CAPS: AN EGLIN RANGE STANDARD FOR PC-BASED TELEMETRY DATA REDUCTION

Tim Thomas
TYBRIN Corporation
Freeman Computer Sciences Center
Central Control Facility
Eglin AFB, Florida

ABSTRACT

A need exists to provide a flexible data reduction tool that minimizes software development costs and reduces analysis time for telemetry data. The Common Airborne Processing System (CAPS), developed by the Freeman Computer Sciences Center at Eglin AFB, Florida, provides a general-purpose data reduction capability for digitally recorded data on a PC. Data from virtually any kind of MIL-STD-1553 message or Pulse Code Modulation (PCM) frame can be extracted and converted to engineering units using a parameter dictionary that describes the data format. The extracted data can then be written to a file, ASCII or binary, with a great deal of flexibility in the output format. CAPS has become the standard for digitally recorded data reduction on a PC at Eglin. New features, such as composing derived parameters using mathematical expressions, are being added to CAPS to make it an even more productive data reduction tool. This paper provides a conceptual overview of the CAPS version 2.3 software.

KEY WORDS

PC, Data Reduction, MIL-STD-1553, PCM, DDS, Engineering Units, UML, C++, Visual C++, MFC, Design Patterns, XML

INTRODUCTION

CAPS is a PC-based data reduction tool developed by the Freeman Computer Sciences Center at Eglin AFB, Florida. CAPS provides the analyst with a flexible, easy to use tool for telemetry data reduction. Before the advent of CAPS, data reduction was done with costly, application-specific hardware and software tools. Each new data reduction application required a costly investment of time, money, and resources to develop new tools. With the generic data reduction capabilities of CAPS, the analyst can now perform data reduction and analysis more effectively in less time and with lower cost. The continued addition of new features makes CAPS an even more powerful data reduction tool.

SYSTEM DESCRIPTION

CAPS operation is centered on three main components, or engines (Figure 1). These are the Input Engine, which reads raw data from an input source, the Engineering Unit Conversion (EUC) Engine, which extracts parameters from the raw data, and the Output Engine, which formats the extracted parameters for output. A fourth engine, the Executive Engine, controls the creation and operation of the other engines.

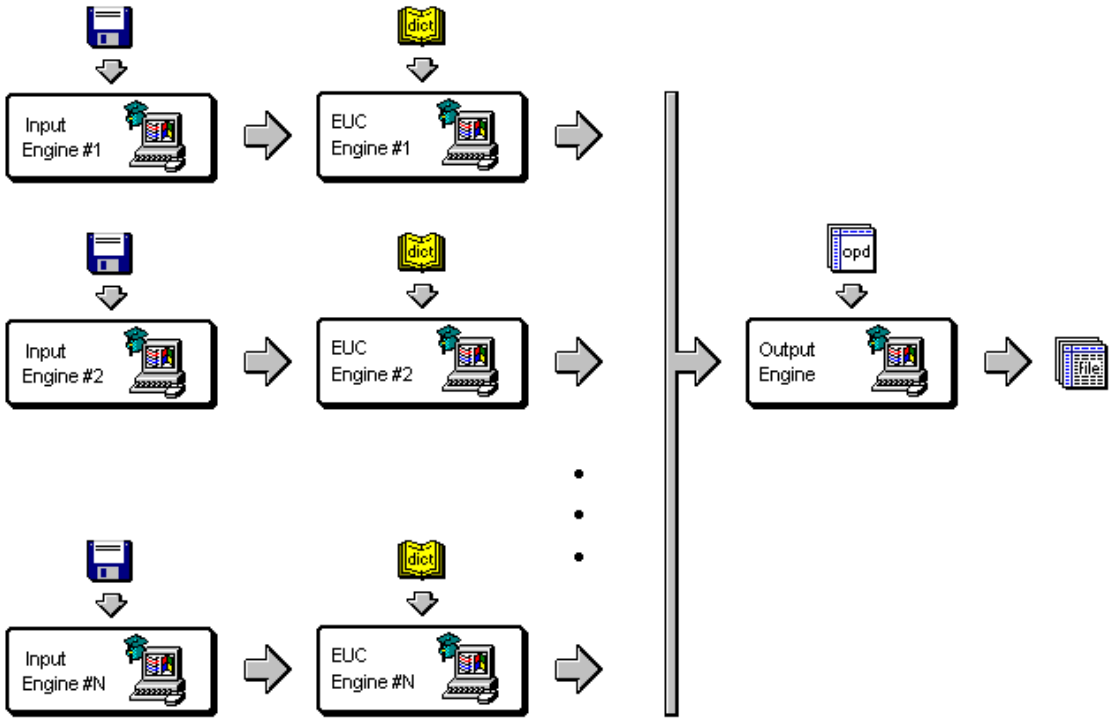

Figure 1. CAPS System Overview

Raw data are read into CAPS with the Input Engine. Each source of data is assigned to a separate Input Engine. Multiple Input Engines can process the same data source, if desired, to simultaneously process a data source in different ways. MIL-STD-1553 data and PCM data are provided to CAPS in a file that conforms to either the Eglin Digital Data Standard (DDS) format or SBS PASS 1000 format files.

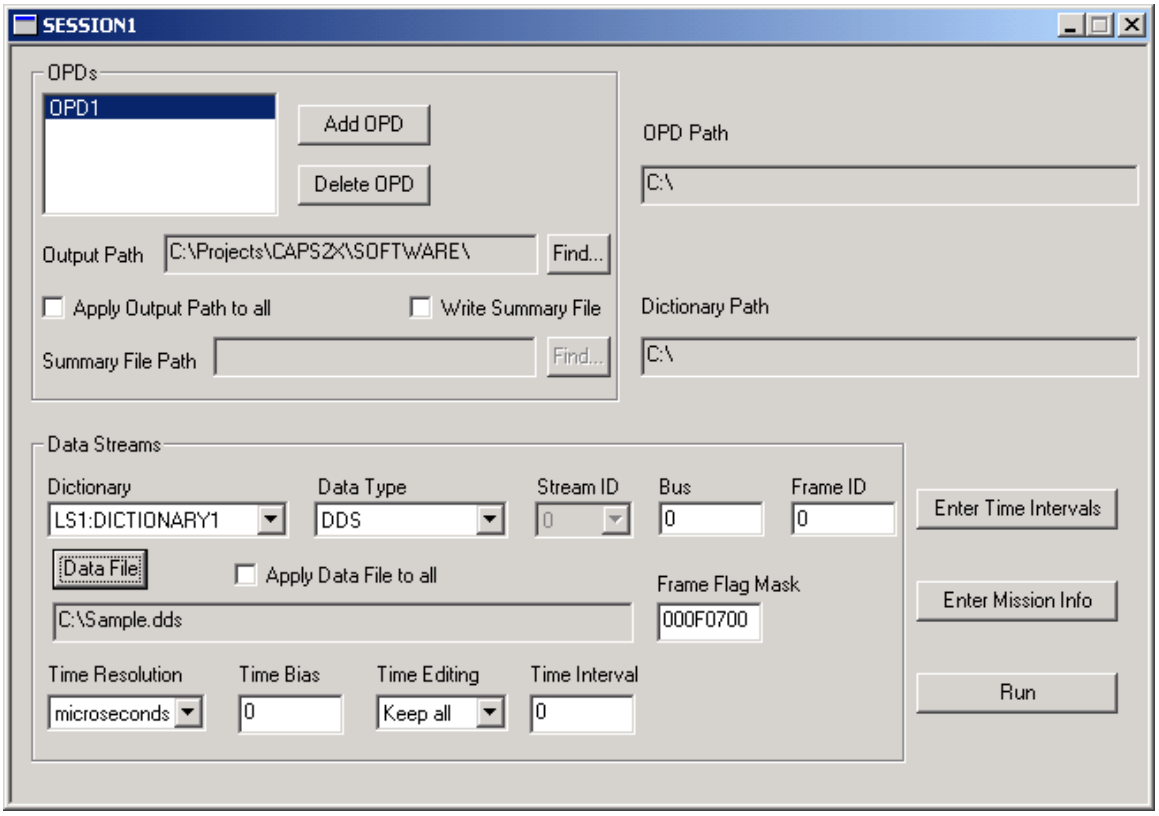

Figure 2. Session Editor

After a frame of data has been read into CAPS, it is processed by the Input Engine according to selections entered in the Session Editor (Figure 2). The frame time is processed by setting its precision to either milliseconds or microseconds, and by adding an optional time bias. Time precision is set by clearing the nanoseconds field of frame time for microsecond precision, or by clearing the nanoseconds and microseconds fields for millisecond precision. (Nanosecond precision is reserved for future enhancements.) This allows the analyst either to keep or to discard the microseconds portion of time, depending on the reliability of the time measurement.

The Input Engine then compares the Frame Information Flags with the Frame Flag Mask from the Session Editor. If a logical AND of these two values produces a non-zero result, the data frame is discarded due to errors. (The DDS standard defines the error flags in the Frame Information Flags field.)

The next step in Input Engine processing is to compare the frame time with a list of time intervals from the Session Editor. If the frame time cannot be found within any of these defined intervals, the data frame is discarded. If no time intervals are defined, then this step is skipped and all data frames are kept for output processing.

The last processing step for the Input Engine is to check for time backups and jumps. If the frame time represents a time back up or jump, and the user has selected to discard time backups and jumps, the data frame is discarded.

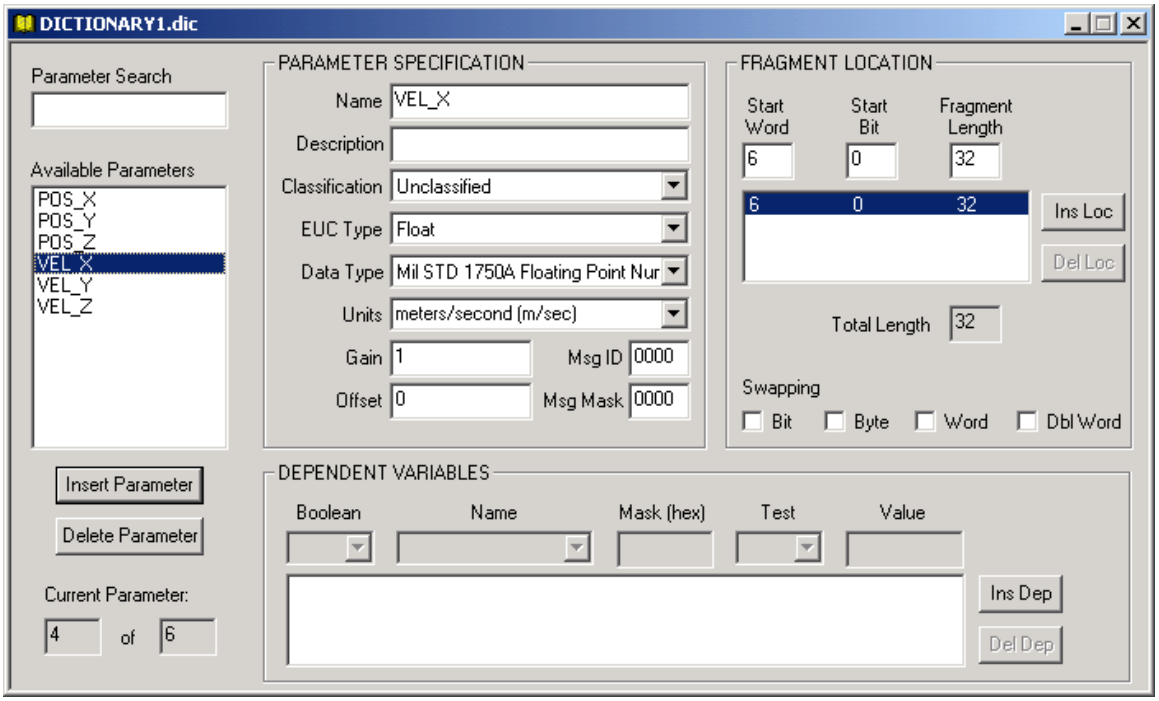

Figure 3. Dictionary Editor

After being processed by the Input Engine, the raw data frame is sent to the EUC Engine for parameter extraction and engineering unit conversion. The EUC Engine performs this extraction and conversion using a parameter dictionary—a low-level description of the data (Figure 3).

For each parameter it processes, the EUC Engine first determines if the parameter is contained in the raw data frame being processed. This is done by comparing the data frame ID with the Msg ID defined for the parameter in the dictionary. The Msg ID is normally set to 0 for PCM frames, but for 1553 messages it is set to the command word (command word 1 for RT-RT messages). The Msg Mask is used to identify a parameter as belonging to a group of messages (such as all receive messages). The Msg Mask is logically ANDed with both the frame ID and the dictionary Msg ID before the comparison is made.

Parameter extraction can also be dependent on the value of other parameters. A parameter's dependencies are specified in the dictionary as a logical expression of the results of other parameters. It is possible, for example, to specify that parameter A can be extracted only if parameter B is equal to 0 and parameter C is not equal to 0. If such a dependency check fails (evaluates as false), the parameter is considered not to exist within the data frame being processed.

Once the EUC Engine determines that a parameter is contained in the data frame, it is extracted and data-type conversion is performed. The parameter is extracted from the location specified in the dictionary. Parameters can be up to 64 bits long and can be composed from multiple, non-contiguous fragments. After being extracted, the parameter is converted from one of many different types of

integer and floating point formats available, to a signed integer (32-bits), an unsigned integer (32-bits), a floating point number (64-bit IEEE format), or a character array.

After extraction and data type conversion, a gain and offset are applied to perform a linear conversion of the form y = mx + b where m and b are the Gain and Offset, respectively, specified in the dictionary.

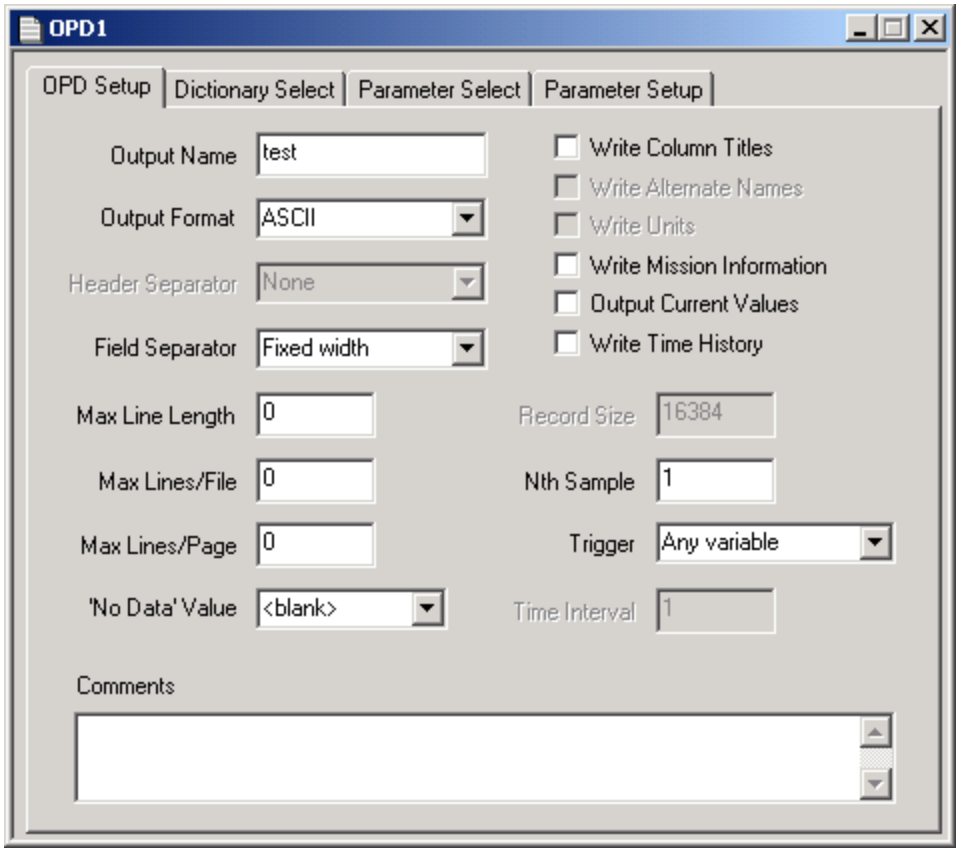

Figure 4. OPD Editor, OPD Setup Page

After parameters have been extracted and converted by the EUC Engine, they are sent to the Output Engine to be formatted for output. For each desired output file, the Output Engine uses an Output Product Description (OPD) defined by the user (Figure 4). The OPD defines the output file format (ASCII, generic binary, or DDS) and other characteristics of the file, such as the number of lines per page and the header information to print at the top of each page. Various triggering options can also be selected to filter the output.

The Output Engine can produce a single output file from multiple input sources. The OPD allows for parameters to be selected for output from multiple dictionaries. Each parameter can be individually formatted for output.

Derived parameters are composed from dictionary parameters (and other derived parameters) using mathematical expressions. They can also be defined for output in the OPD. For example, if the X, Y, and Z positions of an aircraft are available in the raw data file, a new parameter representing the

slant range can be defined as $SR = \sqrt{(X^2 + Y^2 + Z^2)}$ and output along with other parameters extracted from the data.

APPLICATION DESIGN

CAPS was designed using the Unified Modeling Language (UML) and is implemented in Visual C++ 6.0 using Microsoft Foundation Classes (MFC). Various design patterns, such as Factory Method and Composite, were used in the design. The overall architecture of the software is shown in Figure 5.

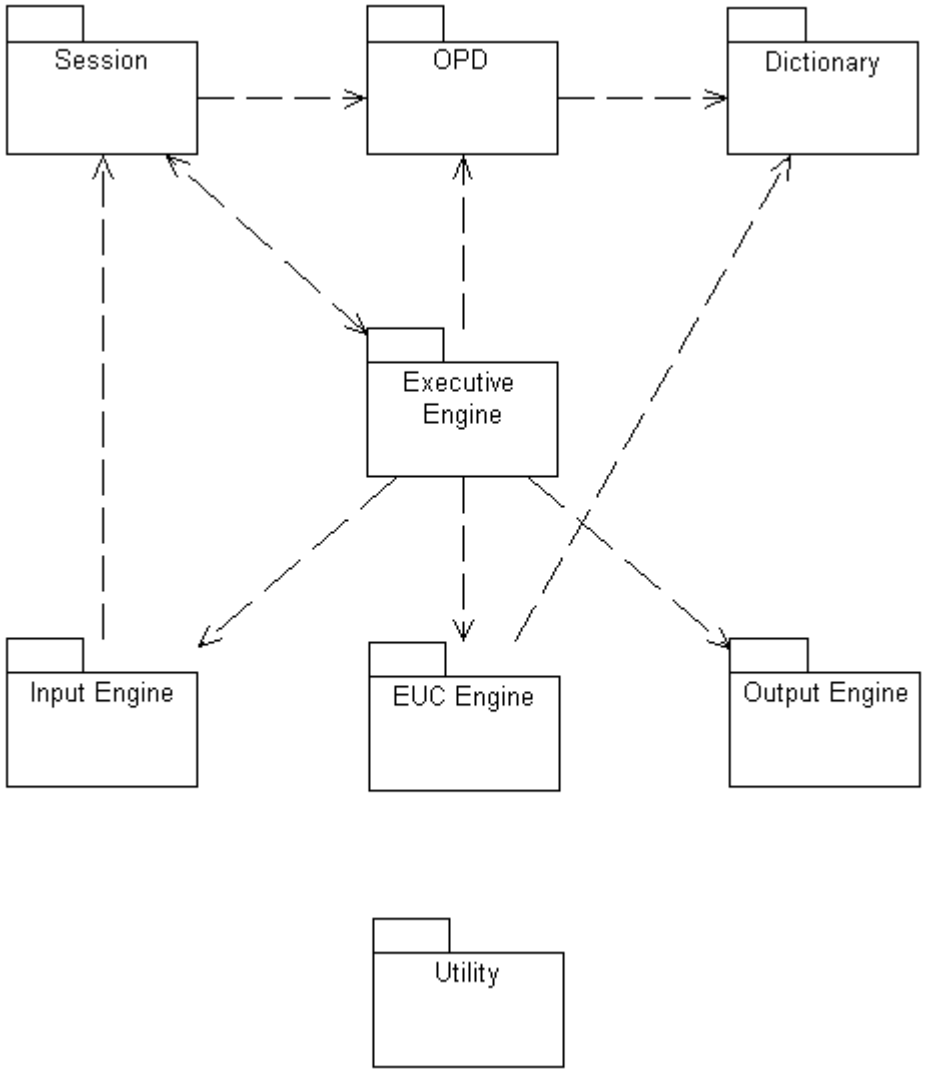

Figure 5. CAPS Software Architecture

A detailed treatment of the CAPS design is beyond the scope of this paper. However, an insight into the flexibility of the CAPS design can be gained by examining the class diagram of the Input Engine (Figure 6).

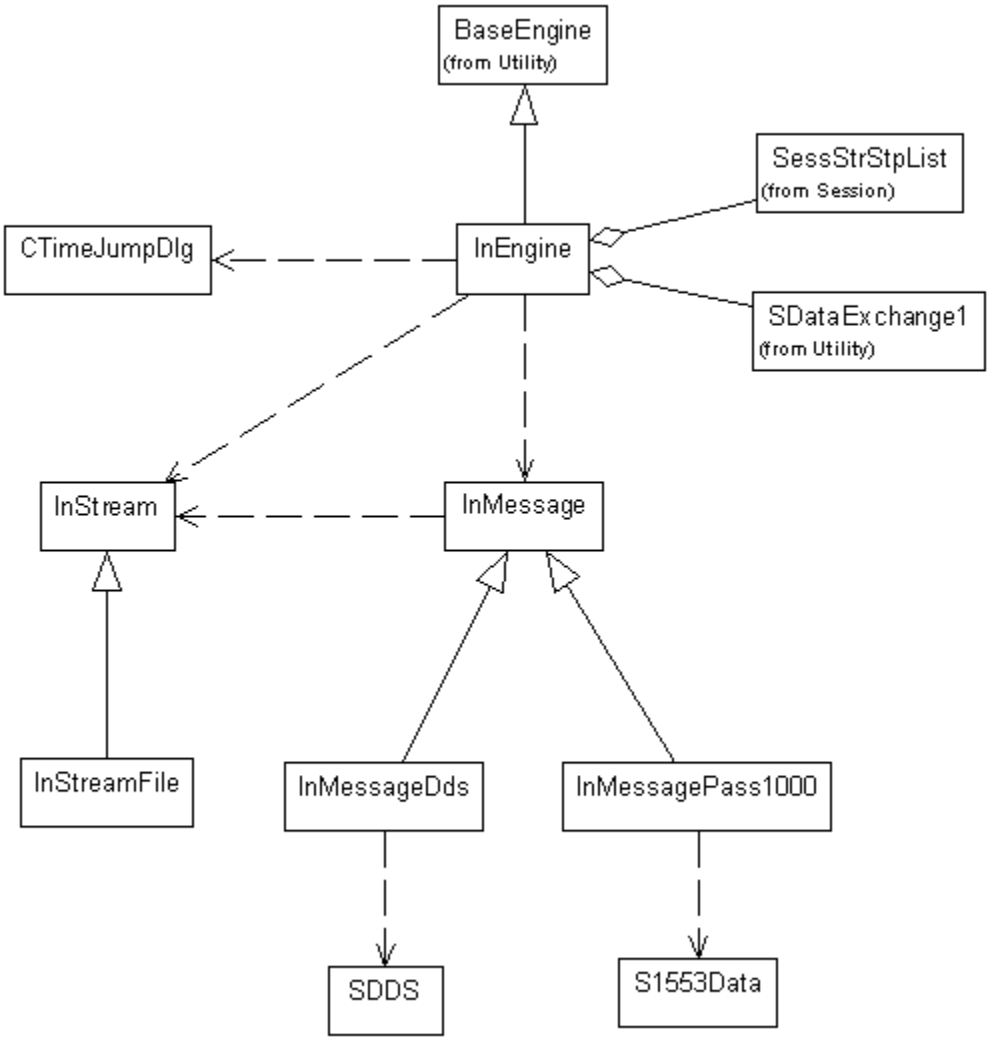

Figure 6. Input Engine Class Diagram

The InEngine class is the main class of the Input Engine. It is derived from BaseEngine, which provides for operation of each engine in its own thread. The Input Engine reads data frames from a source through the interface provided by the abstract class InMessage. The concrete classes derived from InMessage, InMessageDds and InMessagePass1000, provide the capability for reading from either DDS or SBS PASS 1000 files. Additional file formats can easily be supported by implementing a new class derived from InMessage.

InMessage, in turn, reads data from a physical source through the InStream interface. CAPS currently supports reading of data from files only, but the InStream interface allows for easily adding new data sources, such as network sockets.

CONCLUSION

CAPS has succeeded in its goal of providing the analyst with a flexible, easy to use data reduction tool. At the time of the writing of this paper, CAPS version 2.3 was nearing completion. This version will incorporate the addition of derived parameters (described above), add greater flexibility in output configuration with the OPD, and begin the incorporation of newer technologies, such as XML. This will allow CAPS to continue to evolve as the Eglin AFB data reduction tool of choice well into the 21st century.

REFERENCES

Air Armament Center, Eglin AFB, Florida, "Common Airborne Processing System (CAPS) User's Guide." 2002.

Air Armament Center, Eglin AFB, Florida, "Digital Data Standard (DDS): Standardized Digital Instrumentation Data to Achieve DT&E and OT&E Inter-Range Commonality." 1998.

Fowler, Martin, UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, 1997.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Holzner, Steven, Inside XML. New Riders, 2001.