Extensions to the Dynamic Aerospace Vehicle Exchange Markup Language

Geoff Brian[1]
Air Vehicles Division, Platform & Human Systems, Defence Science and Technology Organisation, 506 Lorimer St, Fishermans Bend, Victoria, 3207, Australia

E. Bruce Jackson[2]
NASA Langley Research Center, 100 NASA Road, Hampton, Virginia, 23681-2199, United States of America

The Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) is a syntactical language for exchanging flight vehicle dynamic model data. It provides a framework for encoding entire flight vehicle dynamic model data packages for exchange and/or long-term archiving. Version 2.0.1 of DAVE-ML provides much of the functionality envisioned for exchanging aerospace vehicle data; however, it is limited in only supporting scalar time-independent data. Additional functionality is required to support vector and matrix data, abstracting sub-system models, detailing dynamics system models (both discrete and continuous), and defining a dynamic data format (such as time sequenced data) for validation of dynamics system models and vehicle simulation packages. Extensions to DAVE-ML have been proposed to manage data as vectors and n-dimensional matrices, and record dynamic data in a compatible form. These capabilities will improve the clarity of data being exchanged, simplify the naming of parameters, and permit static and dynamic data to be stored using a common syntax within a single file; thereby enhancing the framework provided by DAVE-ML for exchanging entire flight vehicle dynamic simulation models.

I. Introduction

The Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) is a syntactical language for exchanging flight vehicle dynamic model data[1]. It has been developed in conjunction with the ANSI/AIAA S-119-2011 Flight Dynamics Model Exchange Standard[2] prepared by the American Institute of Astronautics and Aeronautics (AIAA) Modeling and Simulation Technical Committee. The intended purpose of DAVE-ML was to provide a framework for encoding entire flight vehicle dynamic model data packages for exchange and/or long-term archiving[1]. Such data packages are commonly used in research, engineering development, and flight training simulations. DAVE-ML was designed to provide a programming-language-independent representation of aerospace vehicle characteristics, such as the aerodynamics, mass, propulsion, navigation and control properties. This paper describes extensions to DAVE-ML expanding its ability to managing flight vehicle dynamic model data.

The current status of DAVE-ML is Version 2.0.1[1]. It employs a text-based format built upon the eXtensible Markup Language (XML) Version 1.1[3], and Mathematical Markup Language (MathML) Version 2.0[4] open standards developed by the World Wide Web Consortium (W3C). DAVE-ML defines additional grammar to provide a domain-specific language for aerospace flight dynamics modelling, verification, and documentation. It is capable of storing static aerospace vehicle characteristic data in human-readable text form, together with directives for automating the conversion of data into a form suitable for use in vehicle simulations. Furthermore, it provides the capability to include statistical properties for embedded data such as confidence bounds and uncertainty ranges, along with references to reports, contact information and data provenance.

While Version 2.0.1 of DAVE-ML provides much of the functionality envisioned for exchanging aerospace vehicle data, it is limited in only supporting scalar time-independent data. Additional functionality is required to support the exchange of entire flight vehicle dynamic models. This includes supporting vector and matrix data,

[1] Aircraft Flight Dynamics and Performance Engineer, Lifetime Member.
[2] Senior Research Engineer, Dynamics Systems and Control Branch, MS 308, Associate Fellow.

abstracting sub-system models, detailing dynamics system models (both discrete and continuous), and defining a dynamic data format (such as time-history data) to support validation of dynamics system models and the overall simulation package. Proposals are being developed to extend the DAVE-ML Version 2.0.1 syntax to permit the capture of entire flight vehicle dynamic models. Extensions to the DAVE-ML syntax for managing vector and matrix data, together with recording dynamic data, are detailed in this paper.

II. Vector and Matrix Variable Elements

A question that may be posed is *why collate data as vectors or matrices when exchanging aerospace vehicle information*? Aerospace flight behaviour simulation applications often reference and manipulate vehicle data in a component form such as the aerodynamic force coefficients C_x, C_y and C_z. A similar example is addressing a vehicle's individual body-axes mass moment of inertia components when solving the equations of motion instead of utilising a mass moment of inertia tensor matrix. This is typically an artefact of the simulation applications utilising scalar based equations to represent the vehicle and its motion, e.g., Eqs. (1-3) which represent the flat non-rotating earth body-axes linear accelerations of a vehicle, instead of more concise forms of these equations, e.g., Eq. (4). In Eq. (4), $\dot{\boldsymbol{V}}$ represents a vector of the body-axes linear accelerations, $F_{aerodynamic}$ the body-axes aerodynamic forces, $F_{propulsion}$ the body-axes propulsion forces, $\boldsymbol{\Omega}$ the body-axes rotational rates, V the body-axes linear velocities, g the gravitational acceleration, and $[T]^{be}$ a transformation matrix converting data from an earth reference frame to a body reference frame.

$$\dot{u} = \frac{1}{m}(\tilde{q}sC_x + F_{x\ propulsion}) - qw + rv - gsin\theta \tag{1}$$

$$\dot{v} = \frac{1}{m}(\tilde{q}sC_y + F_{y\ propulsion}) - ru + pw + gsin\phi\cos\theta \tag{2}$$

$$\dot{w} = \frac{1}{m}(\tilde{q}sC_z + F_{z\ propulsion}) - pv + qu + gcos\phi\cos\theta \tag{3}$$

$$\dot{V} = \frac{1}{m}(F_{aerodynamic} + F_{propulsion}) - (\Omega \times V) + g[T]^{be} \tag{4}$$

Addressing vehicle data in component form can result in unnecessary complexity when coding simulation applications, as well as increasing the potential of misinterpreting the meaning of data when combined with associated data. Collating data as vectors and matrices permits parameters that have a common basis to be grouped when exchanging vehicle data, e.g., an aircraft's three body-axes force components may be managed as a single vector parameter $F_{aerodynamic}$. Managing data in this way can improve the clarity of data being exchanged.

Vectors and matrices are in essence the same as scalar variables except that they represent a series of values instead of a singular value. Vector and matrix support was accomplished by extending the capability of the variable definition element in DAVE-ML Version 2.0.1[1]. Sub-elements were included to define the dimensions of vectors and matrices, and to specify the data of the vector or matrix. These data could be numeric, references to other defined variables, or a combination. Alternatively, the contents of a vector or matrix could be computed using the calculation sub-element by defining a suitable MathML expression. Figure 1 represents the revised variable definition for DAVE-ML where the additional sub-elements supporting vector and matrix definitions are highlighted in bold text, and listed in Table 1. Figure 2 presents examples of defining a vector and a matrix using this syntax.

```
variableDef : name, varID, units, [axisSystem], sign, alias, symbol,
           [initialValue], minValue, maxValue
    description?
    (provenance | provenanceRef)?
    (dimensionDef | dimensionRef)?
    (calculation | array)?
    (isInput | isControl | isDisturbance)?
    IsState?
    IsStateDeriv?
    IsOutput?
    IsStdAIAA?
    uncertainty?
```

Figure 1. DAVE-ML variable definition with additional sub-elements supporting vectors and matrices.

Table 1. Additional sub-elements for the DAVE-ML variable definition supporting vectors and matrices.

array	The data for the vector or matrix
dimensionDef	A list of dimensions (dim) for the vector or matrix
dimensionRef	A reference to a dimensionDef element

```
<variableDef name="vectorName_nd" varID="vectorID" units="nd">
  <dimensionDef dimID="vector_3">
    <dim>3</dim> <!-- Number of data points in the vector -->
  </dimensionDef>
  <array>
    <dataTable> 1.0, 2.0, 3.0 </dataTable>
  </array>
</variableDef>
```

a) A vector with three entries,

```
<variableDef name="matrixName_nd" varID="matrixID" units="nd">
  <dimensionDef dimID="matrix_2x3">
    <dim>2</dim> <!-- Number of rows in the matrix -->
    <dim>3</dim> <!-- Number of columns in the matrix -->
  </dimensionDef>
  <array>
    <dataTable>
      1.0, 2.0, 3.0, <!-- Row #1 -->
      0.0, 2.0, 5.0, <!-- Row #2 -->
    </dataTable>
  </array>
</variableDef>
```

b) A two-dimensional matrix,

Figure 2. Defining vectors and matrices using the extended DAVE-ML variable definition syntax.

The list of dimensions defined using the **dimensionDef** element specified either the number of data points in a vector, or the size of each dimension in an n-dimensional matrix. The lowest entry of the list specified the number of columns of the base matrix. The subsequent higher entries specified the number of rows in the base matrix, the number of base matrices making up the third dimension, and so forth for n-dimensional matrices. This is illustrated in Fig. 3.

```
<dimensionDef dimID="vector_3">
  <dim>3</dim> <!-- Number of entries in the vector -->
</dimensionDef>
```

a) A vector with three entries,

```
<dimensionDef dimID="matrix_2x2x2">
  <dim>2</dim> <!-- Number of base matrices for the 3rd dimension -->
  <dim>2</dim> <!-- Number of rows in the base matrix -->
  <dim>2</dim> <!-- Number of columns in the base matrix -->
</dimensionDef>
```

b) A three-dimensional matrix,

Figure 3. Defining the dimensions of vectors and matrices.

A reference to a dimension definition could also be used as an alternative to explicitly specifying the dimensions as part of the variable definition. This permitted the reuse of **dimensionDef** elements for vectors or matrices having a common size. The **dimensionRef** referenced an identifier **dimID** associated with the dimension definition.

```
</dimensionRef dimId="matrix_3x3">
```

The data for a vector or matrix were specified using the array element. This was in effect a table of data, and thus the **dataTable** element from DAVE-ML was utilised for encoding the vector and matrix entries. The table of data represented consecutive entries for a vector. The entries for a matrix were specified such that the column entries of the first row were listed followed by column entries for subsequent rows until the base matrix was complete. This sequence was repeated for higher order matrix dimensions until all entries of the matrix were specified. Figure 4 illustrates this procedure.

```
<array>
  <dataTable>
    0.0, eulerInclinationAngle, eulerRollAngle
  </dataTable>
</array>
```

a) A vector with three entries,

```
<array>
  <dataTable>
    <!-- First 2x2 Matrix -->
    1.0, 0.0, <!-- Row #1 -->
    0.0, 1.0, <!-- Row #2 -->

    <!-- Second 2x2 Matrix -->
    inertiaIXX, -20.4,      <!-- Row #1 -->
    -15.6     , inertiaIYY, <!-- Row #2  -->
  </dataTable>
</array>
```

b) A three-dimensional matrix,

Figure 4. Encoding data for vectors and matrices.

The calculation sub-element of the DAVE-ML variable definition permitted the value of a variable to be computed from an equation defined using MathML Version 2.0[4] syntax. MathML contains operators for computing the transpose, inverse, determinant, vectorproduct[*], scalarproduct[†], and outerproduct[‡] of vectors and matrices; in addition to operators for adding, subtracting, and multiplying data. Furthermore, MathML contains an operator, named **selector**, which identifies elements to be extracted from a vector or matrix. Operators such as **plus**, **times**, and **minus** apply equally to scalars, vectors and matrices, and may be used when mixing variable types. The **transpose**, **inverse**, **determinant**, **vectorproduct**, **scalarproduct**, **outerproduct**, and **selector** operators are only relevant to vector and matrix variable types.

Using equations to calculate variables involving vectors and matrices was essentially the same as calculating scalar variables through the equivalent DAVE-ML Version 2.0.1 constructs. However, it was necessary to define the size of the vector or matrix that resulted from the calculation. This was unnecessary if the result of the calculation was a scalar value. An example of multiplying a matrix **M** and a vector **V** to calculate a resultant vector **R**, Eq. (5), is presented in Fig. 5. A more complex example is presented in Fig. 6, where Eq. (6) is encoded using MathML vector and matrix operators.

$$R = [M]V \qquad (5)$$

$$\begin{bmatrix}\square\\\square\end{bmatrix} = \begin{bmatrix}\square&\square&\square\\\square&\square&\square\end{bmatrix}\begin{bmatrix}\square\\\square\\\square\end{bmatrix}$$

```
<variableDef name="R" varID="R" units="">
  <description> Multiplying a matrix and vector</description>
  <dimensionDef>
    <dim>2</dim> <!-- Number of entries in the output vector-->
  </dimensionDef>
  <calculation>
    <math>
      <apply>
        <times/>
        <ci>M</ci>
        <ci>V</ci>
      </apply>
    </math>
  </calculation>
  <isOutput/>
</variableDef>
```

Figure 5. An example of multiplying a matrix by a vector.

The **selector** operator identified entries to extract from a vector, as well as identifying rows, columns, diagonals or sub-matrices to extract from a matrix. The extraction type (i.e., a single entry, a row, a column, etc.) was defined using the **other** attribute of the **selector** operator, and assigning it one of the descriptors listed in Table 2. The dimensions of the originating vector or matrix, together with the choice of extraction type, defined the number of arguments associated with the **selector** operator. For example, the operator had two arguments to extract a single element from a vector. The first argument specified the variable index (varID) of the originating vector, and the second argument defined the index of the element to be extracted, as illustrated in Fig. 7a. Similarly, the list of arguments for extracting a row from a two-dimensional matrix specified the originating matrix and the index of the row to be extracted. Extracting a diagonal from a two-dimensional matrix required three arguments specifying the originating matrix, and the row and column indices of the starting entry for the diagonal, Fig. 7b. Multiple non-consecutive entries from a vector and/or matrix could be extracted by grouping arguments for each extraction within **<apply></apply>** element definitions. Figure 7c illustrates the process of creating a vector by extracting two data entries, one from a vector and the other from a matrix.

[*] The MathML definition for **vectorproduct** is equivalent to that of the **cross product**[5,6].
[†] The MathML definition for **scalarproduct** is equivalent to that of the **dot product**[5,6].
[‡] The **outerproduct** multiplies two vectors to form a matrix: $A = uv^T$, where u and v are vectors[4].

$$R = [I]^{-1}\{\bar{T}(r \cdot r) \times Q\} \tag{6}$$

$$\begin{bmatrix} \square \\ \square \\ \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \end{bmatrix} \left\{ \overline{\begin{bmatrix} \square & \square & \square \end{bmatrix}} \left(\begin{bmatrix} \square \\ \square \end{bmatrix} \cdot \begin{bmatrix} \square \\ \square \end{bmatrix} \right) \times \begin{bmatrix} \square \\ \square \\ \square \end{bmatrix} \right\}$$

```
<variableDef name="R" varID="R" units="">
  <description> Vector and Matrix operations using MathML</description>
  <dimensionDef>
    <dim>3</dim> <!-- Number of entries in the output vector -->
  </dimensionDef>
  <calculation>
    <math>
      <apply>
        <times/>
        <apply>
          <inverse/>
          <ci>I</ci>
        </apply>
        <apply>
          <vectorproduct/>
          <apply>
            <times/>
            <apply>
              <transpose/>
              <ci>T</ci>
            </apply>
            <apply>
              <scalarproduct/>
              <ci>r</ci>
              <ci>r</ci>
            </apply>
          </apply>
          <ci>Q</ci>
        </apply>
      </apply>
    </math>
  </calculation>
  <isOutput/>
</variableDef>
```

Figure 6. An example of coding vector and matrix algebra using the MathML operators.

Table 2. Descriptors for use with the *selector* operator to extract elements from vectors and matrices.

element	Extract a single entry from a vector or matrix
row	Extract a row(s) from a matrix
column	Extract a column(s) from a matrix
diag	Extract a diagonal(s) vector from a matrix
mslice	Extract a sub-matrix from a matrix

The vector and matrix syntax discussed in this paper extends DAVE-ML's capability to manipulate and manage flight vehicle model data. In addition, it offers an approach to improve the clarity of data when exchanged and archived. The next section of this paper further extends the capability of DAVE-ML detailing syntax for recording flight vehicle measured or simulated dynamic data.

```
<variableDef name=”vectorElement” varID=”vectorElementID”>
  <calculation>
    <math>
      <apply>
        <selector other=”element”/>
          <ci>vectorID</ci>
          <cn>2</cn>  <!-- Entry 2 of vectorID -->
      </apply>
    </math>
  </calculation>
</variableDef>
```

a) Extracting an entry from a vector,

```
<variableDef name=”matrixDiagonal” varID=”matrixDiagonalID”>
  <dimensionDef>
    <dim>2</dim>
  </dimensionDef>
  <calculation>
    <math>
      <apply>
        <selector other=”row”/>
          <ci>matrixID</ci>
          <cn>1</cn> <!-- Row number of initial entry -->
          <cn>1</cn> <!-- Column number of initial entry -->
      </apply>
    </math>
  </calculation>
</variableDef>
```

b) Extracting a diagonal vector from a two-dimensional matrix,

```
<variableDef name=”multipleEntries” varID=”multipleEntriesID”>
  <dimensionDef>
    <dim>2</dim>
  </dimensionDef>
  <calculation>
    <math>
      <apply>
        <selector other=”element”/>
        <apply>
          <ci>vectorID</ci>
          <cn>2</cn>  <!-- Entry 2 from vectorID -->
        </apply>
        <apply>
          <ci>matrixID</cn>
          <cn>1</cn> <!-- Row number of entry -->
          <cn>2</cn> <!-- Column number of entry -->
        </apply>
       </apply>
    </math>
  </calculation>
</variableDef>
```

c) Extracting entries from a vector and a matrix to form a new vector,

Figure 7. Examples of extracting data from vectors and matrices using the selector operator.

III. Dynamic Data

Dynamic data, in the context of flight vehicle simulation, refer to the values of any parameter that changes during execution of the simulation. Examples include control commands, as well as vehicle states and responses. Many formats have been defined for recording flight vehicle dynamic data, with some available in open literature while others are proprietary. Irrespective of their source they typically define similar basic information for the dynamic data. This includes the interval between data samples, attributes of the data such as a name and units, and the data.

Comprehensive information about the dynamic data is required when exchanging flight vehicle dynamics models. The data's provenance is particularly important. Information on the source of the data – be it measured flight data, simulation results, or artificially generated – is required as a minimum. The minimum required information for flight measured and simulated data includes details on the vehicle type, the pilot, autopilot and/or control logic, and when and what the vehicle was doing at the time the data were measured or computed. Information on how data are stored is also required so that the data are correctly interpreted in a host application. Quality measures such as data range, resolution, accuracy, and uncertainty, similarly need to be recorded.

A hierarchical structure for storing dynamic data that aligns closely with the DAVE-ML syntax has been proposed, and is described in this paper. The structure is capable of storing arbitrary data having a common independent basis, in addition to the special case of managing time sequenced data. Figure 8 illustrates the structure and elements of the proposed syntax for storing dynamic data. It also presents the linkages with the DAVE-ML syntax. Table 3 lists each of the elements of the dynamic data syntax not directly associated with DAVE-ML. The synergies between the dynamic data syntax and DAVE-ML permit static and dynamic data to be stored using a common syntax within a single file, as well as cross-referencing data for validating a simulation model.

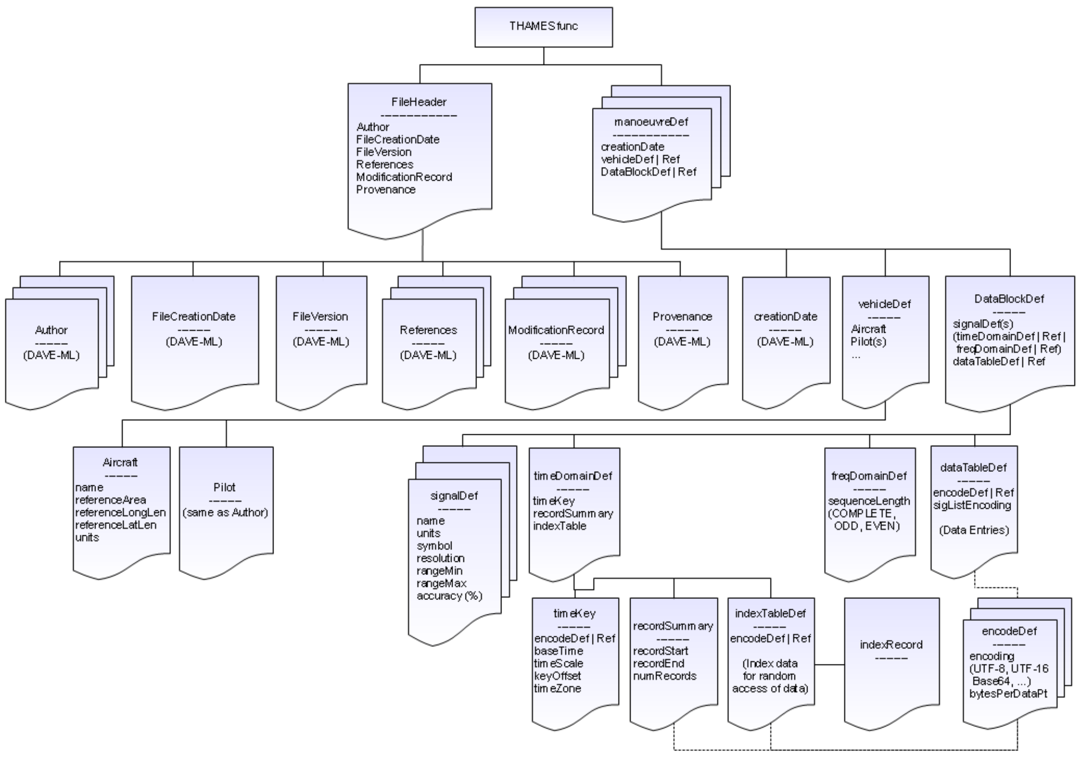

THAMESfunc
FileHeader ----------- Author FileCreationDate FileVersion References ModificationRecord Provenance
manoeuvreDef ----------- creationDate vehicleDef | Ref DataBlockDef | Ref
Author ----- (DAVE-ML)
FileCreationDate ----- (DAVE-ML)
FileVersion ----- (DAVE-ML)
References ----- (DAVE-ML)
ModificationRecord ----- (DAVE-ML)
Provenance ----- (DAVE-ML)
creationDate ----- (DAVE-ML)
vehicleDef ----- Aircraft Pilot(s) ...
DataBlockDef ----- signalDef(s) (timeDomainDef | Ref | freqDomainDef | Ref) dataTableDef | Ref
Aircraft ----- name referenceArea referenceLongLen referenceLatLen units
Pilot ----- (same as Author)
signalDef ----- name units symbol resolution rangeMin rangeMax accuracy (%)
timeDomainDef ----- timeKey recordSummary indexTable
freqDomainDef ----- sequenceLength (COMPLETE, ODD, EVEN)
dataTableDef ----- encodeDef | Ref sigListEncoding (Data Entries)
timeKey ----- encodeDef | Ref baseTime timeScale keyOffset timeZone
recordSummary ----- recordStart recordEnd numRecords
indexTableDef ----- encodeDef | Ref (Index data for random access of data)
indexRecord -----
encodeDef ----- encoding (UTF-8, UTF-16 Base64, ...) bytesPerDataPt

Figure 8. The syntax elements of the dynamic data format.

Table 3. The proposed dynamic data syntax elements.

aircraft	The name and general details of the aircraft, or vehicle
dataBlockDef	The dynamic data and descriptors for a manoeuvre
dataBlockRef	A reference to a data block definition
dataTableDef	The dynamic data associated with a manoeuvre
dataTableRef	A reference to a data table definition
encodeDef	The encoding format for storing dynamic data
encodeRef	A reference to an encoding format definition
freqDomainDef	Dynamic data stored in the frequency domain
freqDomainRef	A reference to a frequency domain definition
indexTableDef	An index table of the dynamic data. This is used to improve random searching of the data
indexTableRef	A reference to an index table definition
indexRecord	The reference and position of a dynamic data record in the index table
pilot	The name and contact information of the pilot(s), or name of the autopilot
recordSummary	Summary information for data stored within a data block
signalDef	The properties of a data signal
signalRef	A reference to a signal definition
signalRefs	A list of signal references
vehicleDef	Information on the vehicle such as the type, pilot, control systems, etc., performing a manoeuvre
vehicleRef	A reference to a vehicle record definition
timeDomainDef	Dynamic data stored in the time domain
timeDomainRef	A reference to a time domain definition
timeKey	Parameters for decoding time stamp data
manoeuvreDef	Defines a manoeuvre for which dynamic data is available

A. Dynamic Data Syntax Elements

The dynamic data are collated under a manoeuvre definition element, **manoeuvreDef**, using the proposed syntax. This element defines the date the data were gathered, generic information about the vehicle, and the associated data. The generic information about the vehicle is defined through a vehicle definition element, **vehicleDef**. The vehicle definition element contains technical details such as the vehicle type or designation. It may also contain reference dimensions, vehicle system components including propulsion, control and sensor systems, together with pilot/autopilot information. The actual dynamic data are defined through a data block definition, **dataBlockDef**, and its various sub-elements. The data block definition provides the structure to manage information on the signals of data stored, whether the data are discrete samples or frequency based, the encoding format of the data, as well as the data itself.

Information on the signals of data stored within a data block is recorded in signal definition elements, **signalDef**. This includes the name for the signal, its unit-of-measure, an identifier (used for cross-referencing when validating system models), and an optional UNICODE symbol that would be displayed when documenting the data. In addition, properties of the signal derived from measured data could be included, such as measurement resolution, minimum and maximum measurement ranges, and percentage accuracy.

Time sequenced and frequency based dynamic data have particular requirements related to the supplementary information required to interpret the associated data. Information such as how the time is encoded, a summary of the time records, and an index table to records of data to improve the speed of random access is necessary. The proposed syntax records this information using the time domain definition element, **timeDomainDef**, and associated sub-elements, as shown in Fig. 8. In the frequency domain, data for each signal represents the coefficients of the equivalent Fourier sequence for the time based dynamic data. These coefficients are typically complex numbers with both the real and imaginary components stored for each signal in a data record. The complete Fourier sequence could be stored; however, utilising the property of conjugate symmetry only half of the coefficients for the Fourier sequence need to be stored. In this case, knowledge is required on whether the original sequence has an even or odd number of entries in order to evaluate the inverse Fourier transform to recreate the history of the signal. The proposed syntax defines this information through the frequency domain definition element, **freqDomainDef**, shown in Fig. 8.

The dynamic data are stored as a table of numeric values defined through the data table definition element, **dataTableDef**, shown in Fig. 8. An encode definition sub-element **encodeDef**, or a reference to such an element, defines the format used to encode data within the table. It stores a label identifying the encoding format; e.g., **ASCII** for encoding data using an *ASCII*[7] representation, or **Base64** for a text equivalent binary format such as *Base64*[8]. The dynamic data syntax does not stipulate the name of this label; however, it would need to be identifiable to, and supported by, an end-use application. The **encodeDef** element could also contain information detailing the precision of the encoded data; this being the number of bytes associated with converting binary data to compliant IEEE Standard 754 floating-point numbers[9]. Dynamic data could be stored as 24-bit (3 byte) floating-point compressed binary numbers, 32-bit (4 byte) single precision floating-point numbers, 64-bit (8 byte) double precision floating-point number, and higher order precision if required.

It is desirable that the dynamic data syntax support the storing of data for only those signals that have changed between successive entries in a time sequenced data set. This functionality is accommodated by prepending a list of signals stored for a time entry to the associated data stream. Listing each signal using either its name or an associated index would be cumbersome; therefore, it is proposed that the list of signals be encoded as a series of bytes where each bit of a byte represents a particular signal. When the bit corresponding to a signal is enabled (i.e., 1) then the data for that signal is stored. If the bit is disabled, (i.e., 0) then the data is not stored. This technique is similar to that used for the storage of compressed binary data by the NASA Dryden Research Center flight test data analysis and aerodynamic parameter estimation applications – getData[§] and pEst[¶]. The resulting list is a binary representation, and therefore, it needs to be encoded using a text equivalent format. An attribute, illustrated in Fig. 8. as the **sigListEncoding**, is required for the **dataTableDef** element to define the choice of format. This attribute would reference a previously defined **encodeDef** element containing the desired encoding format and data precision.

The data table would not use XML markup tags for separating data entities. Instead, an entry would commence with the list of signals stored. The independent reference data for sequenced data would then be included, e.g., the time-stamp for time sequenced data. This would be followed by the data for the stored signals. The real and imaginary components of each signal would be stored consecutively for the frequency domain data. This is illustrated in Fig. 9.

```
dataTableDef
Sequenced Data (Time Domain):
  : signal list, time-stamp, (Signal_1, Signal_2, ...),

or, Frequency Domain Data:
  : signal list, (Signal_1 Real, Imag, Signal_2 Real, Imag, …)
```

Figure 9. Data entries stored in a data table definition element.

§ Richard Maine, *getData Version 3.2.1*, NASA Dryden Research Center, United States of America, 23 Feb 1990.

¶ Richard Maine, *pEst Version 2.3*, NASA Dryden Research Center, United States of America, 23 Feb 1990.

A signal list is not required if only one signal is stored in a data table. In this case defining the **sigListEncoding** attribute of the data table definition is unnecessary.

B. Dynamic Data Examples

Examples of encoding data within a data table using the proposed dynamic data syntax are presented for both time and frequency domain storage. Figure 10 presents the time domain example, which has four signals being angle-of-attack, angle-of-sideslip, true airspeed and pressure altitude. Figure 11 presents the frequency domain example, where time domain data for the angle-of-attack has been converted to the frequency domain. A signal list is not required since angle-of-attack is the only signal being stored. The angle-of-attack sequence for this example has an odd number of time samples, and therefore, the stored sequence would have $(n+1)/2$ entries, where n is the number of time samples. The actual sequence has been truncated in the example; however, it still illustrates the concept of storing frequency domain data.

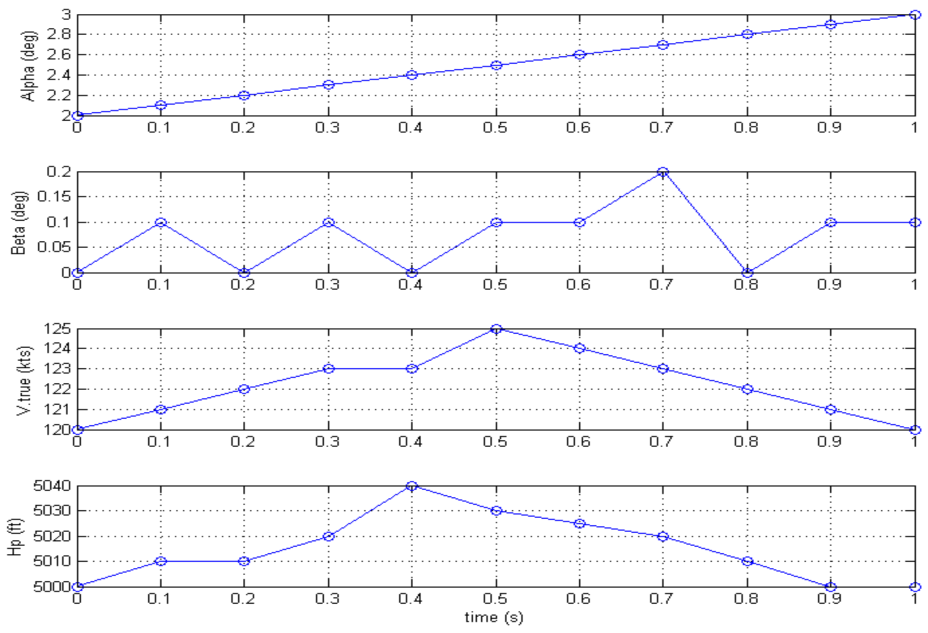
Appendix A presents further examples of the dynamic data syntax illustrating the entries associated with the elements listed in Table 3.

IV. Conclusion

The Dynamic Aerospace Vehicle Exchange Markup Language, together with the ANSI/AIAA-S119-2011 Flight Dynamics Model Exchange Standard, provides a framework for encoding flight vehicle dynamic simulation data packages for exchange between simulation applications and/or long term storage; however, it is limited in only supporting scalar time-independent data.

Syntax consistent with the Dynamic Aerospace Vehicle Exchange Markup Language was developed to manage data as vectors and n-dimensional matrices, offering an approach to improve the clarity of data being exchanged, as well as simplifying the naming of parameters. Furthermore, syntax for managing vehicle dynamic data that closely aligns with the Dynamic Aerospace Vehicle Exchange Markup Language has been proposed. The syntax permits the recording of dynamic data, its provenance, the associated vehicle properties, manoeuvre characteristics, and data quantity metrics. Additionally, it permits static and dynamic data to be stored using a common syntax within a single file.

The vector and matrix syntax, together with the dynamic data syntax, will enhance the ability of the Dynamic Aerospace Vehicle Exchange Markup Language to encode entire flight vehicle dynamic simulation models and their validation data, and simplify the exchange of aerospace vehicle dynamic model data between simulation applications.

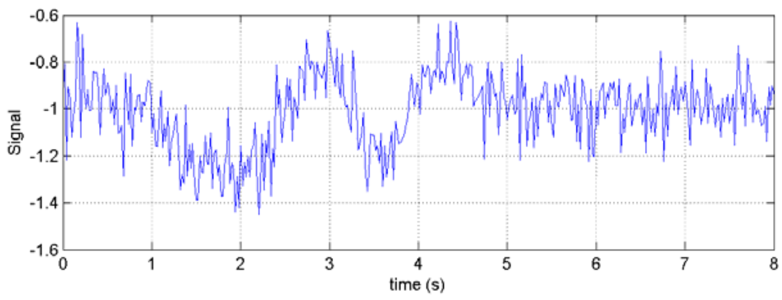


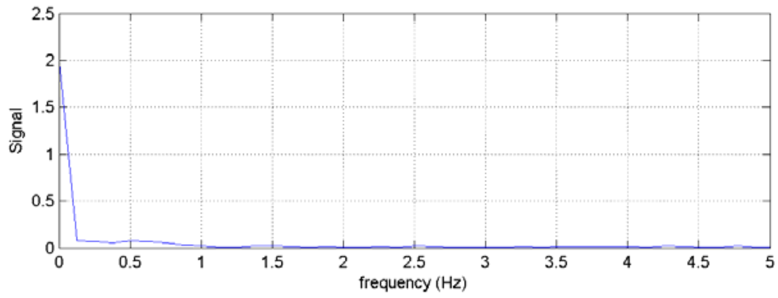
```
<!-- +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Sample Time history data table
     +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Signals: #1 Alpha (deg)  - Angle-of-attack
              #2 Beta  (deg)  - Angle-of-sideslip
              #3 V_true (kts) - True airspeed
              #4 Hp (ft)      - Pressure altitude
-->
<encodeDef encodeID=”encode_1” encoding=”UTF-8”>
<encodeDef encodeID=”signalListEncode_1” encoding=”Base16”>

<dataTableDef dataTabID="dataTab_1">
  <signalListEncodeRef encodeID=”signalListEncode_1”>
  <encodeRef encodeID=”encode_1”/>

  <!-- Signal list, Time-stamp, Signal(s) data -->
  0F, 0.0, 2.0, 0.0, 120.0, 5000.0,
  0F, 0.1, 2.1, 0.1, 121.0, 5010.0,
  <!-- Data for signals #1, #2 & #3. Signal #4 unchanged -->
  07, 0.2, 2.2, 0.0, 122.0,
  0F, 0.3, 2.3, 0.1, 123.0, 5020.0,
  <!-- Data for signals #1, #2 & #4. Signal #3 unchanged -->
  0B, 0.4, 2.4, 0.0,        5040.0,
  0F, 0.5, 2.5, 0.1, 125.0, 5030.0,
  <!-- Data for signals #1, #3 & #4. Signal #2 unchanged -->
  0D, 0.6, 2.6,      124.0, 5025.0,
  0F, 0.7, 2.7, 0.2, 123.0, 5020.0,
  0F, 0.8, 2.8, 0.0, 122.0, 5010.0,
  0F, 0.9, 2.9, 0.1, 121.0, 5000.0,
  <!-- Data for signals #1 & #3. Signals #2 & #4 unchanged -->
  05, 1.0, 3.0,      120.0,
</dataTableDef>
```

Figure 10. An example of encoding time domain data.





```
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Sample Frequency Data Table
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Signals: #1 Alpha (deg) - Angle of attack

     (n+1)/2 samples with Fourier sequence representing an Odd number
     of time domain data entries.

     Note:: The sequence is truncated in this example
-->

<encodeDef encodeID=”encode_1” encoding=”UTF-8”>

<dataTableDef dataTabID="dataTab_3">
  <encodeRef encodeID=”encode_1”/>

  <!-- Real, Imag data -->
  -1.00069,  0.00000,
  -0.01860,  0.03845,
   0.03227,  0.01080,
   0.02103, -0.01227,
   -------,  -------,
   -------,  -------,
  -0.00575,  0.00211,
</dataTableDef>
```

Figure 11. An example of encoding frequency domain data.

Appendix

The following examples illustrate the syntax for various dynamic data elements. They indicate the extent of information that may be assigned to dynamic data records, together with its form. Examples are provided for a file header, a vehicle definition, a time history data table definition, an index table definition, signal definitions, data block definitions, and manoeuvre definitions.

```
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Sample File Header:
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<fileHeader>
  <author name="Geoff Brian" org="" email="">
    <address>
       An address here
    </address>
  </author>
  <creationDate date="2011-05-27"/>
  <description>
    This file is an example of encoding time sequenced data in the dynamic data
    aerospace vehicle modelling exchange syntax
  </description>
  <reference
    refID="REF_1"
    author="Geoff Brian"
    title="Time History for Aircraft Modelling Exchange Syntax"
    classification="Unclassified">
  </reference>
</fileHeader>

<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Vehicle Definitions : General information on the vehicle being
     tested and personnel
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<vehicleDef vehicleID="vehicle_1">
  <description>
    This vehicle definition element could contain technical reference
    details on items such as the vehicle type or designation; reference
    dimensions; vehicle's system components including propulsion, control
    and sensor systems, together with pilot/autopilot information
  </description>
  <aircraft name="TheWing" units="m"
    referenceLonLen="10" referenceLatLen="2" referenceArea="20">
    <description>A hypothetical aircraft</description>
  </aircraft>
  <pilot name="The guru" org="Guru Inc">
    <address>Unknown</address>
  </pilot>
</vehicleDef>

<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Data Table Definition: Sampled time sequenced data table
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     dataEntry+ : Signal list, Time_stamp, Signal(s) data (#PCDATA)

     Entries per line:
     1. Storing data for all signals
     2. Storing data for signal #4. Signals #1, #2 and #3 unchanged
     3. Storing data for signals #1, #2 and #3. Signal #4 unchanged
     4. Storing no data. Signals #1, #2, #3 and #4 unchanged
     5. Storing data for all signals
     Signal list, time, data
     1.   All      0.0   0.5, 25000.0, -39.0, -150.0
     2.   #4       0.2                         -150.56
     3. #1,#2,#3   0.4   0.6, 24500.0, -39.5
     4.  None      0.6   None
     5.   All       1.0   0.7, 22000.0, -39.0, -150.0

     Note: In the following table a new line is used for each time record
           to improve readability. This would not be the case for a production
           file as the carriage return could cause miss-interpretation of data. -->

<encodeDef encodeID=”encode_Base64_precision_4Bytes”
           encoding=”Base64” bytesPerDataPt=”4”>
<encodeDef encodeID=”encode_Base16_precision_1Byte”
           encoding=”Base16” bytesPerDataPt=”1”>
<encodeDef encodeID=”encode_UTF-8” encoding=”UTF-8>

<dataTableDef dataTabID="dataTable_1">
  <!-- Encoding for signal list -->
  <signalListEncodeRef encodeID=”encode_Base16_precision_1Byte”>
  <!-- Encoding for signal data -->
  <encodeRef encodeID=”encode_Base64_precision_4Bytes”/>

  <!-- Signal list, Time_stamp, Signal(s) data -->
  0F0BPwAAAARsNQAAwhwAAAwxYAAA
  080CwxaPXA
  0B0DPxmZmARr9oAAwh4AAA
  000E
  0F10PzMzMgRqvgAAwhwAAAwxYAAA
</dataTableDef>

<!-- +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Index Table Definitions:
       This is used for random access of the data table where the
       position in table represents the location for the start of a
       time record based data including the signal list.
     +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<indexTableDef indexTabID="indexTable_1">
  <encodeRef encodeID=”encode_UTF-8”/> <!-- Encoding for position data -->
  <indexRecord recordIndex="0B" posInTable="0"/>
  <indexRecord recordIndex="0C" posInTable="28"/>
  <indexRecord recordIndex="0D" posInTable="38"/>
  <indexRecord recordIndex="0E" posInTable="60"/>
  <indexRecord recordIndex="10" posInTable="64"/>
</indexTableDef>

<!-- +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Signal Definitions:
     +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<signalDef name="mach" sigID="mach" units="nd">
  <description>Mach number</description>
</signalDef>
<signalDef name="geodeticLatitude" sigID="geodeticLatitude" units="deg">
  <description>Geodetic Latitude</description>
</signalDef>
<signalDef name="longitude" sigID="longitude" units="deg">
  <description>Longitude</description>
</signalDef>
<signalDef name="pressureAltitude" sigID="pressureAltitude" units="ft">
  <description>Pressure Altitude</description>
</signalDef>
```

```
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Data Block Definitions: used by Manoeuvre Definition
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<!-- Sample Data Block -->
<dataBlockDef dataBlkID="dataBlk_1">
  <signalRefs>
    <signalRef sigID="mach"/>
    <signalRef sigID="pressureAltitude"/>
    <signalRef sigID="geodeticLatitude"/>
    <signalRef sigID="longitude"/>
  </signalRefs>

  <timeDomainDef timeDomainID="timeDef_1">
    <!-- time = (Time_stamp – keyOffset)*timeScale – baseTime -->
    <timeKey baseTime="0.2" timeScale="0.2" keyOffset="10"
             timeZone="LOCAL">
      <!-- Encoding for Time_stamp data -->
      <encodeRef encodeID=”encode_Base16_precision_1Byte”>
    </timeKey>
    <recordSummary startRecord="0B" endRecord="10" numRecords="5"/>
    <indexTableRef indexTabID="indexTable_1"/>
  </timeDomainDef>

  <dataTableRef dataTabID="dataTable_1"/>

  <description> This is a sample time history data block </description>
</dataBlockDef>

<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Manoeuvre Definitions:
     ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<manoeuvreDef name="p1f1m2" manID="man_p1f1m2" time="00:00:10.00">
  <description>A manoeuvre performed by an aircraft</description>
  <creationDate date="2011-05-01"/>
  <vehicleRef vehicleID="vehicle_1"/>
  <dataBlockRef dataBlkID="dataBlk_1"/>
</manoeuvreDef>
</THAMESfunc>
```

Acknowledgments

The authors wish to acknowledge the assistance and encouragement of Bruce Hildreth of JF Taylor Inc. who, together with Bruce Jackson from NASA Langley Research Center, initiated the development of the ANSI/AIAA-S119-2011 Flight Dynamic Model Exchange Standard and the Dynamic Aerospace Exchange Markup Language, on which this work has been inspired. Acknowledgement is also extended to members of Air Vehicle Division from the Australian Defence Science and Technology Organisation, Dr. Dan Newman of Quantitative Aeronautics Pty. Ltd., and Mr. Rob Curtin of QinetiQ Consulting Australia, who have provided valuable assistance with developing the Dynamic Aerospace Exchange Markup Language.

References

[1]Jackson, E.B., “Dynamic Aerospace Vehicle Exchange Markup Language, (DAVE-ML) Reference Version 2.0.1”, AIAA Modeling and Simulation Technical Committee, URL: http://daveml.org.

[2]ANSI/AIAA-S119-2011 Flight Dynamics Model Exchange Standard, American Institute of Aeronautics and Astronautics, Reston, VA, USA, March 2011.

[3]World Wide Web Consortium (W3C) “W3C Recommendation: Extensible Markup Language (XML)”, URL: http://www.w3.org/TR/xml/, 2008-11-26

[4]World Wide Web Consortium (W3C) “Mathematical Markup Language (MathML) Version 2.0 (Second Edition)”, URL: http://www.w3.org/TR/MathML2/, 2003.

[5]John H. Hubbard, Barbara Burke Hubbard, *Vector Calculus, Linear Algebra and Differential Forms – A Unified Approach*, Prentice Hall, New Jersey, 1999.

[6]Benjamin C. Kuo, *Automatic Control Systems*, 5th ed., Prentice Hall, New Jersey, 1987.

[7]Cerf V., “ASCII format for Network Exchange”, Internet Engineering Task Force (IETF), RFC 20, URL: http://tools.ietf.org/html/rfc20, October 1969

[8]Josefsson S., “The Base16, Base32, and Base64 Data Encodings”, Internet Engineering Task Force (IETF), RFC 4648, URL: http://tools.ietf.org/html/rfc4648, October 2006

[9]IEEE Std 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI) 1-55937-653.8 [SH10116-NYF].