. For each \texttt{JausMessage} and each \texttt{Struct} there are certain unique items that need to be generated regardless of the other sections of code inside of them. For a \texttt{JausMessage} these include: the type name of the object (e.g. “NewJausMessage”), the beginning portion of each public function name (e.g. the “newJausMessage” portion of \texttt{newJausMessageToJausMessage}), the \texttt{Command Code} that defines the \texttt{JausMessage}, the name of the header and source files being generated (e.g. “NewJausMessage”.c/.h). For a \texttt{Struct} these are just a subset of the \texttt{JausMessage} items which skips over the \texttt{Command Code} since it is unnecessary to the packing and unpacking of a \texttt{Struct} (and it therefore does not exist in the modeling context for a \texttt{Struct}). These are all things that are produced in the visitor function of each \texttt{JausMessage} and \texttt{Struct}. 
JausMessages and Structs The visitor function for each JausMessage and Struct then begins calling some other Template generating functions that then begin to visit each of the child models that are passed to them. The JausMessage and Struct visitor function begins this process by ordering the model’s children according to the Precedence connections connecting the children. For the ordering, first understand that when two objects are connected in GME, the connection has a source and a destination end to it. As such, the interpreter orders the objects so that the first object has only a single connection to it, where it is the source of that connection. The interpreter then adds objects to the sequence by following the connections from object to object until it reaches the last object in the list. If there is only one object then the interpreter skips the ordering process. Subsequent visitor functions then visit the children given to them in order. The sections of code that must be generated for JausMessages and Structs are as follows: data definitions, initialization, data destroy, data from buffer, data to buffer, data size, data to string, additional libraries, initialize tests, and compare tests.

Straight-forward Code The data definitions section of code is the simplest section. It is merely the list of object and primitive definitions for the code that will be stored in the JausMessage or Struct. Similarly the initialization, data destroy, and data to string sections are all relatively simple. Each of these either handles a primitive value in the necessary manner (e.g. setting it to a zero value for initialization) or calls the same function of the object it represents. The additional libraries section of code is also rather simple, but it involves a bit more thought. This section is built over all of the visitor functions for the children of a JausMessage or Struct. These functions report to the main visitor function what libraries they used in generating their code so that the main visitor function can assemble a complete list of necessary libraries (e.g. the math.h library might be necessary if there is any data compression).
**Buffer Code** The most complicated portion of the Message interpreter is the portion that generates the combined data to buffer and data from buffer sections of code. These two sections are generated side by side in a single data buffer visitor function to ensure that objects are removed from the buffer in the same order that they are entered. This is a bit of redundancy on the fact that the data is given to the data buffer visitor function in order. This is because the ordering of JausTypes was retroactively added to the design after the buffer visitor function had been designed to build the to and from code side by side. The visitor function was not redesigned because redesigning it would not make it less complicated.

The data buffer visitor function itself calls visitor functions for each particular object that it visits. These individual visitor functions for each model then generate code for their particular model and return that to the original data buffer visitor function. In the case of StructReferences, JausMessageReferences, and the built in OpenJAUS structs in the list of objects, the visitor functions only return a single line function call to the to and from buffer functions for the objects they reference. In the case of primitives, the appropriate OpenJAUS function calls are made to insert the data into the buffer or to remove it from the buffer. In the case of a JausTime, special OpenJAUS functions are called to insert the time into the buffer as two JausDoubles. For each of these cases the buffer size is also incremented appropriately to reflect the amount of data that was just packaged into it.

**Test Code** The initialize tests and compare tests sections of code that are produced are where the data packaging for each generated JausMessage gets unit tested. These two sections of code are also generated side by side, and for the same reasons that the to and from buffer sections of code are generated together. However, one additional reason for generating the test sections together is that the values that each primitive takes on are randomly generated. It is more complicated to generate these random values and pass them between two independent visitor functions than it is to generate
each value once and use it twice. Also, these sections of code are somewhat simpler than the data buffer section. The initialize tests section simply sets values for the primitive values that are children of the particular \texttt{JausMessage} being tested, and recursively sets primitive values for any \texttt{StructReferences} or \texttt{JausMessageReferences} that are children of the \texttt{JausMessage}. The skeleton artifact for these tests then packages the data for that \texttt{JausMessage} into a buffer and begins unpacking it directly following that. The compare tests section then defines a test case for each primitive value that was set in the initialize tests section. For any data values that were compressed in the process of packaging the data the test case defines a range that the output value must be within to be considered valid. This range can differ depending on the compression method that the user has defined for that object. Finally, note that the OpenJAUS defined structs are not tested in these test cases. It is expected that the developers of OpenJAUS have thoroughly tested these structures and any further testing is unnecessary. These test cases are designed to make sure that the compression methods for the primitive types are producing acceptable values after decompression.

\textit{Makefiles} Lastly, note that for each \texttt{JausMessageTemplate} that gets interpreted, a \texttt{CMake} [55] file is generated. This allows the software to be compiled on any platform that can support the OpenJAUS SDK.

\subsection{Component Interpreter}

The Component interpreter is responsible for generating all of the code for the JAUS Component paradigm. This is limited only to the code necessary for each \texttt{Component}’s executable. In this particular project the code generated for the \texttt{Components} is a minimal skeleton. This means that each \texttt{Component} is generated only with code that allows it to be created, to be destroyed, to sit in its ready state, to run through its initialization state, and to be ready to receive any of the \texttt{JausMessages} defined by
its Receivers. Any other functionality for the generated Components is outside of the scope of this thesis.

As such, the Component interpreter starts at its top level visitor function that visits each Component and generates the simple bits of code that need to be inserted into its skeleton artifacts. For instance, the Component name (e.g. “NewComponent”), the Component identifier (e.g. “NEWCOMPONENT_ID”), and the struct name for the data that the Component needs (e.g. “NewComponentData”). The interpreter then runs through a visitor function for each of the Receiver objects in the Component. These visitor functions generate a stub of code that recognizes a received message of the type included in the Receiver.

At the end of its process, the Component interpreter will have created the source code for a bare bones, but functional Component. The artifacts generated by the interpreter have several stubs inside of them that are marked with a comment that says “TODO: Program specific code goes here.”
Chapter 4

RESULTS

4.1 Artifacts

Up to this point skeleton artifacts have been discussed but not yet properly defined. There are thirteen distinct skeleton artifacts (see Table 4.1) used by the Message and Component interpreters to generate the software for a modeling project. Ten of the skeleton artifacts are used by the Message interpreter while only three are used by the Component interpreter. As previously mentioned, each of these skeleton artifacts contains a lot of code common to each artifact that will be generated from it, as well as some marked off sections of code that will be overwritten by the interpreters with generated code (see Section 4.1.2 for more details).

4.1.1 Messages

One of the files generated by the Message Interpreter is a common header file that contains the command codes for each message and any constants defined for the Messages in the model being interpreted. This file is then used in all other files generated by the Message Interpreter. One of these files is generated for each folder of messages defined within a model built from the Messaging metamodel.

Another file that is generated from the Message Interpreter is a `CMakeLists.txt` file that can be compiled using CMake [55]. The CMake file is created at the directory level above the Message, Test, and Component directories and includes as the Message and Test directories as subdirectories that it utilizes. This allows this project to make use of CMake’s ability to prepare makefiles for any platform to be easily and quickly deployed on any platform with which OpenJAUS is compatible. However, the CMake file does not yet include functionality for compiling the Component classes. This is
due to the fact that the Component classes are only partially filled out skeletons and not the main focus of this research topic.

Message Two types of files generated by the Message Interpreter are the header and source files for the actual messages. These two files define the functions that are integral to any message (i.e. `create`, `destroy`, `toString`, `toBuffer`, `fromBuffer`, etc.) as well as the the data structure that stores all of the message data. Both of these files are generated for each message defined within a model built from the Message Metamodel.

Structs Another two of the types of generated files within the Message paradigm are the header and source files for any user defined data structures. These two files contain all of the relevant code to construct, destruct, package, and un-package one of these data structures, including the data structure definition itself. Both of these files are generated for each user defined structure. It is important to note that user defined data structures that can be generated from the modeling environment are assumed to require no more functionality than storing data. If a user wished to create additional functionality to manipulate the data stored in one of these data structures, it is assumed that he or she will add that into a JAUS Component.

Tests The last four types of files that the Message Interpreter generates are tests for each message. For each folder of messages defined within a model a main unit test header and source file are generated that allow a user to run the individual message tests using CTest [56, 55]. Then, for each message defined within a model a header and source file are generated for the individual message test. These files define the functions necessary to attempt to create a message and set some initial test values, marshall it into a buffer, then de-marshal the data and test the values to ensure the test values were maintained through the process. The information about the success and failure of these tests are then reported to the command line. However, in order
to prevent an information overload to the user running the test cases, the only item reported is success or failure. In the case of success this is most often sufficient, but in the case of failure more information is generally desired. The decision to only report failure (and no further information about the failure) was reached for two reasons. First, it is assumed that any given JAUS system will likely have dozens of messages. In a textual command line interface having potentially dozens of failure notices that each given detailed information was determined to not be simplistic enough to be desirable. Second, it is also expected that the tests will never report failure. The test exist mostly as a means of verifying the functionality of the interpreters that are automatically generating the code. A failure of a JAUS Message test indicates a larger problem that would need to be fixed by the developers of the code generation software instead of the end user.

4.1.2 Skeleton Code

Both Interpreters written for this thesis make use of skeleton artifacts. These skeletons contain code that is common to each object that will be generated from that type (i.e. function names, common constants, helper functions, etc.). These skeletons also contain markers where they expect code to be added by the Interpreter. These markers can be anything, but to avoid issues with blocks that may accidentally be generated by the interpreter or written into the artifact’s other parts a general style for these markers was developed. Each marker is typically a short description of what will be placed there surrounded by hash marks (#). An Interpreter can then read in one of these files, and replace those markers with the generated code. This removes the need for the interpreter to know anything about the generated files other than what is generated. It also allows for some changes to be made to the final artifacts without necessitating a recompile of the Interpreter. Plus, using this method allows the interpreter to generate a single section of code that might be used in many different
places within a skeleton artifact. The interpreter could also generate code that itself contains markers in it that will be replaced later. In fact, the interpreters do make use of this last benefit to insert the function names for the different artifacts. In some of the generated code function calls are made to other functions in the same object. The interpreters handle this by at first inserting code containing the hash marked marker for those function names and afterwards inserting the function name into those marked sections. Figure 4.2 shows an example of what the skeleton code with some markers looks like, while Figure 4.3 shows the filled out version of that code for the drive-by-iPad example. This same sort of copy and paste technique is used in all of the skeleton files to generate artifacts. However, in most cases the generated code is the innards of a function instead of just a few constant definitions.

### 4.1.3 Compression

One of the three concepts that needed to be addressed for JAUS Messages is the ability to compress some of the larger data types into smaller data types in order to reduce the bandwidth necessary for transmitting data. One aspect of OpenJAUS is its ability to compress a JausDouble value into a smaller primitive data type such as a JausUnsignedInteger or a JausUnsignedShort. This functionality is included in the Message Metamodel by an enumerated field that allows a user to select the compression type and two string fields that allow the user to specify a minimum bound and maximum bound on the compressed data. The accuracy of the compressed data depends on the size of the primitive used for the compression and the number of possible data points in the specified range. Testing showed that an unsigned short had error rates up to 3% while unsigned integers had no error for any tested values for a range of 0.0 to 10.0.
The test for waypointsRequestMessage has reported SUCCESS.
The test for verifyPathMessage has reported SUCCESS.
The test for predictionMessage has reported SUCCESS.
The test for velodyneDataMessage has reported SUCCESS.
All tests succeeded!

Figure 4.1: Output from the automatically generated JAUS Message tests for the iPad/Car example used throughout this thesis.

<table>
<thead>
<tr>
<th>Generated Directory</th>
<th>Artifact</th>
<th>Multiples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Messages</td>
<td>Common.h</td>
<td>One per directory.</td>
</tr>
<tr>
<td></td>
<td>Messages.c</td>
<td>One for each JausMessage.</td>
</tr>
<tr>
<td></td>
<td>Messages.h</td>
<td>One for each JausMessage.</td>
</tr>
<tr>
<td></td>
<td>Struct.c</td>
<td>One for each Struct.</td>
</tr>
<tr>
<td></td>
<td>Struct.h</td>
<td>One for each Struct.</td>
</tr>
<tr>
<td>Project Tests</td>
<td>UnitTests.h</td>
<td>One per directory.</td>
</tr>
<tr>
<td></td>
<td>UnitTests.c</td>
<td>One per directory.</td>
</tr>
<tr>
<td></td>
<td>MessageTest.c</td>
<td>One per JausMessage.</td>
</tr>
<tr>
<td></td>
<td>MessageTests.h</td>
<td>One per JausMessage.</td>
</tr>
<tr>
<td>Project Components</td>
<td>Component.cpp</td>
<td>One per directory.</td>
</tr>
<tr>
<td></td>
<td>Component.h</td>
<td>One per directory.</td>
</tr>
<tr>
<td></td>
<td>main.cpp</td>
<td>One per directory.</td>
</tr>
</tbody>
</table>

Table 4.1: The general artifacts generated by both of the interpreters. More than one of each of these artifacts can be generated as noted by the Multiples column. Note that one Project Messages directory and one Project Tests directory is made per JausMessageTemplate while one Project Component directory is made per Component.
4.1.4 Components

For each component defined in a model using the Component paradigm, three files are generated: a “main” file, a class source file, and a class header file. The class header and source files contain all of the information and functions for the Component’s class. At this first stage of development the Component class and header files are mostly just skeletons that are created with "ToDo" stubs declaring where additional code is required to create a valid and functional OpenJAUS Component. Future work will address the issue of automatically generating the functionality of these Components. The main file allows the component to be run from the command line.

4.2 In-depth Example

This thesis will give two examples of projects constructed using the modeling language it describes. The first of these examples will be given in great detail as an example of how the modeling language works, and the artifacts that will be generated. The second, and much larger, example will be given to show the scope of what the modeling language can handle.
```c
#ifndef _azcar_COMMON_H
#define _azcar_COMMON_H

#include <math.h>

// Command Codes
#define JAUS_EXPER_VerifyPathMessage 0xE256
#define JAUS_EXPER_PredictionMessage 0xE257
#define JAUS_EXPER_WayPointRequestMessage 0xE258

// Constant Values
#define ANGLE_MAX JAUS_PI/2
#define DISTANCE_MIN 0
#define ANGLE_MIN -JAUS_PI/2
#define DISTANCE_MAX 10

#endif // _azcar_COMMON_H
```

Figure 4.3: Filled code for the common message header for the drive-by-iPad case study. Note that all markers between hash marks have been eliminated from figure 4.2.
4.2.1 Test Environment

The test case described in this thesis utilizes a ByWire XGV (see Figure 4.4) and simulations of the ByWire XGV as a test environment. As described before, the XGV is a Ford Escape Hybrid modified by TORC Technologies to be a JAUS interoperable drive-by-wire controlled ground vehicle platform [57]. This modified vehicle gives access to various different control inputs, devices, and feedback. The open loop control inputs available are: throttle, steering, brakes, and shifting. The closed loop controls include: speed, acceleration, curvature, and rate of curvature change. The devices that can be controlled in the vehicle aside from the engine are: the vehicle start and disable switch, the horn, and various vehicle lights and signals. The full array of feedback that is available from the vehicle includes: vehicle speed, individual wheel speeds, steering angle, throttle percentage, brake percentage, shifter position, vehicle state, lights and signals status, door status, and any errors with the onboard computers.

For testing purposes, this research also utilizes simulation software that comes with the OpenJAUS SDK, as well as some custom built software to simulate an actual vehicle. This enables the JAUS Components that transmit and receive messages to run on one computer and communicate with another computer that emulates the software interface of the ByWire XGV. From the generated software’s perspective there is no difference between the actual vehicle and the simulation software. The only difference (as far as the software is concerned) is in the obvious physical difference that the simulation software controls a vehicle that is represented by an array of values while the real software controls a vehicle that is represented by plastic, metal, and sensors.
4.2.2 Example Models

This research is used to generate the software for a drive-by-iPad strategy to control a vehicle. A full 360° Velodyne LIDAR sensor [58] has been mounted on the top of the ByWire XGV and supplies a constantly updated three-dimensional view around the vehicle to an onboard computer. These data are then transmitted via JAUS Messages to an iPad through a wireless network where they are displayed. A user is then able to select a position around the vehicle and send that waypoint to the vehicle’s onboard computer that calculates a path to the point and requests user confirmation of the generated path. If all of that is successful and the user confirms the path, the vehicle’s onboard electronics then executes the path. This simple system requires only two JAUS Components and four JAUS Message types.

Figures 4.6 to 4.10 show the whole of the modeling project developed for this example. Note that this test project was developed both by hand and by using the Messaging and Component modeling paradigms. Both versions consist of the same messages, data structures, and components, and both versions operate the exact same way. The only exception to this is the inclusion of the automatically generated test cases for the version created from models.

In this test case there are a total of four JausMessages (two that contain JausArrays), two Structs, and two Components. The two Components in this system are running on two separate computers. The names (iPAD and CARNAVIGATION) also
Figure 4.5: The UML Sequence Diagram for the drive-by-iPad example. Note that the two iPAD and two CARNAVIGATION sequences are running concurrently in each component. That is, while the iPAD is idle it is processing VELODYNEDATA MESSAGES from the CARNAVIGATION component, and while the CARNAVIGATION component is idle it is gathering sensor data to send to the iPAD.

explain which device each Component will be running on (the iPad controller and the vehicle’s computer, respectively). Figure 4.10 shows the iPAD and the CARNAVIGATION Components and how those two Components are interconnected. From Figure 4.10, it is clear that the CarNavigation Component is sending Velodyne data and path prediction data to the iPad while the iPad is sending requests for paths and sends verification messages for the paths it receives. Figure 4.5 shows the UML Sequence Diagram of this example.

Figure 4.8 shows the composition of the message that the iPAD sends to the CARNAVIGATION computer in order to request a path from the vehicle. This message is composed of a single WAYPOINT and a JausTime. The JausTime that gets sent along in this request is what the two Components use to keep track of what particular WAYPOINT, and path they are communicating about. The WAYPOINT itself simply consists of location data in three dimensions (one of each X, Y, and Z for localized
coordinates since the case study does not make use of GPS) and a desired angle of the vehicle once it reaches the WayPoint. The vehicle’s computer then uses this requested WayPoint to generate a series of points that it expects that it will be at while it drives to the WayPoint.

Figure 4.6 shows the composition of the message that the CarNavigation Component sends to the iPad once it has calculated a predicted path from its current position to the previously requested WayPoint. The JausTime that the PredictionMessage packages with its data is the same as the JausTime that was sent with the original WayPointRequestMessage that spawned the whole process of calculating a path. The PredictionMessage also packages a JausArray that contains the list of States that the vehicle’s computer expects the vehicle will path through while executing the path. A State consists of the localized coordinates (again: X, Y and Z values) that the vehicle is expected to be at, and the angle that the vehicle is expected to be at, as well as the angle that the vehicle’s wheels are expected to be at. Note that through this whole process the angles that are being passed back and forth are normalized to the current state of the vehicle, where the current state is always a vehicle angle of zero radians. Upon receiving this message the iPad then plots the expected route onto the screen for the user to see.

Figure 4.7 shows the composition of the message that the iPad sends to the CarNavigation Component once the user has decided to use the predicted path. This message just consists of the JausTime that was originally created when the user first sent a WayPointRequestMessage to the CarNavigation Component. Note that if the user decides not to have the vehicle execute the predicted path, no message is sent to the vehicle. This is why the iPad and vehicle’s computer pass the same JausTime back and forth with this exchange. The CarNavigation Component will know that the user has not accepted a path when it receives a new WayPointRequestMessage instead of a VerifyPathMessage. This ensures that the vehicle does not execute a path that the user has not verified.
Figure 4.9 shows the composition of the message that the CARNAVIGATION computer sends to the iPAD that expresses the view that the Velodyne LIDAR unit is able to sense around the vehicle. This message consists of a JausArray of VELODYNE DATA. These data represent the location of surfaces around the vehicle. The iPAD takes the data in the Vectors JausArray and plots these points on the screen for the user to see. In this case study the vehicle itself ignores this data collected by its LIDAR sensor and instead relies on the user to make sure that it is avoiding obstacles. Note that the vehicle generates a path that consists of it turning to a specific heading, moving along that heading, then turning again to achieve the angle requested in the initial WayPointRequestMessage.

Lastly, Figure 4.10 shows the innards of one of the Ports (this is the same for Receivers and Senders). Note that it contains a single JausMessage which defines what kind of message the Port can handle. In this particular example, the Ports have been named according to the messages that they are responsible for either receiving or sending. However, that was only a stylistic decision and has no bearing on the actual message types that are used in the Ports. Ports can have any name and will operate based upon the JausMessage that each one contains.

4.3 Large Test Case: Modeling OpenJAUS

As another test case for this thesis, all of the JAUS Messages that come with the OpenJAUS libraries were modeled. Note that these are different from the JausStructs discussed in 3. JausStructs were data structures that came with OpenJAUS that were determined to be useful enough to include in the modeling language, while these built in messages are messages that can easily be used by a Component simply by using the name of the message. This was described in 3.1.2 in that a Port is not restricted only to the JausMessages that the user has created. In total there are one-hundred-and-fifty-eight (158) of these messages. This example was done to show that this
Figure 4.6: A PredictionMessage is a message that contains the predicted path that the car will take. This message is sent to the user so he or she may confirm the calculated path. Note that the PredictionMessage contains a JausArray which is an array of State Structs. States are the expected state the vehicle will be in at a given point along the predicted path.
Figure 4.7: A `VerifyPathMessage` is a message that only contains a `JausTime` object that ought to match the `JausTime` that was sent with a `PredictionMessage`. This is used as the confirmation message from the user. This `JausTime` object is the same object that gets created when the user first picks a waypoint to send the vehicle to.

Figure 4.8: A `WayPointRequestMessage` is a message that contains a `WayPoint` and a time stamp. This is a user sent message that requests a path from the car’s computer. Note that this message contains only a `WayPoint Struct` which contains information on the `WayPoint` to which the user wishes to move the vehicle.
Figure 4.9: A *VelodyneDataMessage* consists of an array of *VelodyneData* points. This makes up the view around the vehicle according to the Velodyne LIDAR unit. Note that this message contains a *JausArray* that is an array of *VelodyneData Structs*. These structs contain a single point of 3d data from the Velodyne sensor.
Figure 4.10: The azCARModel MessagingModel shows the visual description of the drive-by-iPad example JAUS Components. Note that this model contains two Components, each Component contains several Ports, and each Port contains a reference to the JausMessage that it will send or receive. Red triangles that point to the right are Senders and blue triangles that point to the left are Receivers.
<table>
<thead>
<tr>
<th>Project</th>
<th>Project Type</th>
<th>NOA</th>
<th>LOC</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenJAUS Messages</td>
<td>Generated</td>
<td>342</td>
<td>113140</td>
<td>3462891</td>
</tr>
<tr>
<td></td>
<td>Tests</td>
<td>318</td>
<td>16568</td>
<td>824577</td>
</tr>
<tr>
<td></td>
<td>Manual</td>
<td>319</td>
<td>119752</td>
<td>4038053</td>
</tr>
<tr>
<td>Drive-by-iPad</td>
<td>Generated</td>
<td>26</td>
<td>5178</td>
<td>148768</td>
</tr>
<tr>
<td></td>
<td>Tests</td>
<td>14</td>
<td>502</td>
<td>13883</td>
</tr>
<tr>
<td></td>
<td>Manual</td>
<td>27</td>
<td>6756</td>
<td>207447</td>
</tr>
</tbody>
</table>

Table 4.2: The number of artifacts, lines of code, and size in bytes of each example project. The generated code and generated tests have been split into two categories for each project to show the scope of the tests. Note that this table does not include tests for the hand written example projects.

modeling language is capable of handling a large data set, and to give a perspective to compare to the drive-by-iPad example (see Section 4.2).

Table 4.2 shows the difference in the number of number of artifacts, lines of code, and data size in bytes between the two examples, their respective manual implementations, and the generated test cases. This comparison is meant to show the possible scope of the projects that can be done with the research from this thesis, and to show how the generated code will be slightly different from code designed by hand. Note that all of these values are rather close. The biggest difference comes from the tests that are generated by the modeling language. Also Figure 4.11 shows partial results from the tests of the generated OpenJAUS messages.

In short, these results show that through creating a modeling language capable of representing and generating working code for the messages and components in an autonomous system, the infrastructure and messages for sensing communication and control for specific autonomous vehicles can be generated.
The test for createEvent has reported SUCCESS.
The test for setDataLinkSelect has reported SUCCESS.
The test for queryImage has reported SUCCESS.
..............
The test for queryManipulatorSpecifications has reported SUCCESS.
The test for pauseMission has reported SUCCESS.
The test for querySelectedCamera has reported SUCCESS.
The test for setSelectedDataLinkState has reported SUCCESS.
All tests succeeded!

Figure 4.11: This is partial output from the tests of the large scale test case. There are a total of 158 messages that are tested in the large scale test. All of the tests report “SUCCESS.”
Chapter 5

FUTURE WORK

The JAUS modeling project described in this thesis is being produced in several different iterations. Each future iteration will incorporate more functionality into the end result until ultimately a complete modeling language for unmanned and autonomous vehicles has been produced. Due to the iterative nature of this process it is roughly the modeling equivalent of either the spiral software development process [59] or the Agile software development process [60]. This method was chosen since it allows for several discrete stages of development with time in between stages for reflection on the design. Also, it can be more readily taken through each development cycle by different developers if necessary.

For clarity’s sake, it should be noted at this point that the final goal of this thesis is not to generate a stand alone product. For instance, incorporating the user interface into the automatic code generation for the drive-by-iPad case study is outside the scope of this thesis and the next planned iteration of this research project. As such, there will generally still be some code-level software development necessary for constructing an unmanned or autonomous system with this JAUS modeling language.

5.1 Modeling Environment Enhancements

Figure 5.1 shows some changes to the design of the metamodel that have already been designed for the next iteration. This metamodel describes several basic concepts, most of which are the same as in the first iteration. First, MessagingModels are composed of Components connected together via Port objects. Second, each Port must either be a Sender or a Receiver of JausMessages. Third, the functionality of Components (which
Figure 5.1: Possible metamodel for the JAUS Component paradigm of the JAUS modeling project’s next iteration. Compare this Figure with Figure 3.2.

is not present in the first iteration Component modeling paradigm) is represented by Commands contained in each Component that have a specific ExecutionOrder. Where the Commands in a Component will be MATLAB and Simulink S-Functions as well as some built in common Component functionality (e.g. conversion of GPS to localized coordinates). Also note that in this context a MessagingModel represents either an entire JAUS System or a portion of a system and its functionality. Meaning that this modeling environment can either generate a full JAUS System on its own, or can generate components that can be integrated with components from other sources (e.g. an existing JAUS System or other third party components.).

Work after the second iteration may involve generating software in other languages than just C and C++. This would also involve either finding software SDKs similar to OpenJAUS in other software languages, or developing an implementation of JAUS in other languages. This would allow a greater degree of freedom for users that wish to use these modeling paradigms. Note that since JAUS can transmit data over UDP
connections, systems could be developed using many different software languages. More likely than that: independent systems could be designed to communicate with one another. This could be useful for systems that are designed to operate in environments where more than one independent and autonomous vehicle is expected to operate (e.g. a roadway with many cars).

Also, future work might entail adding domain specific support for third party sensors. This might be accomplished either by incorporating specific sensors or by constructing interfaces to suspected sensor types. Since it is expected that any autonomous vehicle will have some sort of external input, adding in these sensors or interfaces would further expand the goals of this project by cutting down on the complexity and rigor of the work necessary to develop an autonomous vehicle.

Finally, considering that many autonomous vehicle applications (e.g. an autonomous taxi service) will require some form of human input, it would be advantageous to have a modeling environment that can generate human interface software for common devices. For instance, the user interface to the iPad in the drive-by-iPad example in this thesis could be generated automatically. However, this task itself may be sufficiently complex that a developer may only create a modeling language that can generate an interface that a user interface could then use to interact with one or more of the JAUS Components.

5.2 Code Generator Enhancements

The goal for the second iteration is to be capable of generating JAUS Components that are nearly fully functional. Some of the functionality of processing data and receiving data from external sensors will be left to be filled in later by other means, but the rest of the Components will be generated by the Component Interpreter. To be clear: this means the generated Components will be fully compilable and runnable, but no attempt will be made to offer integration with third party sensor hardware or third
party software other than JAUS Components (e.g. User Interfaces, Non-MATLAB or Simulink function libraries, etc.). Currently generated components are capable of establishing and maintaining connections with other components, detecting when established connections are lost, and maintaining themselves in a their ready states. In the next iteration generated components will be able to deal with received messages and incorporate MATLAB and Simulink functionality via S-functions.
Chapter 6

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under awards CNS-0930919 and CNS-0915010. Thanks to the anonymous reviewers whose comments improved the focus of papers that described earlier versions of the work which this thesis extended. Additional thanks due to Maribel Hudson and Kun Zhang whose feedback on the design and on JAUS were instrumental in the development of the metamodel and code generators.
REFERENCES


[54] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.


Appendix A

Example JAUSMessage and Struct code.

A.1 PredictionMessage model

Figure A.1: This is the PredictionMessage model that the code in Sections A.2 and A.3 was generated from.
A.2 PredictionMessage code

```c
/*
 * predictionMessage.h
 * OpenJaus
 * Created by JausMessageML_Interpreter on 06/07/11.
 * /

#ifndef PredictionMessage_H
#define PredictionMessage_H
#include <jaus.h>
#include "state.h"
#include "azcar_common.h"
#ifdef cplusplus
extern "C"
#endif

typedef struct
{
  // Include all parameters from a JausMessage structure:
  // Header Properties
  struct
  {
    // Properties by bit fields
    #ifdef JAUS_BIG_ENDIAN
    JausUnsignedShort reserved : 2;
    JausUnsignedShort version : 6;
    JausUnsignedShort expFlag : 1;
    JausUnsignedShort scFlag : 1;
    JausUnsignedShort ackNak : 2;
    JausUnsignedShort priority : 4;
    #elif JAUS_LITTLE_ENDIAN
    JausUnsignedShort priority : 4;
    JausUnsignedShort ackNak : 2;
    JausUnsignedShort scFlag : 1;
    JausUnsignedShort expFlag : 1;
    JausUnsignedShort version : 6;
    JausUnsignedShort reserved : 2;
    #else
      #error "Please define system endianess (see jaus.h)"
    #endif
    } properties;
    JausUnsignedShort commandCode;
    JausAddress destination;
    JausAddress source;
    JausUnsignedInteger dataSize;
    JausUnsignedInteger dataFlag;
    JausUnsignedShort sequenceNumber;
  }
  // generated code
  JausByte numprediction;
  JausArray prediction;
  JausTime time;
  // end generated code
}PredictionMessageStruct;

typedef PredictionMessageStruct* PredictionMessage;

// Creates the message of this type.
JAUS_EXPORT PredictionMessage predictionMessageCreate (void);

// Destroys the message of this type.
void predictionMessageDestroy (PredictionMessage);

// Gets the data for this message from a char* buffer.
JAUS_EXPORT JausBoolean predictionMessageFromBuffer (PredictionMessage message, unsigned char* buffer, unsigned int bufferBytes);

// Puts the data for this message into a char* buffer.
JAUS_EXPORT JausBoolean predictionMessageToBuffer (PredictionMessage message, unsigned char* buffer, unsigned int bufferBytes);
```
// Transforms a JausMessage generic type into a message of this type.
JAUS_EXPORT PredictionMessage predictionMessageFromJausMessage ( JausMessage jausMessage );

// Transforms a message of this type into a generic JausMessage.
JAUS_EXPORT JausMessage predictionMessageToJausMessage ( PredictionMessage message );

// Reports the current size of the message data that will be stored in the char* buffer.
JAUS_EXPORT unsigned int predictionMessageSize ( PredictionMessage message );

// Returns a string with the data of this message.
JAUSEXPORT char* predictionMessageToString ( PredictionMessage message );

#ifdef __cplusplus
}
#endif
#endif // PredictionMessage_H

/**
 * predictionMessage.c
 */

static const int commandCode = 0xE256;
static const int maxDataSizeBytes = 0;

static JausBoolean headerFromBuffer ( PredictionMessage message , unsigned char* buffer , unsigned int bufferSizeBytes ) ;
static JausBoolean headerToBuffer ( PredictionMessage message , unsigned char* buffer , unsigned int bufferSizeBytes ) ;
static int headerToString ( PredictionMessage message , char** buf ) ;
static int dataFromBuffer ( PredictionMessage message , unsigned char* buffer , unsigned int bufferSizeBytes ) ;
static int dataToBuffer ( PredictionMessage message , unsigned char* buffer , unsigned int bufferSizeBytes ) ;
static void dataInitialize ( PredictionMessage message ) ;
static void dataDestroy ( PredictionMessage message ) ;
static unsigned int dataSize ( PredictionMessage message ) ;

// USER CONFIGURED FUNCTIONS

// Initializes the message-specific fields
static void dataInitialize ( PredictionMessage message ) {
    // generated code
    message->prediction = jausArrayCreate ( ) ;
    message->time = jausTimeCreate ( ) ;
    // end generated code
}

// Destroys the message-specific fields
static void dataDestroy ( PredictionMessage message ) {
    // generated code
    jausArrayDestroy ( message->prediction , ( void * ) stateDestroy ) ;
    jausTimeDestroy ( message->time ) ;
    // end generated code
}

// Return boolean of success
static JausBoolean dataFromBuffer(PredictionMessage message, unsigned char *buffer, unsigned int bufferSizeBytes)
{
    int index = 0;
    if (bufferSizeBytes >= dataSize(message)) {
        // generated code
        if (!jausByteFromBuffer(&message->numprediction, buffer+index, bufferSizeBytes-index)) {
            printf("Error: removing numprediction from the buffer!!!\n");
            return JAUS_FALSE;
        }
        index += JAUS_BYTE_SIZE_BYTES;
        unsigned int iprediction;
        for (iprediction = 0; iprediction < message->numprediction; iprediction++) {
            State *State_item = stateCreate();
            if (!stateFromBuffer(&State_item, buffer+index, bufferSizeBytes-index)) {
                printf("Error: removing State_item from the buffer!!!\n");
                return JAUS_FALSE;
            }
            index += stateSize(*State_item);
            jausArrayAdd(message->prediction, *State_item);
            if (!jausDateStampFromBuffer(message->time, buffer+index, bufferSizeBytes-index)) {
                printf("Error: removing message->time from the buffer!!!\n");
                return JAUS_FALSE;
            }
            index += JAUS_UNSIGNED_INTEGER_SIZE_BYTES;
            if (!jausDateStampToBuffer(message->time, buffer+index, bufferSizeBytes-index)) {
                printf("Error: removing message->time from the buffer!!!\n");
                return JAUS_FALSE;
            }
            index += JAUS_UNSIGNED_SHORT_SIZE_BYTES;
            // end generated code
            return JAUS_TRUE;
        }
        else {
            printf("Buffer size not appropriate in dataFromBuffer(PredictionMessage)\n");
            return JAUS_FALSE;
        }
    }

    // Returns number of bytes put into the buffer
    static int dataToBuffer(PredictionMessage message, unsigned char *buffer, unsigned int bufferSizeBytes)
    {
        int index = 0;
        if (bufferSizeBytes >= dataSize(message)) {
            // generated code
            if (!jausByteToBuffer(message->numprediction, buffer+index, bufferSizeBytes-index)) {
                printf("Error: placing numprediction into the buffer!!!\n");
                return JAUS_FALSE;
            }
            index += JAUS_BYTE_SIZE_BYTES;
            unsigned int iprediction;
            for (iprediction = 0; iprediction < message->numprediction; iprediction++) {
                State *State_item = message->prediction->elementData[iprediction];
                if (!stateToBuffer(State_item, buffer+index, bufferSizeBytes-index)) {
                    printf("Error: placing State_item into the buffer!!!\n");
                    return JAUS_FALSE;
                }
                index += stateSize(*State_item);
            }
            if (!jausDateStampToBuffer(message->time, buffer+index, bufferSizeBytes-index)) {
                printf("Error: placing message->time into the buffer!!!\n");
                return JAUS_FALSE;
            }
            index += JAUS_UNSIGNED_INTEGER_SIZE_BYTES;
            if (!jausDateStampFromBuffer(message->time, buffer+index, bufferSizeBytes-index)) {
                printf("Error: placing message->time into the buffer!!!\n");
                return JAUS_FALSE;
            }
            index += JAUS_UNSIGNED_SHORT_SIZE_BYTES;
            // end generated code
        }
    }
static int dataToString(PredictionMessage message, char **buf)
{
    unsigned int bufSize = 128 + dataSize(message);
    char *tempStr = (char*) malloc(sizeof(char) * bufSize);
    // generated code
    strcat(tmpStr, "PredictionMessage\n");
    strcat(tmpStr, "prediction\n");
    int iiPrediction;
    for (iiPrediction = 0; iiPrediction < message->numprediction; iiPrediction++) {
        strcat(tmpStr, stateToString(message->prediction->elementData[iiPrediction]));
    }
    char* strtime = (char*) malloc(sizeof(char) * 100);
    jausTimeToString(message->time, strtime, 100);
    strcat(tmpStr, strtime);
    free(strtime);
    // end generated code
    return strlen((*buf));
}

// Returns number of bytes put into the buffer
static unsigned int dataSize(PredictionMessage message)
{
    unsigned int size = 0;
    // generated code
    size += JAUS_BYTE_SIZEBYTES;
    unsigned int iiPrediction;
    for (iiPrediction = 0; iiPrediction < message->prediction->elementCount; iiPrediction++) {
        size += stateSize(message->prediction->elementData[iiPrediction]);
    }
    size += JAUS_UNSIGNEDINTEGER_SIZEBYTES + JAUS_UNSIGNEDSHORT_SIZEBYTES;
    // end generated code
    return size;
}

PredictionMessage predictionMessageCreate(void)
{
    PredictionMessage message;
    message = (PredictionMessage) malloc(sizeof(PredictionMessageStruct));
    if (message == NULL) {
        return NULL;
    }
    // Initialize Values
    message->properties.priority = JAUS_DEFAULT_PRIORITY;
    message->properties.ackNak = JAUS_ACK_NAK_NOT_REQUIRED;
    message->properties.scFlag = JAUS_NOTSERVICE_CONNECTION_MESSAGE;
    message->properties.expFlag = JAUS_EXPERIMENTAL_MESSAGE; // HACK: why isn't this from the
    struct?
    message->properties.version = JAUS_VERSION_33;
    message->properties.reserved = 0;
    message->commandCode = commandCode;
    message->destination = jausAddressCreate();
    message->source = jausAddressCreate();
    message->dataFlag = JAUS_SINGLE_DATA_PACKET;
    message->dataSize = maxDataSizeBytes;
    message->sequenceNumber = 0;
    dataInitialize(message);
    message->dataSize = dataSize(message);
    return message;
  }

void predictionMessageDestroy (PredictionMessage message)
{
  // printf ("About to delete message data...\n")
  dataDestroy (message);
  // printf ("About to destroy address of source...\n")
  jausAddressDestroy (message->source);
  // printf ("About to destroy address of destination...\n")
  jausAddressDestroy (message->destination);
  // printf ("About to free message...\n")
  free (message);
}

JausBoolean predictionMessageFromBuffer (PredictionMessage message,
unsigned char *buffer, unsigned int bufferSizeBytes)
{
  int index = 0;

  if (headerFromBuffer (message, buffer + index, bufferSizeBytes - index))
  {
    index += JAUS_HEADER_SIZE_BYTES;
    if (dataFromBuffer (message, buffer + index, bufferSizeBytes - index))
    {
      return JAUS_TRUE;
    }
  }
  else
  {
    return JAUS_FALSE;
  }

  return JAUS_FALSE;
}

JausBoolean predictionMessageToBuffer (PredictionMessage message,
unsigned char *buffer, unsigned int bufferSizeBytes)
{
  if (bufferSizeBytes < predictionMessageSize (message))
  {
    return JAUS_FALSE; // improper size
  }
  else
  {
    message->dataSize = dataToBuffer (message, buffer + JAUS_HEADER_SIZE_BYTES,
                                       bufferSizeBytes - JAUS_HEADER_SIZE_BYTES);
    if (headerToBuffer (message, buffer, bufferSizeBytes))
    {
      return JAUS_TRUE;
    }
  }
  return JAUS_FALSE;
}

PredictionMessage predictionMessageFromJausMessage (JausMessage jausMessage)
{
  PredictionMessage message = predictionMessageCreate ();
  if (jausMessage->commandCode != commandCode)
  {
    return NULL; // Wrong message type
  }
  else
  {
    // message = (PredictionMessage)malloc (sizeof (PredictionMessageStruct));
    // if (message == NULL)
    // {
    //   return NULL;
    // }
    message->properties.priority = jausMessage->properties.priority;
    message->properties.ackNak = jausMessage->properties.ackNak;
    message->properties.scFlag = jausMessage->properties.scFlag;
    message->properties.expFlag = jausMessage->properties.expFlag;
    message->properties.version = jausMessage->properties.version;
    message->properties.reserved = jausMessage->properties.reserved;
    message->commandCode = jausMessage->commandCode;
    message->destination = jausAddressCreate ();
  }
  return message;
}
message->destination = jausMessage->destination;
message->source = jausAddressCreate();
message->dataSize = jausMessage->dataSize;
message->dataFlag = jausMessage->dataFlag;
message->sequenceNumber = jausMessage->sequenceNumber;

// generated code
message->prediction = jausArrayCreate();

// end generated code

// Unpack jausMessage->data
if (dataFromBuffer(message, jausMessage->data, jausMessage->dataSize))
{
    return message;
}
else
{
    return NULL;
}
}

JausMessage predictionMessageToJausMessage(PredictionMessage message)
{
    JausMessage jausMessage;
    jausMessage = (JausMessage) malloc(sizeof(struct JausMessageStruct));
    if (jausMessage == NULL)
    {
        return NULL;
    }
    jausMessage->properties.priority = message->properties.priority;
    jausMessage->properties.ackNak = message->properties.ackNak;
    jausMessage->properties.scFlag = message->properties.scFlag;
    jausMessage->properties.expFlag = message->properties.expFlag;
    jausMessage->properties.version = message->properties.version;
    jausMessage->properties.reserved = message->properties.reserved;
    jausMessage->destination = jausAddressCreate();
    jausMessage->source = jausAddressCreate();
    jausMessage->dataSize = dataSize(message);
    jausMessage->dataFlag = message->dataFlag;
    jausMessage->sequenceNumber = message->sequenceNumber;
    jausMessage->data = (unsigned char*) malloc(jausMessage->dataSize);
    jausMessage->dataSize = dataToBuffer(message, jausMessage->data, jausMessage->dataSize);

    return jausMessage;
}

unsigned int predictionMessageSize(PredictionMessage message)
{
    return (unsigned int)(dataSize(message) +JAUS_HEADER_SIZE_BYTES);
}

char* predictionMessageToString(PredictionMessage message)
{
    if (message)
    {
        char* buf1 = NULL;
        char* buf2 = NULL;
        int returnVal;

        // Print the message header to the string buffer
        returnVal = headerToString(message, &buf1);
        // Print the message data fields to the string buffer
        returnVal += dataToString(message, &buf2);

        char* buf;
        buf = (char*) malloc(strlen(buf1)+strlen(buf2));
        strcat(buf, buf1);
        strcat(buf, buf2);
        free(buf1);
        free(buf2);

        return buf;
    }
if (bufferSizeBytes < JAUS_HEADER_SIZE_BYTES)
{
    return JAUS_FALSE;
}
else
{
    // unpack header
    message->properties.priority = (buffer[0] & 0x0F);
    message->properties.ackNak = ((buffer[0] >> 4) & 0x0F);
    message->properties.scFlag = ((buffer[0] >> 6) & 0x01);
    message->properties.expFlag = ((buffer[0] >> 7) & 0x01);
    message->properties.version = (buffer[1] & 0xF0);
    message->properties.reserved = ((buffer[1] >> 6) & 0x03);
    message->destination->instance = buffer[4];
    message->destination->component = buffer[5];
    message->destination->node = buffer[6];
    message->destination->subsystem = buffer[7];
    message->source->instance = buffer[8];
    message->source->component = buffer[9];
    message->source->node = buffer[10];
    message->source->subsystem = buffer[11];
    message->dataSize = buffer[12] + ((buffer[13] & 0xF) << 8);
    message->dataFlag = ((buffer[13] >> 4) & 0xF);
    message->sequenceNumber = buffer[14] + (buffer[15] << 8);
    return JAUS_TRUE;
}

}
return JAUS_TRUE;
}
}
static int headerToString (PredictionMessage message, char **buf)
{
// message existence already verified
// Setup temporary string buffer
unsigned int bufSize = 500;
(*buf) = (char*)malloc(sizeof(char)*bufSize);
strcpy((*buf), jausCommandCodeString(message->commandCode));
sprintf((*buf)+strlen(*buf), "%04X", message->commandCode);
strcat((*buf), "\nreserved: ");
jausUnsignedShortToString(message->properties.reserved, (*buf)+strlen(*buf));
strcat((*buf), "\nversion: ");
switch(message->properties.version){
case 0:
    strcat((*buf), " 2.0 and 2.1 compatible");
    break;
case 1:
    strcat((*buf), " 3.0 through 3.1 compatible");
    break;
case 2:
    strcat((*buf), " 3.2 and 3.3 compatible");
    break;
case 3:
    strcat((*buf), ": Reserved for Future: ");
    break;
    default:
    strcat((*buf), "\nexp. flag: ");
    if(message->properties.expFlag == 0)
    strcat((*buf), "Not Experimental");
    else
    strcat((*buf), " Experimental");
    strcat((*buf), "\nservice flag: ");
    if(message->properties.scFlag == 1)
    strcat((*buf), " Service Connection");
    else
    strcat((*buf), " Not Service Connection");
    strcat((*buf), "\nack/nak: ");
    switch(message->properties.ackNak){
case 0:
    strcat((*buf), "None");
    break;
case 1:
    strcat((*buf), " Request ack/nak");
    break;
case 2:
    strcat((*buf), " nak response");
    break;
case 3:
    strcat((*buf), " ack response");
    break;
    default:
    break;
    }
    strcat((*buf), "\npriority: ");
    if(message->properties.priority < 12)
    {
    strcat((*buf), " Normal Priority");
    jausUnsignedShortToString(message->properties.priority, (*buf)+strlen(*buf));
    }
    else
    {
    strcat((*buf), " Safety Critical Priority");
    jausUnsignedShortToString(message->properties.priority, (*buf)+strlen(*buf));
    }
    strcat((*buf), "\nsource: ");
    jausAddressToString(message->source, (*buf)+strlen(*buf));
```c
545  strcat((*buf), "\nDestination:");
546  jausAddressToString(message->destination, (*buf)+strlen(*buf));
547  strcat((*buf), "\nData Size:");
548  jausUnsignedIntegerToString(message->dataSize, (*buf)+strlen(*buf));
549  strcat((*buf), "\nData Flag:");
550  jausUnsignedIntegerToString(message->dataFlag, (*buf)+strlen(*buf));
551  switch(message->dataFlag)
552  {
553    case 0:
554      strcat((*buf), "Only data packet in single-packet stream");
555      break;
556    case 1:
557      strcat((*buf), "First data packet in multi-packet stream");
558      break;
559    case 2:
560      strcat((*buf), "Normal data packet");
561      break;
562    case 4:
563      strcat((*buf), "Retransmitted data packet");
564      break;
565    case 8:
566      strcat((*buf), "Last data packet in stream");
567      break;
568    default:
569      strcat((*buf), "Unrecognized data flag code");
570      break;
571  }
572  strcat((*buf), "\nSequence Number:");
573  jausUnsignedShortToString(message->sequenceNumber, (*buf)+strlen(*buf));
574  return strlen(*buf);
575  }
```
A.3 State code

```c
#ifndef State_H
#define State_H

#ifdef cplusplus
extern "C"
#endif

#include <jaus.h>
#include <string.h>
#include "azcar_common.h"

typedef struct
{
    // generated code
    JausDouble carAngle;
    JausDouble x;
    JausDouble tireAngle;
    JausDouble z;
    JausDouble y;

    // end generated code
} StateStruct;

typedef StateStruct *State;

// State Constructor
JAUS_EXPORT State stateCreate(void);

// State Constructor (from Buffer)
JAUS_EXPORT JausBoolean stateFromBuffer(State *objectPointer, unsigned char *buffer, unsigned int bufferSizeBytes);

// State To Buffer
JAUS_EXPORT JausBoolean stateToBuffer(State object, unsigned char *buffer, unsigned int bufferSizeBytes);

// State Destructor
JAUS_EXPORT void stateDestroy(State object);

// State Buffer Size
JAUS_EXPORT unsigned int stateSize(State object);

// State Data To String
JAUS_EXPORT char* stateToString(State object);

#ifdef cplusplus
}
#endif
#endif // State_H
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <jaus.h>
#include <openJaus.h>

// wayPoint Constructor
State stateCreate(void)
{
```

```c
/* state.c
* OpenJaus
* Created by JausMessageML Interpreter on 06/07/11.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <jaus.h>
#include <openJaus.h>

// wayPoint Constructor
State stateCreate(void)
{
```
State object;
        object = (State) malloc(sizeof(StateStruct));
        if (object) {
          // generated code
          object->carAngle = newJausDouble(0);
          object->x = newJausDouble(0);
          object->tireAngle = newJausDouble(0);
          object->y = newJausDouble(0);
        } else {
          return NULL;
        }
    }
    // end generated code
    return object;
}

WayPoint Constructor (from Buffer)
JausBoolean stateFromBuffer(State *objectPointer, unsigned char *buffer, unsigned int bufferSizeBytes)
{
  unsigned int index = 0;
  State object = stateCreate();
  if (object != NULL) {
    // generated code
    JausUnsignedShort ushort carAngle;
    if (!jausUnsignedShortFromBuffer(& ushort carAngle, buffer+index, bufferSizeBytes-index)) {
      printf("Error removing object->carAngle from the buffer!!!\n");
      return JAUS_FALSE;
    }
    object->carAngle = tan(jausUnsignedShortToDouble(ushort carAngle, ANGLE_MIN, ANGLE_MAX));
    index += JAUS_UNSIGNED_SHORT_SIZE_BYTES;
    JausUnsignedInteger uint x;
    if (!jausUnsignedIntegerFromBuffer(& uint x, buffer+index, bufferSizeBytes-index)) {
      printf("Error removing object->x from the buffer!!!\n");
      return JAUS_FALSE;
    }
    object->x = jausUnsignedIntegerToDouble(uint x, DISTANCE_MIN, DISTANCE_MAX);
    index += JAUS_UNSIGNED_INTEGER_SIZE_BYTES;
    JausUnsignedShort ushort tireAngle;
    if (!jausUnsignedShortFromBuffer(& ushort tireAngle, buffer+index, bufferSizeBytes-index)) {
      printf("Error removing object->tireAngle from the buffer!!!\n");
      return JAUS_FALSE;
    }
    object->tireAngle = tan(jausUnsignedShortToDouble(ushort tireAngle, ANGLE_MIN, ANGLE_MAX));
    index += JAUS_UNSIGNED_SHORT_SIZE_BYTES;
    JausUnsignedInteger uint y;
    if (!jausUnsignedIntegerFromBuffer(& uint y, buffer+index, bufferSizeBytes-index)) {
      printf("Error removing object->y from the buffer!!!\n");
      return JAUS_FALSE;
    }
    object->y = jausUnsignedIntegerToDouble(uint y, DISTANCE_MIN, DISTANCE_MAX);
    index += JAUS_UNSIGNED_INTEGER_SIZE_BYTES;
    // end generated code
    *objectPointer = object;
    return JAUS_TRUE;
  } else {
    printf("Problem creating way point\n");
    return JAUS_FALSE;
  }
}

// Returns number of bytes put into the buffer
static unsigned int dataSize(State object) {
  unsigned int size = 0;
  // generated code
  object->carAngle = newJausDouble(0);
  object->x = newJausDouble(0);
  object->tireAngle = newJausDouble(0);
  object->y = newJausDouble(0);
  // end generated code
  return JAUS_TRUE;
}
unsigned int place(z, buffer+index, bufferSizeBytes)

unsigned int index = 0;

index += JAUS_UNSIGNED_SHORT_SIZE_BYTES;

index += JAUS_UNSIGNED_INTEGER_SIZE_BYTES;

index += JAUS_UNSIGNED_INTEGER_SIZE_BYTES;

// end generated code

return index;

//*

stateDestroy(

size = JAUS_UNSIGNED_SHORT_SIZE_BYTES;

size = JAUS_UNSIGNED_INTEGER_SIZE_BYTES;

size = JAUS_UNSIGNED_SHORT_SIZE_BYTES;

size = JAUS_UNSIGNED_INTEGER_SIZE_BYTES;

size = JAUS_UNSIGNED_INTEGER_SIZE_BYTES;

// end generated code

return size;

} /*

// External interface to tell how much data in a way point

unsigned int stateSize(State object)

{ return dataSize(object); }

} /*

// WayPoint To Buffer

JausBoolean stateToBuffer(State object, unsigned char *buffer, unsigned int bufferSizeBytes)

if(object && (bufferSizeBytes >= dataSize(object)))

{ /*

// generated code

short carAngle = jausUnsignedShortFromDouble(atan(object->carAngle), ANGLE_MIN, ANGLE_MAX);

if(!jausUnsignedShortToBuffer(carAngle, buffer+index, bufferSizeBytes-index))

printf("Error placing object->carAngle into the buffer!! \n");

return JAUS_FALSE;

} /*

index += JAUS_UNSIGNED_SHORT_SIZE_BYTES;

unsigned Integer uint_x = jausUnsignedIntegerFromDouble(object->x, DISTANCE_MIN, DISTANCE_MAX);

if(!jausUnsignedIntegerToBuffer(uint_x, buffer+index, bufferSizeBytes-index))

printf("Error placing object->x into the buffer!! \n");

return JAUS_FALSE;

} */

index += JAUS_UNSIGNED_INTEGER_SIZE_BYTES;

unsigned Integer uint_y = jausUnsignedIntegerFromDouble(object->y, DISTANCE_MIN, DISTANCE_MAX);

if(!jausUnsignedIntegerToBuffer(uint_y, buffer+index, bufferSizeBytes-index))

printf("Error placing object->y into the buffer!! \n");

return JAUS_FALSE;

} /*

index += JAUS_UNSIGNED_INTEGER_SIZE_BYTES;

} /*

// end generated code

return JAUS_TRUE;

} /*

// WayPoint Destructor

void stateDestroy(State object)

{ /*

nothing to see here...

} /*

// generated code

} /*

// end generated code

} /*

char* stateToString(State object)

{ unsigned int bufSize = 128+dataSize(object);

char* tempStr = (char*)malloc(sizeof(char)*bufSize);

// generated code

char* strcarAngle = (char*)malloc(sizeof(char)*100);

jausDoubleToString(object->carAngle, strcarAngle);

strcpy(tempStr, strcarAngle);

free(strcarAngle);

87
184  strcat (tempStr, "\n");
185  char* strx = (char*)malloc(sizeof(char)*100);
186  jausDoubleToString(object->x, strx);
187  strcat (tempStr, strx);
188  free(strx);
189  strcat (tempStr, "\n");
190  char* strtireAngle = (char*)malloc(sizeof(char)*100);
191  jausDoubleToString(object->tireAngle, strtireAngle);
192  strcat (tempStr, strtireAngle);
193  free(strtireAngle);
194  strcat (tempStr, "\n");
195  char* strz = (char*)malloc(sizeof(char)*100);
196  jausDoubleToString(object->z, strz);
197  strcat (tempStr, strz);
198  free(strz);
199  strcat (tempStr, "\n");
200  char* stry = (char*)malloc(sizeof(char)*100);
201  jausDoubleToString(object->y, stry);
202  strcat (tempStr, stry);
203  free(stry);
204  strcat (tempStr, "\n");
205  // end generated code
206  return tempStr;
207  }