Transformation Capabilities in Configurable Common Services

A Lesson Learned from CMS Development

Army FACE™ TIM Paper by:

Christopher J. Edwards, Systems Engineering Lead, CMS Team

Steven P. Price, Software Engineer, CMS Team/FACE TSS SC Lead

William G. Tanner, Data Modeler, CMS Team

September 2018

Distribution Statement A - Approved for Public Release
- Distribution Unlimited (control number PR 4053)

Table of Contents

Executive Summary 4

A Configurable Generic Core System 5

The Status and Menu Messages 7

A Hosted Capability 8

Useful Aspects of the FACE Technical Standard 10

Transformation Capability 10

TSS Abstraction 10

UoC Specific Abstractions with Transformation Capability 11

Conclusion 14

References 15

About the Author(s) 16

About The Open Group FACE™ Consortium 17

About The Open Group 17

Executive Summary

The development of the Crew Mission Station (CMS), and subsequent realization of the Rapid Integration Framework (RIF), feature a core system of fully-configurable software components that would host capability software. To reduce the need to recompile core system components, the core system provides functionality to hosted capabilities using generalized messages rather than specific messages addressing the true nature of the hosted capability. Through these generic messages, the specifics of the hosted capabilities are abstracted away so the core system software components do not need to change when hosted capabilities change, or new ones are added.

Currently, the data model that describes these generic messages is based on concepts from the perspective of the configurable core software components. For example, to a core software component, hosted capability status is just a number to display on a screen or to compare to a value obtained from a configuration file in order to perform the correct functionality. Likewise, to a configurable software component, a data event sent to a specific hosted capability is represented simply as a numeric ID.

However for the hosted capability every status and each piece of data has specific meaning. Each one is likely defined by a concrete enumeration that should be described in the data model. The current use of these generic messages by hosted capability is achievable only because the hosted capability UoC is developed with the foreknowledge of the message context used in the generic core system UoC.

A pure interpretation of the Future Airborne Capability Environment ™ (FACE) Technical Standard and its intent would have the incoming and outgoing message data for each UoC be based on the nature and subject matter of the UoC, not on those of the UoC that is the source or destination of the message. Doing so, however, leads to a contextual (or semantic) mismatch between messages: the data of the message being sent from the core software UoC is not the same as the data of the message being received by the hosted software UoC, and vice-versa. Resolving this mismatch calls for a Transport Service that was not readily available within the development timeframe of the original CMS project.

As CMS and the Rapid Integration Framework begin to incorporate changes for transitioning to FACE Technical Standard, Edition 3.0, the use of the TSS Transformation Capability can provide a better representation of the hosted capabilities in the data model.

A Configurable Generic Core System

The Crew Mission Station (CMS) was developed as a means to deliver new capabilities to the field as quickly as possible. The Rapid Integration Framework (RIF) extends these concepts and applies them to a broader range of computing systems known as Rapid Integration Platforms (RIPs). This rapid integration concept led to the development of a Core System and portable Hosted Capabilities. In order to improve speed to field, the Core System is constructed with generic, configurable software components, called Core Software Capabilities, which can adapt to changing Hosted Capabilities.

The Menu System

One of the Core Software Capabilities is the Menu System. The Menu System provides a user interface for selection of functionality provided by the Hosted Capabilities. It provides the interface between bezel inputs and specific functionality, allowing inputs to be displayed based on the current state of the Hosted Capability. By abstracting this function from the Hosted Capabilities themselves, the Core System can provide controls best suited for the hardware platform; and provide a common look-and-feel abstracted from the specific Hosted Capability. To provide this user interface the Menu System needs to know the status of the Hosted Capability and have a way of sending commands the capability understands. The Menu System and its interfaces were presented in "A Common Command Interface for Interactive FACE Units of Conformance (UoC)" presented at the Air Force FACE TIM in February of 2017.

As an example, consider a stopwatch capability that has three functions: start, stop, and reset. These functions might be supported by three separate buttons, or by two. The single Start/Stop button could act as a Start button when the stopwatch capability is not running, or as a Stop button when it is. The Menu System needs to know if the stopwatch capability is currently running in order to properly display the button state and subsequently send the correct command to the capability. The stopwatch capability states might include "idle", "running" and "have results". The Start/Stop button name would depend on the "idle" and "running" states; the enabled/disabled state of a "reset" button could depend on the "have results" state.

To provide maximum configurability without recompiling the Menu System, these state values must be abstracted as simple numerical values. The stopwatch capability sends a message with the values of all three states where index 0 contains the value of the "idle" state, index 1 contains the value of the "running" state, and index 2 contains the value of the "have results" state. To the Menu System, these are interpreted in software as simply State Message [0], State Message [1], and State Message [2].

Configuration files for the Menu System indicate which specific Hosted Capability state values activate or deactivate menu features. The configuration file includes the names of these states, but the software abstracts it down to just the numeric ID.

Similarly, when the Menu System sends a command to a Hosted Capability, it simply sends a number indicating the command ID. Once again, the Menu System treats this as a generic ID from a configuration file. The Menu System source code has no enumerations associated with each Hosted Capability.

In our stopwatch example, there are three possible commands: start, stop, and reset. The Menu System sends command ID 0, 1, or 2 depending on the state of the buttons and which one was pressed; the stopwatch would include a specific enumeration for the commands. Once again, the enumeration in the stopwatch code would be reflected in the Menu System configuration file using a name, although the Menu System relies solely on the command ID.

The System Status Capability

Another core capability in the CMS is the System Status Capability. This capability keeps track of the status of each capability installed on the CMS, including all core and hosted capabilities. It also provides a means to display the current state and status of each capability. The display of current application states can greatly improve the ability to debug issues that can arise in the field. It is for this reason that the states sent to the Menu System should also available for display in the System Status Capability.

The combination of system status and application status into a single message simplifies some of the modeling and the communications in the transport, reducing the number of sends required from each application.

The data added to the application status includes a heartbeat counter, built-in-test (BIT) results, and overall specific status messages that may not be needed by the Menu System, but provides useful information to the System Status Capability.

Equipment Status

This same message used by applications can be used by other equipment within the CMS system. The Equipment ID designates what equipment the BIT/Status data relates to. This extension also allows the status of capabilities running on multiple displays within the CMS to designate both their application id and the equipment id.

The result is a combined equipment status, application status, and menu state message that can be sent across a common TSS service to multiple destinations. This combined message has multiple uses within the core system, all of which are factored into the data model for this view.

The Status and Menu Messages

Currently, the data model for the messages sent between the Menu System and the Hosted Capabilities specify state and command ID values as generic numbers, matching only the perspective of the Menu System and the System Status Capability.

Within the CMS, a generic status message is sent from the Hosted Capabilities. Both the Menu System and System Status applications read this message. This message is called the AC_Equipment_Status and is depicted to the right.

This message includes an array of 256 possible status bytes, and treats each as a simple number with a range of values to be compared to a value in a configuration file.

```
namespace FACE
{
    namespace DM
    {
        struct AC_Equipment_Status
        {
            FACE::DM::tEquipmentID      equipment_id;
            FACE::DM::tApplicationID    application_id;
            FACE::DM::tHeartBeatCounter counter;
            FACE::DM::tNumStatusBytes   num_status_bytes;
            FACE::DM::tStatusByte       byte[256];
        };
    }
}
```

```
namespace FACE
{
  namespace DM
  {
    struct AC_MenuCommand
    {
            FACE::DM::tApplicationID application_id;
            FACE::DM::tCommandID command_id;
    };
  }
}
```

When the Menu System receives user input that relates to Hosted Capability functionality, an AC_MenuCommand message is sent as depicted to the left.

In this case, the Menu System is sending a number from its configuration file that is simply a command ID.

Other fields in these messages are there to support the System Status Capability, which tracks health of the applications and can display current states to the user. All of these fields fall into the category of implementation specifics for the CMS Rapid Integration Platform. Ideally, the Hosted Capability would be developed without knowledge of these fields.

In these messages, the application_id and equipment_id fields serve as source/destination identifiers. These fields could have been implemented in many different ways; however, they were implemented in the actual message due to 1) a need for the System Status application and the Menu System to know the source of the data and 2) the simplicity of the original CMS TSS component and its use of common transport connections for these messages. A more robust TSS component could hide these fields from the Hosted Capability entirely, burying the knowledge in the TSS library linked into each application.

A Hosted Capability

One of the Hosted Capabilities developed for the CMS is a Fuel Calculator. One aspect of this capability is a burn rate calculation that can be started by the operator. After a set time the results are shown in a pop-up, which the user can clear when done.

Much like the stopwatch example, the Menu System is given the Fuel Calculator's state to determine how to display the Start and Ack buttons. The Menu System sends a command ID associated with either the Start button or the Ack button being pressed. From the perspective of the Menu System, these are generic IDs. From the perspective of the Fuel Calculation capability, these are specific values with specific enumerations.

```
Fuel Burn Rate Source:
    enum StatusByteEnum
    {
        . . .
        MENU_START = 14,
        MENU_STOP = 15,
        MENU_RESET = 16,
        MENU_SNAP_OPEN = 17,
        MENU_NOT_AVAILABLE = 18,
        MENU_COMPLETED = 19,
        MENU_POPUP_VISIBLE = 20
    };

    enum OverallStatusEnum
    {
        GO = 0,
        FAILED = 1,
        DEGRADED = 2,
        TEST = 3,
        INITIALIZATION = 4
    };


    enum MenuEnum
    {
        OFF = 0,
        ON = 1
    };
```

```
Menu System Configuration:
[UA_FuelCalc]
ApplicationID = 5
DataItemName1 = Burn_Rate_Start_Enabled
DataItemName2 = Burn_Rate_Stop_Enabled
DataItemName3 = Burn_Rate_Reset_Enabled
DataItemName4 = Burn_Rate_Snapshot_Open
DataItemName5 = Burn_Rate_Not_Available
DataItemName6 = Burn_Rate_Completed
DataItemName7 = Fuel_Popup_Visble
DataItemID1 = 14
DataItemID2 = 15
DataItemID3 = 16
DataItemID4 = 17
DataItemID5 = 18
DataItemID6 = 19
DataItemID7 = 20
NumberOfDataItems = 7
CommandName1 = Burn_Rate_Start
CommandName2 = Burn_Rate_Stop
CommandName3 = Burn_Rate_Reset
CommandName4 = Burn_Rate_Take_Snapshot
CommandName5 = Burn_Rate_Close_Snapshot
CommandName6 = Fuel_Popup_Ack
CommandID1 = 1
CommandID2 = 2
CommandID3 = 3
CommandID4 = 4
CommandID5 = 5
CommandID6 = 6
NumberOfCommands = 6
```

```
Fuel Burn Rate Source:
    enum MenuCommandEnum
    {
        START = 1,
        STOP = 2,
        RESET = 3,
        TAKE_SNAPSHOT = 4,
        CLOSE_SNAPSHOT = 5,
        ACK_POPUP = 6
    };
```

The current CMS software uses the data model for the generic Menu System when building the Fuel Calculation UoC.

This is a legitimate interpretation of the messages sent to and from the Fuel Calculation UoC, and it provides portability to any system in the RIF that uses similar Menu System messages. Yet, the Fuel Calculation UoC must translate its concepts to those generic ones specified in the Menu System data model.

This approach does not promote portability outside of the RIF and does not support the true intent of the data model, as it hides the enumerations from the data model for the UoC. Ideally, each status for a UoC should be data modeled to include an enumeration for each status. A separate enumeration of Menu Commands would

also be modeled. These items would be realized into their own platform data types, which would lead to a message composed of individual elements, rather than the array of generic status bytes used by the Menu System.

If we were to model the data from the perspective of the Hosted Capability, without thought to the integration with the Core System, we would likely model each of these items separately and include the enumerations.

```
BURN RATE APPLICATION STATE MODEL
namespace FACE
{
    namespace DM
    {
        enum tTimerState
        {
            STOPPED,
            RUNNING
        };
        struct AC_FuelTimerState
        {
            FACE::DM::tTimerState state;
        };
    }
}
```

```
BURN RATE APPLICATION STATUS MODEL
namespace FACE
{
    namespace DM
    {
        enum tOverallStatus
        {
            GO,
            FAILED,
            DEGRADED,
            TEST,
            INITIALIZATION
        };
        struct AC_FuelApplicationStatus
        {
            FACE::DM::tOverallStatus  app_status;
        };
    }
}
```

```
BURN RATE APPLICATION COMMAND MODEL
namespace FACE
{
    namespace DM
    {
        enum tFuelBurnRateCommand
        {
            START,
            STOP,
            RESET,
            TAKE_SNAPSHOT,
            CLOSE_SNAPSHOT,
            ACK_POPUP
        };
        struct AC_FuelBurnRateCommand
        {
            FACE::DM::tFuelBurnRateCommand cmd;
        };
    }
}
```

If we model the UoC as a portable component to the intent of the Data Model, the component would not be sending or receiving the same message that the Menu System must be constructed to.

If the Hosted Capability had been developed without the CMS Core system in mind, integration with the CMS Core would require writing translation logic.

Fortunately, the authors of the FACE Technical Standard accounted for this.

Useful Aspects of the FACE Technical Standard

There are some aspects of the TSS in the FACE Technical Standard that provide ways of dealing with this mismatch. As CMS and the Rapid Integration Framework move to Technical Standard, Edition 3.0, a refactoring of the TSS interface can take advantage of these elements to improve software efficiency while improving the Data Model documentation of what the Hosted Capabilities are actually doing.

Transformation Capability

To combine the various modeled statuses with application IDs and a status counter into a single message the way the Menu System expects it, we would use the Data Transformation Capability described in the Transport Services Segment requirements of the FACE Technical Standard. Using the Data Transformation Capability, a TSS component can take individual status enumerations and place them at the appropriate location in the generic array of status IDs used by the Menu System.

TSS Abstraction

The Type Abstraction interface was added into the FACE Technical Standard, Edition 2.1 to allow a TSS to pass conformance and allow the addition of Type-Specific interfaces after conformance testing of the entire TSS component was completed. Just as in the case of the Menu System, there is a desire to create configurable generic code as part of the Core System.

UoC Specific Abstractions with Transformation Capability

When a Type Abstraction Capability is used, it is the only place in the TSS component that is developed specifically to the message structure. If we put our data marshalling in this layer, we can write integration specific marshalling without modification of the PCS/PSSS components or the core TSS components.

There is a desire for auto-generation or configuration driven TSS component implementations to avoid the same kind of compile-and-certify-once development to which the rest of the Rapid Integration Framework core system is built. Using the Type Abstraction, the core TSS can be developed once, and use the generic interfaces with configuration files.

The Data Marshalling code requires specific knowledge of each structure and is much harder to auto-generate or configure. For simplicity, the Type Abstraction Capability would be written by the system integrator, with knowledge of how the various UoCs in the system must communicate with each other in order to achieve the desired system functionality.

In this model the Type Abstraction Capability is written specific to each UoC and linked directly to the UoC code to produce the final application/partition binary. Because this linking creates a single binary, evaluation of performance and airworthiness qualification should treat the combined UoC and its TSS component (library) as a single entity.

With this tailored abstraction, we can move the knowledge of Equipment ID and Application ID into the TSS, removing this system specific data from the Hosted Capability UoC. We can also move the knowledge of the combined "application status" into the TSS component. The Hosted Capability is then coded only with knowledge of the data needed for its processing.

The Type Abstraction Capability would be developed with individual message interfaces for each status and command enumeration. When each status is "sent" by the Hosted Capability, the individual elements of the entire Status Message is constructed. One flag within the Type Abstraction Capability can keep track of the need to send the message due to a change, and another flag can track if the message should be sent due to watchdog timing.

An added advantage of this Type Abstraction logic is that it can be built to the specific capabilities of the selected TSS component. If the TSS component provides filtering and history features, the Data Marshalling code can take advantage of it. If not, the Data Marshalling code can add something to assist.

Generation of TSS Abstractions

The FACE Technical Standard, Edition 3.0 includes modeling of transforms and implies that a transform can be generated directly from an integration model. While code generation is not required, investment in code generation tools would greatly speed up integration efforts.

Although the FACE Technical Standard does not explicitly require the use of the Integration Model (i.e., no "shall" statements are used), Section 3.8.6: "Transport Services Segment FACE Data Architecture Requirements" states that a TSS component can use the Integration Model to build configuration data for message associations and/or data transformations.

The Integration Model defines a set of "transport node" types that can be used to document the manipulation of message data sent across the TSS component. One of these transport node types is called a View Aggregation and consists of two or more input Views and a single output View. The implementation of a View Aggregation transforms the message data from the structures specified by the input Views to the structure specified by the output View. Without getting into all of the details required for an Integration Model, the following figure illustrates this concept using a generalized subset of the graphical notation used by the Data Architecture Working Group:

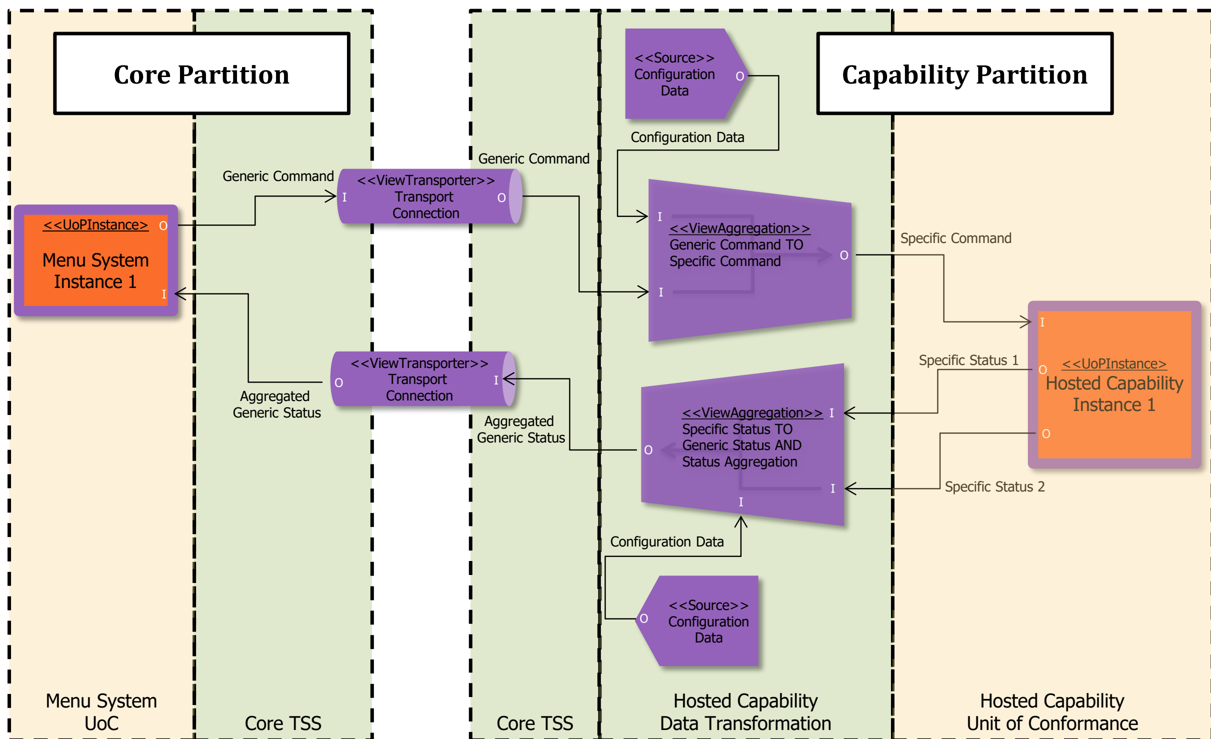

Figure 1- Data Transformation in the Hosted Capability Type Abstraction

Transformation Capabilities in Configurable Common Services

The example shows a) a Generic Command being sent from an instance of the Menu System UoC that is transformed using View Aggregation into a Specific Command being received by the Hosted Capability UoC instance, and b) two Specific Statuses being sent from two instances of the Hosted Capability UoC that are aggregated and transformed using View Aggregation into a single Aggregated Generic Status being received by the Menu System UoC instance. Generic Command, Specific Command, Specific Status 1/2, Aggregated Generic Status, and Configuration Data all represent Platform Views that define the structure of the data used to support communication between Menu System and Host Capability UoC instances.

Using code generation techniques, a TSS Type Abstraction Capability can be generated with the combined information from the:

- Integration Model
- UoC Supplied Models (USMs) for Menu System
- UoC Supplied Models (USMs) for Hosted Capability, and
- Configuration data definition

This code generation could be simple, generating a prototype for the transformation that can later be filled in by the system integrator to effect the exact implementation for message data mappings from generic to specific data, and vice-versa. Or it could be complex, generating the entirety of the transform – if enough information exists in the various models – to derive the transformation logic.

Conclusion

Use of the Transformation Capability in the TSS Type Abstraction allows:

- UoC code to be developed and modeled to the data used within the UoC, not the target system. This meets the intent of documenting the interfaces in the Data Model for the portable software, thus improving the portability and integratability of the UoC.
- System integrators to have control over the way data flows through the resulting system, thus allowing for transforms to happen at the most efficient point for the specific integration.
- The potential to generate software for the transforms to be written for the specific TSS implementation, potentially improving efficiencies and reducing errors.

Future modifications to Rapid Integration Platforms should adopt these methods as they move to conformance to the FACE Technical Standard 3.0.

References

Technical Standard for Future Airborne Capability Environment (FACE), Edition 2.1, 24 Jun 2014

Technical Standard for Future Airborne Capability Environment (FACE), Edition 3.0, December 2017

"A Common Command Interface for Interactive FACE Units of Conformance (UoC)", Air Force TIM published by The Open Group, February of 2017.

"The Impact of the FACE Technical Standard on Achieving the Crew Mission Station (CMS) Objectives", NAVAIR TIM Paper published by The Open Group, October 2017

About the Author(s)

Christopher J. Edwards has been working in the avionics industry for over 20 years, primarily on cockpit systems for military aircraft. In those years, he has served in leadership roles in Software, Requirements, System Design, PVI development, Qualification Testing, and Project Management. Mr. Edwards has been the primary author of the FACE Conformance Certification Guide and the Problem Report/Change Request (PR/CR) Process and a contributor to several other documents in both the Technical Working Group (TWG) and Business Working Group (BWG). Mr. Edwards currently serves as a co-lead of the FACE TWG Conformance Verification Matrix Subcommittee, a co-lead on the FACE EA PR/CR Process, the facilitator of the FACE Verification Authority Community of Practice and is the Systems Engineering Lead for the CMS Project.

Steven P. Price has been working in avionics and embedded software for 30 years. He has worked on several different graphic user interfaces including cockpit systems. He has been a leader in the design and implementation of some of these systems, along with being involved with the testing of some of these systems. Currently Mr. Price is one of the Software Engineers for CMS, and the principal developer of the CMS Menu System. He is a FACE Verification Authority Subject Matter Expert (SME).

William G. Tanner has over 25 years of embedded software and embedded software application development experience. Bill is a contributing author to the FACE Data Architecture and actively serves in the FACE Data Architecture and Guidance Working Groups. He is currently the primary data modeler for several Army projects including Crew Management System (CMS), Synergistic Unmanned Manned Intelligent Teaming (SUMIT), Degraded Visual Environment (DVE), and has contributed significantly to the Joint Common Architecture (JCA) data model.

About The Open Group FACE™ Consortium

The Open Group Future Airborne Capability Environment (FACE) Consortium, was formed as a government and industry partnership to define an open avionics environment for all military airborne platform types. Today, it is an aviation-focused professional group made up of industry suppliers, customers, academia, and users. The FACE Consortium provides a vendor-neutral forum for industry and government to work together to develop and consolidate the open standards, best practices, guidance documents, and business strategy necessary for acquisition of affordable software systems that promote innovation and rapid integration of portable capabilities across global defense programs.

Further information on FACE Consortium is available at www.opengroup.org/face.

About The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through technology standards. Our diverse membership of more than 600 organizations includes customers, systems and solutions suppliers, tools vendors, integrators, academics, and consultants across multiple industries.

The Open Group aims to:

- Capture, understand, and address current and emerging requirements, and establish policies and share best practices
- Facilitate interoperability, develop consensus, and evolve and integrate specifications and open source technologies
- Offer a comprehensive set of services to enhance the operational efficiency of consortia
- Operate the industry's premier certification service

Further information on The Open Group is available at www.opengroup.org.