THE OPENJAUS APPROACH TO DESIGNING AND IMPLEMENTING THE NEW SAE JAUS STANDARDS

Thomas Galluzzo,[*] Danny Kent[†]

The Joint Architecture for Unmanned Systems (JAUS) is an international standard of the SAE AS-4 Unmanned Systems Steering Committee. The OpenJAUS project team has recently undergone an effort to update their software to support the new SAE JAUS standards (AS5684, AS5669 and AS5710). This paper will discuss the critical design issues that needed to be overcome in order to develop an effective JAUS solution. Also, several features that were added to the OpenJAUS design to address advanced requirements currently outside of the scope of JAUS will be discussed. The paper will highlight how the OpenJAUS team conducted their development work in a distributed online manner, and give examples of the tools used for this process. An approach will be given for how the new OpenJAUS features will be offered to AS-4 for future standardization.

INTRODUCTION

The Joint Architecture for Unmanned Systems (JAUS) is a standard that was originally chartered by the Department of Defense (DoD). Its original purpose was to define an open communication standard to support interoperability of robotic systems in the military. Many programs and vendors became involved with JAUS and have successfully demonstrated its use on unmanned systems. Some examples of these systems include several robots from the 2004 and 2005 DARPA Grand Challenges and the 2007 DARPA Urban Challenge. The third place Urban Challenge vehicle, from Virginia Tech, was developed using the JAUS standard; showing its potential to support complex missions.

Beginning in 2004, the JAUS Working Group transitioned to the Society of Automotive Engineers (SAE) standards body under the title AS-4 Unmanned Systems. Since then the JAUS standard has undergone significant changes to improve its functionality, reliability and interoperability. In recent years a number of documents such as: AS5684 JAUS Service Interface Definition Language (JSIDL)[1], AS5710 JAUS Core Service Set[2], AS6009 JAUS Mobility Service Set[3] and AS5669 JAUS Transport Specification[4] have been published by SAE. These documents provide the foundation for the next-generation of JAUS. Most of the new standard is built upon JSIDL, which defines an XML schema that enables formal specification of JAUS Services, Messages and Message Protocol. This schema aids robust and reliable interoperability by removing some of the ambiguities often found in hand-written standards.

[*] Senior Robotics Engineer, National Robotics Engineering Center, Ten 40th St. Pittsburgh, PA 15201
[†] Cofounder, OpenJAUS, danny@openjaus.com

OpenJAUS is an open source Software Development Kit (SDK) based on the JAUS standard, which was founded in 2006 by researchers from the University of Florida (UF). Based on proven technology developed by UF for the DARPA Grand Challenge; OpenJAUS has quickly become one of the leading JAUS implementations. It has been readily adopted by industry and educational organizations alike. Since 2006 the OpenJAUS team has continued to support and evolve their original codebase by maturing the code, fixing bugs and supporting a number of customers across the unmanned systems community.

In late 2009, the OpenJAUS team began development of a next-generation code base to implement the new SAE JAUS standard using a variety of powerful tools through the design, development and testing process. This paper will examine the new OpenJAUS design and implementation process. Additional focus is placed on the powerful modeling, compilation and testing tools used by the team to facilitate collaboration and development with minimum manpower.

The remaining sections of this paper are organized as follows. An overview of JAUS terminology and the rise of interest in the standard is provided in the next section. The Design section reviews the goals set out by the OpenJAUS team for implementing the new JAUS standards, and discusses the difficult challenges that needed to be solved during this process. The new OpenJAUS design itself is then discussed in the Technical Approach, in which four specific technical enhancements to the JAUS standard are highlighted. A review of the implementation process and the tools used during this process is given in the Development section. The Standardization section explains the JAUS standard balloting process, and how the OpenJAUS team plans to bring new enhancements into the standard. Finally, a summary of findings and plans for future work are given in the Conclusion.

JAUS OVERVIEW

This paper describes in detail the new OpenJAUS design and the technical problems it solves. It is assumed that the reader has a thorough knowledge of the JAUS standard and its nomenclature. Many of the concepts and terms discussed will be foreign to anyone newly introduced to JAUS. It is highly recommended that the reader refer to the current SAE JAUS documents to understand the specific concepts. In the interest of those who cannot access the JAUS documents, a brief summary of basic concepts is provided here. This summary is intended to serve as an introduction to the key terms and definitions used hereafter.

The main goal of JAUS is to structure communication and interoperation of unmanned systems within a network. At the highest level, JAUS systems are made up of a number of Subsystems. A Subsystem typically represents a physical entity in the system network, such as an unmanned vehicle or operator control unit (See Figure 1). The JAUS network is further subdivided into hierarchical layers. At the next level down Subsystems are divided into Nodes, which represent a physical computing end-point in the system. For example, a Node might be a computer or microcontroller within a Subsystem. Nodes can then host one or more Components, which are commonly applications or threads running on the Node. Finally, Components are made up of one or more Services. A Service simply provides some useful function for the system, and has a well-defined message interface and protocol. The JSIDL standard formally defines how Services, Messages and Message Protocol are specified.

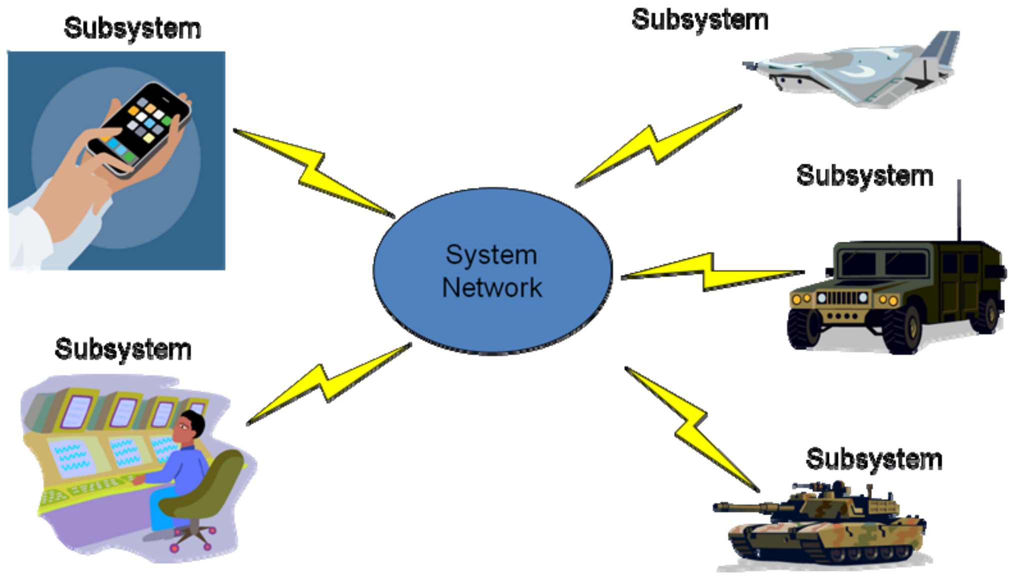

Figure 1 High-level JAUS System Architecture

Growth of JAUS

Interest in JAUS and OpenJAUS is growing. Evidence for this is provided here in two ways. First, the OpenJAUS team has monitored and compiled statistics to measure downloads, and website traffic since 2006. These statistics show that approximately 50% of all traffic comes from search engine queries with keywords focused on JAUS. The website traffic has seen steady growth over the past three years, growing to almost two thousand visitors and over 50,000 page views per month (see Figure 2). Given that half of this traffic is generated from people searching for JAUS, it is clear that much of the growth is attributable to rising interest in the standard itself.

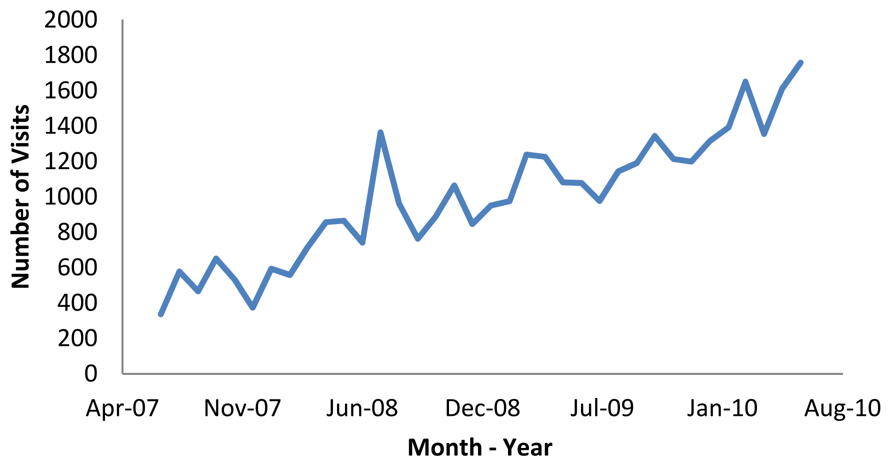

Figure 2 OpenJAUS.com Monthly Visitor Growth

The second indicator of growth is shown by increasing requirements for JAUS. Recently the US Navy's AEODRS project has specified the JAUS standard for the logical / communications interface. Also, in June 2010 the Robotic Systems Joint Project Office (RSJPO) held an industry forum on Unmanned Ground Vehicle (UGV) interoperability standards in which JAUS was specified as the standard of choice for the logical layer. In addition to these programs, the Association for Unmanned Vehicle Systems International (AUVSI) sponsors five yearly student competitions covering a wide range of unmanned systems, including ground, air, surface and underwater systems; each of these student competitions includes a JAUS portion that challenges student teams to use the standard with varying degrees of complexity.

DESIGN

The new JAUS standard under SAE has significant differences from its previous versions. The most prominent of which is its migration to Service Oriented Architecture (SOA)[5]. The new SOA version of JAUS attempts to formalize the message format and protocol interaction between system components[6]. This approach is standardized by the JAUS Service Interface Definition Language (JSIDL); an XML-based language that provides the basic structure and syntax for specifying JAUS Services. All of the Services that are standardized by JAUS must be specified in valid JSIDL syntax.

In December 2009 the OpenJAUS development team began the process of outlining the design requirements for a new codebase centered on the new SOA JAUS standard. Significant experience from the design, development and deployment of the original OpenJAUS products provided the foundation for the design requirements. It was important that the best features of the existing OpenJAUS SDK were incorporated into the next generation code. The development team also operated off several design "tenants" which are based on past experiences, such as:

- A focus on a simple & robust end-user (developer) interface
- Clarity and completeness of the API (such as function names, variable names, etc)
- Bitwise compliance with the published JAUS standards

Another key development focus has been on primarily supporting the published standards. Experience in the past with the OpenJAUS project has shown that many users are only interested in using the published standard and are not interested in experimental messages or services. This is often done because the source of the design requirement of a given project is some compliance or interoperability effort. Therefore, the majority of OpenJAUS development efforts have been to focus on the users of the standard as published rather than providing a sandbox in which new services and messages are easily and quickly developed.

Design requirements were discussed in an online-collaborative environment using the Google Wave tool to: track the requirements and design discussion in real-time, document conceptual drawings, and collaborate on things such as a code style guide. Many features outside the scope of the JAUS standard, such as message time stamps, data compression and data encryption, were included in the new OpenJAUS design. These enhancements were based on the most requested features from current users of OpenJAUS.

To organize development documents and pseudo-code, the OpenJAUS development team also deployed an online code repository based on Mercurial. This distributed revision control repository allowed rapid development in a collaborative and distributed effort. Early prototypes and pseudo-code allowed the development team to prototype a number of interfaces and capabilities such as the message library, protocol state machine interfaces, dynamic configuration and discovery.

Design Challenges

The key challenge presented by the new JAUS standard comes from JSIDL, which requires an implementation design that is in line with its SOA approach and conducive to direct (or automatic by machine) translation of XML Service specifications to code. This means that given the JSIDL code for a Service, there must be a clear way to structure the resultant Messaging and Protocol code for the actual software that will be executed online in the JAUS-based system.

Since most JAUS implementations prior to JSIDL had a significantly different flow and structure, porting old JAUS code into this new form has proven to be non-trivial. Indeed many organizations that pursued implementing their own JAUS code in the past have not yet migrated to SAE JAUS due to the cost associated with reworking their software. Furthermore, in many instances the JSIDL standard document does not provide clear guidance or examples on how software can be structured to meet the specification. Thus the onus is on the system developer to design their software in a form that is compatible with JSIDL.

In addition to general software architecture, another difficult aspect of JSIDL is its use of protocol state machines. In JSIDL, the message protocol for each Service is defined as a Finite State Machine (FSM). Each FSM can contain any number of states, each of which structure how different messages are processed within the Service. Additionally, JSIDL allows any Service to extend or inherit protocol from another Service. The result of which is a nested FSM, made up of states within states. This nested structure is what specifies the exact protocol for how each message is processed. The complexities associated with these nested FSMs make finding an elegant implementation very hard. The OpenJAUS team struggled through several design iterations before converging on a well-posed approach.

TECHNICAL APPROACH

All of the changes from the old JAUS Reference Architecture to the new JAUS design under SAE have created the need to transform the OpenJAUS software. This required change is drastic, but it was used by the OpenJAUS team as an opportunity to “push the envelope” and enhance what can be done with JAUS. Presented here are some key enhancements that OpenJAUS will offer in its upcoming releases.

State machine changes

As previously stated, the nested FSMs that are used in JAUS Services present a difficult implementation problem. The new OpenJAUS design uses an approach that simplifies the use of these State Machines. The key feature enabled by this design is the ability for multiple Services to operate in parallel within a single Component.

With the existing JSIDL standard approach, States may contain Sub-States, and thus a nested state machine structure is possible (and often required). The problem with this approach arises when multiple sibling States need to execute simultaneously. In this case, the State responsible for processing an incoming Message is ambiguous. Allowing both States to process the Message is not a viable solution, because this may result in conflicting behaviors or responses from the Service. There needs to be a mechanism to ensure only one State processes a given Message. Currently JSIDL does not specify a clear approach on how to do this, yet the new JSS Core Standard implies a need to have multiple Services (all of which inherit from a single parent Service) operating in parallel within a single Component.

The new OpenJAUS design solves this problem by compartmentalizing nested States within their own State Machines. In other words, States are allowed only to contain Sub-State-Machines, rather than Sub-States directly. This allows the system to enforce a simple rule; parallel (or sibling) State Machines must be mutually exclusive in the Messages they can receive. Therefore when a Message is received by a State, one and only one contained State Machine can process the Message. A State Machine can only have one currently active State, so this ensures the Message will be processed by only a single State. All of this capability and design is abstracted from the end OpenJAUS developer, which greatly simplifies how Services are programmed.

Transport Policies

Some of the enhancements the new OpenJAUS were requested by existing OpenJAUS users. These capabilities, such as Message compression, time stamping and data encryption, were difficult to incorporate into the existing OpenJAUS SDK without breaking interoperability with other JAUS systems. With the development of the new SAE JAUS SDK, it was important to include these features from the start in a way that did not break compliance or interoperability. The concept of a transport policy was developed to solve these issues.

A transport policy describes the communications behavior between any two interfaces (typically an Ethernet interface). The policy describes what additional features are supported by the interface, if any. Policies are discovered online, allowing different transport behavior on a case-by-case basis. For example, while one system may support data compression, another system may support both data compression and encryption. Generic data compression, message encryption and a per-message timestamp fields are some of the capabilities to be supported by the transport policy mechanism.

It is very important that the addition of these capabilities does not impact the standard messaging protocol for compliance or interoperability. This was mitigated by using of the type field defined by the JUDP standard. A value for the type field is standardized for JAUS messages but it is left open for other uses. The OpenJAUS design has extended the use of this field by defining a new type code for the exchange of transport policies. This allows standard JAUS components which do not expect or understand these messages to ignore them. The OpenJAUS system is designed to fail gracefully in that situation and revert to standard JAUS messaging protocols, i.e. a transport policy of standard AS5669 JUDP.

An example of the message protocol for exchanging transport policy information and then using it online is shown in Figure 3. In this example, Service 1 initiates the conversation by transmitting a QueryJausAddress Message using broadcast semantics. The receiving component, Service 2, responds with the ReportJausAddress Message. Receipt of the report inside Service 1 causes the JudpInterface object (part of the OpenJAUS SDK) to trigger the discovery of Service 2's transport policy. It sends a QueryTransportPolicy Message to Service 2 (with type code 2). Service 2 receives this inquiry and responds with its ReportTransportPolicy Message. When Service 1 receives the report message the transport policy map is configured according to the report. Once the transport policy is established, messages are exchanged between entities according to the least-common-denominator of supported capabilities. These transformed messages are transmitted using the OpenJAUS type code, which informs Service 2 to decode the message according to the previously discovered transport policy.

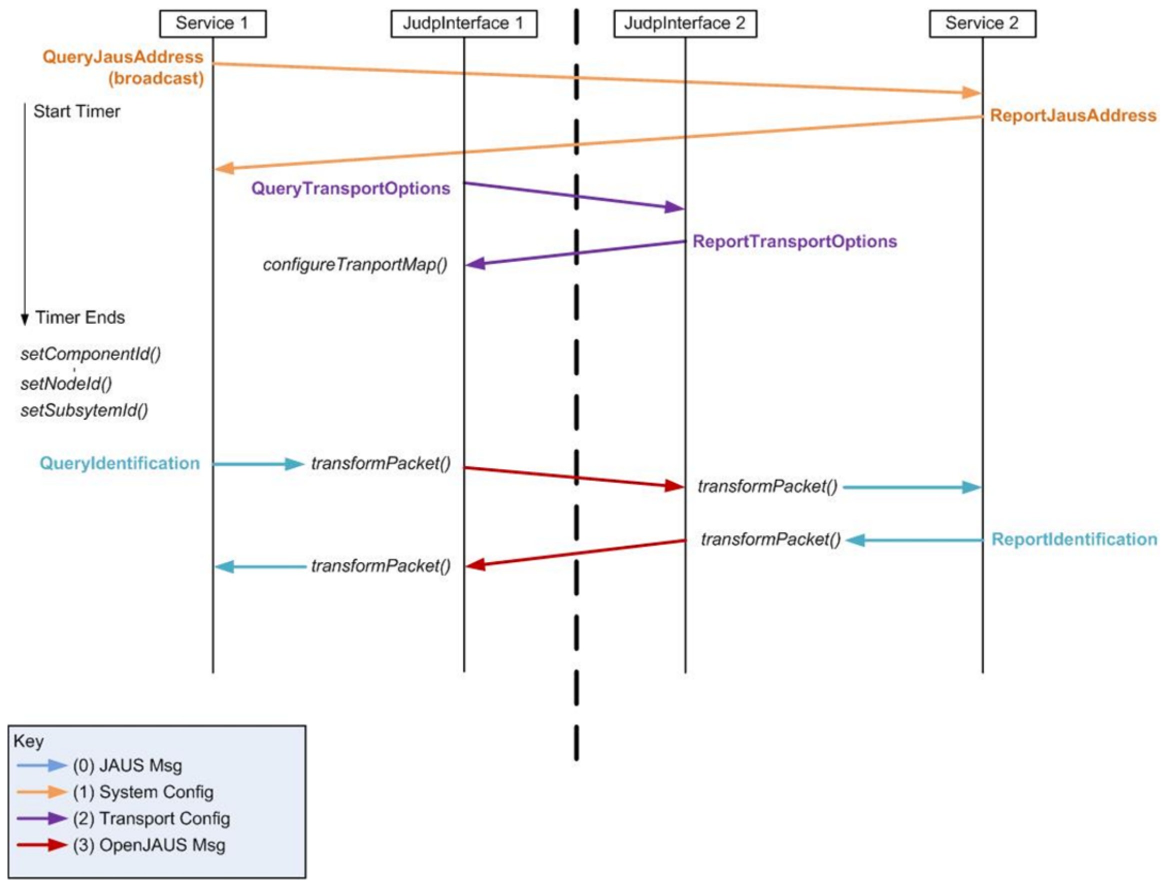

Figure 3 OpenJAUS Transport Policy

Online dynamic configuration

JAUS supports the ability for Components to discover one another online. This process is called dynamic discovery. Dynamic discovery greatly aids the integration process because it allows JAUS developers to quickly network their systems together.

In order for dynamic discovery to work, each Component in the JAUS system must have a unique address. The address is made up of three parts: Subsystem number, Node number and Component number. The existing JAUS standard does not specify how addresses should be assigned. It is currently left to the system designer to establish an addressing scheme. This greatly dampens the ability for “plug-and-play” like interoperability. Some level of manual configuration or programming is always required in order for two separately developed systems to work together.

OpenJAUS-based systems are capable of dynamic configuration. This automatically allows Components to configure their addresses online. Once addresses are established, the standard dynamic discovery process ensues. Dynamic configuration enhances the integration process such that “plug-and-play” style interoperability is possible. Additionally dynamic configuration makes more advanced JAUS networking capabilities possible. One example of how OpenJAUS systems are able to dynamically configure is by connecting to a central address server. The server is responsible for ensuring each Component in the network has a unique address. This simple capability allows JAUS systems to interoperate globally via the internet. By connecting to an address server at a fixed URL, (example: “server.openjaus.net”) OpenJAUS systems will be able to discover any other OpenJAUS system connected to the internet. This is a very powerful feature that allows systems integrators to test basic communication and interoperability between each other without the need for expensive face to face testing. Additionally, users will be able to run their own private servers. OpenJAUS will provide free online address servers for users that are not concerned with communication privacy. However, there will be no requirement to use them. They will be provided for convenience only.

Access control enhancements

Some JAUS Components are responsible for controlling critical system functions such as propulsion and mobility. It is imperative that these types of components only receive commands from a single source. This prevents the component from receiving conflicting commands, which may result in undefined or undesirable behavior. JAUS has a mechanism that enforces this exclusive control requirement, namely the Access Control Service.

The Access Control Service is designed to deliver exclusive control of a Component with almost no restriction on the controlling source. While this is desirable for ease of use, it leaves the system vulnerable to misuse. The ability to strictly and robustly regulate access to a Component is something very desirable to systems concerned with security.

OpenJAUS intends to address this issue by providing mechanisms for mandatory access control. These mechanisms will be similar to operating system level permissions, user access lists and password protection. In conjunction with encryption transport policies, this will enable OpenJAUS-based systems to operate on open networks with far less risk of attack or misuse from nefarious sources. All of this is provided to the developer in software, requiring no need for external data security or encryption hardware.

Enhanced security allows for a broader level of control and information to be brought into a JAUS network. Components with this type of protection will be able to allow permission to change internal application parameters or deliver private information such as internal event or error reporting. Integration, development and online management of systems will benefit from these capabilities.

DEVELOPMENT

Computer Aided Software Engineering (CASE) techniques were used for development of the new OpenJAUS. A specific type of CASE called Model Driven Development (MDD) was employed for the engineering process. In MDD, software classes, methods and data members are first modeled in diagrammatical form. From this model you can then prototype an implementation by using automatic code generators. These generators take input from the model along with code template files. They can then automatically generate source code stubs, header files, build files, reference documentation, and unit tests; all using model specific information.

There are many reasons to use MDD for OpenJAUS. First, the number of developers working on OpenJAUS is very limited. MDD serves as a force multiplier that reduces effort to manage and write a large codebase. Second, the demand for quality in unmanned systems software is critical. MDD reduces the risk for hand coded errors. For example, by validating models of Services and Messages, automatic code generators can be used to create error-free Message serialization and parsing code. This ensures proper interoperation and communication with other JAUS-based systems. Finally, the development of OpenJAUS is decentralized. This means that the people working on it are geographically dislocated. Using a single model as the source of design enables group members to share a common understanding of the project. Communicating the design and implementation is greatly aided by a single view of the model.

Development Tools

The OpenJAUS team has recently completed an online survey of over 80 unmanned systems professionals. This survey shows that a significant percentage of unmanned systems developers are familiar with Eclipse and use it for development. Figure 4 shows a histogram of development tools selected by survey participants. The most significantly used tool is Microsoft Visual Studio. Although OpenJAUS is primarily developed in Eclipse, release versions for Visual Studio will be available.

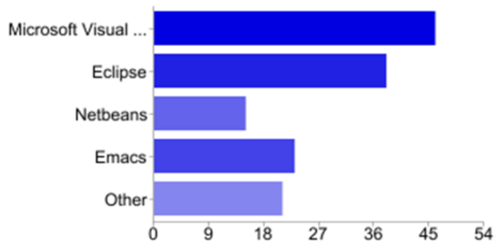

Figure 4 Histogram of Preferred Development Tools

At the center of the MDD process are the tools used to create and maintain the software model. In this case, the team chose Eclipse and the Eclipse Modeling Framework (EMF)[7] as the primary tools for developing OpenJAUS. Figure 4 shows that this choice is in line with the preference for Eclipse within the unmanned systems community.

Eclipse is a cross-platform integrated development environment that supports software development for many different programming languages. It is a free tool that has many plug-ins, which allow integration with other software development tools such as revision control systems, compilers and debuggers.

The EMF has the ability to create and validate software models. This framework supports modeling in many different ways. For example, it allows modeling in predefined languages such as UML, but it also allows the user to define and create their own Domain Specific Language (DSL) model along with the tools for editing models with their DSL[8]. In this way the OpenJAUS team was able to create a JAUS-based DSL. The EMF then provides auto-generated code and editing tools (in the form of Eclipse plug-ins) for this model.

Eclipse defines a minimalistic meta-modeling language called Ecore that simplifies design of a DSL. This was the language chosen to model the new OpenJAUS design. An example of the OpenJAUS Ecore model is shown in Figure 5. Ecore enables specification of simple object-oriented constructs such as packages, object classes, operations (or methods), attribute data references, and data members. The figure specifically shows the primary model for JAUS, in which the standard defines entities such as Services, States, Transitions, Messages, etc. Each of these is modeled as a single class within the "openjaus.model" package.

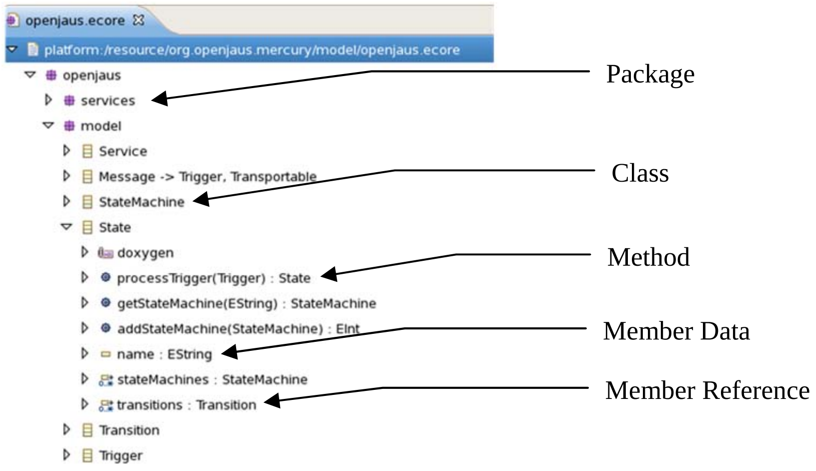
```
openjaus.ecore
platform:/resource/org.openjaus.mercury/model/openjaus.ecore
  openjaus
    services
    model
      Service
      Message -> Trigger, Transportable
      StateMachine
      State
        doxygen
        processTrigger(Trigger) : State
        getStateMachine(EString) : StateMachine
        addStateMachine(StateMachine) : EInt
        name : EString
        stateMachines : StateMachine
        transitions : Transition
      Transition
      Trigger
```

Figure 5 Ecore Model of OpenJAUS Design

A key advantage of MDD is the ability to use automatic code generation tools. These tools are able to create an initial set of code or modify an existing codebase given a model specification. The OpenJAUS team selected the Acceleo plug-in for Eclipse as the primary code generation tool. Acceleo integrates directly with EMF and allows the user to specify an input set of code templates that can be used for generation. In the Acceleo process the user first creates a set of template files. These templates use a special syntax that allows data to be extracted from a given model and converted directly into template formatted text. For example, templates can specify how to create and format: C++ code, HTML documentation, or any other text-based information.

OpenJAUS uses Acceleo templates for two separate model types. The first is the aforementioned Ecore model, the second is a model specified in the OpenJAUS DSL. As mentioned, the EMF is able to automatically create editing tools for a DSL given its specification in an Ecore model. The OpenJAUS team uses this editor to create models of JAUS Services and Messages. The models contain information about each Service such as its contained Messages, State Machines and Actions. OpenJAUS has Acceleo templates that create code for the defined Services. Given a Message specification in a Service model, Acceleo automatically generates the parsing and serialization code that is necessary for the message to be transported online.

Implementation Structure

The MDD workflow starts at the creation of the model, and then continues through code generation, implementation, compiling and testing the code. This process is iterated through small changes in the model, templates and implementation, which ultimately refine the product software. This process involves three separate types of source files: models, code templates, and implementation code. In the case of OpenJAUS, the model is specified in Ecore, the templates are in Acceleo syntax, and the implementation code is in C++. Given that users of OpenJAUS are most likely to interact with only the implementation code, it is desirable to separate it from the model and template files. The OpenJAUS code is therefore separated into two projects. The first project is called Mercury. It contains all of the model and template files needed to generate the implementation code. The second project is called Gemini. It contains the implementation code that is partly generated by Mercury, but the majority of the Gemini code comes from manual implementation.

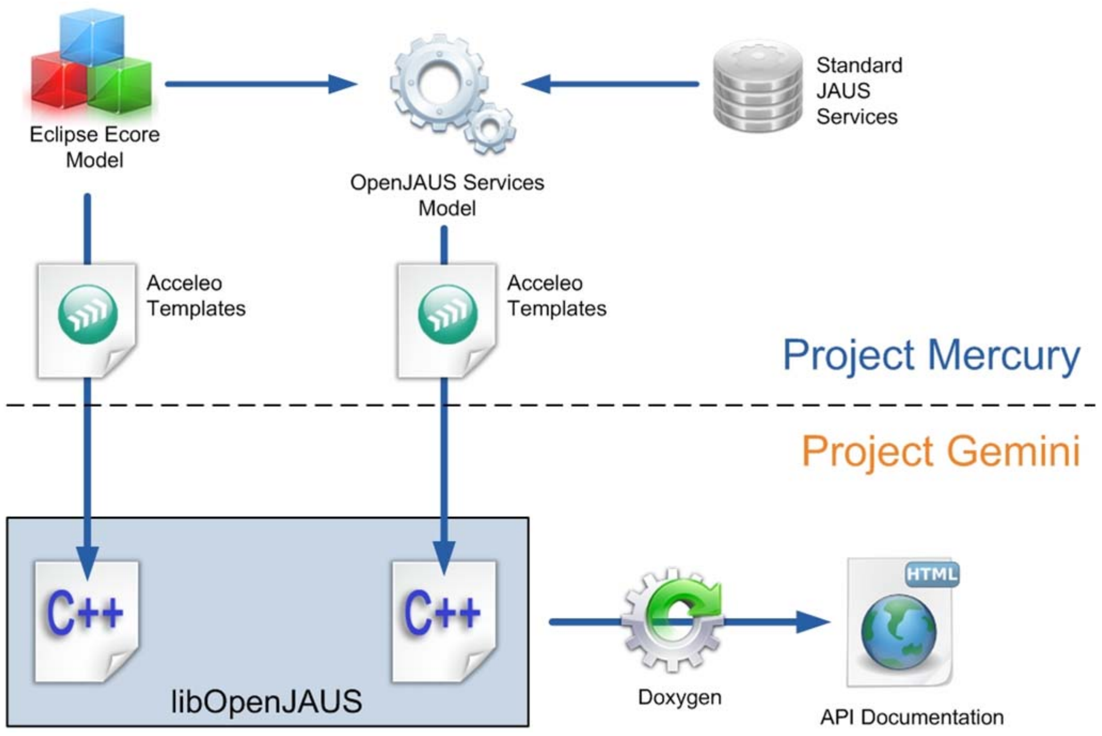

Figure 6 OpenJAUS MDD Workflow

STANDARDIZATION

The AS-4 working group uses a document sponsor and balloting process to introduce or change its standards, recommended practices, and information reports. This process requires a member of the AS-4 committee to sponsor new documents or changes to existing ones. The sponsor is then responsible for submitting the documents for committee review and ballot vote. The documents are usually created or changed by a small sub-committee over the course of a few weeks or months. They are then submitted for review and ballot, which takes approximately two months. Almost all submitted documents are approved after review comments have been addressed.

OpenJAUS intends to sponsor enhancements to several existing JAUS standards and recommended practices. The technologies described in this paper span the JAUS transport, service, message, and protocol definitions. The OpenJAUS software lifecycle process will include a period of review and sponsorship of the most successful and widely used additions made to the implementation. Only after these new technologies have been developed, tested and proven to work as intended between multiple OpenJAUS users, they will be proposed to the AS-4 committee. The AS-4 committee will then have the opportunity to review and request changes as necessary. If no significant changes are requested by AS-4 then OpenJAUS will continue sponsorship of the enhancements through the balloting phase. However, if the committee requests significant changes, OpenJAUS will attempt to implement those changes into new software releases. Then after the changed software has been successfully vetted new enhancements will be re-proposed to the AS-4 committee.

In the past, the JAUS standard documents have led the development of software implementations. This results in designs that appear to be well-structured on paper, but in fact are difficult to elegantly transition to code. The drive of OpenJAUS is to let software implementation lead the standardization process. This will help ensure implementations are realizable, reduce errors in documentation, and greatly reduce the lifecycle cost of JAUS systems.

CONCLUSIONS

There is a clear trend that shows an increasing interest and rising customer demand for JAUS. While the standard has been shown to support integration and interoperability, implementation design remains challenging, costly and largely unguided by documentation.

OpenJAUS seeks to aid the use of JAUS by providing a simplified, well-planned implementation. The implementation design is statistically proven to be in line with community preferences for tools (Eclipse, Visual Studio) and programming language (C/C++, Java). It is also designed to meet the needs and requests of unmanned systems developers. This means that OpenJAUS extends the present scope of JAUS and offers simple mechanisms for better timing, data compression, system configuration, and more.

The newest version of OpenJAUS will be released for beta testing in the fall of 2010. This will be a closed beta test. Interested parties will be asked to fill out an application, and the OpenJAUS team will select a few organizations to work with based on their reported needs. A commercial version of OpenJAUS is anticipated for wide release in early 2011. The latest information on these topics can be found online at www.openjaus.com.

ACKNOWLEDGMENTS

The OpenJAUS team would like to thank the National Robotics Engineering Center at Carnegie Mellon University for their kind support of this paper.

REFERENCES

[1] AS5684 JAUS Service Interface Definition Language *http://standards.sae.org/as5684a*

[2] AS5710 JAUS Core Service Set *http://standards.sae.org/as5710*

[3] AS6009 JAUS Mobility Service Set *http://standards.sae.org/as6009*

[4] AS5669 JAUS / SDP Transport Specification *http://standards.sae.org/as5669a*

[5] Erl, Thomas. Service-Oriented Architecture: Concepts, Technology, and Design. Upper Saddle River, NJ: Prentice Hall, 2005.

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

[7] Budinsky, Frank., et al. Eclipse Modeling Framework. Addison-Wesley Professional, 2003

[8] Cuadrado, JS., Molina, JG. "Building Domain-Specific Languages for Model-Driven Development." IEEE Software, 2007.