SpaceOps-2025, ID # 307

XTCE as a Unifying Model for Flight and Ground Software: Experience with the VIPER Mission

Mark Rose[a]*, Philip Cooksey[b], Matt Deans[c], Lorenzo Flückiger[d], Leigh Garbs[e], Brian Hamilton[f], Matt Knudson[g], Ethan Massey[h], Arno Rogg[i], Antoine Tardy[j], Hans Utz[k]

[a] *NASA Ames Research Center, mark.rose@nasa.gov*
[b] *NASA Ames Research Center – KBR Wyle Services, LLC, philip.cooksey@nasa.gov*
[c] *NASA Ames Research Center, matthew.deans@nasa.gov*
[d] *NASA Ames Research Center – KBR Wyle Services, LLC, lorenzo.fluckiger@nasa.gov*
[e] *NASA Ames Research Center, leigh.s.garbs@nasa.gov*
[f] *NASA Ames Research Center – KBR Wyle Services, LLC, brian.m.hamilton@nasa.gov*
[g] *NASA Ames Research Center, matt.knudson@gmail.com*
[h] *NASA Ames Research Center – KBR Wyle Services, LLC, ethan.o.massey@nasa.gov*
[i] *NASA Ames Research Center – KBR Wyle Services, LLC, arno.rogg@nasa.gov*
[j] *NASA Ames Research Center – KBR Wyle Services, LLC, antoine.tardy@nasa.gov*
[k] *NASA Ames Research Center – KBR Wyle Services, LLC, hans.utz@nasa.gov*
* Corresponding Author

Abstract

Spacecraft and ground systems need a common definition of telemetry and command formats to be able to communicate with each other. The complexity of modern missions makes a traditional document-based interface definition difficult, so a common modeling language is preferable. The NASA VIPER mission uses the XML Telemetric and Command Exchange (XTCE) standard[1] as this common definition for modeling telemetry and command formats, raw-to-engineering conversions, encodings, and algorithms.

While the use of XTCE as a unifying language for both flight and ground operations has been successful, there are areas where plain XTCE has been insufficient, requiring additional processing and information to support different testing and flight environments. In addition, differing needs of flight and ground systems have necessitated other workarounds to limitations in XTCE modeling. This paper will describe the use of XTCE within the VIPER mission, showing ways that XTCE mechanisms have proven useful as well as methods that have been used to overcome XTCE limitations. The conflict between the goals of mission software development and that of XTCE will be analyzed and possible mitigations suggested.

Keywords: XTCE, CCSDS, Telemetry & Command Dictionary, VIPER

Nomenclature

CCDD	Core Flight System Command and Data Dictionary utility
CCSDS	Consultative Committee for Space Data Systems
cFE	NASA Core Flight Executive, a component of the NASA Core Flight System
cFS	NASA Core Flight System, a C/C++ flight software framework
CLPS	NASA Commercial Lunar Payload Services
ITOS	NASA Integrated Test and Operations System, a telemetry and commanding system
VIPER	Volatiles Investigating Polar Exploration Rover
XSLT	XML Transformations, an XML processing language
XTCE	XML Telemetric and Command Exchange
Yamcs	Yamcs Mission Control, an open-source telemetry and commanding system

1. Introduction

The NASA Volatiles Investigating Polar Exploration Rover[2] (VIPER) mission is designed to explore the extreme environment of the Moon's polar regions in search of ice and other potential resources. VIPER will be launched as a payload on a Commercial Lunar Payload Services[3] (CLPS) mission as part of NASA's Artemis program. The flight

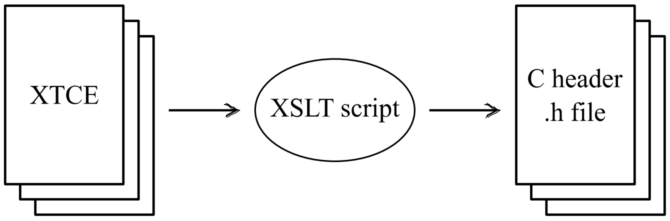

Figure 1: Generating C headers from XTCE

software for VIPER is developed by adapting NASA's Core Flight System[4] (cFS) framework, while the telemetry and command component of the Ground Data System (GDS) is an adaptation of the open-source Yamcs Mission Control system[5].

Early in the development of the flight and ground systems, the team decided to use XTCE as the common definition of telemetry and commanding formats for both the flight and ground software. This is a natural choice for the Yamcs system, since it is natively supported. The cFS-based flight software, on the other hand, relies on C/C++ header files to define telemetry and command formats using structure definitions. To generate those header files, the team created a processor in the XSL Transformations[6] (XSLT) language which reads the XTCE definitions and generates C header files (see figure 1). This technique was used not only for telemetry and commands which are VIPER-specific, but also for all native cFS telemetry and commands.

1.1 Comparison with other approaches

All missions need some form of command and data dictionary for interpreting telemetry and formatting commands. One popular telemetry and command system, the NASA Integrated Test and Operations System[7] (ITOS), available commercially as The Hammers Company *Galaxy*[8] system, allows for the definition of command and telemetry records, and will export those records to other formats, including C header files. Some NASA tools, including *System Configuration Information & Mission Interfaces*[9] (SCIMI) and the *Core Flight System (cFS) Command and Data Dictionary*[10] (CCDD) utility use a relational database to create an interactive system for defining the command and data dictionary, and support both direct- and script-based export to other formats, including XTCE and header files. And other missions have used XTCE as a definition language for data interchange between ground system components, but not as the primary mission database format[11].

VIPER uses XTCE as the primary mission database format. This has two main advantages and one disadvantage. First, the source format is textual, making it amenable to version control using the same system as is used for the software source. Second, the entire language defined by the XTCE specification (subject to any limitations imposed by the chosen telemetry and command system) can be used when defining telemetry and command formats, raw-to-engineering conversions, and algorithms. The main disadvantage is that there are no existing tools for the conversion of XTCE to C/C++ header source files or other outputs.

2. Material and methods

The VIPER team defines the command and telemetry formats into a collection of separate XML files using the XTCE schema. This includes base definitions of telemetry and command packets which follow the standards of the Consultative Committee for Space Data Systems[12] (CCSDS), as required by the cFS-based flight software and the communications systems of NASA's Deep Space Network[13].

The XTCE files form a hierarchical namespace of arbitrary depth for telemetry and command definitions based on the named *SpaceSystem* elements in the XTCE files. References within the XTCE or to the definitions are similar to UNIX pathnames, such as /ViperRover/Thermal/Housekeeping. Each XTCE file is used to create a corresponding C header file, using the name of the SpaceSystem (plus extension) as the header file name. Definitions within the XTCE that describe scalar type definitions (in <ParameterTypeSet> and <ArgumentTypeSet>) become either "enum" or "typedef" definitions in the generated headers. Telemetry and command packet definitions (<Container> and <CommandContainer>), and aggregate types for telemetry parameters or command arguments are converted to C struct definitions. The struct name is an abbreviated form of the XTCE path. (Putting some of the path into the struct name is used instead of C++ namespaces because compatibility with legacy cFS C code is required.)

2.1 XTCE preprocessing

It became clear very early that some mission requirements were not directly facilitated by XTCE. These include support for different endianness of test and flight hardware, and the repetition of common parameter values in many housekeeping packets. For this reason the team decided to preprocess the XTCE before using it for C header generation or within the telemetry and command system (see figure 2). This means the final XTCE files are not authored directly. Instead, template XTCE files are edited and versioned, which are converted to actual XTCE during the builds of the flight and ground software. This section will list those types of preprocessing.

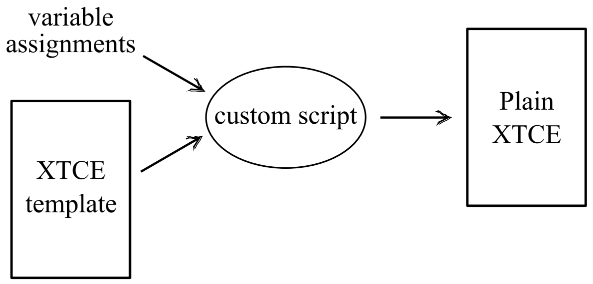

Figure 2: Generating XTCE from a template

2.1.1 Processor endianness

XTCE allows specifying the endianness of telemetry parameter and command argument types, but the cFS-based flight software does not. Instead, the native layout of the C struct defining the telemetry or command packet is used as the transmission format. In some VIPER test environments the flight software runs natively on little-endian processors, while in other test environments and in flight a big-endian processor is used or emulated. This requires specifying in the XTCE a varying endianness which will be filled in during the XTCE preprocessing. An example using an integer parameter type:

```
<IntegerParameterType name="SomeType">
  <IntegerDataEncoding sizeInBits="16" byteOrder="%PLATFORM_BYTE_ORDER%" encoding="unsigned"/>
</IntegerParameterType>
```

Variables within the templates are indicated with surrounding “%” signs, as can be seen above.

2.1.2 Common parameters

Various parameters are used the same way in different telemetry packets. It is convenient to use a parameter to refer to them as a group rather than repeating their definitions.

```
<SequenceContainer name="SomeHousekeeping">
  <EntryList>
    %VIPER_HOUSEKEEPING_FIELDS%
    <ParameterRefEntry parameterRef="anotherParameter"/>
...
</SequenceContainer>
```

A similar variable is used to insert the definitions of those parameters in the <ParameterSet> of the same XTCE file. (XTCE facilities such as base containers and container references do not help here, since each cFS application needs separate housekeeping parameters.)

2.1.3 Common definitions between telemetry and commands

XTCE does not have a good way of sharing types between telemetry parameters and commands. (The standard does not specify whether references to types outside of the current section are allowed, or what the semantics would be.) This is not a large problem except for certain enumerations which need to be shared and identical between the telemetry and command definitions. One example of this is the definition of Space Packet Application Process Identifiers[14] (APIDs). For this reason, a variable is used in the XTCE template to allow sharing of the same enumeration values between the two definitions. (Note: This differs slightly from the actual VIPER source.)

In the parameter type definitions:

```
<EnumeratedParameterType name="Apid">
  <IntegerDataEncoding sizeInBits="11" byteOrder="mostSignificantByteFirst" encoding="unsigned"/>
  <EnumerationList>%apidList%</EnumerationList>
</EnumeratedParameterType>
```

And in the command argument type definitions:

```
<EnumeratedArgumentType name="Apid">
  <IntegerDataEncoding sizeInBits="11" byteOrder="mostSignificantByteFirst" encoding="unsigned"/>
  <EnumerationList>%apidList%</EnumerationList>
</EnumeratedArgumentType>
```

2.1.4 Multiple processors using the same XTCE

There are two main processors in the VIPER rover that run the cFS-based flight software. Some of the cFS applications run on both processors, using the same code and message definitions. It would be cumbersome to duplicate the XTCE for both those application instances. Instead, the same XTCE template is used to generate two XTCE files for the telemetry and command system, with different names.

```
<SpaceSystem name="CfeEventServices%CPU%">
```

This is also needed for setting the arguments for some commands which can be routed to either CPU:

```
<ArgumentAssignment argumentName="apid" argumentValue="VIPER_%CPU%_EVS_CMD"/>
```

2.2 Augmenting the XTCE information

There are some instances either during header generation or during telemetry processing where additional information about the data is required. The XTCE facility <AncillaryDataSet> can be used to place additional information into the XTCE. Since in Yamcs the ancillary data is not available to client applications, in one case the XTCE *alias* feature is used instead.

2.2.1 Enumeration prefixes

In VIPER, various enumeration types for parameters and command arguments have the same enumeration labels. Since each enumeration type becomes a C enumeration, and since C enumeration constants all reside in a single namespace, those labels need to have a prefix added before generating the C enum type. For example, from a SpaceSystem called “Subsys”:

```
<EnumeratedParameterType name="FlagBit">
  <AncillaryDataSet>
    <AncillaryData name="VIPER_ENUMERATION_PREFIX">SUBSYS_</AncillaryData>
  </AncillaryDataSet>
  <IntegerDataEncoding sizeInBits="1" encoding="unsigned"/>
  <EnumerationList>
    <Enumeration value="0" label="LO"/>
    <Enumeration value="1" label="HI"/>
  </EnumerationList>
</EnumeratedParameterType>
```

```
enum SubsysFlagBit {
    SUBSYS_LO=0,
    SUBSYS_HI=1
};
```

2.2.2 Base telemetry rates

XTCE allows defining the expected rates of telemetry containers to facilitate reporting staleness in telemetry values.

```
<RateInStream streamRef="telemetry" basis="perSecond" minimumValue="0.9" maximumValue="1.1"/>
```

However, this is not sufficient for detecting staleness in telemetry originating from cFS-based flight software. cFS allows for the configuration of different *filter tables*, which filter out some fraction of packets from a single source. The filter table can be changed at any time, so a fixed rate defined in the XTCE is not sufficient for specifying the expected arrival rate, and therefore the staleness.

To handle this variability, the *base rate* of telemetry in packets per second is defined in VIPER using ancillary data, and the actual rate is calculated at runtime based on the fractional value in the filter table combined with the base rate.

```
<SequenceContainer name="Housekeeping">
  <AncillaryDataSet>
    <AncillaryData name="VIPER_BASE_RATE">10</AncillaryData>
  </AncillaryDataSet>
…
```

2.2.3 Non-XTCE type indications

The primary telemetry display system for VIPER is the open-source Open MCT framework[15]. In that framework some string parameter values may be URLs to images. To distinguish plain string parameters from URL parameters, VIPER uses an XTCE *alias*. This alias is available to client applications, like OpenMCT, when querying the mission command and telemetry database.

```
<Parameter parameterTypeRef="Url" name="Navcam_left_image">
  <AliasSet><Alias nameSpace="OpenMCT:type" alias="yamcs.image" /></AliasSet>
</Parameter>
```

2.2.4 C typing hints

In some instances it was convenient to override the XTCE definition of a data type in favor of another definition that was more suitable for coding in C or C++. For example, there were a few instances where telemetry values were more convenient for the flight system to treat as an array, but which were more convenient on the ground as separate, scalar parameters. This was accomplished by forcing the header file to use an array type using ancillary data. In the PoseEstimator SpaceSystem:

```
<AggregateParameterType name="AngleInfo">
  <AncillaryDataSet>
    <AncillaryData name="VIPER_ARRAY"/>
  </AncillaryDataSet>
  <MemberList>
    <Member typeRef="rotation_angle" name="yaw"/>
    <Member typeRef="rotation_angle" name="pitch"/>
    <Member typeRef="rotation_angle" name="roll"/>
  </MemberList>
</AggregateParameterType>
```

The C definition generated from this:

```
typedef rotation_angle PoseEstimatorYprAngles[3];
```

For the VIPER header generator it is an error if an array is requested but the aggregate members are not all of the same type.

2.2.5 C/C++ compiler structure packing

Another occasion where additional information was encoded into ancillary data was in cases where the C compiler would not pack struct members together in the same way that the telemetry and command system would interpret the XTCE. This could occur when telemetry did not have the natural alignment of numeric types on the target processor. In those cases ancillary data was added to the container definition which resulted in a compiler directive in the generated header file to force a packed structure layout, requiring unaligned access to structure members.

```
<Container name=”someTelemetryPacket”>
  <AncillaryDataSet>
    <AncillaryData name=”VIPER_ALIGNMENT”>1</AncillaryData>
  <AncillaryDataSet>
  <EntryList>…</EntryList>
</Container>
```

(This technique of forcing the structure alignment caused problems in some instances, so was used only occasionally in VIPER. Instead, dummy parameters were inserted into the telemetry or command containers to ensure proper alignment in the generated C structs.)

2.3 Other XTCE features

Several other XTCE features, presented below, have proven useful to the VIPER mission.

2.3.1 Calibrators

Calibrators are used in XTCE to specify the conversion between the raw form of a telemetry value, as packed into a telemetry container, and the engineering value, which may be of a different numeric type and have associated units. Three calibrator forms are available for use in XTCE, and all have found uses in VIPER:

- Polynomial calibrators: The engineering value is specified as a polynomial in the raw value.
- Math operation calibrators: The engineering value is an arbitrary math operation based on the raw parameter value, other constant values, and math operations.
- Spline calibrators: The engineering value is interpolated based on a lookup table. All instances of spline calibrators in VIPER specified all possible table locations, so they were strictly lookup tables with no interpolation required.

2.3.2 Derived parameters

In many instances there were parameters needed on the ground which were a combination of two or more telemetry values. These were created in VIPER by defining a new parameter which was not part of any telemetry container, but instead was written to by an algorithm, and triggered by the arrival of telemetry containing the input parameter(s) to that algorithm. These algorithms can be specified in an XML language within the XTCE (math operations) or externally in custom code (custom algorithms). Both forms are use in VIPER.

2.3.3 Multiple decodings of the same bits or bytes

Occasionally there were teams within VIPER that wanted the same telemetry to be interpreted in different ways. For example, a hardware group might want a 2-byte sequence to be displayed as a numeric value, while another group might want it parsed into two numbers of 10 and 6 bytes. This can be accomplished in XTCE by specifying both encodings in the telemetry container, and specifying a relative offset for the second interpretation to “back up” over the first interpretation.

```
<EntryList>
…
  <ParameterRefEntry parameterRef="theTwoByteParameter"/>
  <ParameterRefEntry parameterRef="the10BitParameter">
    <LocationInContainerInBits referenceLocation="previousEntry">
      <FixedValue>–16</FixedValue>
    </LocationInContainerInBits>
  </ParameterRefEntry>
  <ParameterRefEntry parameterRef="the6BitParameter"/>
…
```

2.3.4 Command significance

Some commands to the VIPER rover require some caution when used. They may only be appropriate if some prior procedures are accomplished, for example. To make these commands safer to use, the *significance* of the command is specified as a non-default value. The configuration of the command queuing and approval in the Yamcs system can then require another level of approval before sending the command.

```
<MetaCommand name="SampleDangerousCommand">
…
  <DefaultSignificance consequenceLevel="critical"/>
</MetaCommand>
```

2.4 Other features of the XTCE processing

The processing of XTCE to C headers file proved quite useful, and, over time, there were requests both to enhance the content of the header files and to generate other output formats.

2.4.1 Bitfield processing

Both to maximize telemetry bandwidth and to accommodate telemetry generated by hardware, VIPER defined some numeric or enumerated parameters as bit fields not occupying an integral number of bytes. It is natural to expect such parameters either to be represented in the C header files as struct bitfield members or to have macros available for getting and setting such bit field values. The latter approach was used in VIPER.

2.4.2 Other representations of the command and data dictionary

Other output formats were requested, and the XSLT technique was used for generation.

- A CSV summary of telemetry container (packet) sizes, for use in ensuring that filter tables satisfied telemetry bandwidth budgets.
- A spreadsheet output format for wider dissemination within VIPER. For example, the fault management team used data from the spreadsheet format of the telemetry definitions to define alarm limits. This proved so useful that an enhanced converter was developed using Python, so that Excel format could be generated directly.

3. Results and discussion

3.1 Things that went well

Overall, the decision to use XTCE as the source for modeling the command and data dictionary on the VIPER project has worked well. At the same time, the textual nature of the command and telemetry definition fit well into the existing source code control practices of the mission teams.

3.1.1 Ease of XTCE processing

A relatively small amount of scripting (under 2k raw lines) was used to process about 30k lines of XTCE templates (55 files) and variable configuration which generated about 60k raw lines of C header files for the flight software. The same XTCE that generated those header files was used to configure the telemetry and commanding in the Yamcs system.

3.1.2 Revision control and reviews

The textual nature of the XTCE made it easy to incorporate into existing software management practices and Git revision control. Breaking the XTCE up into multiple files allowed work on separate subsystems to proceed without checking conflicts. And the existing code review process worked well for reviewing XTCE changes, once reviewers learned the basics of the XTCE representation.

3.1.3 Richness of XTCE language

VIPER made extensive use of several XTCE features supported by the Yamcs telemetry and command system which are less flexible in non-XTCE-based telemetry and command systems that the team is familiar with.

- Calibrators for raw-to-engineering conversions
- Algorithms for ground-derived telemetry
- Aggregated types, especially for avoiding repetition when supporting redundant or multiple systems
- Enumerated types, where the enumeration labels are simultaneously available as enum constants in flight software source code and in the telemetry client displays

3.1.4 Ground-specific XTCE
In addition to the roughly 30k lines of XTCE templates used to define telemetry and commanding for the flight software, there were about 20k lines of XTCE for ground-derived telemetry, and about 30k lines of machine-generated XTCE to define parameters coming from the Deep Space Network monitoring facilities. In this way XTCE became a common language for describing most of the data processed by the entire Ground Data System. The rich features provided by XTCE formed a feature set that different development groups could count on as they developed new mission features and capabilities.

3.2 Things that were less successful
Some features of XTCE were underused, and some aspects of the XTCE usage were cumbersome or problematic.

3.2.1 Non-XTCE-friendly telemetry
Occasionally, there was telemetry which was not possible to represent in XTCE. One example was a telemetry packet generated by a hardware device. Two 9-bit numeric parameters were contiguous bits when sent over a serial line from the hardware to the flight software, but not when considered as part of a sequence of bytes. The bits in those numeric fields were, instead, separated by intervening parameters. There was no way in XTCE to specify that two bit ranges from a packet should be combined to form a single number. However, this was easily worked around by using a ground algorithm to combine the two parts into a new parameter.

A more problematic example is the case of telemetry not having a fixed packet format. One example of this is the name-value format used by the Deep Space Network for reporting ground station telemetry. Defining the parameters themselves in XTCE was not a problem, but it was impossible to define a packet format where parameters could appear in any order, and in which numeric codes would indicate what parameter value is next. Instead, VIPER wrote custom processing code for these data, writing the parameter values directly to the Yamcs telemetry and command system, without using any XTCE definition of the telemetry packet format.

3.2.2 Lack of sharing between telemetry and command type definitions
The inability to share the same types for telemetry parameters and command arguments required two features be implemented in the XTCE processing scripts. First, to avoid name conflicts in the generated header files, any time a command argument type definition was processed, there first had to be a check to see if a type with the same name had already been defined as a parameter type. If so, the types were assumed to be the same, and the argument type definition was not written to the output header file.

Second, the inability to share enumerated type definitions between telemetry parameter and command argument types was a major motivator toward implementing the XTCE template processing outlined earlier. The variable substitution performed during template processing allows creating identical definitions with a minimum of duplicated code. But if XTCE had defined semantics for sharing types, some of that processing may not have been necessary.

3.2.3 XSLT knowledge is rarely common among developers
As mentioned earlier, the processing of XTCE into C header files was implemented in a script using the XSL Transformations language XSLT. Whenever a lot of XML transformation needs to be done, for one author the obvious first choice is XSLT. However, knowledge of XSLT proved too rare among most developers on the VIPER project, so few people attempted to make changes to the XSLT scripting. In retrospect it would likely have been better to write the processing in Python, or some other scripting language more generally familiar to the developer pool. The resultant script might not have been as small, but would have been more understandable just because so few are knowledgeable about the XSLT language or paradigms.

3.2.4 Developers like general types, parameters need specific types
If you have twenty parameters in a subsystem that are all 2-byte integers, a developer will likely approach that in XTCE by defining one parameter type that is shared among all those parameters. However, in XTCE both raw-to-

engineering conversion and alarms based on limits are applied to the data type, not the parameter itself. This means that it may be better instead to have separate data types for each of those integer parameters.

Similarly, a developer may think that the set of twenty parameters *might occur again*, in which case it may be better to put them all into an aggregate type in XTCE. In the extreme case – which occurred once on VIPER – a developer might decide to put the entire set of parameters for a telemetry packet into an aggregate and have the <Container> just have one parameter in its <EntryList>. From a software developer standpoint it doesn't matter: in VIPER both the aggregate type and the <Container> become C structures. But from the telemetry client standpoint there are two issues. First, there is another level of nesting that may show up in the client displays of the telemetry. Second, aggregates and their members have been treated as an afterthought by the XTCE standardization process, and so are not as well supported by the telemetry and command systems.

In VIPER, this tension between what looks good from a software development standpoint and what makes a good telemetry user experience has caused some rework and some problems in telemetry alarms processing that needed to be worked around. As will be mentioned later, some XTCE standardization changes might help in this area.

3.2.5 Command verifiers were underused

Everyone on VIPER was new to XTCE, so some features were underutilized, at least at first. The biggest example of this is the ability to define *verifiers* for a command. In the XTCE one can define, for a command, telemetry values to look at to determine whether the command was received, whether it is executing, and whether it finished, as well as failure at any of those points.

In VIPER, this feature is not utilized, for two main reasons. First the cFS flight-software framework is not designed with XTCE command verification in mind. cFS applications tend to report failure, for example, by sending event packets. This means that success or failure of any command cannot generally be monitored by looking at a unique telemetry parameter. Instead, a single set of common parameters is reported in the event packet. It is true that cFS applications report command and error counts in their housekeeping data, but that data may not automatically be sent, depending on the filter table, and may be misleading if multiple commands to the same application are sent in a short timeframe. The telemetry reported by cFS applications may need to be augmented to make command verifiers easier to configure and more reliable.

Second, lack of knowledge about how to configure command verifiers, and how to set up telemetry so it is useful to command verifiers, was lacking. This is an area that might have been done differently if we were redesigning the mission based on acquired knowledge.

3.3 New XTCE features, or a higher-level language

The areas in which VIPER customized the XTCE templates through the addition of ancillary data could serve as a model for enhancements to the XTCE language or the motivation for a similar, slightly higher-level language.

3.3.1 Parameterization

XTCE could benefit from the introduction of parameters, such as is possible in the XSL Transformations language[6]. In that language, a parameter may be declared at the top level like this:

```
<xsl:param name="platformByteOrder"/>
```

The parameter may be referenced elsewhere by prefixing with a dollar sign. Such a technique could be used to replace the VIPER substitution %PLATFORM_BYTE_ENCODING% with a parameter at build time.

```
<IntegerParameterType name="Sample">
  <IntegerDataEncoding sizeInBits="16" byteOrder="$platformByteOrder" encoding="unsigned"/>
</IntegerParameterType>
```

The use of a parameter in this instance would not reduce the complexity of the specification but would standardize it so that other systems wishing to use the same XTCE would understand those parameters that need to be supplied.

3.3.2 Alternate representations

XTCE was designed as a language for interchanging the definitions necessary to interpret telemetry and formulate commands. But VIPER is using it to model representations that might differ in structure between the flight software and the telemetry and command system on the ground. Alternate representations for the flight software were indicated in added ancillary data. A slightly more expressive description language could include alternate representations as a built-in feature. For example, the case described earlier where the ground systems wanted separate parameters in an aggregate type while the flight system software preferred an array representation might be handled something like this, defining both representations as parameter types:

```
<AggregateParameterType name="AnglePose">
  <MemberList>
    <Member typeRef="rotation_angle" name="roll"/>
    <Member typeRef="rotation_angle" name="pitch"/>
    <Member typeRef="rotation_angle" name="yaw"/>
  </MemberList>
</AggregateParameterType>
<ArrayParameterType arrayTypeRef="rotation_angle" name="AngleArray">
  <DimensionList>
    <Dimension>
      <StartingIndex><FixedValue>0</FixedValue></StartingIndex>
      <EndingIndex><FixedValue>2</FixedValue></EndingIndex>
    </Dimension>
  </DimensionList>
</ArrayParameterType>
```

Then, referencing both the type used for telemetry processing as well as an alternate representation for another purpose, in this case the flight software:

```
<Parameter parameterTypeRef="AnglePose" name="pose">
  <ParameterProperties>
    <Representation target="flight-software" parameterTypeRef="AngleArray"/>
  </ParameterProperties>
</Parameter>
```

A validation tool could infer that the two representations should occupy the same amount of space, and test for that condition.

3.3.2 Parameter-level specifications

Many aspects of a spaceflight system are parameter-specific, such as the raw-to-engineering calibration method and alarm limits. However, XTCE is highly focused on data types. This means that declarations that are parameter-specific require parameter-specific types. When XTCE is used as initially intended, for *interchange* of telemetry and command definitions, this extra level of indirection does not matter. But when humans are writing and reviewing the XTCE the extra level of indirection is a burden. It is as if when declaring a variable in a high-level language one could not write:

```
float temperature;
```

but instead must write a new type declaration first:

```
typedef float TemperatureType;
TemperatureType temperature;
```

It would be convenient if some parts of data type declarations could also be applied at the parameter level.

4. **Conclusions**

The need for a common model between the flight and ground software systems for the telemetry and command definitions is a key factor in ensuring smooth communications. XTCE has proven to be a worthy language for specifying those definitions, and, though extensions to the standard XTCE modeling were required by VIPER, they were facilitated by XTCE features in the standard, such as the ability to specify ancillary data and aliases.

An area for further research is to what extent a higher-level language might be utilized to avoid custom representations of ancillary information and to reduce the amount of mission-specific scripting required.

Acknowledgements
The authors would like to thank those who provided helpful comments during the review process. In addition, they would like to acknowledge the many participants in the VIPER project, both inside and outside of NASA, who have made the VIPER project such a pleasure to work on.

References

[1] Object Management Group, *XML Telemetry and Command Exchange Format version 1.2*, January, 2019, https://www.omg.org/spec/XTCE/1.2/PDF.
[2] Colaprete, Anthony, *Volatiles Investigating Polar Exploration Rover*, Accompaniment to ROSES Proposal Call ROSES-21, https://ntrs.nasa.gov/citations/20210015009.
[3] Schonfeld, Julie, *Commercial Lunar Payload Services (CLPS) For Lunar Science*, Lunar Biology Technology Workshop, 2025, April 20–21, https://ntrs.nasa.gov/citations/20220006317.
[4] *Core Flight System – BUNDLE*, NASA Goddard Space Flight Center's Flight Software Systems Branch, https://github.com/nasa/cFS.
[5] *Yamcs Mission Control*, https://github.com/yamcs/yamcs.
[6] World Wide Web Consortium, *XSL Transformations (XSLT) Version 3.0*, W3C Recommendation, June 8, 2017, https://www.w3.org/TR/xslt-30/.
[7] *Integrated Test and Operations System (ITOS)*, The ITOS Group at NASA Goddard Space Flight Center, https://itos.gsfc.nasa.gov/.
[8] *Galaxy*, The Hammers Company, https://hammers.com/galaxy.
[9] Pires, Craig, *Study of Tools for Command and Telemetry Dictionaries, Flight Software Workshop*, December 4, 2017, https://ntrs.nasa.gov/api/citations/20180000774/downloads/20180000774.pdf
[10] Software, Robotics and Simulation Division, *Core Flight System (CFS) Command and Data Dictionary (CCDD) utility*, NASA Lyndon B. Johnson Space Center, https://github.com/nasa/CCDD.
[11] Gal-Edd, Jonathan, et. al., *XTCE and XML Database Evolution and Lessons from JWST, LandSat, and Constellation*, American Institute of Aeronautics and Astronautics, 2008, https://ntrs.nasa.gov/api/citations/20080044045/downloads/20080044045.pdf.
[12] Consultative Committee for Space System Systems web site, https://public.ccsds.org/default.aspx.
[13] Deutsch, Leslie J., et. al., *Deep Space Network: The Next 50 Years*, SpaceOps 2016, Daejon, Korea, May 16–20, 2026, https://dataverse.jpl.nasa.gov/file.xhtml?fileId=54055&version=2.0.
[14] The Consultative Committee for Space Data Systems, *The Space Packet Protocol*, CCSDS 133.0-B-2, June, 2020, https://public.ccsds.org/Pubs/133x0b2e2.pdf.
[15] NASA Ames Research Center, *Open MCT*, https://github.com/nasa/openmct.