



**DOCUMENT 127-17**

**DATA DISPLAY MARKUP LANGUAGE (DDML) HANDBOOK**

**ABERDEEN TEST CENTER  
DUGWAY PROVING GROUND  
REAGAN TEST SITE  
REDSTONE TEST CENTER  
WHITE SANDS MISSILE RANGE  
YUMA PROVING GROUND**

**NAVAL AIR WARFARE CENTER AIRCRAFT DIVISION  
NAVAL AIR WARFARE CENTER WEAPONS DIVISION  
NAVAL UNDERSEA WARFARE CENTER DIVISION, KEYPORT  
NAVAL UNDERSEA WARFARE CENTER DIVISION, NEWPORT  
PACIFIC MISSILE RANGE FACILITY**

**30TH SPACE WING  
45TH SPACE WING  
96TH TEST WING  
412TH TEST WING  
ARNOLD ENGINEERING DEVELOPMENT COMPLEX**

**NATIONAL AERONAUTICS AND SPACE ADMINISTRATION**

**DISTRIBUTION A: APPROVED FOR PUBLIC RELEASE,  
DISTRIBUTION IS UNLIMITED**

This page intentionally left blank

**DOCUMENT 127-17**

**DATA DISPLAY MARKUP LANGUAGE (DDML) HANDBOOK**

**January 2017**

**Prepared by**

**Telemetry Group**

**Published by**

**Secretariat  
Range Commanders Council  
White Sands Missile Range  
New Mexico 88002-5110**

This page intentionally left blank.

## Table of Contents

<b>Preface</b> .....	<b>v</b>
<b>Acronyms</b> .....	<b>vii</b>
<b>Chapter 1. Introduction</b> .....	<b>1-1</b>
1.1 The Range Commanders Council (RCC) and DDML.....	1-1
1.2 eXtensible Markup Language.....	1-5
1.3 DDML Schema.....	1-7
<b>Chapter 2. Getting Started with a Simple Example</b> .....	<b>2-1</b>
2.1 A Simple Data Display Example.....	2-1
2.2 A Description of the Simple Example.....	2-1
2.3 The “Look and Feel” of DDML.....	2-2
2.4 Display Object Containers.....	2-2
2.5 Display Object Common Components.....	2-3
2.6 Parameters.....	2-6
2.7 Data Display Container Custom Parameters.....	2-7
2.8 Data Source Pool Definitions.....	2-7
2.9 Data Variable Pool Definitions.....	2-8
<b>Chapter 3. General Structure, Semantics, and the Display Object Group</b> .....	<b>3-1</b>
3.1 Layered Structure.....	3-1
3.2 Display Object Group.....	3-2
<b>Chapter 4. DDML Translation</b> .....	<b>4-1</b>
4.1 Translator Development Methodology.....	4-1
4.2 Development of External Translators.....	4-3
4.3 Development of Internal Translators.....	4-4
<b>Appendix A. Citations</b> .....	<b>A-1</b>

## List of Figures

Figure 1-1. A Sample Data Display System.....	1-2
Figure 1-2. Code Development Effort for Translators Needed for Current System: $O(n^2)$ ...	1-3
Figure 1-3: Code Development Effort for Translators to and from DDML: $O(n)$ . ....	1-4
Figure 1-4. XML Snippet.....	1-5
Figure 1-5. Components of an XML Element.....	1-6
Figure 1-6. Example Schema Diagram.....	1-7
Figure 1-7. High-Level DDML Schema Diagram.....	1-8
Figure 2-1. Simple Example Illustration.....	2-1
Figure 2-2. Look and Feel Example.....	2-2
Figure 2-3. Container Definitions.....	2-3
Figure 2-4. Common Display Object Schema.....	2-4
Figure 2-5. Common Name and Location Definitions.....	2-4

Figure 2-6.	Common Title Definitions .....	2-4
Figure 2-7.	Common Rule XML Schema.....	2-5
Figure 2-8.	IF/THEN Rule Definition .....	2-5
Figure 2-9.	Specific Type Definitions .....	2-6
Figure 2-10.	Miscellaneous Common XML Schema .....	2-6
Figure 2-11.	Custom Parameter Schema .....	2-7
Figure 2-12.	Data Display Container Custom Parameter Example XML .....	2-7
Figure 2-13.	Data Display Container Custom Parameter Example XML Schema .....	2-7
Figure 2-14.	Data Source Pool Example XML .....	2-8
Figure 2-15.	Data Source Pool Example XML Schema.....	2-8
Figure 2-16.	Data Variable Pool Example XML.....	2-9
Figure 2-17.	Data Variable Pool Example XML Schema .....	2-9
Figure 3-1.	DDML Layered Structure .....	3-1

### **List of Tables**

Table 3-1.	The Display Object Group .....	3-2
Table 4-2.	DDML Data Dictionary Sample .....	4-1

## Preface

This standard was prepared by the Data Multiplex Committee of the Telemetry Group (TG), Range Commanders Council (RCC). The DDML Handbook is a common reference for use by organizations that produce DDML files, by ranges that receive DDML files, and by vendors who incorporate DDML files into their telemetry processing systems. The use of this handbook will eliminate inconsistencies and differing interpretations of DDML files so that all parties will benefit from its usage.

The RCC gives special acknowledgement for production of this document to the TG Data Multiplex Committee. Please direct any questions to the committee point of contact or to the RCC Secretariat as shown below.

Telemetry Group Chairman: Mr. Jon Morgan  
412 TW, Edwards AFB  
Bldg 1408 Room 5  
301 East Yeager  
Edwards AFB, CA 93524  
Phone: DSN 527-8942      Com (661) 277-8942  
Fax: DSN 527-8933      Com (661) 277 8933  
email [jon.morgan.2.ctr@us.af.mil](mailto:jon.morgan.2.ctr@us.af.mil)

Secretariat, Range Commanders Council  
ATTN: CSTE-WS-RCC  
1510 Headquarters Avenue  
White Sands Missile Range, New Mexico 88002-5110  
Phone: DSN 258-1107      Com (575) 678-1107  
Fax: DSN 258-7519      Com (575) 678-7519  
email [usarmy.wsmr.atec.list.rcc@mail.mil](mailto:usarmy.wsmr.atec.list.rcc@mail.mil)

This page intentionally left blank.



## Acronyms

API	application programming interface
DDML	Data Display Markup Language
DOM	document object model
DTD	document type definition
IRIG	Inter-range Instrumentation Group
MathML	Mathematical Markup Language
RCC	Range Commanders Council
SAX	simple API for XML
SVG	Scalable Vector Graphics
T&E	test and evaluation
TG	Telemetry Group
TM	telemetry
TMATS	Telemetry Attributes Transfer Standard
W3C	Worldwide Web Consortium
XML	eXtensible Markup Language

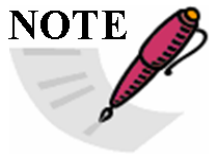
This page intentionally left blank.

## CHAPTER 1

### Introduction

This Data Display Markup Language (DDML) Handbook is intended to supplement Chapter 9 of RCC IRIG 106 Telemetry Standards.<sup>1</sup> Practical guidance for properly generating and using DDML files is provided and examples of some of the more commonly used DDML features are given. Since there may be multiple ways of describing these features in DDML, the examples are intended to illustrate “best practices.” The overall purpose of this handbook is to improve the use of DDML as a standard by presenting clear guidelines and thereby eliminating any misinterpretations that may exist.

The RCC IRIG 106 sets forth standards for various aspects of telemetry (TM) data transmission, recording, and processing. These standards constitute a guide for the orderly implementation of TM systems and provide the necessary criteria on which to base equipment design and modification. Their purpose is to ensure efficient spectrum utilization, interference-free operation, interoperability between ranges, and compatibility of range user equipment at the ranges.

 <p><b>NOTE</b></p>	<p>The RCC IRIG 106 is the master source of all information for TM data transmission, recording, and processing. Therefore, the RCC IRIG 106 is assumed to be correct if a discrepancy is found between it and this handbook. If a discrepancy is found, it should be immediately reported to the RCC Secretariat or to the Telemetry Group (TG). The RCC IRIG 106 can be viewed or downloaded from the RCC public web site, <a href="http://www.wsmr.army.mil/RCCsite/Pages/default.aspx">http://www.wsmr.army.mil/RCCsite/Pages/default.aspx</a>.</p>
---	---

#### 1.1 The Range Commanders Council (RCC) and DDML

The RCC held its first meeting in August 1951. In March 1952, the RCC Commanders established the Inter-Range Instrumentation Group (IRIG) to make recommendations for improvement of range instrumentation and conservation of the resources of the ranges. After a few meetings, the IRIG recognized the need to expand and specialize, and the IRIG Steering Committee was created to oversee several IRIG technical working groups. In 1971, the IRIG Steering Committee was disbanded, and the IRIG working groups became known as the RCC working groups. To this day, the RCC standards documents are still commonly referred to as IRIG standards.

Data display is a critical component for test and evaluation (T&E) environments in aircraft, space, and energy systems under test. The TM functions associated with these systems produce too much data for a single person to comprehend as alphanumeric information. Displays ease the task of interpreting raw measurands faster than the eye can fathom. Moreover, they depict when measurands are within safe and meaningful limits, show relationships between measurands, and spot trends. To assist with these efforts, data display systems provide a wide

---

<sup>1</sup> Range Commanders Council. . “Telemetry Attributes Transfer Standard,” in *Telemetry Standards*. IRIG 106-15. July 2015. May be superseded by update. Retrieved 1 July 2015. Available at [http://www.wsmr.army.mil/RCCsite/Documents/106-15\\_Telemetry\\_Standards/Chapter9.pdf](http://www.wsmr.army.mil/RCCsite/Documents/106-15_Telemetry_Standards/Chapter9.pdf).

variety of customizable display objects, including strip charts, bar charts, vertical meters, round gauges, cross plots, tabular displays, orientation displays, and bit maps. Each display type can be tailored with respect to size, foreground and background colors, fonts, grids, time, and data format to name a few.

Each data display object has peculiarities of its own. Not only is there a wide range of parameters and attributes, but also these values are often a function of the state of the data that they display. For example, the attributes of an object can change as the color of a curve or numeric value changes when a measurand approaches a limit or is out of a limit. In addition to processing algorithms that detect changes, large time scales make it easier to visualize trends. Dynamic 3-D models of objects under testing can be used to show orientation, as opposed to interpreting a table of numeric orientation values. Multiple objects can be grouped into a single window to form instrument panels. Windows can be created for a test plan that is used over and over either with the same measurands and processed parameters or with new ones as required. Measurands and parameters can be changed in real time. Similarly, attributes such as data limits can also be changed. Standard drawing and graphics tools may be used in creating process diagrams and embellishing control panels. Snapshots of events can be sent to color printers or saved to disk for inclusion in reports. Features such as local disk and ring buffers that are associated with video displays and are independent of system archiving give operators the ability to recreate data leading to an event of interest.

This description illustrates how complex a singular data display system can be. To compound this situation, there is a variety of vendors offering software packages for data acquisition and display with such features, each requiring its own data display specification. [Figure 1-1](#) shows a snapshot of a data display.

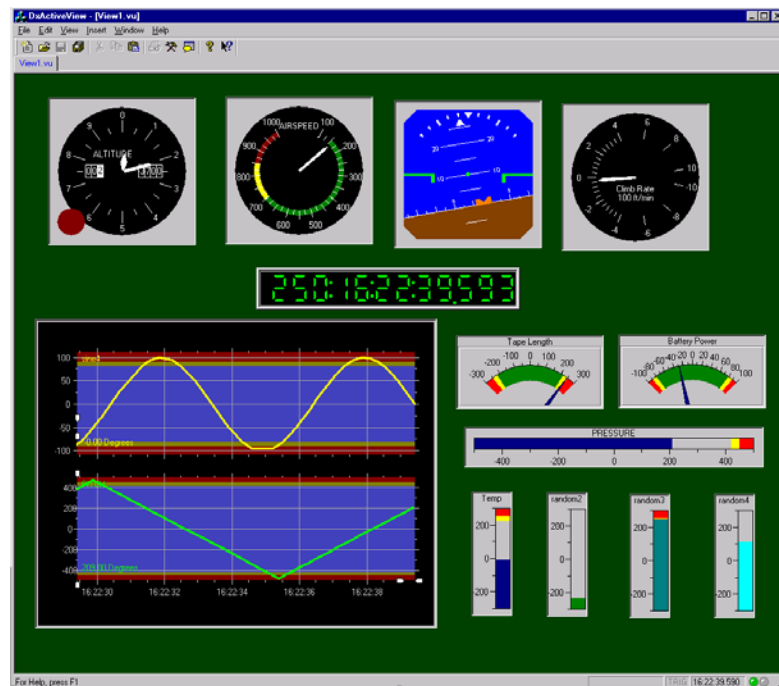


Figure 1-1. A Sample Data Display System

Use of Telemetry Attributes Transfer Standard (TMATS) (Range Commanders Council 2015) is an increasingly popular method for transferring files between non-compatible systems. Because each system uses a different internal format, translators are required to convert data to and from the TMATS intermediate format. The purpose of TMATS is to provide a common format for the transfer of information between the user and a test range or between different ranges. This format will minimize the activities unique to stations that are necessary to support any test item. In addition, the format is intended to eliminate the labor-intensive process currently required to reformat the information by providing the information on computer-compatible media, thus reducing errors and requiring less preparation time for test support.

Even though TMATS provides a powerful means for transferring TM information, it does not provide any support for capturing display objects and their layout for systems that require common data displays. Moreover, the tendency of T&E is towards a plug-and-play-like data acquisition system that requires standard languages and modules for data displays. Currently, the only way to transfer data displays between display applications is to manually recreate displays using an experienced programmer. Absence of a neutral format also requires the programmer to manually craft the display transfers between each system pair in the application domain.

For example, a T&E system of six data display systems requires 30 unidirectional data display transfers, as shown in [Figure 1-2](#). This can be formulated as  $n(n-1)$  transfers, where  $n$  is the number of applications. Also, since there are no automatic translators between display systems, a small change in one of the systems requires manual changes in the other related transfers.

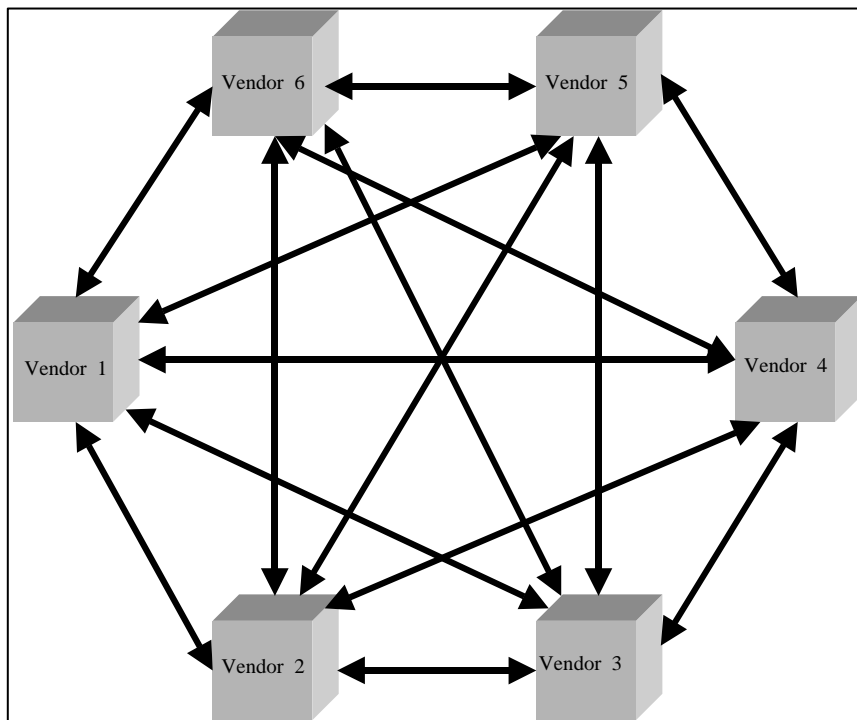


Figure 1-2. Code Development Effort for Translators Needed for Current System:  $O(n^2)$

The DDML standard is a specification of an eXtensible Markup Language (XML)-based neutral format that is intended to be the inter-lingua of data displays. The DDML format has the requirement of being generic enough to encompass various vendor-specific data display formats and at the same time being unified (not a loose grouping of XML-ized vendor formats). In addition, it is required to support reusable concepts (such as variables and data sources), be robust (e.g., use of cross-references), and support future objects without warranting a change of the DDML format.

Availability of DDML as the inter-lingua drastically reduces the number of unidirectional translators to two per vendor-specific format. Returning to the T&E system of six data display systems, we would require 12 unidirectional display transfers, as shown in [Figure 1-3](#). In general, for a system of  $n$  formats, the translator development effort is  $O(n)$ . Also, the task of developing translators would be highly simplified because of two reasons. First, because DDML is an XML format, there is ample support by way of free software to parse and generate DDML. Second, there is a high degree of reuse of a number of translator components because of the new hub-and-spoke translator framework. Our proposed solution, therefore, includes the development of highly reusable, customizable, well-documented translator components along with well-documented end-to-end processes for rapid translator development. This translator framework will then be used to develop the bi-directional translators between DDML and the vendor formats. As a result, DDML along with the system of translators is a practical and cost-effective solution to the current manual method of recreating displays in the target format.

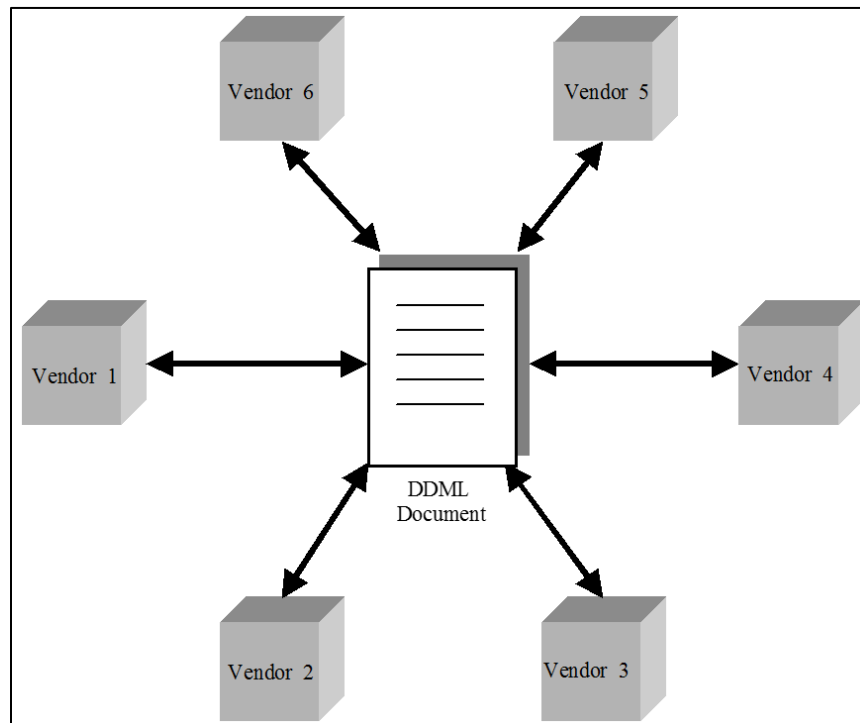


Figure 1-3 Code Development Effort for Translators to and from DDML:  $O(n)$ .

Changes in one of the vendor formats would require re-coding only in the translators between that format and the neutral format. While the effort to accommodate the changes is

mitigated by the design of modular components, we can also *automatically generate* key components of the translator code that can be easily compiled, tested, and deployed. This capability will significantly mitigate the effort to keep up with “moving targets” or evolution of source/target vendor languages because now the focus of the effort will be on modeling the data display specification and not on the translator code development. In that sense, it will be similar to using Computer-Assisted Software Engineering tools to develop object-oriented software and automatically generate the target code for compiling.

Files in the DDML format are usually produced and read by software. Automating this process reduces the time needed to prepare a data display configuration and eliminates errors that inevitably result from entering this information manually.

## 1.2 eXtensible Markup Language

The XML specification is produced by the World Wide Web Consortium (W3C)<sup>2</sup> whose original intent was to provide a standard, machine-readable format for describing documents. Because of its popularity, wide adoption, and prevalence on the Internet, its use has expanded to describe arbitrary data structures such as web services and T&E metadata. Here we provide a brief overview of XML. More detail can be found in the RCC Style Guide.<sup>3</sup>

The example in [Figure 1-4](#) shows a portion of an XML document (an XML snippet).

```
<ddmlbarchart:barchart id="BAGC1">
  <ddmlcommon:name>BAGC1</ddmlcommon:name>
  <ddmlcommon:point>
    <ddmlcommon:x>500000</ddmlcommon:x>
    <ddmlcommon:y>0</ddmlcommon:y>
  </ddmlcommon:point>
  <ddmlcommon:point>
    <ddmlcommon:x>500000</ddmlcommon:x>
    <ddmlcommon:y>250000</ddmlcommon:y>
  </ddmlcommon:point>
  <ddmlcommon:point>
    <ddmlcommon:x>750000</ddmlcommon:x>
    <ddmlcommon:y>250000</ddmlcommon:y>
  </ddmlcommon:point>
  <ddmlcommon:point>
    <ddmlcommon:x>750000</ddmlcommon:x>
    <ddmlcommon:y>0</ddmlcommon:y>
  </ddmlcommon:point>
  <ddmlcommon:titleFont>Arial</ddmlcommon:titleFont>
  <ddmlcommon:title>Bar Chart</ddmlcommon:title>
  <ddmlcommon:titleColor> 65280</ddmlcommon:titleColor>
  <ddmlcommon:titleFontSize>24</ddmlcommon:titleFontSize>
  <ddmlcommon:scrollDirection>DOWN</ddmlcommon:scrollDirection>
</ddmlbarchart:barchart>
```

Figure 1-4. XML Snippet

<sup>2</sup> <http://www.w3.org/>

<sup>3</sup> Range Commanders Council. *XML Style Guide*. RCC 125-15. July 2015. Retrieved 13 January 2017. Available at [http://www.wsmr.army.mil/RCCsite/Documents/125-15\\_XML\\_Style\\_Guide/](http://www.wsmr.army.mil/RCCsite/Documents/125-15_XML_Style_Guide/).

In XML, each piece of data, or element, is surrounded by a “tag” such as `<ddmlbarchart:barchart>` and `<ddmlcommon:point>`. The structure of an XML file is such that tags can be enclosed in other tags to an arbitrary depth (`<ddmlcommon:x>` is a sub-element of `<ddmlcommon:point>`, `<ddmlcommon:title>` is a sub-element of `<ddmlbarchart:barchart>`, etc.). This is the basic idea behind the structure of an XML document.

The component parts of an XML element are identified in [Figure 1-5](#). Each of these components is defined below the figure.

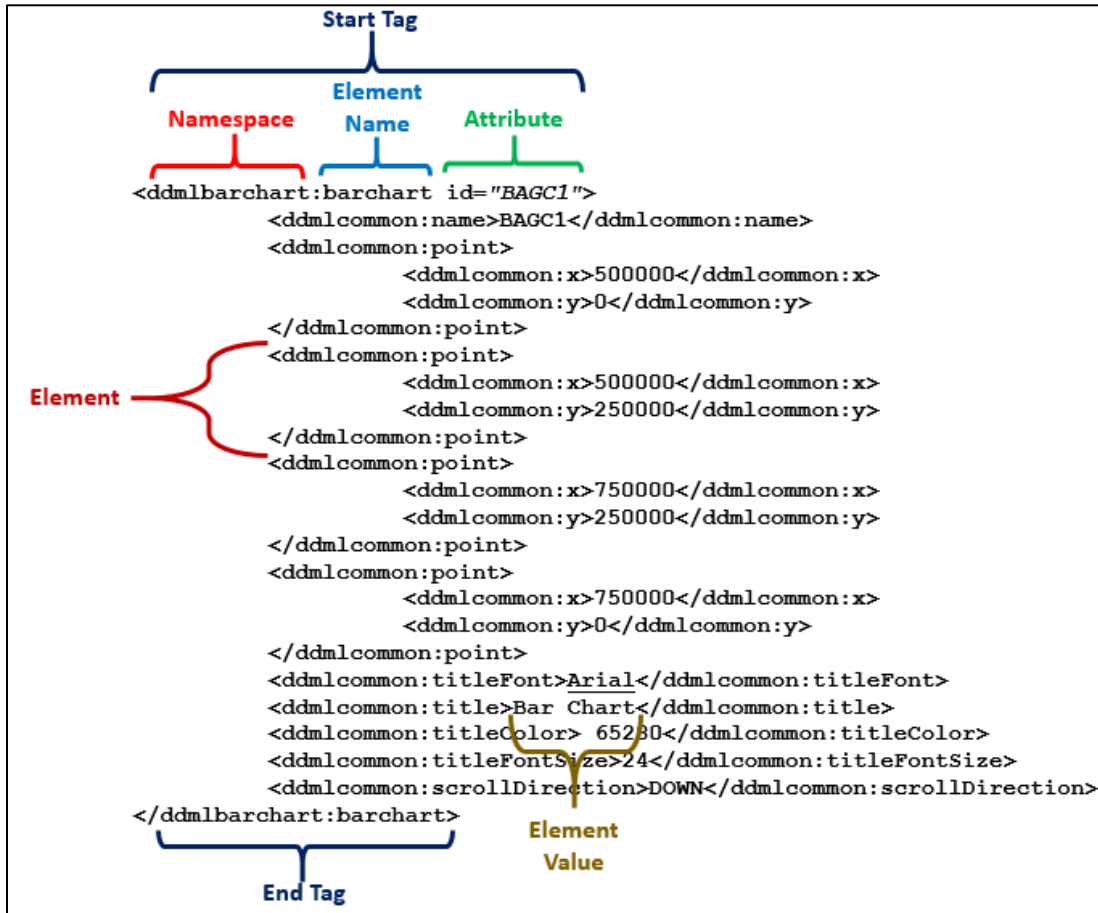


Figure 1-5. Components of an XML Element

- **Element:** “Element” is the term used to define a complete unit of XML information. It begins with a start tag and ends with an end tag. The value of an element can be a simple value or one or more sub-elements (children).
- **Start Tag:** The start tag identifies the beginning of the element and consists of the element’s name (and possibly a namespace and attributes) included between a “<” and a “>” symbol.
- **End Tag:** The end tag identifies the end of the element and looks identical to the start tag, except it includes a “/” (forward-slash) after the “<” symbol. An end tag does not contain attributes.



- **Namespace:** The namespace is optional in XML, but can be used to define the scope within which the element is defined. In our TMATS example, we define a “ddmlbarchart” namespace (for bar chart display objects) and make all of the elements for describing bar charts members of it.
- **Element name:** The name of the element is what appears in the start and end tags and is what actually identifies the piece of information being defined.
- **Attribute:** Attributes are another method of associating information with an XML element. An attribute consists of a name followed by a “=” sign followed by a value enclosed in quotes.
- **Element value:** The value of the element is everything that lies between the start tag and the end tag. The value can either be a single value (e.g. 7, “John”, true, etc.) or a collection of one or more sub-elements (children).

An XML schema is a design document used to describe a specific language that is based on XML. The rules for formatting proper XML are very simple and unrestricted. A schema defines which element names are valid, which elements can have which children, and which values are valid for each element. Types organize XML documents by defining the allowed structure for specific groupings of elements.

Even though an XML schema is itself a document, it is usually more useful to view the schema as a diagram. In this representation, boxes represent XML schema complex types (types that have structure). The bold title in the box is the name of the type. Inside the box, attributes are shown under the *Attributes* heading and simple types (types that do not have structure) are shown under the *Elements* heading. Elements that are complex types are represented by a line that connects the element with its type. The line is labeled with the name of the element. [Figure 1-6](#) contains an example schema diagram.

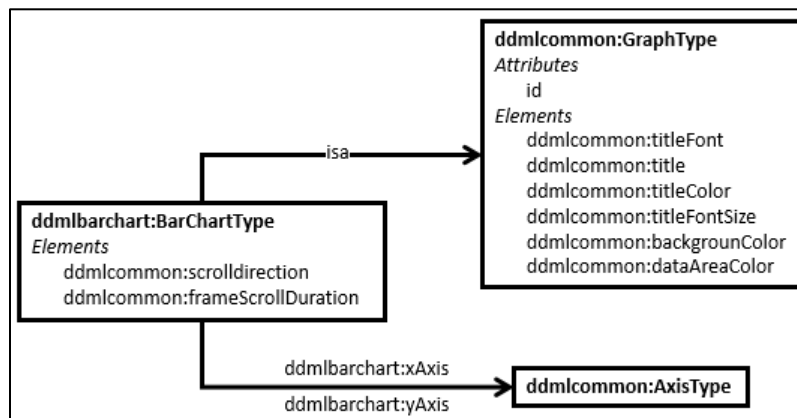


Figure 1-6. Example Schema Diagram

### 1.3 DDML Schema

[Figure 1-7](#) shows a diagram of the DDML schema at a high level. The diagram shows the core items of DDML including projects, models, pools, graphics resources, and generic parameters (the “param” element).

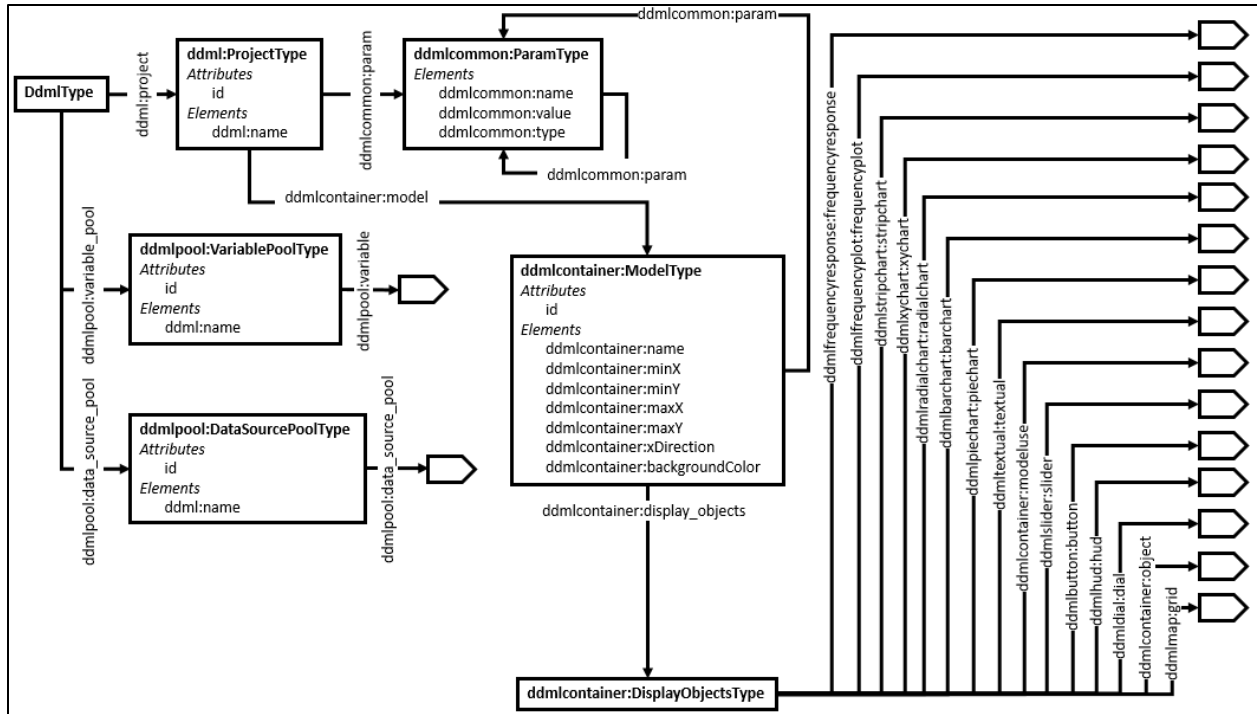


Figure 1-7. High-Level DDML Schema Diagram

A DDML document consists of a project description (element `ddml:project` of type `ddml:ProjectType`), a list of variables (element `ddmlpool:variable_pool` of type `ddmlpool:VariablePoolType`), and a list of data sources (element `ddmlpool:data_source_pool` of type `ddmlpool:DataSourcePoolType`). Each project can have one or more custom parameters (element `ddmlcommon:param` of type `ddmlcommon:ParamType`) and a model description (element `ddmlcontainer:model` of type `ddmlcontainer:ModelType`). Each model contains a description of the specific display objects.

## CHAPTER 2

### Getting Started with a Simple Example

#### 2.1 A Simple Data Display Example

In order to demonstrate the use of DDML, a notional data display is shown in [Figure 2-1](#). In the following sections, all the DDML constructs that could be utilized to store information on this data display will be described.

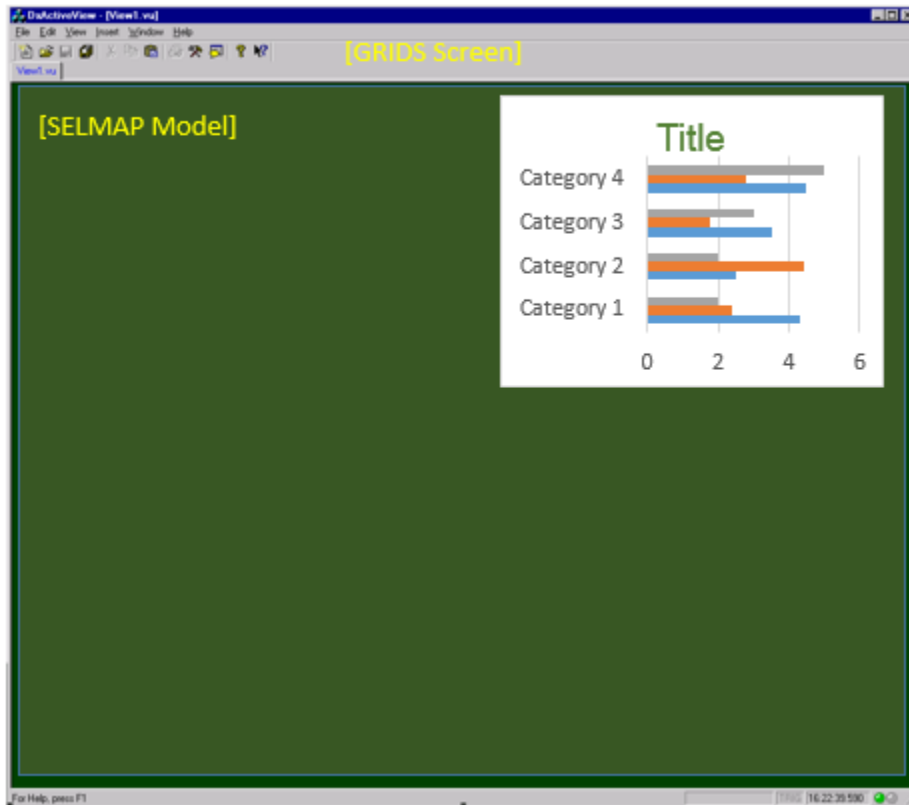


Figure 2-1. Simple Example Illustration

#### 2.2 A Description of the Simple Example

The display is extremely basic with just a single bar chart object; however, the notional example requires all the fundamental building blocks of DDML: projects, models, and graphic resources. The display contains a horizontally oriented bar chart object stored inside a model that is located on the screen. These display attributes will all need to be stored in DDML. Additionally, the orientation and location for the model and bar chart will need to be captured. It is important to note that all graphic resource locations must be contained within the bounds of their parent model.

**A note about DDML Graphics Resources:** Since graphics resources are stored inside models, graphic resource locations are restricted to the bounds of their parent model.

## 2.3 The “Look and Feel” of DDML

An example XML instance document that implements the bar chart data display example is shown in [Figure 2-2](#) to demonstrate the look and feel of DDML. This shows a DDML instance that contains all the minimal constructs needed to validate a DDML file: a project, a model, a display object container, and a display object (bar chart). Some of the elements are minimized for clarity. In the following sections, the instance document will be broken down and explained.

```

<ddml:project id="PROJ1">
  <ddml:name>Schema Validation Task</ddml:name>
  <ddmlcontainer:model id="MOD1">
    <ddmlcontainer:name>SELMAP</ddmlcontainer:name>
    <ddmlcontainer:minX>0</ddmlcontainer:minX>
    <ddmlcontainer:minY>0</ddmlcontainer:minY>
    <ddmlcontainer:maxX>1300000</ddmlcontainer:maxX>
    <ddmlcontainer:maxY>1000000</ddmlcontainer:maxY>
    <ddmlcontainer:xDirection>RIGHT</ddmlcontainer:xDirection>
    <ddmlcontainer:yDirection>DOWN</ddmlcontainer:yDirection>
    <ddmlcommon:param>
      <ddmlcommon:name>Screen</ddmlcommon:name>
      <ddmlcommon:value>GRIDS</ddmlcommon:value>
    </ddmlcommon:param>
    <ddmlcontainer:display_objects>
      <ddmlbarchart:barchart id="BAGC1">
        <ddmlcommon:name>BAGC1</ddmlcommon:name>
        <ddmlcommon:dynamics/>
        <ddmlcommon:titleFont>Arial</ddmlcommon:titleFont>
        <ddmlcommon:title>Bar Chart</ddmlcommon:title>
        <ddmlcommon:titleColor> 65280</ddmlcommon:titleColor>
        <ddmlcommon:titleFontSize>24</ddmlcommon:titleFontSize>
        <ddmlbarchart:xAxis/>
        <ddmlbarchart:yAxis/>
        <ddmlcommon:scrollDirection>DOWN</ddmlcommon:scrollDirection>
      </ddmlbarchart:barchart>
    </ddmlcontainer:display_objects>
  </ddmlcontainer:model>
</ddml:project>
<ddmlpool:variable_pool>
  <ddmlpool:variable id="var0">
    <ddmlcommon:name>IRIG_DAYS</ddmlcommon:name>
    <ddmlpool:data_source_ref>S1SRP</ddmlpool:data_source_ref>
  </ddmlpool:variable>
</ddmlpool:variable_pool>
<ddmlpool:data_source_pool>
  <ddmlpool:data_source id="S1SRP">
    <ddmlcommon:name>S1SRP</ddmlcommon:name>
  </ddmlpool:data_source>
</ddmlpool:data_source_pool>

```

Figure 2-2. Look and Feel Example

## 2.4 Display Object Containers

All display objects in DDML must be stored within a set of containers. In particular, each object must minimally have a model parent with a corresponding project parent. Additionally, grid and map containers can be utilized inside models to add more storing features

to the models. Essentially, grids and maps act as containers inside a container. The following describes the display object containers available in DDML.

- **Project.** For each DDML instance, a project definition is needed to contain all display object information. A project is basically a simple grouping mechanism so that all related data display information can be stored together.
- **Model.** A model is the starting point for storing data display information. Models contain one or more data display objects or containers and constrain the location and direction of those objects. A model is required under the project element of a DDML instance document.
- **Grid.** Grids are used within a model to add additional features for arranging display objects. Grid rows and columns can be defined and display objects can be placed within specific rows or columns to space them evenly.
- **Map.** Maps are display object containers than have an image, typically of a map, as the background. This allows display objects to be shown on top of a map.

For the bar chart example, the project “PROJ1” is defined with the name “Schema Validation Task.” A model “MOD1” is then placed inside the project with the name “SELMAP” and location and orientation are defined. [Figure 2-3](#) shows the XML snippet where these definitions are stored.

```
<ddml:project id="PROJ1">
  <ddml:name>Schema Validation Task</ddml:name>
  <ddmlcontainer:model id="MOD1">
    <ddmlcontainer:name>SELMAP</ddmlcontainer:name>
    <ddmlcontainer:minX>0</ddmlcontainer:minX>
    <ddmlcontainer:minY>0</ddmlcontainer:minY>
    <ddmlcontainer:maxX>1300000</ddmlcontainer:maxX>
    <ddmlcontainer:maxY>1000000</ddmlcontainer:maxY>
    <ddmlcontainer:xDirection>RIGHT</ddmlcontainer:xDirection>
    <ddmlcontainer:yDirection>DOWN</ddmlcontainer:yDirection>
  </ddmlcontainer:model>
</ddml:project>
```

Figure 2-3. Container Definitions

## 2.5 Display Object Common Components

In general, display objects all have common properties and features such as titles, fonts, and color. For this reason, the DDML common schema can be utilized for all display objects. [Figure 2-4](#) shows the common schema for display objects

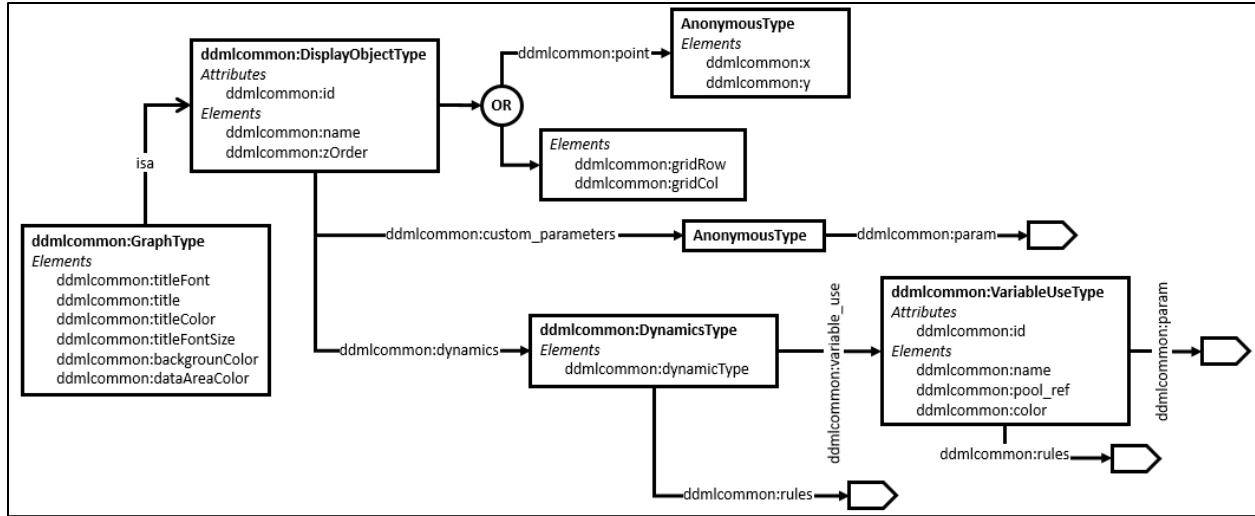


Figure 2-4. Common Display Object Schema

In the example DDML, common parameters are defined in order to place the bar chart in a specific location on the screen as well as define the name of the bar chart (see [Figure 2-5](#)). Additionally, a title name, text color, font, and font size are defined (see [Figure 2-6](#)).

```
<ddmlbarchart:barchart id="BAGC1">
  <ddmlcommon:name>BAGC1</ddmlcommon:name>
  <ddmlcommon:point>
    <ddmlcommon:x>500000</ddmlcommon:x>
    <ddmlcommon:y>0</ddmlcommon:y>
  </ddmlcommon:point>
  <ddmlcommon:point>
    <ddmlcommon:x>500000</ddmlcommon:x>
    <ddmlcommon:y>250000</ddmlcommon:y>
  </ddmlcommon:point>
  <ddmlcommon:point>
    <ddmlcommon:x>750000</ddmlcommon:x>
    <ddmlcommon:y>250000</ddmlcommon:y>
  </ddmlcommon:point>
  <ddmlcommon:point>
    <ddmlcommon:x>750000</ddmlcommon:x>
    <ddmlcommon:y>0</ddmlcommon:y>
  </ddmlcommon:point>
</ddmlbarchart:barchart>
```

Figure 2-5. Common Name and Location Definitions

```
<ddmlbarchart:barchart id="BAGC1">
  <ddmlcommon:name>BAGC1</ddmlcommon:name>
  <ddmlcommon:titleFont>Arial</ddmlcommon:titleFont>
  <ddmlcommon:title>Bar Chart</ddmlcommon:title>
  <ddmlcommon:titleColor> 65280</ddmlcommon:titleColor>
  <ddmlcommon:titleFontSize>24</ddmlcommon:titleFontSize>
</ddmlbarchart:barchart>
```

Figure 2-6. Common Title Definitions.

[Figure 2-7](#) shows the common schema for if/then/else rules.

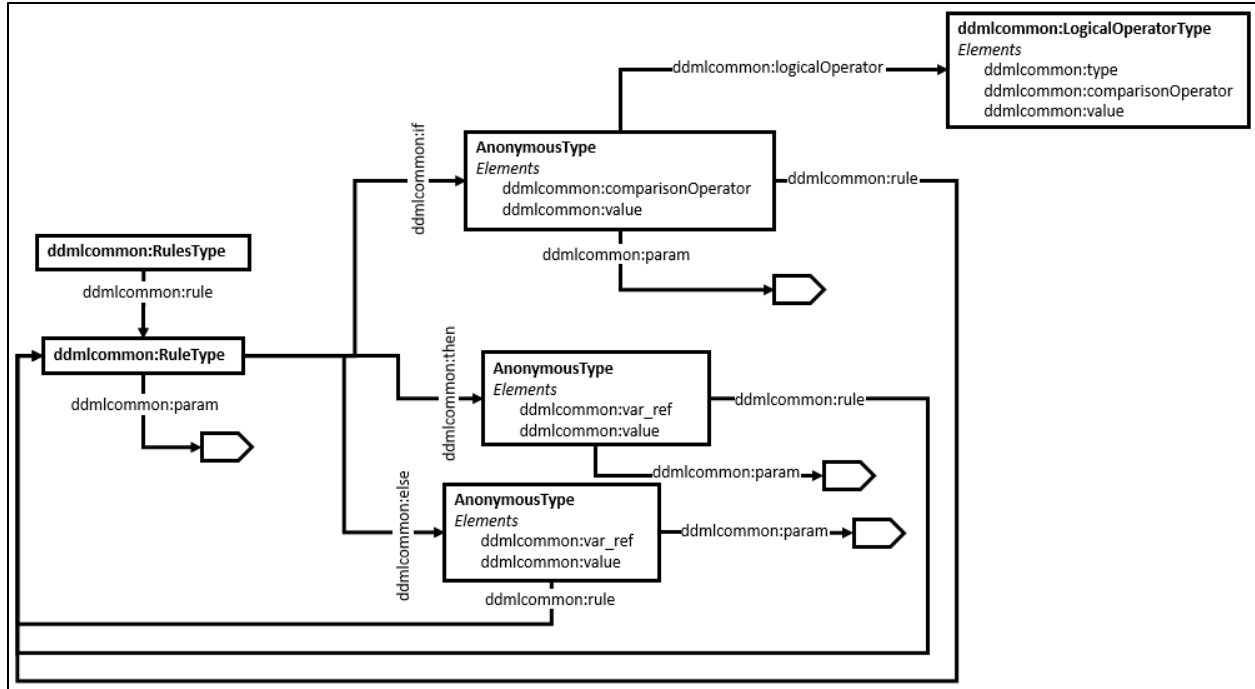


Figure 2-7. Common Rule XML Schema

In the example DDML, a rule is introduced to set the value of a specific variable to 0 whenever the value of that variable is negative (see [Figure 2-8](#)). In addition, this rule sets the color of the variable value in the display.

```

<ddmlbarchart:barchart id="BAGC1">
  <ddmlcommon:dynamics>
    <ddmlcommon:dynamicType>builtin</ddmlcommon:dynamicType>
    <ddmlcommon:variable_use id="vu0">
      <ddmlcommon:name>TMLF001F</ddmlcommon:name>
      <ddmlcommon:pool_ref>var19</ddmlcommon:pool_ref>
      <ddmlcommon:color>65280</ddmlcommon:color>
    </ddmlcommon:variable_use>
    <ddmlcommon:rules>
      <ddmlcommon:rule>
        <ddmlcommon:if>
          <ddmlcommon:comparisonOperator>LT</ddmlcommon:comparisonOperator>
          <ddmlcommon:value>0</ddmlcommon:value>
        </ddmlcommon:if>
        <ddmlcommon:then>
          <ddmlcommon:value>0</ddmlcommon:value>
        </ddmlcommon:then>
      </ddmlcommon:rule>
    </ddmlcommon:rules>
  </ddmlcommon:dynamics>
</ddmlbarchart:barchart>

```

Figure 2-8. IF/THEN Rule Definition

Type-specific elements must be stored for specific graphics resources. Type-specific elements are used when specific graphics resources have unique attributes such as formatting of

a text object or axis for different types of graphics. In [Figure 2-9](#), the type-specific elements of “xAxis” and “yAxis” are defined for the bar chart as well as the common elements for each of the axes.

```

<ddmlbarchart:barchart id="BAGC1">
  <ddmlbarchart:xAxis>
    <ddmlcommon:axisType>TIME</ddmlcommon:axisType>
    <ddmlcommon:axisColor>0</ddmlcommon:axisColor>
  </ddmlbarchart:xAxis>
  <ddmlbarchart:yAxis>
    <ddmlcommon:axisType>VALUE</ddmlcommon:axisType>
    <ddmlcommon:axisMin>0</ddmlcommon:axisMin>
    <ddmlcommon:axisMax>5</ddmlcommon:axisMax>
    <ddmlcommon:axisLabelForegroundColor>
      0
    </ddmlcommon:axisLabelForegroundColor>
    <ddmlcommon:axisGrid>
    <ddmlcommon:axisGridInterval>2.5</ddmlcommon:axisGridInterval>
    <ddmlcommon:axisGridColor>16711935</ddmlcommon:axisGridColor>
    <ddmlcommon:tickColor>0</ddmlcommon:tickColor>
    <ddmlcommon:tickLabelFormat>%4.2f</ddmlcommon:tickLabelFormat>
    </ddmlcommon:axisGrid>
  </ddmlbarchart:yAxis>
</ddmlbarchart:barchart>

```

Figure 2-9. Specific Type Definitions

[Figure 2-10](#) shows common schema elements for parameters, axes, and XY coordinates.

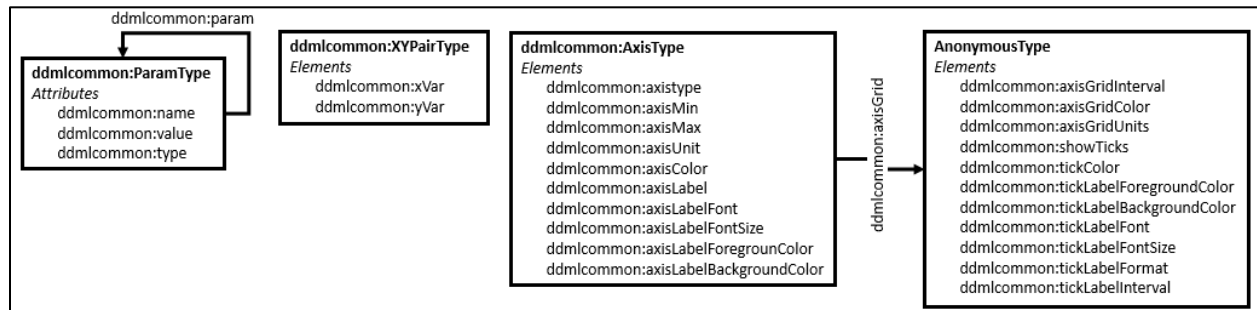


Figure 2-10. Miscellaneous Common XML Schema

## 2.6 Parameters

There are two types of parameter elements available for all DDML elements: DDML sub-elements and custom parameters. The DDML sub-elements describe common and required pieces of information for each DDML element. These parameters are stored as named sub-elements in the DDML schemas. Custom parameters, however, are utilized when vendor-specific information that is not explicitly defined as a DDML sub-element needs to be stored. These parameters are stored in generic DDML param elements. [Figure 2-11](#) shows the schema for the param element.



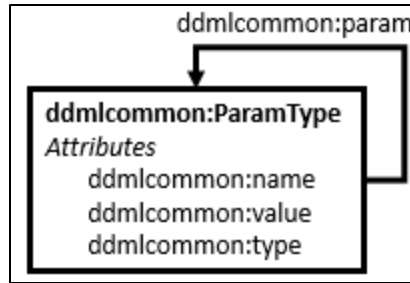


Figure 2-11. Custom Parameter Schema

## 2.7 Data Display Container Custom Parameters

In DDML, custom parameters can be utilized for all graphics resources. These generic parameters allow for unsupported or context-specific information elements to be managed in DDML. In [Figure 2-12](#), a custom parameter called “Screen” is defined for the model (MOD1) and stores the value of “GRIDS.” This indicates that the model should be placed on the “GRIDS” screen in a multiple-monitor configuration.

```
<ddml:project id="PROJ1">
  <ddml:name>Schema Validation Task</ddml:name>
  <ddmlcontainer:model id="MOD1">
    <ddmlcommon:param>
      <ddmlcommon:name>Screen</ddmlcommon:name>
      <ddmlcommon:value>GRIDS</ddmlcommon:value>
    </ddmlcommon:param>
  </ddmlcontainer:model>
</ddml:project>
```

Figure 2-12. Data Display Container Custom Parameter Example XML

[Figure 2-13](#) shows the schema diagram for custom parameters.

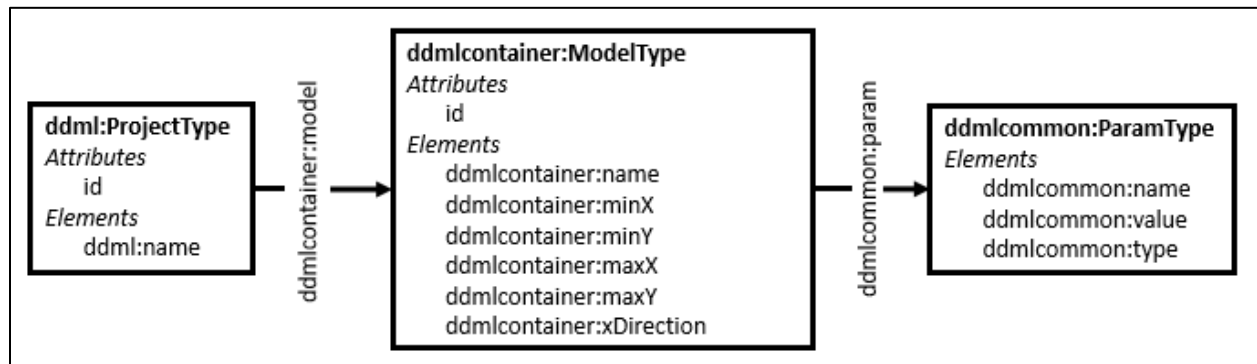


Figure 2-13. Data Display Container Custom Parameter Example XML Schema

## 2.8 Data Source Pool Definitions

Since all data displays depend on sources of data, a data source pool must be defined for all instances of DDML. [Figure 2-14](#) demonstrates a notional data source

**A note about DDML Pools:** Since most graphics resources require data to display, a data source and data variable pool are required for all DDML instance documents. It is recommended that these items be completed prior to defining display objects in DDML.

stored in the DDML source pool. As shown, custom parameters are utilized for each of the data sources to store specific information, such type, scale, and symbol.

```

<ddml:project id="PROJ1">
  <ddmlpool:data_source_pool>
    <ddmlpool:data_source id="S1SRP">
      <ddmlcommon:name>S1SRP</ddmlcommon:name>
      <ddmlcommon:param>
        <ddmlcommon:name>Type</ddmlcommon:name>
        <ddmlcommon:value>FIXED-REF</ddmlcommon:value>
      </ddmlcommon:param>
      <ddmlcommon:param>
        <ddmlcommon:name>Save</ddmlcommon:name>
        <ddmlcommon:value>0</ddmlcommon:value>
      </ddmlcommon:param>
      <ddmlcommon:param>
        <ddmlcommon:name>Stream</ddmlcommon:name>
        <ddmlcommon:value>0</ddmlcommon:value>
      </ddmlcommon:param>
      <ddmlcommon:param>
        <ddmlcommon:name>Symbol</ddmlcommon:name>
        <ddmlcommon:value>SQUARE</ddmlcommon:value>
      </ddmlcommon:param>
    </ddmlpool:data_source>
  </ddmlpool:data_source_pool>
</ddml:project>

```

Figure 2-14. Data Source Pool Example XML

[Figure 2-15](#) shows the schema diagram for data sources.

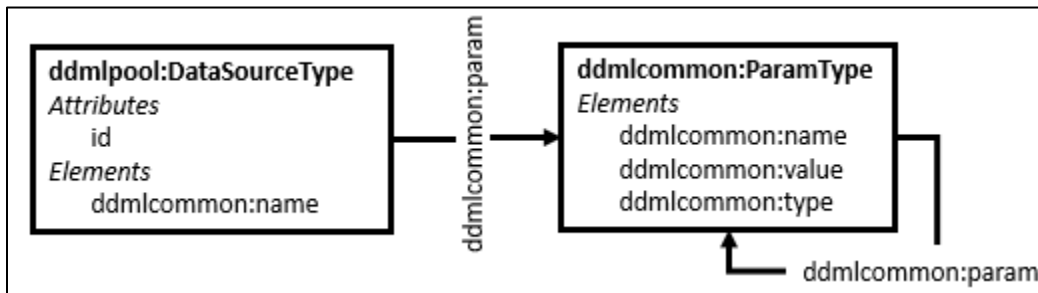


Figure 2-15. Data Source Pool Example XML Schema

## 2.9 Data Variable Pool Definitions

Similar to the data source pool, each instance of DDML must contain a data variable pool. This pool allows DDML to manage the links between data sources and data variables. As shown in [Figure 2-16](#), each data variable must have a name and data source pool reference.

```

<ddml:project id="PROJ1">
  <ddmlpool:variable_pool>
    <ddmlpool:variable id="var0">
      <ddmlcommon:name>IRIG_DAYS</ddmlcommon:name>
      <ddmlpool:data_source_ref>S1SRP</ddmlpool:data_source_ref>
    </ddmlpool:variable>
    <ddmlpool:variable id="var1">
      <ddmlcommon:name>IRIG_HOURS</ddmlcommon:name>
      <ddmlpool:data_source_ref>S1SRP</ddmlpool:data_source_ref>
    </ddmlpool:variable>
  </ddmlpool:variable_pool>
</ddml:project>

```

Figure 2-16. Data Variable Pool Example XML

[Figure 2-17](#) shows the schema diagram for variable pools.

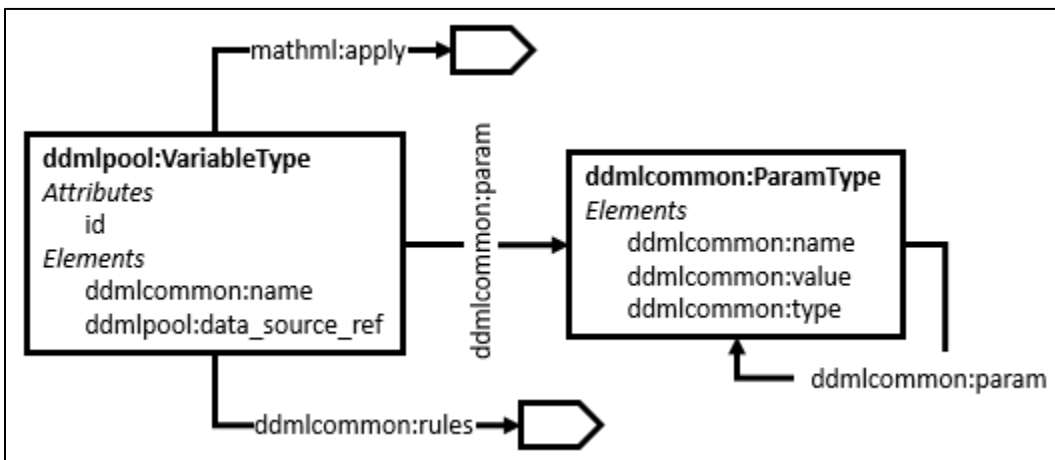


Figure 2-17. Data Variable Pool Example XML Schema

This page intentionally left blank.

## CHAPTER 3

### General Structure, Semantics, and the Display Object Group

Now that we've looked at a complete example, let's back up and look at the overall structure of DDML.

#### 3.1 Layered Structure

The DDML format is built off of a layered structure with layers to support everything from data sources to the visualization of data coming from the data sources on data displays. This structure is similar to a typical software layered architecture composed of graphics resources, visualization and user interfaces, information management, and persistence modules. The DDML layer structure, alongside a typical software structure, is shown in [Figure 3-1](#).

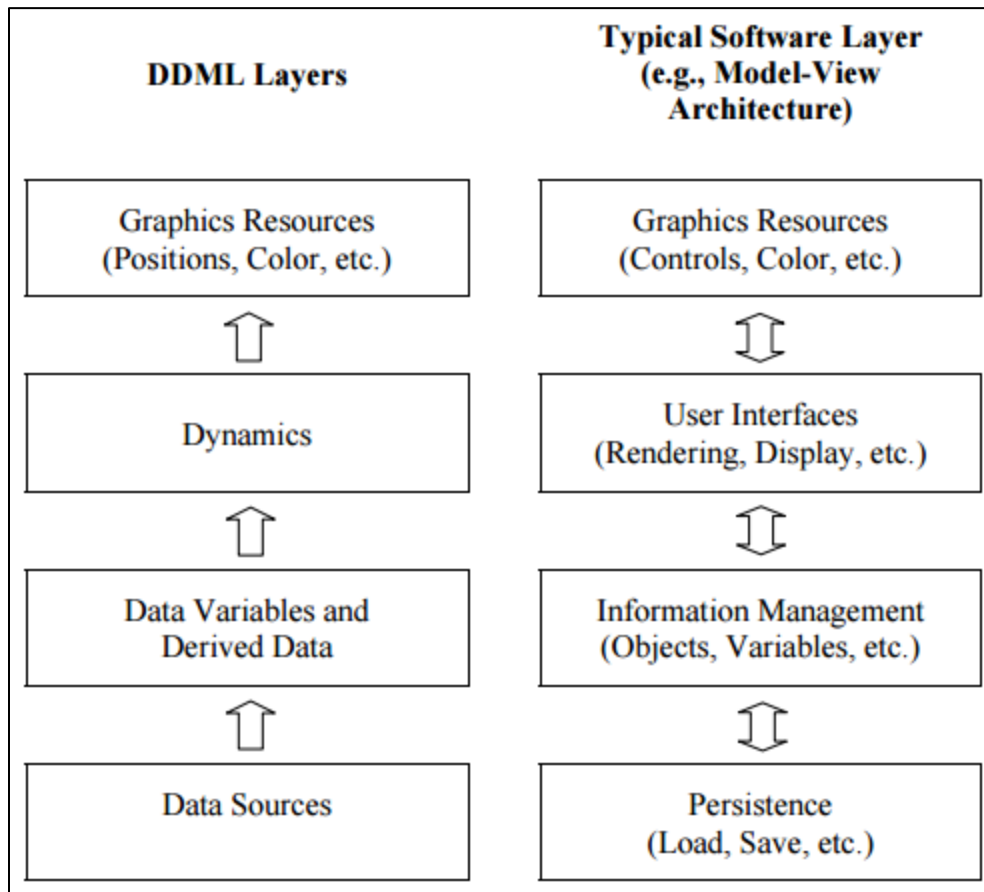


Figure 3-1. DDML Layered Structure

Similar to software modules, DDML is also composed of layers as depicted in [Figure 3-1](#). The first layer in the structure is graphics resources. This layer is similar to graphics resources utilized in software systems. In DDML, this layer is composed of visual components used in data display systems such as charts, buttons, and sliders as well as low-level graphic elements such text, lines, and rectangles. Simple graphical shapes are modeled using Scalable Vector Graphics, which is recommended by W3C. The second layer of DDML is dynamics. The

dynamics layer models behaviors of display objects. It manages the variable and data source use as well as rules associated with data displays. The data variables and derived data layer handle the links between the display objects and data sources. Data variables can be atomic or derived using mathematical expressions in MathML (also recommended by W3C). The last layer of the DDML architecture is the data sources layer. This layer handles various data sources such as text files, Open Database Connectivity, network ports, and ports on data acquisition cards.

### 3.2 Display Object Group

[Table 3-1](#) shows the various common display object classes that are available in DDML. These classes have display object type specific elements that are unique to each object.

**A note about Display Objects:** If a class for a specific display object is not available in DDML, a custom or “generic” display object can be stored. This allows uncommon display objects to be managed in DDML.

<b>Table 3-1. The Display Object Group</b>	
<b>Element Name</b>	<b>Description</b>
Bar Chart	A display object that shows data as vertical or horizontal bars whose values correspond to the lengths of the bars.
Button	A display object used to assign a variable when clicked.
Dial	A circular display object that displays a data value as a needle point on a circular axis.
Custom	A custom display object.
Frequency Plot	A display object that displays frequency domain data.
Frequency Response Plot	A display object that contains frequency and magnitude axes to plot frequency data.
Heads-Up Display	A display object that is typically used to display velocity, pitch, and altitude as well as heading.
Pie Chart	A display object that shows percentage data values as slices of a circle.
Radial Chart	A circular display object that represents data values as line-connected points with distance outward from the center point based on the magnitude of the values.
Slider	A display object that displays a single data value with a marker on a vertical or horizontal axis.
Strip Chart	A display object that displays values vs. time on a scrolling grid.
Textual	A display object that shows text such as a label.
XY Chart	A display object showing x and y data points.

## CHAPTER 4

### DDML Translation

#### 4.1 Translator Development Methodology

The overall development methodology of a bidirectional DDML translator can be broken into four main steps: mapping DDML objects to vendor objects in a data dictionary; choosing an XML parser; writing the translator code; and testing. These four steps apply to the development of both external and internal translators.

The first step in developing a DDML translator for a specific vendor's format is to list that vendor's display objects in a data dictionary. This process involves using the vendor's tool, any available documentation, and source code (when available) to form a comprehensive list of the attributes, their definitions, and their possible values for each display object in the vendor format. These attribute lists are then stored in the data dictionary.

Once these lists have been created, each vendor display object can be matched with a corresponding DDML display object. For example, a vendor format may have a display object called 'meter' that is similar in appearance and functionality to a DDML dial. Thus, the vendor's meter will be matched with DDML's dial.

For each display object, this matching process must then be repeated at the attribute level. The DDML definition contains certain basic attributes for each DDML element. These attributes comprise the most common and most necessary information needed to describe that element. Some of these attributes are required, while others are left optional. For example, a DDML strip chart must have coordinates and may have a title. When an object from a vendor's format is added to the data dictionary, all of the required DDML attributes and as many of the optional DDML attributes as possible are matched to equivalent vendor attributes. Any vendor attributes that are not matched to DDML attributes will become custom parameter elements in generated DDML files.

[Table 4-1](#) shows a section of a data dictionary that compares DDML attributes to DataViews attributes for strip charts. Not all DDML attributes correspond to DataViews attributes, and some DDML attributes must be calculated from one or more DataViews attributes.

<b>Table 4-1. DDML Data Dictionary Sample</b>	
<b>Strip Chart Attributes</b>	
<b>DDML</b>	<b>DataViews</b>
ScrollDirection	Derived from graph type
Title	Title
TitleColor	TitleColor
TitleFont	TitleFont
TitleFontSize	TitleFontSize / 10
BackgroundColor	BackgroundColor
DataAreaColor	DataAreaColor

FrameScrollDuration	
ValueAxisMin	YlowRange
ValueAxisMax	YhighRange
TimeAxisMin	TimeStart / 60
TimeAxisMax	(TimeStart + numSlots) / 60
TimeAxisUnit	Seconds
ValueGrid1Color	GridColor (DV only has 1 grid)
ValueGrid2Color	
ValueGrid3Color	

The mappings detailed in the data dictionary are used for both directions of the translator: vendor format to DDML, and DDML to vendor format; however, in some cases, the relationship between an item in the vendor format and an item in DDML is one-to-many. In this situation, special rules must be established for determining which DDML item should be created when performing a translation in the vendor format to DDML direction. Similarly, if the relationship is many-to-one, rules need to be created for the DDML to vendor format direction. These rules should be stored in the data dictionary along with the mappings.

Once the vendor format has been added to the data dictionary, an XML parser must be chosen before development of the DDML translator can begin. The XML parser will be used both to read the DDML file when translating from DDML to the vendor format, and to build the DDML file when translating from the vendor format to DDML. There is a number of XML parsers freely available, and they come in two types: Simple API for XML (SAX)<sup>4</sup> and Document Object Model (DOM)<sup>5</sup>. The first parser decision to be made is whether to use a SAX or a DOM parser. This decision is influenced by any memory or performance constraints and the developers' own preferences. Once a parser type has been selected, a parser must be chosen from those available for that type. This decision is most influenced by the language and development environment used in creating the translator. For example, if a DOM parser is to be chosen, Microsoft<sup>®</sup> provides its MSXML DOM parser for nonmanaged applications or the System.Xml DOM parser for applications that make use of managed classes and the .NET framework.

With the parser chosen, the next step in the development process is to write the translator code. The development of the translator code can be made easier and more flexible with the creation of a DDML helper module that contains constants and functions useful for performing common translations, conversions, and other often-repeated functions. For example, in DDML, colors are always stored as 24-bit integers, with the red, green, and blue components encoded as 0xRRGGBB. By adding functions to the helper module that converts between the DDML color format and the format used by the vendor's tool, the amount of code written for the translator will be reduced. Additionally, if future versions of the vendor tool use a different color format, the change will only have to be made in the helper module. Similarly, this module can contain functions that convert coordinates from the incoming DDML coordinate system to the vendor tool's coordinate system. In addition, the helper module can contain functions for performing common XML-related tasks such as finding a node, adding a subnode, and getting the value of

<sup>4</sup> Official website of the *Simple API for XML* (SAX), <http://www.saxproject.org/>.

<sup>5</sup> "Document Object Model (DOM) Level 3 Core Specification," W3C Recommendation <http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/>, April 2004.



an attribute. Having all of these XML-related tasks in one module also makes it easier to switch to a different XML parser in the future if needed.

By using the DDML helper module and following the mappings defined in the data dictionary, writing the translator code is just a matter of detecting the objects in the source format and saving them as the appropriate objects in the destination format. It is best to write the translator code for the highest-level objects first, starting with the DDML project element, working down to DDML model level and, finally, to the individual charts, graphs, and other display objects. When writing the translator code for a given object, both directions of the translator should be written before moving on to another object, starting with the vendor format to DDML direction. Once this direction is completed for an object, it is easy to take that code and simply reverse it for the DDML to vendor format direction. Some additional code may need to be added to the reverse direction, such as default values for certain vendor format parameters, to ensure that DDML files that were translated from a different vendor format are translated to this vendor format correctly.

When the translator code is written for each direction and each object, the mappings and attribute lists for each object created in the data dictionary are used to translate each item in the vendor format to and from the correct item in DDML. Each attribute that maps to a DDML attribute is translated to or from the attribute list for the current object. Each remaining attribute in the vendor format is loaded or saved as a custom parameter element in the custom parameters section associated with the current object. During this development process, it is sometimes necessary to modify the mappings in the data dictionary as more information about each attribute is obtained.

Some additional details of the code-writing step are different depending on whether the translator is internal or external. These differences are described in later sections for each translator type.

During translator development and after completion, it is important to test the translation of each object type for both the forward (vendor format to DDML) and the reverse (DDML to vendor format) directions. The DDML files generated by the translator must be tested, using a program such as XMLSpy<sup>6</sup> by Altova<sup>®</sup>, to validate the files against the DDML Document Type Definition.<sup>7</sup> In addition, the translators must be tested for completeness by performing round-trip translations from the vendor tool to DDML and back. The data display should be identical before and after the translations. Finally, the correctness of the object and attribute mappings should be tested by translating displays from the vendor tool to several other vendor tools and vice versa. This can be accomplished by making use of the external translators or by using an internal translator in that vendor's tool.

## 4.2 Development of External Translators

If the vendor tool uses a text-based or well-documented binary format or if the vendor tool has an application programming interface (API) for accessing data display information, the translator can be developed as an external translator without requiring access to the vendor's

---

<sup>6</sup> Altova's XMLSPY web page, <https://www.altova.com/xmlspy.html>.

<sup>7</sup> "Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1," <http://www.w3.org/XML/1998/06/xmlspec-report>, June 1998. Retrieved 31 January 2017.

source code. If the translator is being developed as an external translator, some additional steps and considerations must be taken that do not apply to internal translators.

For external translators, a way to access and create data display information in the vendor format must be created. This can be done by accessing an available API for the vendor tool or by writing a parser for the vendor format. Since the translator is bidirectional, the parser must be capable of not only reading and interpreting the vendor format, but of building new files in the vendor format as well. Functions that read and write basic elements in the vendor format should be created and added to the DDML support module previously described. Separating this format parser code from the actual translator code as much as possible makes it easier to modify the translator if the vendor syntax changes.

The code that performs the translations in each direction should be organized according to the DDML object being translated. This makes it easier to modify the mappings of one particular object without affecting the translations of other objects. For each DDML object, the code should be further split into code for DDML attribute translation and code for custom parameter translation. This makes it easy for the developer to look at the code and see which vendor parameters are mapped to DDML attributes and which parameters are not.

### **4.3 Development of Internal Translators**

If the vendor tool uses an undocumented binary format and does not have an API, the translator will have to be integrated into the tool itself as an internal translator. The process of writing the translator code is slightly different for internal translators than for external translators. Like external translators, special considerations must be accounted for internal translators as well.

Since internal translators are developed as part of the vendor tool, they have access to all of the information stored in a data display through calls to class members and functions. Consequently, it is not necessary to develop a format parser for the vendor's format.

To ensure that an internal translator is comprehensive, it is best to model the translator's code on the existing save and load functions for the vendor's native format. Every class that contains a function to save to or load from the vendor format should also have a function to perform the same task in DDML. Any information that is saved to or loaded from the native format should also be saved to or loaded from DDML. This will ensure that no information is lost when performing a round-trip translation from the vendor tool to DDML and back. Using the same structure as the existing save and load functionality also prevents problems with trying to access private variables from an external class.

## Appendix A

### Citations

Altova's XMLSPY web page, <https://www.altova.com/xmlspy.html>.

“Document Object Model (DOM) Level 3 Core Specification,” W3C Recommendation  
<http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/>, April 2004.

“Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1,”  
<http://www.w3.org/XML/1998/06/xmlspec-report>, June 1998. Retrieved 31 January 2017.

Official website of the Simple API for XML (SAX), <http://www.saxproject.org/>.

Range Commanders Council. “Telemetry Attributes Transfer Standard,” in *Telemetry Standards*. IRIG 106-15. July 2015. May be superseded by update. Retrieved 1 July 2015. Available at [http://www.wsmr.army.mil/RCCsite/Documents/106-15\\_Telemetry\\_Standards/Chapter9.pdf](http://www.wsmr.army.mil/RCCsite/Documents/106-15_Telemetry_Standards/Chapter9.pdf).

———. *XML Style Guide*. RCC 125-15. July 2015. Retrieved 13 January 2017. Available at [http://www.wsmr.army.mil/RCCsite/Documents/125-15\\_XML\\_Style\\_Guide/](http://www.wsmr.army.mil/RCCsite/Documents/125-15_XML_Style_Guide/).

**\*\*\*\*\* END OF DDML HANDBOOK \*\*\*\*\***