A translation architecture for the Joint Architecture for Unmanned Systems (JAUS)

Scott Cutler*[a]

[a]SPAWAR Systems Center Pacific, 53560 Hull St., San Diego, CA, USA 92152

ABSTRACT

JAUS is an open architecture designed to support interoperability between unmanned vehicles, payloads and controllers.[1] However, it competes against a plethora of other open architecture technologies and standards. In many cases, there is much to be gained by merging multiple open architecture components within a single system. One such case is with the Navy's Common Control System (CCS) architecture utilized by the Multi-robot Operator Control Unit version 4 (MOCU4). CCS has primarily focused on Group 3-5 aircraft whereas MOCU4 is focused on ground and maritime unmanned vehicles. To utilize both of these architectures within a single system, a translation architecture called the SAE JAUS Vehicle Interface Service (VIS) has been designed and implemented by the MOCU4 team at the Space and Naval Warfare Systems (SPAWAR) Center Pacific (SSC Pacific). This paper will explore the design considerations and decisions of this VIS, as well as provide details of its implementation. It will also describe briefly how the VIS has been developed and utilized for the following projects: Navy Common Control System (CCS) integration with Large Training Vehicle (LTV), Control Station Human Machine Interface (CaSHMI), and the Universal Tactical Controller (UTC) for the Common Robotic System - Individual (CRS(I)).

Keywords: JAUS, open architecture, MOCU4, CaSHMI, CRS(I), robot controller, UCS-MDE, UCS

1. INTRODUCTION

The author is a lead developer on the MOCU4 team at SSC Pacific. MOCU4 is a government-owned, versatile suite of robot controller software utilizing many modern software technologies and standards such as Object Management Group's (OMG) Data Distribution Service (DDS), UAS Control Segment Multi-domain Extension (UCS-MDE), and modular UI design utilizing HTML5 and Angular UI directives. However, the focus of this paper is to discuss how two major unmanned systems architectures (JAUS and UCS) are brought into parity through the architecture of one service within MOCU4: the SAE JAUS VIS.

1.1 SAE JAUS and IOP

At present, the Society of Automotive Engineers (SAE) maintains the Joint Architecture for Unmanned Systems (JAUS). This architecture has been developed since 1995[2] and has become an international standard.[3] It is described in great depth in a series of documents that detail the JAUS transport specification, the JAUS data model and the details of the standard JAUS service sets (including mobility, manipulator, environment sensing, etc).[4] These documents amount to many hundreds of pages and are available from SAE.

The JAUS architecture specifies how a complete system of robots, controllers, and other devices may interact, including both the message transport details and the data model. To illustrate, a JAUS system might consist of two robots and an operator control unit (OCU); each of these is considered a JAUS subsystem. Each robot may have one or more computers onboard that handle the processing and control of the hardware available, such as the drive motors or manipulator arm; each of these computers is a JAUS node. On each node, there are one or more JAUS components, which are essentially groupings of related services that may only be controlled by a single operator at a time. For example, there may be a mobility component that contains all drive services (regular throttle/steering, vector driving,

*scott.h.cutler@navy.mil

waypoint driving, etc.) This single component may be controlled by one and only one operator at a time, while another component on the same node or subsystem, such as a manipulator arm component, could be controlled by a separate operator. The OCU is also considered a JAUS subsystem, and the computer it runs on is considered a node. It may also implement some components and services, but its primary purpose is to convert user input to actionable JAUS command messages and send them to the robots.

The current JAUS standard documents define over 60 services which together include over 200 JAUS messages. Each field of a JAUS message has a specified name, primitive type, units, optionality, and interpretation that typically includes a specified range or enumeration of values, as well as a description of the data. A typical JAUS message is only a few bytes on the wire. Auto-generated code to consistently encode and access the data contained in each message can be produced by the open source JAUS Toolset (JTS).[5] JTS also: autogenerates component and service code that includes finite state machines for the pre-defined JAUS service behaviors, allows for the development of custom JAUS messages and services, and can also auto-generate documentation for these custom messages and services. It also includes the JAUS software framework and reference implementation, including the ubiquitous node manager that performs the actual message routing within a JAUS system.[6]

In recent years, the U.S. Army has produced unmanned ground vehicle (UGV) Interoperability Profiles (IOP), often referred to as "IOP" in the UGV community.[7] IOP provides an additional set of specifications for unmanned systems, including hardware connectivity and additional JAUS services and messages.[8] These are detailed in a set of large documents that are available from the National Advanced Mobility Consortium (NAMC). IOP is essentially a super set of SAE JAUS; it specifies the use of nearly all the standard JAUS services and messages and it also adds 43 additional custom JAUS services and over 200 more JAUS messages to increase the interoperability of conforming IOP systems to include capabilities not included in the JAUS standard. It also adds interpretation detail and additional default values to the standard JAUS services.

1.2 UCS-MDE over DDS

Data Distribution Service (DDS) is a standard provided by the Object Management Group (OMG) that provides data-centric connectivity for networked services.[9] This standard provides a very robust set of quality of service (QoS) configuration capabilities, but those will not be discussed here. DDS utilizes topics, which are essentially a related set of data fields not unlike a database table. The various participants of a DDS bus publish and subscribe to these topics and DDS handles the underlying details of keeping the data synchronized between all participants and notifying subscribers when data has been published.

DDS in an excellent medium for data sharing between multiple applications, but it does not define the data that is to be shared. The Unmanned Aircraft System (UAS) Control Segment (UCS) Multi-domain Extension (UCS-MDE) fulfills this need in MOCU4. UCS-MDE, as the name implies, is an extension of the UCS architecture.

UCS is a framework and data model that defines a set of open standard command and control (C2) interfaces for Unmanned Air Vehicles (UAVs) and their sensors and weapons. It uses a Platform Independent Model (PIM) that defines a standard set of services capturing common functionality required for a specific domain and provides semantic interoperability. A system implementation typically only consists of a subset of the defined services based on its specific requirements and an intermediate model then defines this subset. Finally, a Platform Specific Model (PSM) binds the services defined by the intermediate model to a particular middleware (such as DDS) and it is no longer platform independent.

UCS-MDE extends UCS by incorporating additional services needed by the ground, sea surface, and underwater unmanned system domains. The UCS core components were initially utilized to encourage eventual merging of UCS-MDE back into its parent architecture (UCS), which is currently in process.[10] In MOCU4, most of the UCS-MDE services are implemented and bound to the DDS middleware for transport.

1.3 MOCU4

Multi-robot Operator Control Unit 4 (MOCU4) is a suite of software under active development by the US Navy, Army, and industry. The MOCU4 architecture aligns with the Navy's CCS by integrating software components using

UCS-based UCS-MDE over a DDS bus in a service-oriented architecture. MOCU4 supports the planning, management, and control of small air, ground, and maritime programs in the Navy and Army.

A major component of MOCU4 is the user interface framework which makes use of modern web technologies to provide a highly configurable and quickly composable user interface. This Human Machine Interface (HMI) portion of MOCU4 is displayed at the top of Figure 1. A MOCU4 control system can include multiple MOCU clients to allow multiple operators a common picture and control capability. Each MOCU client interacts with other MOCU4 services via the DDS Weblink, which is a web server that is also a participant on the DDS bus. It allows the MOCU client to both subscribe to and publish UCSMDE topics, essentially providing a Javascript interface to the DDS bus.

MOCU4 communicates to unmanned vehicles through vehicle interface services (VIS) which translate the over-the-air communications protocol to the UCS-MDE standard; thus the primary capability of any MOCU4 VIS is to reside within the native interface architecture of a given unmanned vehicle system and simultaneously reside within the MOCU4 UCS-MDE architecture. The VIS publishes information about all discovered vehicles and subscribes to pertinent control topics on DDS so that operators may control those robots through a MOCU client. On the native interface side of a VIS, this same status and control information is communicated in the vehicle's native protocol. Each VIS can interact with multiple UxVs that use the same interface. At the same time, multiple VIS may be part of a single MOCU4 control system, allowing myriad robots of varying types and protocols to be controlled by one or more users.

MOCU4 also provides some core services, which include C2 interface services to integrate with combat systems, radio interface services to configure and control radios, and other management services. Figure 1 provides a simplified view of the MOCU4 architecture.

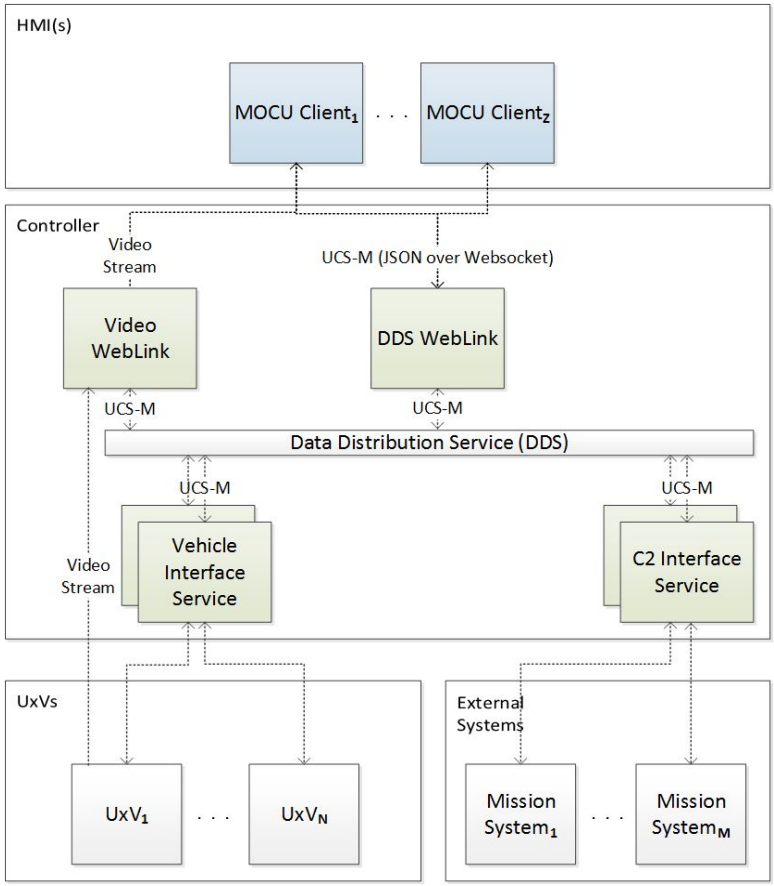


Figure 1. High-level MOCU4 Architecture.

2. SAE JAUS VIS

The subject VIS of this paper is the SAE JAUS VIS. This VIS was so named to differentiate it from earlier versions of JAUS which are not fully compatible with the current JAUS standards. In recent months, the capability to translate a large portion of IOP version 2 (IOPv2) services has been added as well, so the VIS may simultaneously be considered an IOPv2 VIS.

2.1 Design Considerations

2.1.1 Interacting with JAUS

Perhaps the fundamental design decision for this VIS was how to interact with the JAUS system. As previously described, the JAUS Toolset (JTS) automatically generates JAUS service code, and that code is then populated by a developer to interact with the underlying hardware on a robot. A JAUS operator control unit (OCU) would presumably utilize this same methodology. However, the JTS-generated service code is bulky and somewhat hard to follow. The actual developer code typically makes up only a minor part of the total code lines. This is not difficult to overcome when implementing a handful of services on a robot, but a controller that must control the majority of available JAUS services could get very unwieldy. A developer of such code would have to spend an extensive amount of time just navigating to the pertinent lines of code. In addition, the JTS-provided node manager is required to run on any JAUS node, so a JAUS OCU must run the service code as well as a node manager to actually communicate with the JAUS system.

The UCS-MDE data model is based largely on the JAUS data model, so there is perhaps 90% commonality. This means when a JAUS robot reports information such as robot name, latitude, longitude, speed, heading, etc., these data fields are easily copied from the JAUS report messages that contain them to the UCS-MDE topics that communicate this data. Because the mapping is so nearly one to one, it seemed there could be a simpler way to perform translation than to use the bulky JTS-generated service code.

2.1.2 JAUS Message Code

The JAUS message code is another factor in this design. This message code is also generated by JTS and contains classes that perform the functions of storing the full set of data for a given JAUS message and encoding/decoding it from the byte stream that is sent over the network. (A typical JAUS message is around 60-80 total bytes on the wire, with about 40 bytes containing the Ethernet, IPv4, and UDP headers.) The JTS-generated JAUS message code ensures that the data encoded and decoded from these bytes is consistent, as long as all JAUS entities on the network are using the same message code. When JTS generates code for a component, it generates code for each service as well as each message. Nearly all JAUS services inherit at least one or more other services, so any service will include its own unique JAUS messages as well as the messages from all the services it inherits from. This means that generated message code from certain core services that are almost universally inherited, such as the Event service, will be included in the list of files for each set of service code generated. The SAE JAUS VIS presently translates over 90 services, so if JTS-generated code had been used directly, it would include around 90 copies of each header and source file for the twelve JAUS messages utilized just by the Event service (that's over 2000 files!) If any changes were needed to one of these messages, all 90 instances would have to be changed, which is obviously not ideal. JTS-generated code is certainly useful to create a starting point for a JAUS-enabled robot, but it has some major inefficiencies that are greatly exacerbated as the needed set of JAUS services grows.

2.1.3 A Better Way - Expanding the Existing Node Manager

With these considerations in mind, the author and team determined there was a simpler path forward: build the VIS out of the existing node manager code. The JAUS messages already pass through the node manager, so it was a relatively simple matter to implement functions to inspect each message and translate those which can be translated to UCS-MDE. It was likewise simple to add code to inject JAUS messages back into the system. All messages in the node manager are handled generically as all JTS message code inherits from a message base class, so no JAUS message code is even required in the inspection part of the VIS. This does beg the question, however, where does the translation actually happen?

2.1.4 Modularity

The current SAE JAUS documents detail a handful of service sets, including Mobility, Environment Sensing, Manipulator, and UGV among others. JTS also allows for the creation of custom JAUS messages, which are often used by robots with capabilities not currently contained in the SAE JAUS data model. The VIS designers knew that the need to handle both standard and custom messages would be vital to the success and usability of the VIS, so a modular design was adopted. Each translator module would be loaded as a shared library at runtime. Each translator would register the messages it could translate with the JAUS message inspection code so the appropriate messages could be forwarded to the module. With over 200 existing standard JAUS messages and substantially more custom messages, creating a shared library for each message would be overkill and run into many of the same inefficiencies as the JTS-generated service code. Even creating a module for each of the 60 standard JAUS services was a bit too much. Instead, the modules were divided by JAUS service set. There are presently translator modules for Core, Environment Sensing, Manipulator, Mobility, and UGV. There are also multiple custom modules, one of which encapsulates many of the 43 custom JAUS services specified by the IOPv2 specification. This modularity provides a mechanism for custom capabilities to be added as needed, both by the MOCU4 team and by other interested organizations. It will also allow for support of the continuously incrementing versions of each JAUS service set (Core and Manipulator are already beyond their 1.0 versions, and a new version of Mobility is pending). Translator modules that are not needed or allowed for a particular system can be easily removed without adversely affecting the rest of the VIS or requiring any code to be changed. An additional efficiency is that the majority of the JAUS message code utilized by the services in a given module need only be compiled once with only a handful of common message classes being compiled by each module.

2.1.5 Potential Drawbacks

This design provides flexibility and modularity, but it does have some drawbacks. A custom JAUS message was once used to contain an entire image file, which resulted in the JAUS message being broken up into multiple messages on the wire (this is allowed in the JAUS transport specification and handled at the application layer). When this message was received by the VIS, it didn't contain the full image file, thus no image could be displayed. Extensive investigation revealed that the JAUS common code used by each service is able to reconstruct large messages, but the node manager has no need to do that because it just passes all messages along, whether they are fragments or not. JTS common code was used to add this capability to the VIS, but this problem would have been avoided if the VIS had been based on JTS service code.

JAUS services utilize a finite state machine to allow or disallow certain actions. This usually doesn't affect operation of the translation modules, but in some cases, such as when a JAUS component is being controlled, the workings of this state machine must be manipulated by the VIS. For example, the VIS must ensure a JAUS component being controlled has granted control and is in the ready state, otherwise commands sent will be ignored. Additionally, certain JAUS messages are required to maintain the liveliness of some functions, such as events, drive commands, and handoff requests. Also, it is possible for a MOCU client to publish too many control messages within a given amount of time. The VIS must throttle these messages down to a reasonable rate so as not to overload the more sensitive robots. These situations all require some state information to be managed in the VIS, but handling those few cases is far simpler than attempting to work with state machines for every single service.

2.2 Architecture

A JAUS system contains one or more subsystems (usually robots), which contain one or more nodes (computers), which contain one or more components, which contain one or more services. In a JAUS system with a typical IP network configuration, a node manager runs on each node because each node has its own network stack and IP address. The SAE JAUS VIS is also run on a computer that is networked with the other JAUS nodes in the system. In the JAUS system, the VIS appears as a single component on a single node on a single OCU subsystem. It initiates discovery of JAUS robots by sending regular identification queries. Figure 2 shows the encapsulation of the JAUS system as well as the boundaries of the SAE JAUS VIS.

The translation manager, shown attached to the node manager in Figure 2, is the glue code that passes specific JAUS messages to any translator module that registered for them. It also passes JAUS messages translated from UCS-MDE topics to the node manager for distribution to the greater JAUS system. On the right side of Figure 2, the translation modules are shown. The basic functionality of these modules was explained in the previous section, and further details

will be provided in the following section. Figure 2 simply illustrates that these modules "plug in" to the SAE JAUS VIS and when present, provide translation of JAUS messages to UCS-MDE topics and vice versa.

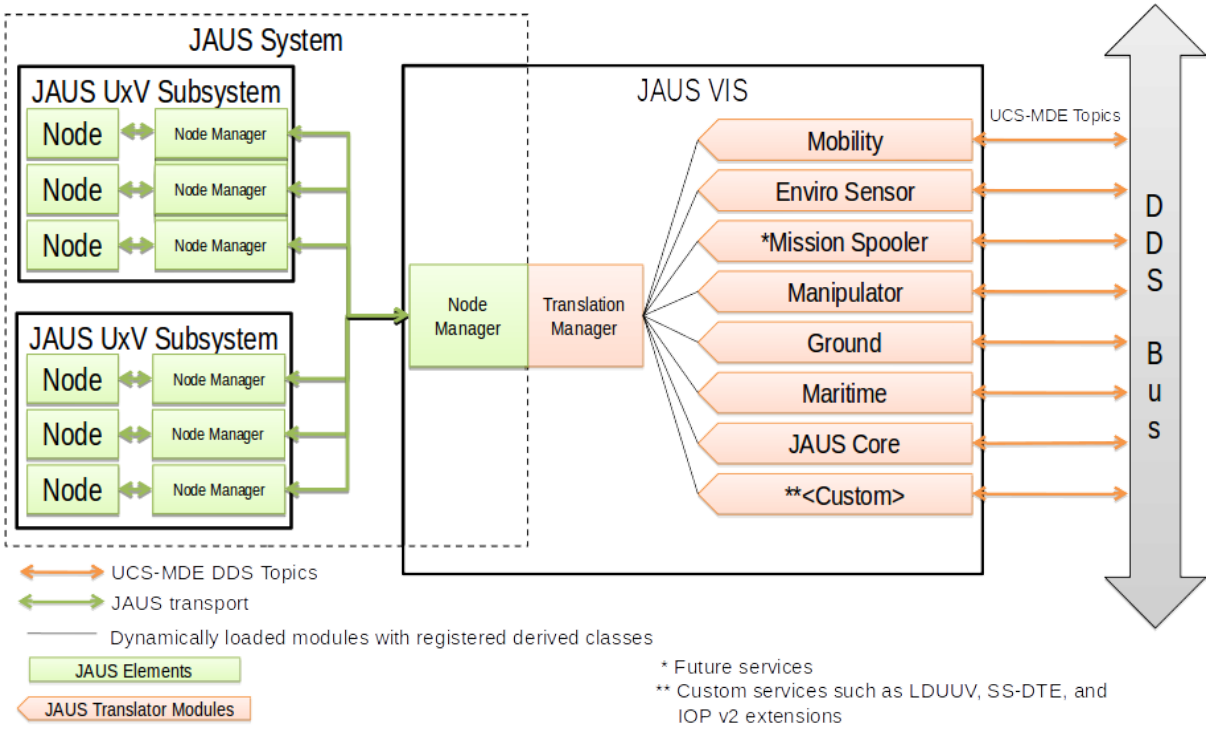


Figure 2. SAE JAUS VIS Architecture. The encapsulation of the JAUS system and SAE JAUS VIS are displayed.

2.3 Software Description

2.3.1 Translation Manager

As previously described, the SAE JAUS VIS is built on top of the JTS node manager, which is freely available along with the rest of JTS from the *jaustoolset.org* website.[5] Care was taken to include in the VIS only the header and source files from the JTS common directory that were strictly required to run the node manager. The JTS node manager is only available in C++, so the rest of the VIS code is also written in C++ to accommodate this. A translation manager class was added to the node manager code, which loads in translator modules using Linux or Windows-specific classes, depending on which OS it is running on. At present, shared modules are loaded only from the same directory as the executable. Even then, modules are only loaded that contain a specific function name and signature which indicates the shared library is an SAE JAUS VIS translation module. After a module is loaded, a module-specific translator class is created, which registers the JAUS messages it can translate. In addition, each translator class instance is also provided its own thread by the translation manager to execute any specific startup code or repeating code that is necessary. The translation manager also provides a callback function to each translator class so it may inject JAUS messages generated by the translator module into the JAUS system.

The main node manager loop executes in a function called RunJuniorRTE. A handful of code lines were added to that function in order to initialize the translation manager, extract messages for inspection by the translation manager, and inject new messages from the translation manager. When the translation manager finds a JAUS message that can be translated by one or more of the loaded modules, the message is passed to the module(s) via a registered callback function. Messages that have been passed back to the translation manager from the translator modules are queued up and sent as if they were messages received from another node that need to be properly routed.

2.3.2 Translator Classes

All translator classes inherit from a translator base class which provides two main capabilities: 1) reconstruct JAUS messages at the application layer that have been fragmented; 2) extract JAUS report messages embedded inside Event messages. Each module then has its own specific translator class which instantiates a robot class for each robot discovered that can send or receive JAUS messages translated by said module. Once a robot class is created, the translator calls functions on the specified robot class to handle each JAUS message. There are some exceptions to this, where the translator performs some basic functionality outside of the robot classes. One such exception is the discovery code, wherein the Core translator class broadcasts an identification query to any vehicle or OCU on the JAUS system network. Another exception is the subsystem ID allocation service, which assigns each JAUS subsystem a system-unique ID. Other than those exceptions, however, the translator classes are typically very simple.

2.3.3 Robot Classes

Each module contains a module-specific robot class, which inherits from a base class as well. The robot base class contains all common code utilized by the module-specific robot classes, such as converting JAUS timestamps to milliseconds from epoch, or maintaining a period tick function that for those messages that require throttling or repeated sending and querying. Each module-specific robot class contains a static function that inspects the service list reported by a JAUS subsystem. If a robot supports any of the services translated by the given module, a robot instance is created for it in that module. The module-specific robot class then begins to send JAUS queries where applicable. Wherever possible, JAUS events are set up which prompt the robot to send reports at a specified periodic rate, rather than requiring a query for each report. This can reduce message traffic by close to 50% in some cases. With both queries and events, the robot is prompted to continuously send information updates from each of its services. These are then translated to UCS-MDE topics, published to the DDS bus, and consumed by any subscriber (usually the MOCU client at a minimum, and sometimes other services such as a control arbiter, data aggregator, image server, or interested external system).

In addition to publishing continuous status updates, the module-specific robot class subscribes to any UCS-MDE topics that can be translated by its module. UCS-MDE topic updates trigger a registered function callback. In most cases, the updated topic is promptly translated to a corresponding JAUS message and injected into the JAUS system. This is the basic way in which the MOCU client can control a given robot. Some commands can only be received at a specified rate by a robot (usually no more often than 10 Hz), so the robot class must throttle down these commands which may be received at 60-100 Hz from the MOCU client. The robot class also performs basic state maintenance, such as sending periodic control requests (JAUS robots often need several control requests per second to ensure there is no “dead man” control).

3. MOCU4 SAE JAUS VIS PROJECTS

MOCU and its various versions have been under development since 2001[11], with the overarching goal of providing a single controller for as many unmanned systems as possible, thereby escaping the typical “stovepipe” systems that are developed for the Department of Defense (DoD).[12] MOCU has been used over the years in many DoD research and development projects, including Advanced Concept Technology Demonstrations (ACTDs) and Joint Capability Technology Demonstrations (JCTDs). The latest version of MOCU, version 4, was being developed in earnest by 2015 and incorporated the UCS-MDE architecture over a DDS bus among other changes. With these architecture changes, MOCU4 can allow multiple operators to control multiple robots simultaneously, something that cannot be done with earlier MOCU versions.

Development of the SAE JAUS VIS began at a similar date as MOCU4 and enabled even more robots to be controlled by MOCU. Some of these projects include the Navy Common Control System (CCS) integration with an Large Displacement Unmanned Underwater Vehicle (LDUUV) surrogate called Large Training Vehicle (LTV) 38, the Control Station Human Machine Interface (CaSHMI), and the Universal Tactical Controller (UTC) for the Common Robotic System - Individual (CRS(I)).

3.1 CCS LTV

In December 2015, the Navy successfully tested in-water control via radio link with CCS of an LTV 38 (shown in Figure 3).[13] This was the first live demonstration of the Common Control System.[14] As previously stated, CCS utilizes the UCS architecture in a way similar to MOCU4, and contains some of the same capabilities. However, at that stage of development, CCS did not have a robust vehicle interface service that could communicate via JAUS with an LTV. The SAE JAUS VIS, as a standalone executable utilizing a custom translator module, provided that capability for this event. This testing demonstrated two major benefits of the VIS design: 1) speed of integration, and 2) ability to translate custom JAUS messages.

The SAE JAUS VIS had not yet existed for a year when this demonstration took place. The initial development efforts of the VIS were to provide translation for the basic services of the standard Mobility and Environment Sensing service sets. However, about 3 months before this live demonstration, development work commenced to provide translation for several custom JAUS messages developed by labs at Johns Hopkins and Penn State Universities. The fundamental modular design of the VIS made it easy to add these new capabilities without compromising any existing translation ability.

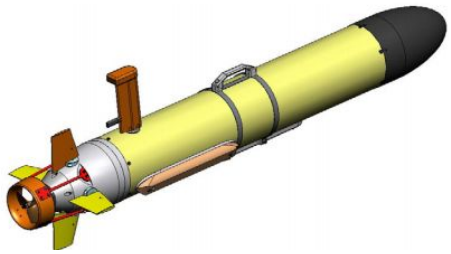
Figure 3. LTV 38 being lowered into water for control testing in Keyport, Washington.

3.2 CaSHMI

Starting in October 2015, the MOCU4 team began extensive development of the Control Station Human Machine Interface (CaSHMI) under the Office of Naval Research (ONR) Unmanned Aerial Systems Interface, Selection, and Training Technologies (UASISTT) Future Naval Capability (FNC). This interface was intended to provide a futuristic supervisory control capability with a single operator managing several UxVs simultaneously, but requirements soon changed. There was an immediate need to provide detailed control of the AeroVironment Blackwing UAV, which was to be flown to provide intelligence, surveillance, and reconnaissance (ISR). An additional capability was also identified: use the UAV to provide a radio relay to surfaced unmanned underwater vehicles (UUVs). This concept of cross-domain communication called for further utilization of the SAE JAUS VIS. In this case, the unmanned submarines communicated via JAUS messages, some of which were standard and some of which were custom. Along with communicating various environmental parameters such as ocean temperature and current flow, there was also a desire to transmit automatic target recognition (ATR) image files captured by the UUVs during underwater missions. Again the modular design of the JAUS VIS proved vital to success. While the UUVs used some standard JAUS messages, they did not perform any standard method of discovery. A custom module was developed which was able to discover these UUVs and control them.

Communication with these JAUS UUVs via relay from the UAV was demonstrated on three occasions: 1) Naval Undersea Warfare Center (NUWC) Advanced Naval Technology Exercise (ANTX) 2016 in Newport, Rhode Island, 2)

Unmanned Warrior 2016 in Kyle of Lochalsh, Scotland, and 3) Ship-to-Shore Maneuver Exploration and Experimentation Advanced Naval Technology Exercise (S2ME2 ANTX) 2017 at Camp Pendleton, California. JAUS messaging was increased at each exercise, and in the final exercise the custom translator module in the VIS not only translated several custom JAUS services, but it also established a separate network connection with the UDT protocol to reliably transmit the ATR image files.[15]



Figures 4 (left). Blackwing UAV in flight. Figure 5 (right). Drawing of Iver 2 UUV which was ran missions and communicated via JAUS during the ANTX and Unmanned Warrior exercises.

3.3 CRS(I) UTC

In 2016, MOCU4 was selected for use as the prototype control software for the Universal Tactical Controller (UTC) being developed for the Common Robot System - Individual (CRS(I)). The CRS(I) robot design calls for use of approximately 90 JAUS standard and custom IOP services (incorporating over 350 JAUS messages). The prototype development was performed in spirals, with the SAE JAUS VIS adding incremental capability for each.

After a successful demonstration at Ft. Benning, Georgia in August 2017, the Army's Product Manager Unmanned Ground Vehicles (PdM UGV) requested the SAE JAUS VIS be vastly expanded to translate all the messages required for CRS(I). Since that time, the VIS has gone from translating approximately 50 messages to translating now well over 350.

The CRS(I) project is planned to become a Defense Acquisition Program of Record (POR), with contract award planned for late March 2018. The intention is that the winning contractor(s) will be handed MOCU4 and particularly the SAE JAUS VIS as a robust starting point for a controller so that more focus can be placed on design of the robot hardware and internal software.

4. SAE JAUS VIS FUTURE

The SAE JAUS VIS development work for CRS(I) is still underway. When that work is complete (scheduled for late March 2018), the SAE JAUS VIS will be able to translate around 90% of JAUS standard services and approximately 79% of the IOPv2 services. The intent is to expand the VIS to translate IOPv2 in its entirety so that it may be used for other IOPv2 systems planned for fielding in the near future.

An upcoming Navy project will require the translation of a new draft JAUS standard for Unmanned Maritime Vehicles (UMV) as well as a pending update to the Mobility service set. Development work for those modules will begin shortly. Other projects, such as CRS(I)'s bigger brother Common Robotic System - Heavy (CRS(H)) have also expressed interest in using the SAE JAUS VIS and work will be ongoing. The JAUS standards will continue to be revised and updated, IOP will release version 3 later this year, and there is no theoretical end to the possible custom JAUS messages.

Ongoing SAE JAUS VIS efforts include refactoring and other improvements to make the code more accessible, understandable, and flexible. For example, a design factor being considered is to utilize Cython to access the required C++ code and then use the much simpler Python for the translation code. This would be a major rewrite, but could ultimately reduce development time over the long term.

5. CONCLUSION

The unification of functional unmanned systems standards and architectures is a continuously progressing theme among DoD research and development and has lead to the implementation of this open and extensible software architecture that will support current and future unmanned systems programs. The SAE JAUS VIS bridges two widely used standards: SAE UCS and SAE JAUS in a modular way that supports future extensions. The lessons learned from design, development, and demonstration of this capability will further the state of the art of unmanned systems controllers, and the general concepts may be successfully applied to other architectures.

REFERENCES

[1] Serrano, D. “Introduction to JAUS for Unmanned Systems Interoperability”, *sto.nato.int*, 2015, <https://www.sto.nato.int/publications/STO%20Educational%20Notes/STO-EN-SCI-271/EN-SCI-271-02.pdf>, (accessed 3 March 2018)

[2] Clark, M. N., “ JAUS Compliant Systems Offers Interoperability across Multiple and Diverse Robot Platforms”, *andrew.cmu.edu*, 2005, <http://www.andrew.cmu.edu/user/mnclark/(4)Clark.pdf>, (accessed 15 February 2018).

[3] Open JAUS, LLC, “Understanding JAUS”, *openjaus.com*, 2017, <http://openjaus.com/understanding-jaus/>, (accessed 15 February 2018)

[4] Various Authors, “JAUS”, *wikipedia.org*, 2017, <https://en.wikipedia.org/wiki/JAUS>, (accessed 15 February 2018).

[5] Neya Systems, LLC, “JTS: The JAUS Toolset”, *jaustoolset.org*, 2013, <http://jaustoolset.org/>, (accessed 15 February 2018).

[6] Authors Unknown, “JTS Users Guide Version 2.0”, *github.com*, 2013, <https://github.com/dvmartin999/jaustoolset/blob/master/GUI/doc/JTS_UsersGuide.pdf>, 16-17, (accessed 15 February 2018).

[7] Various Authors, “UGV Interoperability Profile”, *wikipedia.org*, 2017, <https://en.wikipedia.org/wiki/UGV_Interoperability_Profile>, (accessed 15 February 2018).

[8] Mazzara, M., "UGV Interoperability Profiles (IOPs) Update for GVSETS”, *esd.org,* 2014, <http://ww2.esd.org/gvsets/pdf/ags/1500mazzara_skalny.pdf>, (accessed 15 February 2018).

[9] Object Management Group, “DDS: The Proven Data Connectivity Standard for the IoT”, *portals.omg.org*, 2017, <http://portals.omg.org/dds/>, (accessed 13 February 2018).

[10] Ernst, R. “UCS Architecture Overview”, *ndiastorage.blob.core.usgovcloudapi.net*, 2016, <https://ndiastorage.blob.core.usgovcloudapi.net/ndia/2016/GRCCE/Ernst.pdf>, (accessed 15 February 2018)

[11] Powell, D., Gilbreath, G., Bruch, M., "Multi-robot operator control unit", Proc. SPIE 6230, Unmanned Systems Technology VIII, 62301N (12 May 2006).

[12] Powell, D., Barbour, D., Gilbreath, G., “Evolution of a common controller”, *dtic.mil*, 2012, <http://www.dtic.mil/get-tr-doc/pdf?AD=ADA563958>, (accessed 5 March 2018).

[13] Eckstein, M., “Navy Successfully Tests Common Control System On Unmanned Underwater Vehicle,” *news.usni.org,* 2016, <https://news.usni.org/2016/01/29/navy-successfully-tests-common-control-system-on-unmanned-underwater-vehicle>, (accessed 23 February 2018).

[14] Author Unknown., “Navy’s UxS Common Control System completes first live demonstration,” *navair.navy.mil,* 2016, <http://www.navair.navy.mil/index.cfm?fuseaction=home.PrintNewsStory&id=6171>, (accessed 23 February 2018).

[15] Cutler, S., "A search for the optimal file transfer protocol from surfaced UUVs to UAV relays and beyond", pending SPIE publication, Unmanned Systems Technology VIII, (21 March 2018).