Proceedings of the 14th International Symposium on Unmanned Untethered Submersible Technology, Durham, NH, August 2005

AUTOMATED PARSING AND CONVERSION OF VEHICLE-SPECIFIC DATA INTO AUTONOMOUS VEHICLE CONTROL LANGUAGE (AVCL) USING CONTEXT-FREE GRAMMARS AND XML DATA BINDING

Duane T. Davis, Naval Postgraduate School
Watkins Annex Room 265
1 University Circle
Monterey, CA 93943
(831) 656-3733
dtdavis@nps.navy.mil

ABSTRACT

Numerous languages and data formats are utilized for autonomous underwater vehicle (AUV) operations. In particular, missions are generally written and data archived using vehicle-specific languages and formats. Development of an unmanned vehicle (UV) ontology and automated conversion of vehicle-specific information into a data format constrained by this ontology can prove useful by enabling the development of planning and analysis tools for arbitrary vehicles and enabling facilitating interoperability between dissimilar vehicles. Implementation of this ontology using the Extensible Markup Language (XML) is the obvious choice, in part because of the ease with which it can be transformed to other formats using the Extensible Stylesheet Language for Transformations (XSLT). Additionally, consistent mappings of output telemetry to XML files permits results to be self-documenting and self-validating. Nevertheless, automated conversion of vehicle-specific data to a common XML format can be problematic necessitating the development of conversion methodologies suited to this task.

Although tools such as the Extensible Stylesheet Language for Transformations (XSLT) do not exist for arbitrary data formats, a similar methodology can be developed and implemented. The key to a potential solution to this problem is recognizing that vehicle-specific languages and formats can be defined using context-free grammars (CFG). Specifically a set of productions rules in Chomsky Normal Form (CNF) is developed that completely defines both the structure and semantics of a vehicle-specific data format. The Cocke-Younger-Kasami (CYK) algorithm is then utilized to generate a parse tree for a vehicle-specific data file. The parse tree contains all of the implicit information corresponding to the grammar provided by an XML document. The implementation of routines to generate an XML data object corresponding to the common format is now a relatively simple matter that is analogous to the use of XSLT to convert from XML to vehicle-specific formats.

This paper provides a brief discussion of a Naval Postgraduate School-developed, XML-based common autonomous vehicle data model—the Autonomous Vehicle Control Language (AVCL)—which is used as the basis for vehicle-independent mission planning and rehearsal in the Naval Postgraduate School (NPS) Autonomous Unmanned Vehicle Workbench (AUVW). Additionally, implementation results for the automated conversion of four vehicle-specific data formats to this common data model are presented.

1. INTRODUCTION

Significant research in recent years has investigated methodologies and protocols to foster coordinated operations among unmanned vehicles (UV). However, much of this research has assumed that the vehicles involved are inherently compatible. That is, either the multi-vehicle system consists solely of one type of vehicle, or all vehicles use the same language for mission specification and/or inter-vehicle communication. Unfortunately, this is unrealistic given current inventories of legacy vehicles and the parallel development of vehicles by various commercial, academic, and government entities.

Ongoing research at the Naval Postgraduate School (NPS) is attempting to address the issue of dissimilar vehicle compatibility through the use of a common ontology for UV tasking, communications and mission results. In the context of this research, ontology refers to a formal description of a vocabulary, including word meanings, assumptions and relationships, that can be used to describe and represent an area of knowledge [4], in this case UV operations. In addition to the ontology itself,

methods are being developed to foster automated translations between vehicle-specific data formats and an ontology-compliant format. In this way, an UV tasking ontology can be used to facilitate coordination between vehicles that are not designed or programmed to work together. Similarly, this ontology can be used as the basis for planning and data analysis tools.

For reasons discussed in [7] and expanded in [17] and [14], the UV ontology is being developed as a schema-governed Extensible Markup Language (XML) tagset [19]. XML is well-suited to this role for a number of reasons. First, an XML document can be easily understood and processed by both humans and computers. This facilitates task development, monitoring of mission progress, and interpretation of mission results by human operators as well as programmatic parsing and interpretation by computer. Additionally, the structure and content of an XML document can be rigorously defined through the use of an XML schema [20][21] or a Data Type Document (DTD) [19]. This strict content governance facilitates mapping to and from other data formats, makes document validity and correctness easily verifiable, and even serves as the basis for compression algorithms that make transmission of XML content over noisy and bandwidth-limited communications paths feasible. Finally, XML lends itself to automatic translation to different data formats through the use of the Extensible Stylesheet Language for Transformations (XSLT) [18] and XML data binding [16].

XSLT is a declarative programming language that is used to transform one XML document into another text-based format. Primarily defined for transforming one XML document into another [18], XSLT is by no means limited to this application. As demonstrated in [15], it can be used effectively to convert XML documents into virtually any text-based format containing the same information, or derivable information, as the original XML document. Among the increasingly common uses of XSLT is the transfer of data between applications requiring different formats [10], a process that is semantically identical to transforming an UV-ontology constrained XML document to a vehicle-specific format.

XML data binding is another tool that proves useful in processing schema- or DTD-governed XML documents. While general XML parsers such as the Simple Application Programmer's Interface (API) for XML (SAX) [13] and the Document Object Model (DOM) [22] are appropriate in many instances, they can be somewhat cumbersome because of the general nature of the APIs. XML data binding, on the other hand, uses an XML schema or DTD to automatically generate high-level programming language code (Java implementations are the most common) that is specific to a particular type of document. This makes loading, manipulating and generating schema-compliant documents significantly simpler than would be the case using SAX or DOM. With XML data binding, ontology-compliant UV data can be programmatically generated and transformed to vehicle-specific formats as required in a relatively straightforward manner.

While the preceding discussion clearly indicates the utility of XML, XSLT and XML data binding in converting from an UV ontology to vehicle-specific data formats, it does not address the reverse transformation—a significantly more difficult problem. The fact of the matter is that vehicle-specific data formats are not, in general, XML. Nevertheless, use of an XML-based common data model as an intermediary between different vehicle-specific formats requires a mechanism for converting non-XML vehicle-specific data into XML.

A simple observation that vehicle-specific data formats, while not schema-governed XML, are still rigorously defined lexically, semantically, and structurally provides the basis for a methodology for the automatic parsing and conversion of vehicle-specific data to ontology-compliant XML. Implementation of this methodology involves definition of vehicle-specific formats using context-free grammars (CFG). Once defined in this manner, there are a number of well-known algorithms that can be used to generate parse trees for arbitrary vehicle-specific data, thereby providing a data structure capable of supporting automated translation into ontology-compliant XML.

The implementation of this methodology in the NPS Autonomous and Unmanned Vehicle Workbench (AUVW) [6][11] will be the focus of the remainder of this paper. Section 2 will consist of a brief discussion of the UV ontology into which vehicle-specific data formats will be translated, the Autonomous Vehicle Command Language (AVCL). Section 3 will provide a mathematical overview of CFGs and their use in defining vehicle-specific data formats, while section 4 will describe the use of CFGs in the AUVW for the generation and translation of parse trees. Finally, section 5 will give a brief description of the CFG definitions, translations and results for four vehicle-specific data formats currently implemented in the AUVW.

2. AUTONOMOUS VEHICLE COMMAND LANGUAGE (AVCL)

Before specifically covering CFG-based translations it is appropriate to briefly discuss the structure and semantics of the proposed data model into which vehicle-specific data is to be translated—the Autonomous Vehicle Control Language (AVCL). AVCL is an exemplar ontology encapsulating data requirements for the operations of arbitrary UVs. Structurally speaking, the schema is divided into three parts: mission results, communication and mission preparation. The mission results portion of the schema is utilized to record synchronous data such as telemetry and control orders as well as asynchronous data including contacts and messages sent or received. The communications portion is used to format messages sent to or received from an UV. Finally, AVCL's mission preparation vocabulary is used to define the mission requirements. This mission preparation portion of AVCL will be the primary focus of the remainder of this paper, however the methods discussed are equally applicable to the other portions of the ontology.

The overall structure of the mission preparation element of an AVCL document is shown graphically in Figure 1. This element includes child elements for the units of measure that are used throughout the rest of the document, the origin of the Cartesian coordinate system in geographic coordinates (if required), and the configuration and capability requirements of the vehicle for which the mission is intended. Finally, the element contains the actual description of the mission (i.e. what the vehicle is required to do).

Mission requirements can be specified in one of two ways. An example of the first method, mission specification as a sequence of task-level script commands, is shown in Figure 2. The simplest task-level commands reset vehicle state parameters (e.g. commanded depth, rudder deflection, etc.) and will generally execute in a single vehicle control loop. More complex task-level commands, waypoints for example, will require an indeterminate amount of time to complete. Missions specified in this way will proceed sequentially with individual commands executed one at a time in the order that they are specified. Conceptually simple, this method of mission specification is appropriate for many current commercial and research UVs including NPS' Acoustic Radio Interactive Exploratory Server (ARIES) AUV, the Naval Oceanographic Office (NAVO) Seahorse AUV and the Hydroid Remote Environmental Monitoring Units (REMUS) AUV.

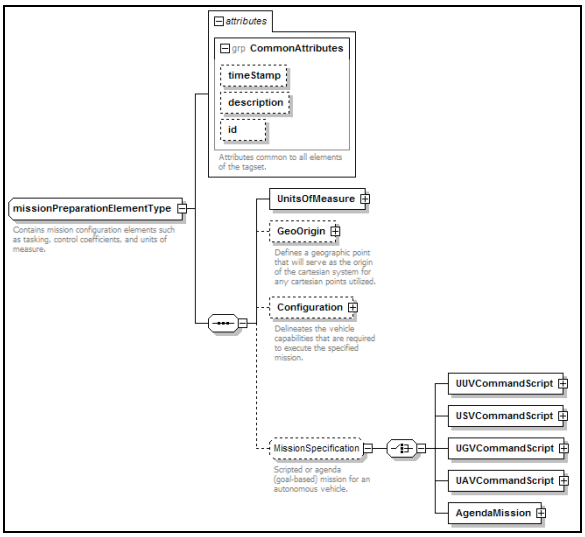

Figure 1: Mission preparation element structure in an Autonomous Vehicle Control Language (AVCL) document.

```
<UUVCommandScript/>
   <Position>
       <XYPosition x="0.0" y="0.0"/>
       <Depth value="0.0"/>
   </Position>
   <Thrusters value="false"/>
   <Waypoint>
       <XYPosition x="100.0" y="100.0"/>
       <Depth value="45"/>
       <SetPropeller>
           <AllPropellers value="100.0"/>
       </SetPropeller>
   </Waypoint>
   <Waypoint>
       <XYPosition x="500.0" y="100.0"/>
       <!-- use prior depth! -->
   </Waypoint>
   <Waypoint>
       <XYPosition x="500.0" y="200.0"/>
       <Depth value="25"/>
   </Waypoint>
   <Waypoint>
       <XYPosition x="0.0" y="0.0"/>
   </Waypoint>
   <MakeDepth value="0.0"/>
   <Quit/>
</UUVCommandScript>
```

Figure 2: A scripted mission for an autonomous underwater vehicle (AUV) written with AVCL.

The second method of AVCL mission specification is via a set of goals and constraints that are to be completed in the course of the mission. The overall format consists of a finite state machine (FSM) where each node represents a single high-level goal. FSM links determine which goal is to be accomplished upon the completion (or failure) of the currently executing goal. The AVCL element representing an individual goal (Figure 3) includes a description of the goal itself, the location or operating area and

circumstances under which reports are to be made. Attributes of the current goal are used to reference the goal to be attempted upon completion (nextOnSucceed) or failure (nextOnFail). Missions specified in this method are appropriate for vehicles such as the Naval Undersea Warfare Center (NUWC) UUV-21 AUV and other vehicles using more advanced hierarchical and hybrid control architectures.

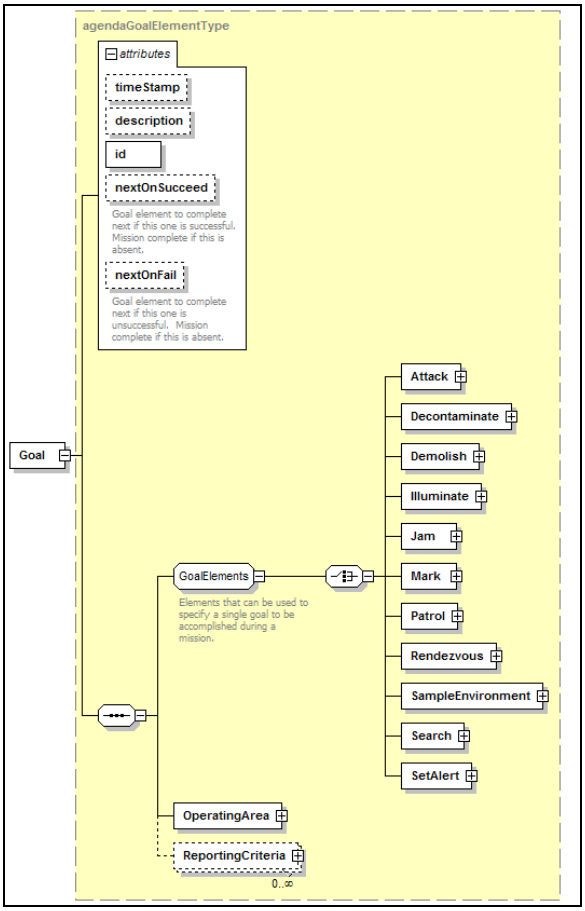

Figure 3: AVCL declarative mission goal element format.

3. DEFINITION OF VEHICLE-SPECIFIC DATA FORMATS USING CONTEXT-FREE GRAMMARS (CFG)

As stated in the previous section, vehicle-specific data formats are by necessity rigorously defined both structurally and semantically. In fact, vehicle-specific data formats take the form of specific context-free languages (CFL) [3]. Mathematically speaking, a CFL is the set of strings derivable from a CFG [8]. The implication of the previous statement is that there exists a CFG corresponding to any vehicle-specific data format in which we might be interested. It stands to reason that the CFG can be used to both generate and parse instances of the vehicle-specific data format.

A CFG is commonly defined as a set of productions or rules of the form

$$A \to \alpha$$

where A is a variable, α is a sequence of variables and terminal symbols (the tokens that make up the alphabet of the language) plus null (ε), and the production symbol ($\to$) indicates that the variable A can be expanded into α.

A CFG can be formally specified with four components: *V*, *T*, *P*, and *S*, where *V* is the set of variables, *T* the set of terminal symbols, *P* the set of productions, and *S* the set of available start symbols (a non-empty subset of *V*) [8]. As a simple example, the productions

$$P \to (\ ) \text{ and } P \to (P)$$

can be used to construct all strings of balanced parentheses (i.e., strings of the form "((()))"). An appropriate CFG can therefore be fully defined as

$$G = (\{A\}, \{(,)\}, \{A \to (), A \to (A)\}, \{A\}).$$

Simplicity of the preceding example notwithstanding, CFGs can a powerful tool for recursively defining a variety of complex languages. In fact, CFG production rules of the form described above provide the basic building blocks of the XML DTD [19] and can be used to define high-level programming languages [8].

While CFGs are expressively powerful in their basic form, there are a number of potential issues that can complicate their application. The most obvious is that there are a number of CFGs that can be used to generate a specific CFL. For instance we could generate the same CFL as the example by replacing the productions in the grammar (*G*) with

$$\{A \to (A), A \to BC, B \to (, C \to)\}.$$

Further, it is easy to define a mathematically correct grammar (i.e. one that correctly generates the CFL) that has unreachable or redundant rules that make its use unnecessarily complex. Finally, many CFGs are ambiguous, meaning that there are multiple ways to apply the productions to derive a given string. It is

therefore, often advantageous to constrain the form of the production rules of the CFG by defining it in accordance with a normal form.

In general a normal form is a way of defining the format and characteristics of the productions of a CFG. Commonly utilized normal forms used in the definition of CFGs are Chomsky Normal Form (CNF), Greibach Normal Form (GNF) and Backus-Naur Form (BNF) [8]. Because the parsing algorithm implemented in the AUVW requires CFGs defined in CNF, the remainder of this section will consist of a discussion of this normal form and its practical use in defining CFGs capable of generating UV-specific documents.

The productions of a CFG defined in CNF have three significant characteristics:

1. No useless symbols (i.e., variables or terminal symbols that do not appear in any terminal-string derivation beginning with the start symbol).

2. No ε productions (i.e., those of the form $A \rightarrow \varepsilon$).

3. All productions must be of the form

   $A \rightarrow BC$ or
   $A \rightarrow a$

   where *A*, *B* and *C* are variables and *a* is a terminal symbol.

It can be proven that for any CFL not containing the empty string (ε), there exists a CNF CFG capable of generating that CFL [8]. Defining CFGs in CNF addresses a number of the potential pitfalls of CFG definition (although ambiguity, in particular, can still be an issue), and provides a convenient form for application of the Cocke-Younger-Kasami (CFK) parsing algorithm that will be discussed in the next section.

From a practical standpoint, defining a CFG to generate a particular vehicle-specific data format involves determining the terminal symbols and writing the productions. Terminal symbols may consist solely of numbers as in the ARIES AUV control language [12] or can consist of keywords, symbols and numbers [9]. Production definition can be fairly arbitrary. Nevertheless, since the production rules ultimately determine the structure of the resultant parse tree, the AUVW CFG definitions attempt to utilize production rules that generate homogeneous constructs at their particular derivation level (e.g., position, command or mission) as demonstrated by the following productions which were utilized in the CFG definition corresponding to the AUV mission programming language described in [5].

COMMAND $\rightarrow$ *WAYPT_TOKEN + POSIT_3D*
POSIT_3D $\rightarrow$ *LAT_LONG + DOUBLE*
LAT_LONG $\rightarrow$ *DOUBLE + DOUBLE*

Similar production rules are developed for different commands that reuse variables such as "LAT_LONG" and "POSIT_3D" when feasible.

Programmatic implementation of individual CFGs in the AUVW is accomplished through a CFG-specific dictionary class that lexically maps tokens to their production rule, and a CFG-specific parser class that defines the set of binary production rules of the grammar. Both of these classes inherit from abstract classes that implement the CYK parsing algorithm (described in the next section). Actual parsing in the AUVW is therefore CFG-independent, and arbitrary languages can be implemented quickly. All that is required is to define a suitable CFG through extended dictionary and parser classes.

4. CONTEXT-FREE LANGUAGE (CFL) PARSING AND TRANSFORMATION

Utilization of CFGs defined as described in the previous section to transform vehicle-specific instance documents is implemented in the AUVW using a two-step process. First, a parse tree is generated for the instance document using the CFG productions. Second, the parse tree is traversed and templates applied at individual nodes to generate an XML data bound Java object using the Java Architecture for XML Binding (JAXB) [16]. The second step of this process is analogous to the use of XSLT in that both traverse a document tree in a predictable manner and generate output based on the contents of the tree. These methods differ, however, in their traversal and generation methodologies. In the case of XSLT, the document tree is traversed in an arbitrary fashion and allows multiple visits to individual nodes but generates the output document serially. The CFG-based method described here, on the other hand, traverses the tree in depth-first order and visits each node only once, but generates the output document in an arbitrary fashion.

CFG parsing in the AUVW is accomplished using the CYK algorithm. This algorithm utilizes dynamic programming [2] to parse CFL instances in a bottom up fashion. When utilizing CFGs specified in CNF, the CYK algorithm can be expected to generate a binary parse tree (similar to the one shown in Figure 4 which corresponds to a waypoint command derivable from the sample productions in the preceding section) in $O(n^3)$ time based on the length of the input string [8]. As shown in Figure 5, the algorithm simply recognizes string membership in the CFL corresponding to the CFG and does not generate a parse tree. However, replacing the Boolean array with an array of partial parse trees is all that is required to fully implement a parser. Upon completion of the algorithm, the array elements *P[1, n, x]* will contain valid parse trees (if the grammar is unambiguous, at most one will be non-null).

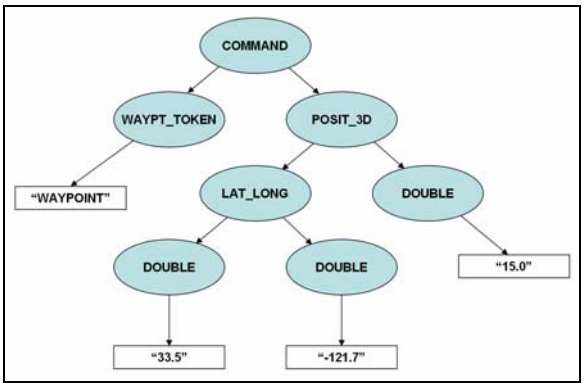

Figure 4: A Cocke-Younger-Kasami (CYK) algorithm generated parse tree corresponding to an AUV waypoint command as defined in [5] and the context-free grammar (CFG) productions example of section 3.

Following generation of a parse tree corresponding to a vehicle-specific CFL instance, it is possible to generate AVCL based on the parse tree structure and content. XSLT utilizes a template-based approach to generate output based on an XML content tree, and a similar approach is utilized here to generate AVCL based on the parse tree. As previously stated, the tree is processed in depth-first order. Templates are applied at each node that define actions to take based on the content of that particular node. Actions in most cases will include recursively processing the current node's left and then right child, and will in some cases include the generation of output.

As an example, upon arriving at the "COMMAND" node depicted in Figure 4, output can be generated. Since the structure of the parse tree is deterministic, upon reaching this node, further traversal in this case not necessary, so the node's children are not recursively processed. All of the information required to generate the corresponding AVCL is in a known location relative to the current node. For instance, the type of command is specified by the left child of the current node, and given that it is a "Waypoint" command, the latitude is specified by the left child of the left child of the right child of the current node.

```
Let the input string be a sequence of n letters
    a1…an

Let V1…Vr be the set of CFG symbols (V)

Let S be the set of indices of V corresponding
    to CFG start symbols

Let P[n, n, r] be a Boolean array initialized
    to false

For i = 1 to n

    For each unit production Vj → ai

        P[i, 1, j] = true

For i = 2 to n – Length of span

    For j = 1 to n – i + 1 – Start of span

        For k = 1 to i – 1 – Partition of span

            For each production VA → VBVC

                if P[j, k, B] = true and

                   P[j + k, i – k, C] == true

                    then P[j, i, A] = true

if P[1, n, x] is true (x is an element of S)

    then the string is a member of the CFL

    else the string is not a member of the CFL
```

Figure 5: The CYK algorithm for recognizing string membership in a context-free language (CFL) [8].

A parse tree of the form shown in Figure 6, on the other hand, will probably not generate output upon reaching the "COMMAND_LIST" node. Rather, it will recursively process the left child and then the right child. While processing the left child, output will be generated that corresponds to the type of command represented. When processing the right child no output will be generated until its left and right children are recursively processed.

Still other cases might require the generation of output as well as recursive processing of the current node's left and/or right child. Additionally, nodes may include state information that must be maintained during traversal of the rest of the tree. An example is provided by the REMUS AUV language's reuse of program elements through references. Upon encountering a reference, the template must cache the relevant data for use when the reference is invoked in other portions of the program.

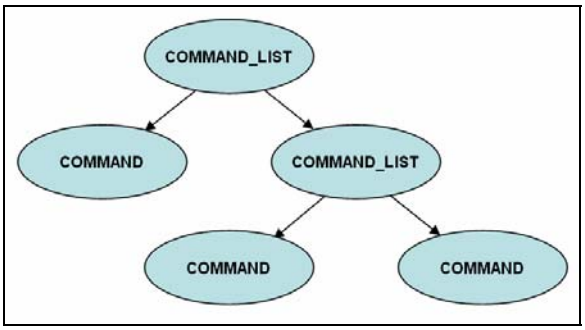

Figure 6: A partial parse tree of a vehicle-specific CFL instance suitable for recursive translation into AVCL.

A noteworthy characteristic of the depth-first traversal is that the instance document is processed in order. For vehicle-specific data formats along the lines of those used in the ARIES or Seahorse AUVs, the corresponding AVCL will be generated in the same order. However, the sequence and structure of the AVCL corresponding to instances of more complex data formats may differ significantly from the vehicle-specific document. Since the parse-tree traversal is fixed, AVCL generation must be arbitrary (i.e. it must be possible to generate the AVCL document in any order). This is accomplished through the use of XML data binding. Since the entire document is maintained in memory throughout the generation process, it is a fairly simple matter to access or modify existing data, add new data, and move or copy existing data from one location in the document to another.

5. IMPLEMENTATION AND RESULTS

To date CFGs have been developed and implemented in the AUVW for the vehicle-specific mission programming languages of four AUVs: NPS ARIES, NPS Phoenix, NAVO Seahorse and Hydroid REMUS. Missions specified in any of these languages are translatable into task-level missions in AVCL.

The ARIES AUV

Of the implemented data formats, the simplest is used to task the NPS ARIES AUV. ARIES mission files consist of individual waypoint commands that are to be achieved sequentially in the order specified. Each waypoint is specified with 11 double-precision floating point numbers. Waypoint parameters include Cartesian coordinate position of the waypoint, depth below the surface or altitude above the bottom to use during transit, left and right propeller speed, maximum time to allow for reaching the waypoint and a binary flag indicating whether or not a global positioning system (GPS) fix is to be obtained during transit. [12]

Because ARIES' data consists entirely of numbers, only one production rule mapping variables to terminals is required:

$$DOUBLE \rightarrow number$$

This and 14 binary productions fully define the ARIES mission specification language. Resultant parse tree nodes corresponding to an ARIES waypoint often translate directly into a single AVCL waypoint script command. In some circumstances, most notably when different left and right propeller settings are ordered or a GPS fix is required during transit, additional AVCL task-level commands are required prior to the waypoint command as depicted in the example of Figure 7.

```
<SetPropeller>
    <PortPropeller value="75.0"/>
</SetPropeller>
<SetPropeller>
    <StarboardPropelelr value="50.0"/>
</SetPropeller>
<GpsFix>
    <StartGpsFix/>
</GpsFix>
<Waypoint>
    <XYPosition x="500.0" y="200.0"/>
    <Altitude value="25"/>
    <Standoff value="10"/>
    <TimeOut value="125"/>
</Waypoint>
```

Figure 7: An AVCL task-level command sequence corresponding to an ARIES AUV waypoint command with dissimilar left and right propeller settings and an enroute global positioning system (GPS) fix.

The Phoenix AUV

The second vehicle-specific data format implemented in the AUVW is a script-based mission specification described in [5]. The language was utilized in the NPS Phoenix AUV and is still an available option in the AUVW. Translation of Phoenix missions into AVCL requires a larger subset of the AVCL task-level command set and includes a number of commands that are not directly representable in AVCL.

A Phoenix AUV mission takes the form of a series of script commands. Each command consists of a keyword and zero or more numerical or text parameters. In some instances, different tokens can be used for the same keyword (e.g., "thrusters" and "thrusters-on" can be used interchangeably). This is

dealt with in the dictionary class by mapping each of the various forms of a keyword to the same production rule. Additionally, many commands can have varying numbers of parameters. In each case, the number of parameters determines what each parameter represents. For instance, the command

```
WAYPOINT 100 500 15 10
```

orders the vehicle to transit to the waypoint specified by the Cartesian coordinate (100, 500) at a depth of 15 meters and to proceed to the next command upon reaching a standoff distance of ten meters. The command

```
WAYPOINT 100 500 15
```

on the other hand, specifies the same waypoint and transit depth, but does not specify a standoff distance (in this case, the most recently specified standoff distance applies). It is not possible, however, to specify a waypoint with a Cartesian coordinate and standoff distance without also specifying a transit depth.

Terminal tokens in the Phoenix command language can take the form of keywords, numbers or literal string tokens. Keywords for parameterless commands (e.g., gpsfix) are all mapped to the same rule:

$$COMMAND \rightarrow keyword$$

This greatly simplifies the CFG, however it requires actual interpretation of the keyword when the parse tree is translated into AVCL. Keywords for commands requiring parameters are mapped to more specific unary rules along the lines of

$$HOVER\_TOKEN \rightarrow \text{"}hover\text{"}$$

that can be utilized during translation to infer the potential parameter semantics. In total, 22 unary and 39 binary production rules are utilized to define the Phoenix CFG.

It is worth noting that the *COMMAND* variable is used on the left side of both unary and binary rules in the Phoenix CFG. This results in numerous potential parse tree structures with a "COMMAND" node at their root. Nevertheless, in all cases the specific keyword contained in the leftmost leaf of the parse tree node (referenced directly from the "COMMAND" node in the AUVW implementation without further recursive traversal) can be utilized during translation to determine how to process individual commands.

Translation of Phoenix command language documents into AVCL is almost a one to one mapping from Phoenix commands to AVCL task-level commands. For instance, each of the waypoint commands described previously translates directly to a single AVCL waypoint command. There are, however, a number of Phoenix commands that are not directly represented in AVCL. For instance, "pause," "trace" and "real-time-off" commands were implemented to support mission and control testing in a virtual environment in [1]. This functionality has been implemented directly in the AUVW, however, and is no longer required in the vehicle-command language. Nevertheless, capture of the semantics of these commands in AVCL is required in order to facilitate the accurate generation of Phoenix missions from AVCL scripts as described in [6]. AVCL, therefore, incorporates a "meta" command that can encode command data that is relevant only to a specific vehicle. Conversions from AVCL to most vehicle-specific data formats will ignore meta commands, however, enough information is contained in the command to allow accurate translations from AVCL to the data format of the relevant vehicle. For example, a meta command representing a Phoenix "pause" command might have the form

```
<MetaCommand value="pause"
     description= "Phoenix AUV
     pause command"/>
```

Parameterized vehicle-specific commands requiring meta command representation in AVCL will include all of the parameters in the MetaCommand value parameter.

The Seahorse AUV

The third vehicle-specific CFG implemented in the AUVW is used for NAVO Seahorse AUV tasking. While still script-based and mappable to the AVCL task-level command set, the syntactic complexity of the Seahorse CFL is greater than that of ARIES or Phoenix, so the CFG implementation in the AUVW is also comparatively complex.

Syntactic complexity notwithstanding, the Seahorse CFL contains just four general purpose commands. These can be used to direct the vehicle to a waypoint, cause it to loiter at a specified location, obtain a GPS fix or surface for communications. Additionally, the first command in a Seahorse mission must be a

launch command that initializes the vehicle and the final command must be a rendezvous command that directs the vehicle to the retrieval location. Each of the six commands has a form similar to the station-keeping command depicted in Figure 8 (which commands the vehicle to loiter at a specified geographic position) which is comprised of a series of parameter name-value pairs. Parameter order is fixed, however CFG complexity is introduced by the fact that some parameters are optional and others have more than one potential form (e.g., positions can be specified as latitude and longitude or as north and east offsets from the current position, distances can be specified using feet, meters, kilometers or kiloyards, etc.). In all, the Seahorse AUV CFG implemented in the AUVW utilizes 47 terminal and 136 binary production rules, most of which generate subportions of individual commands.

```
Start_Order               : Station_Keep_Order
Scheduling_Info_Is_Timed  : True
Destination_Latitude      : 33.0 Degrees
Destination_Longitude     : -122.5 Degrees
Until_when                : 90 Minutes
Transit_Altitude          : 15 Meters
Loiter_Depth              : 15 Feet
```

Figure 8: A Seahorse AUV command directing the vehicle to proceed to a geographic position and remain there for 90 minutes.

Utilization of the parse tree to generate AVCL is relatively straight forward despite the number of production rules and correspondingly deep parse tree branches for individual commands. As with ARIES and Phoenix, once a parse tree node representing a command is encountered, the location in the branch of various parameter names and values is known, so they can be accessed directly without further traversal.

In the simplest cases (e.g., a waypoint command with no GPS fix during transit), a Seahorse command can be represented by a single AVCL task-level command. However, more complex cases such as the command depicted in Figure 9, require a series of task-level commands to accurately capture the semantics (Figure 10).

The REMUS AUV

The final vehicle-specific CFG implemented in the AUVW is utilized for tasking of the REMUS AUV. A REMUS mission consists of a sequence of objectives that are to be accomplished in order. There are 14 objective types available including mandatory start and end objectives, various forms of waypoints and a number of vehicle-specific commands. Like Seahorse AUV commands, REMUS objectives use a keyword to identify the objective type and a number of mandatory and optional name-value pairs to parameterize the objective resulting in objectives similar to the one shown in Figure 11. In addition, the REMUS CFL allows locations to be specified by reference. This facilitates location reuse throughout the mission file and supports the specification of relative positions. [9]

```
Start_Order               : Rendezvous_Order
Scheduling_Info_Is_Timed  : False
Rendezvous_Latitude       : 28.0 Degrees
Rendezvous_Longitude      : 88.0 Degrees
Transit_Depth             : 30.0 Meters
Transit_Speed             : 4.0 Knots
GPS_Fix_Before_Departure  : True
GPS_Fix_After_Arrival     : True
```

Figure 9: A Seahorse AUV rendezvous command directing the vehicle to transit at 30 meters depth and a speed of 4 knots to a geographic rendezvous point and to obtain GPS fixes before starting and after arriving.

```
<GpsFix>
    <StartGpsFix/>
</GpsFix>
<Waypoint>
    <LatitudeLongitude latitude="28.0"
                       longitude="88.0"/>
    <Depth value="30"/>
    <SetPropeller>
        <AllPropellers value="75.0"/>
    </SetPropeller>
</Waypoint>
<GpsFix>
    <StartGpsFix/>
</GpsFix>
<Loiter>
    <LatitudeLongitude latitude="28.0"
                       longitude="88.0"/>
    <LoiterDepth value="0.0"/>
    <Duration value="120.0"/>
</Loiter>
<Quit/>
```

Figure 10: An AVCL task-level command sequence corresponding to the Seahorse AUV command of Figure 9.

The CFG implemented in the AUVW to support the REMUS tasking language 40 unary and 40 binary production rules and generates both the location and objective portions of a REMUS mission file.

While advantageous at parsing time, maintenance of both objective and location information in the same parse tree requires some special handling during translation. In the AUVW implementation, the parse tree is traversed twice: once to build the location reference table and once to translate the objectives into AVCL. The nature and sequential execution of REMUS objectives allows most of them to be mapped directly to AVCL task-level commands. Of the 14 objective types, nine can be represented a

sequence of one or more task-level commands. The remaining six objective types are REMUS-specific and are not directly representable in AVCL. As with the Phoenix AUV CFL, these commands require the use of AVCL meta commands. Additionally, a number of parameters to other commands (such as the "Track ping interval" and "Sidescan range" parameters in Figure 11) are vehicle-specific enough require the use of meta commands.

```
[Location]
Type= Waypoint
Label= WPT
Destination= 41N31.070 70W41.935
Offset direction= 0.0
Offset distance (meters)= 0.0
Offset Y axis(meters)= 0.0

[Objective]
Type= Navigate
Destination= WPT
Offset direction= 0.0
Offset distance (meters)= 0.0
Offset Y axis(meters)= 0.0
Minimum range (m.)= 10
Depth control mode= Depth (meters)= 5.0
Speed= 2.3 m/s
Timeout (seconds)= 300
Follow trackline= Yes
Track ping interval (secs.)= 0
Sidescan range= 30

[Objective]
Type= Navigate
Destination= WPT
Offset direction= 0.0
Offset distance (meters)= 1000.0
Offset Y axis(meters)= 200.0
Minimum range (m.)= Same
Depth control mode= Depth (meters)= 5.0
Speed= 4.0 knots
Timeout (seconds)= Auto
Follow trackline= No
Track ping interval (secs.)= 0
Sidescan range= Same
```

Figure 11: REMUS AUV objectives defining two waypoints. The first is specified using a reference to the location "WPT" while the second is defined as an offset from "WPT".

6. CONCLUSIONS

The implementation of automated parsing and translation of four AUV command languages into a common data format (in this case AVCL) using CFGs and XML data binding demonstrates the viability of this approach for the automated conversion of structured non-XML data to an XML format. Generally speaking, it is complementary to XSLT in that it provides a fairly simple method of converting between XML and non-XML data formats. From an AUV data-management standpoint, the implication is that a common data model can be utilized for more or less arbitrary AUVs with conversions providing the vehicle-specific support.

Related NPS research is being conducted into the development use of a common data model for UV operations. Subtopics in this area in addition to the one described in this paper include the development of higher-level declarative UV mission specification, conversion between task-level and declarative mission specifications and the development of XSLT-based conversions from AVCL to various vehicle-specific data formats. Additionally, the AUVW is being utilized as a testbed for arbitrary UV planning, physics-based mission rehearsal and playback, automated UV data archiving and runtime monitoring and control of UVs [6].

ACKNOWLEDGEMENT

The author wishes to acknowledge the financial support of the Navy Modeling and Simulation Office (NMSO), Naval Undersea Warfare Center (NUWC) CARUSO project, Navy Research Laboratory-Stennis Space Center, and the Singapore Defense Science and Technology Agency and Defense Science Organization National Laboratories.

REFERENCES

[1] Brutzman, D. P., *A Virtual World for an Autonomous Underwater Vehicle*, Ph.D. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, December 1994. http://web.nps.navy.mil/~brutzman/dissertation

[2] Cormen, T. H., Leiserson, C. E. and Rivest, R. L., *Introduction to Algorithms*, McGraw-Hill Book Company, 1990.

[3] Crangle, C. and Suppes, P., *Language and Learning for Robots*, Center for the Study of Language and Information Publishing, 1994.

[4] Daconta, M. C., Obrst, L. J. and Smith, K. T., *The Semantic Web, A Guide to the Future of XML, Web Services, and Knowledge Management*, Wiley Publishing, 2003.

[5] Davis, D. T., *Precision Maneuvering of the Phoenix Autonomous Underwater Vehicle for Entering a Recovery Tube*, Masters Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, September 1996.

[6] Davis, D. T. and Brutzman, D. P., "The Autonomous and Unmanned Vehicle Workbench: Mission Planning Mission Rehearsal, and Mission Replay Tool for Physics-Based X3D Visualization," *Proceedings of the 14th International Symposium on Unmanned Untethered Submersible Technology*, Durham, NH, August 2005.

[7] Hawkins, D. L. and Van Leuvan, B. C., *An XML-Based Mission Command Language for Autonomous Underwater Vehicles*, Masters Thesis, Department of Information Sciences, Naval Postgraduate School, Monterey, California, June 2003.

[8] Hopcroft, J., Motwani, R. and Ullman, J., *Introduction to Automata Theory, Languages, and Computation*, 2nd Edition, Addson-Wesley, 2001.

[9] Hydroid, Inc., *Technical Manual Operations and Maintenance Instructions, Remote Environmental Measuring Units (REMUS)*, 2001.

[10] Kay, M., *XSLT Programmer's Reference*, 2nd Edition, Wiley Publishing, 2003.

[11] Lee, C. S., *NPS AUV Workbench: Collaborative Environment for Autonomous Underwater Vehicles (AUV) Mission Planning and 3D Visualization*, Masters Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, California, March 2004.

[12] Marco, D., "Procedure to Run Missions with the ARIES," Naval Postgraduate School Center for Autonomous Underwater Vehicle Research internal document, September 2001.

[13] Means, W. S. and Bodie, M. A., *The Book of SAX, The Simple API for XML*, No Starch Press, 2002.

[14] Naval Postgraduate School (NPS) Technical Report NPS-MAE-04-002, *Naval Postgraduate School Center for Autonomous Underwater Vehicle Research 2003 Annual Report*, edited by Kragelund, S., 15 March 2004.

[15] Neushul, J. D., *Interoperability, Data Control and Battlespace Visualization Using XML, XSLT, and X3D*, Masters Thesis, Naval Postgraduate School, Monterey, California, September 2003.

[16] Sun Microsystems, Inc., *Java Architecture for XML Binding (JAXB)* Version 1.0, edited by Fialli, J. and Vajjhala, S., January 2003. Available at http://www.sun.com/xml/jaxb/jaxb-docs.pdf

[17] Weekley, J., Brutzman, D., Healey, A., Davis, D. and Lee, D., "AUV Workbench: Integrated 3D for Interoperable Mission Rehearsal, Reality and Replay," *Proceedings of the Mine Warfare Association Australian-American Mine Warfare Conference*, Canberra, Australia, February 2004. Available at http://www.movesinstitute.org/xmsf/projects/AUV/AUVWorkbenchIntegrated3D-MinwaraCanberraAustraliaFebruary2003.pdf

[18] World Wide Web Consortium, XSL Transformations (XSLT) Version 1.0 Recommended Specification, edited by Clark, J., October 2001. Available at http://www.w3.org/TR/2001/REC-xsl-20011015

[19] World Wide Web Consortium (W3C), Extensible Markup Language (XML) 1.0 (Third Edition) Recommended Specification, edited by Bray, T., Paoli, J., Sperberg-McQueen C. M., Maler, E., and Yergeau, F., February 2004. Available at http://www.w3.org/TR/REC-xml

[20] World Wide Web Consortium, XML Schema Part 1: Structures Second Edition Recommended Specification, edited by Biron, Thompson, H. S., Beech, D., Maloney, M. and Mendelsohn, N., October 2004. Available at http://www.w3.org/TR/xmlschema-1

[21] World Wide Web Consortium, XML Schema Part 2: Datatypes Second Edition Recommended Specification, edited by Biron, P. V. and Malhotra, A., October 2004. Available at http://www.w3.org/TR/xmlschema-2

[22] World Wide Web Consortium, Document Object Model Level 3 Core Specification Recommendation, edited by Le Hors, A., Le Hegaret, P., Wood, L., Nicol, G., Robie, J., Champion, M. and Byrne, S., April 2004. Available at http://www.w3.org/DOM