The Dynamic Compact Control Language: A Compact Marshalling Scheme for Acoustic Communications

Toby Schneider
and Henrik Schmidt
Center for Ocean Engineering
Department of Mechanical Engineering
Massachusetts Institute of Technology
Cambridge, MA 02139
tes@mit.edu and henrik@mit.edu

***Abstract*—The Dynamic Compact Control Language (DCCL) extends the ubiquitous Extensible Markup Language (XML) to provide a structure for defining very short messages comprised of bounded basic variable types, suitable for transmission over a low throughput acoustic channel. Algorithms are provided to consistently encode and decode the fields of these messages, and an implementation of DCCL with encryption is provided as a open source C++ library. Furthermore, DCCL has been incorporated into a publish/subscribe robotic autonomy architecture and used on numerous simulations and field trials involving heterogeneous networks of vehicles; we present the results of several. The ease of reconfiguration and error checking provided by DCCL make it well suited for collaborative autonomous underwater vehicle operations, where the flexibility to quickly change the message set, combined with low incidence of error, is necessary for success.**

I. Introduction

Sound is the most practical carrier for digital messages over any appreciable distance (i.e. O(1) km or further) in the sea. However, due to a variety of physical realities pertaining to sound transduction and propagation in water, acoustic communications (especially between moving nodes) is characterized by a list of undesirables: low data rates, high latency, a large number of errors, and drop-outs. These challenges are well summarized by Baggeroer [1] [2] and Partan [3].

Much work has been done on the communications systems required to perform acoustic telemetry [4], and mature systems have been developed such as the WHOI Micro-Modem [5]. Also, substantial research on the lower level networking protocols such as medium access control (MAC) has been performed: a number of underwater MAC schemes and their relative merits are discussed in [6]. Now that the hardware is relatively robust, it is possible to make meaningful strides in higher level acoustic networking. The subject of this paper, data marshalling, resides in this category.

In response to this state of acoustic communications, we developed the Dynamic Compact Control Language (DCCL), a language for defining highly compact messages. Given constraints on data rates afforded by modern acoustic modems (and enforced by physics at a more fundamental level), it appears that minimizing the size of command and data messages is a necessary goal for usable subsea networks.

DCCL is comprised of two components: 1) a structure language based on XML with which to define messages (described in section II); and 2) a C++ library (*libdccl*, detailed in section III) that validates the XML structure and implements consistent encoding and decoding of each message. *libdccl* is provided with the `goby-acomms` project, freely available under the GNU General Public License from <http://launchpad.net/goby>.

Thus far, DCCL has been used primarily with the Mission Oriented Operating Suite (MOOS), a publish/subscribe autonomy architecture for autonomous underwater vehicles (AUVs). Thus, several additional features (discussed in section IV) were developed for *libdccl* to facilitate use of DCCL with this and similar architectures. DCCL is presently being used by a number of institutions: NURC (La Spezia, Italy), NUWC (Newport, RI), WHOI (Woods Hole, MA), NAVSEA (Panama City, FL), and MIT (Cambridge, MA). In section V, we mention results from a subset of field trials which used DCCL.

A. Design goals

In order to produce messages as small as possible, DCCL offers these features:

- Defined bounded field types with customizable ranges. For example, an integer with minimum value of 0 and maximum value of 5000 takes 13 bits instead of the 32 bits often used for the integer type, regardless of whether the full integer type is needed.
- Dissolved byte boundaries (unaligned messages): fields in the message can be an arbitrary number of bits. Octets (bytes) are only used in the final message produced.
- Delta encoding of correlated data (e.g. CTD instrument data): rather then sending the full value for each sample in a message, each value is differenced from both a pre-shared key and the first sample within the message.

We also wanted to remove some of the complexity and potential sources of human error involved in binary encoding and bit arithmetic. To make DCCL straightforward, we made several design choices:

- All bounds on types can be specified as any number, such as powers of ten, rather than restricting the message designer to powers of two. This leads to a small inefficiency since the message is encoded by powers of two, but this drawback is balanced by the value of simplicity since the human mind is much more comfortable with powers of ten than powers of two.
- XML is the basis of the markup language that defines the structure of a DCCL message. XML was chosen for its ubiquity (e.g. XHTML for the web, RSS for news, KML for Google Earth), which means a host of tools are already available for editing and checking the validity of DCCL messages.
- Encoding and decoding for basic types are predefined and handled automatically by the DCCL C++ library (*libdccl*), meaning that in the vast majority of the cases no new code needs to be written to create or redefine a DCCL message. Writing code on cruises is always a risky endeavor, and minimizing that risk is important to maximizing use of ship time. However, flexibility to define custom algorithms to assist with encoding is provided for the fairly rare case when the basic encoding does not satisfy the needs of a particular message.

B. State of the art

1) Compact Control Language: This work owes inspiration and part of the name to the Compact Control Language (CCL) developed at WHOI by Roger Stokey and others for the REMUS series of AUVs. An overview of CCL is available in [7], and the specification is given in [8]. In our experience, before DCCL, CCL was the *de facto* standard data marshalling scheme for acoustic networks based on the WHOI Micro-Modem.

DCCL is intended to build on the ideas developed in CCL but with several notable improvements. DCCL provides the ability for messages to adapt quickly to changing needs of the researchers without changing software code (i.e. *dynamic*). CCL messages are hard coded in software while DCCL messages are configured using XML.

Also, significantly smaller messages are created with DCCL than with CCL since the former uses unaligned fields, while the latter, with the exception of a few custom fields (e.g. latitude and longitude), requires that message fields fit into an even number of bytes. Thus, if a value needs eleven bits to be encoded, CCL uses two bytes (sixteen bits), whereas DCCL uses the exact number of bits (eleven in this case). DCCL also offers several features that CCL does not, including encryption, delta-differencing, and data parsing abilities.

To the best of the authors' knowledge (which is supported by Chitre, et al. in [9]), CCL is the only previous effort to provide an open structure for defining messages to be sent through an underwater acoustic network. Other attempts have been ad-hoc encoding for a specific project. In order not to trample on Stokey's work and maximize interoperability, we have made DCCL compatible with a CCL network, giving DCCL the CCL initial byte flag of 0x20 (decimal 32). This allows vehicles using CCL and DCCL to interoperate, assuming all nodes have appropriate encoders for both message languages.

2) Text Encoding: Two approaches to encoding that have proven useful in other applications for compressing data are dictionary coders (e.g. LZW [10]) and entropy coders (e.g. Huffman coding [11]). Both of these are successful on sparse data, such as human readable text. Their utility for the types of messages encountered commonly in marine robotics is limited, however. These messages tend to be short and full of numeric values, whose information entropy is much greater than that of human generated text.

Furthermore, the overhead cost incurred by these text encoders means that the compressed message may not be more efficient than the original message until a sizable amount of data (perhaps several kilobytes) has been encoded. This exceeds the size of individual frames in the WHOI Micro-Modem, meaning that in messages would have be split across frames and reassembled. Given the low throughput and high error rate of the acoustic channel, it is impractical to attempt to send a message that is more than several frames before being decodable. Furthermore, the resulting message from these text encoders is variable length, as the compressibility depends on the input data. This can cause further difficulties transporting these data across the acoustic network.

Given these considerations, we decided that currently available text encoders would not an acceptable solution to the problem at hand, i.e. creating short messages for acoustic communications.

3) Abstract Syntax Notation One: Abstract Syntax Notation One (ASN.1) is a mature and widely used standard for abstractly representing data structures (or messages) in a human-readable textual form. It also specifies a variety of rules for encoding data using the ASN.1 structures. In both these areas, ASN.1 is similar to DCCL: DCCL also provides a structure language (based on XML in this case), and a set of encoding rules. In fact, the rules used by DCCL are very similar to the ASN.1 unaligned Packed Encoding Rules (PER). For a good treatment of ASN.1, see Larmouth's book [12].

If DCCL used the ASN.1 notation, it could hope to draw on the advantages of being standards compliant. However, DCCL does not currently use the ASN.1 representation at this time for two main reasons:

1) Given the severe restrictions on message size due to the acoustic modem hardware, existing ASN.1 structures are unlikely to be useful, unless the designers were originally careful in specifying bounds on numerical types (e.g. INTEGER) and minimizing use of string types (UTF8STRING/IA5STRING). Thus, for simplicity of the DCCL specification, the authors prefer the XML specification given in section II and currently used by DCCL.

2) ASN.1 structures are commonly "compiled" into source code which is then compiled into the finished program. This does not allow for dynamic message structures, which is at the core of the DCCL goal. DCCL does not compile the message structure, but rather translates it into a collection of objects at runtime. We argue that for the underwater robotics research community, at least, changes to messages should not need recompilation of code. Perhaps as the field matures and messages become widely used and standardized, support for compiling of messages will become more desirable.

Support for ASN.1 may become a desirable goal in the future to take advantage of the knowledge base and experience of this well accepted standard. However, we will likely have to choose a tightly reduced subset of the ASN.1 specification to meet the restrictive demands of the underwater acoustic channel. One possible path would be to match the XML definition of DCCL to the ASN.1 XML Encoding rules. Then, either the ASN.1 definition or XML definition could be used to encode messages using the Packed Encoding Rules, which are similar to the rules already used in DCCL (see section III).

C. Hardware Layer

DCCL was developed initially for the WHOI Micro-Modem acoustic modem, a relatively mature and widely deployed system originally presented in [5]. The WHOI Micro-Modem appends error checking bits and destination/source addressing to the user's data (in our case, the DCCL message). The WHOI Micro-Modem has several fixed length frame sizes (32, 64, and 256 bytes) corresponding to different data rates (and modulation schemes).

However, DCCL can be used to encode a message of any fixed length, and thus can be used in any communications scenario where compact short messages are desirable. In our field trials, we use DCCL for both subsea (via the WHOI Micro-Modem) and surface communications (via an IEEE 802.11 UDP or TCP/IP network). This allows for a seamless transition between surface and subsea networks.

II. Defining Messages

DCCL messages are defined using a custom language built from XML. Thus, the message structure is given by a text file composed of a series of nested tags (e.g. `<message>`). Such files can be edited by any text editor or any of a large of tools designed specifically for composing XML. The basic tags needed to define a message are given in this section. A number of additional tags are available for interacting with the vehicle's autonomy architecture; these tags are described in section IV.

A. XML Specification

The full XML schema is available with the source code at <http://launchpad.net/goby>; here we give a summary of the tags. A DCCL message file always consists of the root tag `<message_set>` which has one or more `<message>` tags as its children. The `<message>` children are as follows:

- `<id>`: an identification number (9 bits, so `<id>` $\in [0, 511]$) representing this message to all decoding nodes [`unsigned integer`].
- `<name>`: a name for the message. This tag and `<id>` must *each* be a unique identifier for this message. [`string`].
- `<size>`: the maximum size of this message in bytes [`unsigned integer`]. DCCL may produce a smaller message, but will not validate this message XML file if it exceeds this size.
- `<header>`: the children of this tag allow the user to rename the header parts of the DCCL message. See Fig. 1 for a sketch of the DCCL header format. These names are used when passing values at encode time for the various header fields.
  - `<time>`: seconds elapsed since 1/1/1970 ("UNIX time"). In the DCCL encoding, this reduced to seconds since the start of the day, with precision of one second. Upon decoding, assuming the message arrives within twelve hours of its creation, it is properly restored to a full UNIX time.
    - `<name>`: the name of this field; optional, the default is "_time". [`string`]
  - `<src_id>`: a unique address (`<src_id>` $\in [0, 31]$) of the sender of this message. For a given experiment these short unique identifiers can be mapped on to more global keys (such as vehicle name, type, ethernet MAC address, etc.).
    - `<name>`: default is "_src_id". [`string`]
  - `<dest_id>`: the eventual destination of this message (also an `unsigned integer` in the range [0,31]). If this destination exists on the same subnet as the sender, this will also be the hardware layer destination id number.
    - `<name>`: default is "_dest_id". [`string`]
- `<layout>`: the children of this tag define the generic data fields of the message, which can be drawn from any combination of the following types, summarized in Table I.
  - `<bool>`: a boolean value.
    - `<name>`: the name of this field. [`string`]
  - `<int>`: a bounded integer value.
    - `<name>`: see `<bool><name>`.
    - `<max>`: the maximum value this field can take. [`real number`].
    - `<min>`: the minimum value this field can take. [`real number`].
  - `<float>`: a bounded real number value.
    - `<name>`: see `<bool><name>`.
    - `<max>`: see `<int><max>`.
    - `<min>`: see `<int><min>`.
    - `<precision>`: specifies the number of decimal digits to preserve. For example, a precision of "2" causes 1042.1234 to be rounded to 1042.12;

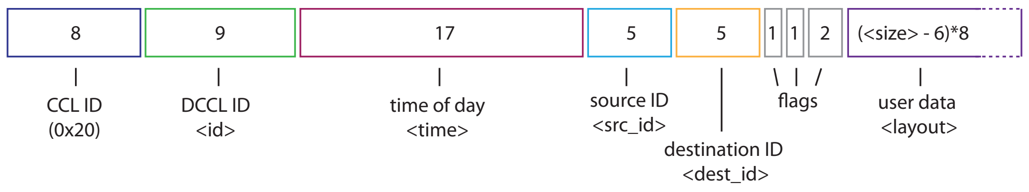


Fig. 1: Layout of the DCCL header, showing the fixed size (in bits) of each header field. The user cannot modify the size of these header fields, but can access and set the data inside through the same methods used for the customizable data fields specified in `<layout>`. The flags are not used by DCCL, but are included for use by the lower level networking.

TABLE I: Types supported by the Dynamic Compact Control Language

Type Name	DCCL XML Tag	C++ Type[a]
Bounded integer	`<int>`	`long int`
Bounded real	`<float>`	`double`
String	`<string>`	`std::string`
Enumeration	`<enum>`	`std::string`
Boolean	`<bool>`	`bool`
Pre-encoded hexadecimal	`<hex>`	`std::string`

[a] the *preferred* C++ type when encoding using *libdccl*, however any meaningful casts from other types (using streams from the `std` library) will be made.

a precision of “-1” rounds 1042.1234 to 1.04e3. [`integer`].

- `<string>`: a fixed length string value.
  - `<name>`: see `<bool><name>`.
  - `<max_length>`: the length of the string value in this field. Longer strings are truncated. `<max_length>4</max_length>` means “ABCDEFG” is sent as “ABCD”. [`unsigned integer`].
- `<enum>`: an enumeration of string values.
  - `<name>`: see `<bool><name>`.
  - `<value>`: a possible value the enumeration can take. Any number of values can be specified. [`string`].
- `<hex>`: a pre-encoded hexadecimal value.
  - `<name>`: see `<bool><name>`.
  - `<num_bytes>`: the number of bytes for this field. The string provided should have twice as many characters as `<num_bytes>` since each character of a hexadecimal string is one nibble (4 bits or $\frac{1}{2}$ byte). [`unsigned integer`].

B. Message Design

When designing a DCCL message, a few considerations must be made. Each message needs to be given a `<name>` and `<id>` unique within the DCCL network that this message is intended to live. Sometimes messages may have limited scope or may be mutually exclusive, in which case duplicate `<id>` numbers may be assigned.

Furthermore, the overall size of the message needs to be determined. This may be a constraint imposed by the hardware layer that this message is intended to traverse. In the case of the WHOI Micro-Modem, this should match the frame size of the intended data rate to be used (32 bytes for rate 0, 64 bytes for rate 2, and 256 bytes for rates 3 and 5). The size of the message is given by the header overhead (six bytes) and the sum of the sizes of the fields. The field sizes are calculated using the expressions given in the ”Size” column of Table II. These sizes are calculated at runtime with *libdccl*, so it is rarely necessary to calculate these by hand. However, these expressions give a sense of how much space a given field will typically take, which is important when considering how to type and bound the data.

An example XML message file, showing all the field tags, is provided in Fig. 2.

III. Algorithms and Implementation

Along with the XML message structure defined in section II, DCCL provides a set of consistent encoding and decoding tools in the C++ *libdccl* library, a piece of the freely available `goby-acomms` project (<http://launchpad.net/goby>). The class structure and sequence of using *libdccl* is modelled in Fig. 3. The tools provided by *libdccl* include:

- XML file parsing and validation using the Xerces-C++ XML Parser [13]. This ensures that the syntax of the XML file is valid and structure matches that of the DCCL schema.
- Calculation of message field sizes and comparison to the mandated maximum size (specified in the `<size>` tag). Messages exceeding this size are rejected and the designer must choose to remove and/or reduce fields or increase the message `<size>`.
- Encoding of DCCL messages using the expressions given in Table II. The user passes values of the C++ types given in Table I for all the fields in `<layout>` and desired fields in `<header>`. Fig. 2 provides an example of the encoding process for a DCCL message.
- Decoding of DCCL messages using the reciprocal of the expressions used for encoding. The user of *libdccl* will receive values of the C++ types as given in Table I for all header and layout fields.

TABLE II: Formulas for encoding the DCCL types.

DCCL Type	Size (bits)	Encode[a]
`<bool>`	2	$x_{enc} = \begin{cases} 2 & \text{if } x \text{ is true} \\ 1 & \text{if } x \text{ is false} \\ 0 & \text{if } x \text{ is undefined} \end{cases}$
`<enum>`	$\lceil \log_2(1 + \sum \epsilon_i) \rceil$	$x_{enc} = \begin{cases} i+1 & \text{if } x \in \{\epsilon_i\} \\ 0 & \text{otherwise} \end{cases}$
`<string>`	$length \cdot 8$	ASCII[b]
`<int>`	$\lceil \log_2(max - min + 2) \rceil$	$x_{enc} = \begin{cases} \text{nint}(x - min) + 1 & \text{if } x \in [min, max] \\ 0 & \text{otherwise} \end{cases}$
`<float>`	$\lceil \log_2((max - min) \cdot 10^{precision} + 2) \rceil$	$x_{enc} = \begin{cases} \text{nint}((x - min) \cdot 10^{precision}) + 1 & \text{if } x \in [min, max] \\ 0 & \text{otherwise} \end{cases}$
`<hex>`	$num\_bytes \cdot 8$	$x_{enc} = x$

· x is the original (and decoded) value; x_{enc} is the encoded value.
· $min, max, length, precision, num\_bytes$ are the contents of the `<min>`, `<max>`, `<max_length>`, `<precision>`, and `<num_bytes>` tags, respectively. ϵ_i is the ith `<value>` child of the `<enum>` tag (where $i = 0, 1, 2, \ldots$).
· nint(x) means round x to the nearest integer.
[a] for all types except `<string>` and `<hex>`, if data are not provided or they are out of range (e.g. $x > max$), they are encoded as zero ($x_{enc} = 0$) and decoded as not-a-number (NaN).
[b] the end of the string is padded with zeros to $length$ before encoding if necessary.

A. *Encryption*

libdccl provides encryption of the `<layout>` portion of the message using the Advanced Encryption Standard (AES or Rijndael) [14]. AES is a National Institute of Standards and Technology (NIST) certified cipher for securely encrypting data. It has been certified by the National Security Agency (NSA) for use encrypting top secret data.

libdccl uses a SHA-256 hash of a user provided passphrase to form the secret key for the AES cipher (see [15] for the specification of SHA-256). In order to further secure the message, an initialization vector (IV) is used with the AES cipher. The IV used for DCCL is the most significant 128 bits of a SHA-256 hash of the header of the message. Since the message header contains the time of day, it provides the continually changing value required of an IV. This ensures that the ciphertext created from the same data encrypted with the same secret key will only look the same in the future on a given day on the exact second it was created. The open source Crypto++ library available at [16] is used to perform the cryptography tasks.

IV. Incorporation with Autonomy Architecture

The primary use of *libdccl* thus far has been with the Mission Oriented Operating Suite (MOOS) autonomy architecture, explained by Benjamin, et al. in [17]. MOOS is a publish/subscribe infrastructure, where processes *publish* data to a central data bus (the `MOOSDB`) and receive messages from the data bus for which they had previously *subscribed*. To facilitate operation with such an architecture, *libdccl* provides an additional set of XML tags that allow messages to: 1) define the source variables from which to encode an outgoing message; 2) provide publish (i.e. destination) variables to post decoded data from an incoming message; and 3) provide "trigger" events that cause the creation of a DCCL message. These tags are optional and are ignored when using the regular encode/decode functions described in section III.

A. *Source variables*

As a child of any of the DCCL types[1] or header variables[2], the tag `<src_var key="">` indicates the name of a variable in the autonomy architecture that should be used to provide the value for this field when encoding. In the case of MOOS, this is a double or string MOOS variable. *libdccl* will perform a number of parsing and casting tasks on the value provided in order to fill the field. If the provided value is a `std::string` and the parameter "key" is given for the `<src_var>`, *libdccl* assumes the string is of the form `key1=value1,key2=value2,key3=value3...` and extracts the value for the given key from the string. This value then forms the value encoded into the given field of the DCCL message.

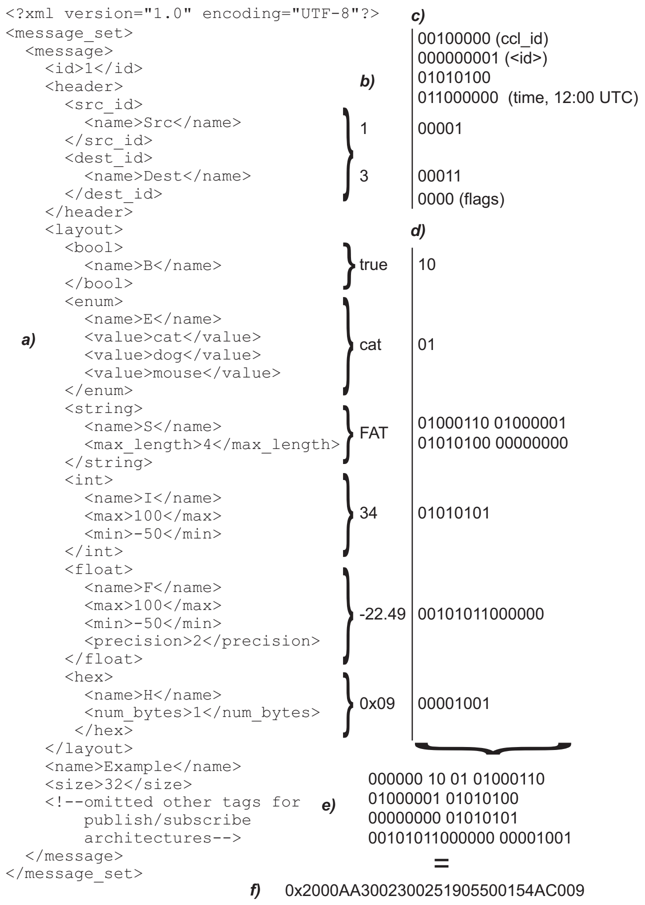
For all other C++ types, casting is done using by the `MessageVal` class to attempt to transform the data into a form acceptable for the given field of the DCCL message. `MessageVal` uses unbiased rounding and `std` streams to perform these casts. For example, the `double` 3.5 would be placed in an `<int>` field as 4. Similarly, the `std::string` "24.5" would be placed as 24.

B. *Trigger*

The `<trigger>` tag allows an event to be chosen that will be used to create a new DCCL message. Currently, two triggers are provided, one event driven and one time-based:

1) "publish": create a message when the variable specified by `<publish_var>` is published (i.e. written to) in the autonomy architecture.

[1]`<int>`, `<float>`, `<string>`, `<bool>`, `<enum>`, `<hex>`
[2]`<time>`, `<src_id>`, `<dest_id>`



```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <id>1</id>
    <header>
      <src_id>
        <name>Src</name>
      </src_id>
      <dest_id>
        <name>Dest</name>
      </dest_id>
    </header>
    <layout>
      <bool>
        <name>B</name>
      </bool>
      <enum>
        <name>E</name>
        <value>cat</value>
        <value>dog</value>
        <value>mouse</value>
      </enum>
      <string>
        <name>S</name>
        <max_length>4</max_length>
      </string>
      <int>
        <name>I</name>
        <max>100</max>
        <min>-50</min>
      </int>
      <float>
        <name>F</name>
        <max>100</max>
        <min>-50</min>
        <precision>2</precision>
      </float>
      <hex>
        <name>H</name>
        <num_bytes>1</num_bytes>
       </hex>
    </layout>
    <name>Example</name>
    <size>32</size>
    <!--omitted other tags for
        publish/subscribe
        architectures-->
  </message>
</message_set>
```

```
c)
00100000 (ccl_id)
000000001 (<id>)
01010100
011000000  (time, 12:00 UTC)
1       00001
3       00011
        0000 (flags)
d)
true    10
cat     01
FAT     01000110 01000001
        01010100 00000000
34      01010101
-22.49  00101011000000
0x09    00001001

e)
000000 10 01 01000110
01000001 01010100
00000000 01010101
00101011000000 00001001
=
f) 0x2000AA300230025190550015 4AC009
```

Fig. 2: Example of the DCCL encoding process. The process of encoding starts with the DCCL XML file (a). Data are provided by the application (b). *libdccl* encodes these data to binary via the algorithms given in Table II to form the header (c) and layout (d), concatenates and zero fills the encoded layout from most significant bit to closest byte (e) to produce the full encoded message (f). Finally, this point the message is encrypted (if desired).

2) "time": create a message every <`trigger_time`> seconds using on the newest available values of the <`src_var`>s in the architecture's database.

C. Publish variables

The <`on_receipt`> section of the XML file provides the user a place to specify any number of formatted variables to be published to the database of the publish/subscribe architecture once a DCCL message is received and decoded. Each <`publish`> tag defines a single variable to be published from some combination of the message fields. The children of <`publish`> include:

- <`format`>: a format string using the boost::format library conventions (which are a generalization of the `printf` specifiers, see [18]).
- <`publish_var`>: a variable name for where to publish this formatted value in the autonomy database.
- <`message_var`>: the <`name`> of one of the fields (<`int`>, <`float`> ...), the value of which will replace one of the specifiers in <`format`>. The order of these tags map onto the order of the specifiers given in the <`format`> tag.
- <`all`>: a shortcut for including all the fields of the message. This is equivalent to specifying <`message_var`> for every field in the message in the order declared in <`layout`>.

D. User supplied algorithms

While the basic encoding expressions given in Table II are sufficient for representing most data, occasionally the user wants to provide a simple pre-encode and post-decode algorithm of their own. An example of this would be to encode a logarithmic value or wrap an angle into the range $[0, 2\pi]$. In this case, the field tags (i.e. <`int`>, <`float`>, <`string`>, <`bool`>, <`enum`>, or <`hex`>) all take an optional parameter `algorithm`. If the `algorithm` parameter is provided, *libdccl* calls the user provided algorithm corresponding to a callback provided on startup of the library.

For example, the user provides a callback function called `log_function` which it passes to *libdccl* as the algorithm "log". Now, when *libdccl* encounters <`int algorithm="log"`> it passes the value intended for that field to the `log_function`. The return value of `log_function` is then used to encode the corresponding field of the message.

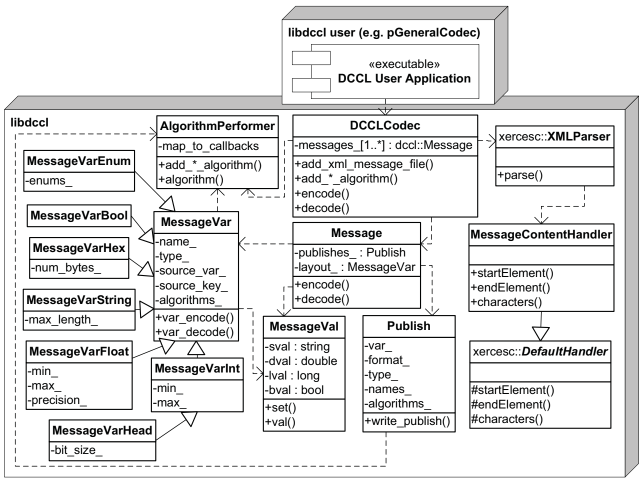
Similarly, the <`message_var`> tag used in the <`publish`> sections also takes the `algorithm` parameter, allowing for post-decoding algorithms to be processed.

E. MOOS Processes that use DCCL

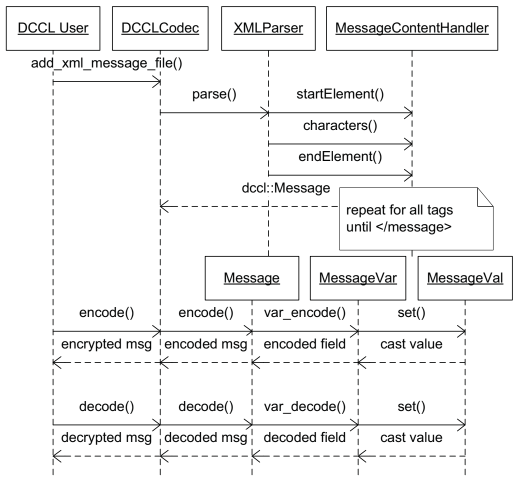
While DCCL is an entirely standalone project from MOOS, the MOOS processes that call *libdccl* are still the primary users of the library. Thus, a brief explanation of each process is given.

1) pGeneralCodec: This MOOS process acts as an interface between *libdccl* and the MOOS community. It subscribes for and publishes variables on behalf of *libdccl*. Given that *libdccl* already has substantial features for interacting with a publish/subscribe architecture (as detailed in other parts of this section), `pGeneralCodec` is little more than a shell around *libdccl* that handles the configuration and communication details specific to MOOS.

2) pAcommsHandler: This program acts as an interface between MOOS and the entire `goby-acomms` suite of libraries, which includes *libdccl* as well as libraries for handling medium access control (MAC), buffering, and low-level serial communication with the modem. `pAcommsHandler` calls the exact same libraries that `pGeneralCodec` does for the DCCL functionality, making `pGeneralCodec` unnecessary when `pAcommsHandler` is being run. `pGeneralCodec` is provided for users (such as surface vehicles) who wish a standalone MOOS DCCL encoder / decoder.



(a) Structure diagram. Dependencies between classes are indicated by a dashed arrow. Class generalizations are indicated with a solid arrow (e.g. `MessageVarInt` *is a* `MessageVar`).



(b) Sequence of using *libdccl*. The user initializes the DCCLCodec with one or more XML files and then proceeds to use the encode/decode methods as needed.

Fig. 3: Unified Modeling Language diagrams of *libdccl*. `XMLParser` and its dependencies handle the parsing of the XML file(s) into `Messages`. Each `Message` has one or more `MessageVars` that represent each field's structure. The `MessageVal` provides mapping of C++ types onto DCCL types (e.g. `std::string` → <`string`>) and performs casting if necessary (e.g. `double` → `std::string`). Each `Publish` represents a <`publish`> block and the `AlgorithmPerformer` calls user provided pre-encode and post-decode algorithms defined by the "algorithm" parameter to the field tags and <`message_var`> tags, respectively.

3) iCommander: A number of DCCL messages are being used as commands for changing the behavior of underwater vehicles during operations. `iCommander` provides a terminal-based graphical user interface (GUI) for a human to type in the fields for a given DCCL (command) message. Since `iCommander` uses *libdccl*, it reads the same XML files being used to actually encode and decode messages. Thus, any change to the XML files being used for commands is propagated to the command software (`iCommander`) without any further work.

V. Experimental Results

We have used the Dynamic Compact Control Language in several field trials involving autonomous surface and subsurface craft since its development. The acoustic communications hardware used was the WHOI Micro-Modem. Table III summarizes the location and assets involved in each trial. As we developed DCCL, we realized that messages could be classified into three rough categories: commands, collaboration, and data. Commands are messages sent from a topside vehicle operator to change the mission or redeploy the vehicle(s) to a different location to carry out the task at hand (acoustic sensing and/or environmental monitoring in these experiments). Collaboration messages are used to coordinate autonomous tasks amongst two or more vehicles, and data messages are sent from the vehicles to the topside operator with some kind of measured or computed data. Table IV lists all the messages, a total of seventeen, which we have created and used in field experiments.

The ease of defining and redefining DCCL messages allows for rapid prototyping of new experimental ideas during the field trial, rather than being rigidly confined to previously defined messages. For example, in SWAMSI09, we used two AUVs to perform bistatic acoustic detection of mine-like targets on the seafloor. Both AUVs traversed a circular pattern around the potential target, maintaining a constant bistatic angle. Entering into this collaboration and maintaining the correct angle required handshaking and data transfer between both vehicles. We were able to command the vehicles into this collaborative state with `LAMSS_DEPLOY`, and the `LAMSS_STATUS` message (with additional fields added to support this experiment) was passed between vehicles to maintain the correct positioning autonomously.

In GLINT09, DCCL messaging made another collaborative experiment possible. We had a mobile acoustic gateway (an autonomous surface craft with a WHOI Micro-Modem) available to stream high rate environmental and other data messages. By virtue of the surface craft staying near the AUV (made possible by the AUV's `LAMSS_STATUS` message), the AUV had a short acoustic propagation path to the surface craft. From there, the surface craft relayed data to the operators via IEEE 802.11 wireless ethernet. Also, the depth of the modem was controlled by a winch that the surface vehicle could command autonomously. Using the `WINCH_CONTROL` message, the AUV commanded the surface craft a depth at which to set the modem to improve communications. The

TABLE III: Summary of field trials.

Name	Summary	Assets	Experiment Datum[a]
SWAMSI09	Mine detection using bistatic acoustics.	2 Bluefin 21 AUVs, 1 WHOI Comm Buoy	30.045°N, 85.726°W
GLINT09	Interoperability of marine vehicles for passive acoustic target detection	1 NURC OEX AUV, 1 OceanServer Iver2 AUV, 2 Robotic Marine Kayaks, 2 Ship-deployed WHOI Micro-Modems	42.47°N, 10.9°E
DURIP09	Engineering test for collaborative autonomy and towed array improvements	2 Bluefin 21 AUVS, 2 Robotic Marine Kayaks, 1 Ship-deployed WHOI Micro-Modem.	42.35°N, 70.95°W
CHAMPLAIN09	Thermocline gradient following.	1 OceanServer Iver2 AUV, 1 Ship-deployed WHOI Micro-Modem.	42.2511°N, 73.3612°W

[a] The experiment datum is a location in the southwest corner of the operation region from which all vehicle positions are referenced using the Universal Transverse Mercator projection with the WGS 84 ellipsoid [19].

TABLE IV: Summary of DCCL Messages used in field experiments

Message Name	Category	Experiments Used[a]	Size (bytes)	Field Count	Description
SENSOR_DEPLOY	Command	SWAMSI09	28	17	DCCL Mimic of CCL Sensor Command - Deploy
SENSOR_PROSECUTE	Command	SWAMSI09	32	22	DCCL Mimic of CCL Sensor Command - Prosecute
SENSOR_STATUS	Data / Collaboration	SWAMSI09	32	24	DCCL Mimic of CCL Sensor Report - Status
SENSOR_CONTACT	Data	SWAMSI09	32	24	DCCL Mimic of CCL Sensor Report - Contact
SENSOR_TRACK	Data	SWAMSI09	31	24	DCCL Mimic of CCL Sensor Report - Track
ACTIVE_CONTACTS	Data	SWAMSI09	32	22	Active acoustic contact report message.
ACTIVE_TRACKS	Data	SWAMSI09	32	20	Active acoustic tracking message.
LAMSS_DEPLOY[b]	Command	All	31	22	Underwater vehicle command message.
LAMSS_STATUS[b]	Data / Collaboration	All	26	20	Vehicle Status message (position, speed, Euler angles, autonomy state)
LAMSS_CONTACT	Data	GLINT09	29	24	Passive acoustic contact report message.
SURFACE_DEPLOY	Command	GLINT09, DURIP09	13	10	Command message for surface vehicles.
ACOUSTIC_MOOS_POKE	Command	GLINT09, DURIP09	32	3	Underwater debugging / safety message.
SOURCE_ACTIVATION	Collaboration	GLINT09	5	2	Vehicle to buoy source command message.
WINCH_CONTROL	Collaboration	GLINT09	3	1	Underwater vehicle to surface vehicle command message.
BTRCODEC	Data	GLINT09, DURIP09	256	Varies	Beam-Time Record Data from a towed passive acoustic array.
CTDCODEC	Data	All	256	Varies	Salinity, temperature, depth data from a CTD instrument (delta-encoded).

[a] See Table III for a list of the experiments.
[b] See the appendix for the full XML definition of these messages.

AUV was performing a bistatic acoustic detection of a mid-water column depth target. The source, mounted on a buoy, was autonomously turned on and off by the AUV using the `SOURCE_ACTIVATION` message. The AUV, which was towing an acoustic array, was the receiver. None of this multi-robot collaboration would have been possible without the ability to define new messages quickly and with a high degree of confidence in their syntactical correctness.

The third case study is the CHAMPLAIN09 adaptive environmental experiment. In this experiment, a small AUV outfitted with a Conductivity-Temperature-Depth (CTD) instrument was deployed to study the thermocline structure of Lake Champlain. The AUV was commanded, using a updated `LAMSS_DEPLOY` message, on the task of adaptively surveying the thermocline. The vehicle accomplished this task by performing series of sinusoidal ("yoyo") depth maneuvers and streamed its samples back using the delta-encoded `CTDCODEC` message. In this manner, the environmental data was made available in near realtime (i.e. delayed by no more than a few minutes) to the AUV operator (see Fig. 4). Currently, the delta encoding was provided by a separate assistant process to *libdccl*, but this feature will shortly be incorporated into *libdccl* itself.

VI. Conclusion

Dynamic Compact Control Language (DCCL) provides a framework for *defining* messages in a reconfigurable manner using XML and *encoding* them consistently. The language can be used to make efficient short messages suitable for sending through presently available acoustic modem hardware,

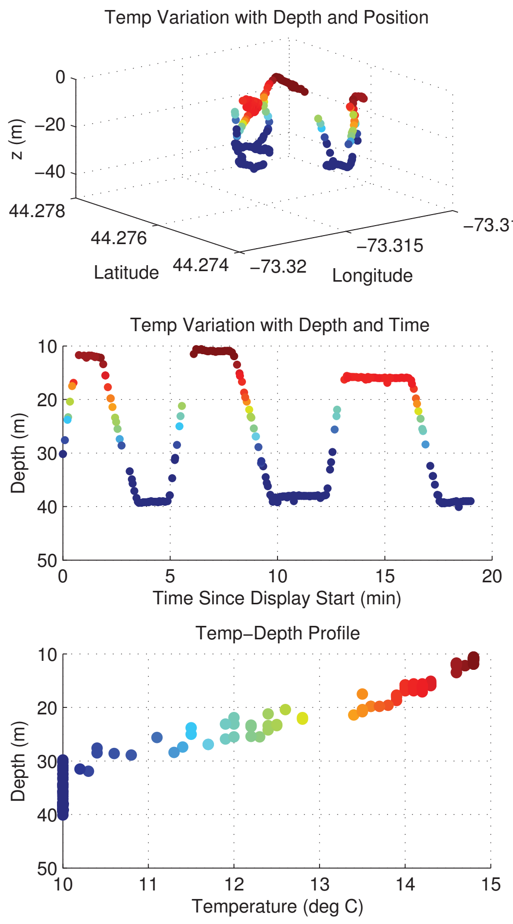


Fig. 4: Temperature data available to the AUV topside operator in near realtime from the CTDCODEC message at the CHAMPLAIN09 experiment. The "time since display start" is October 05, 2009 at 16:26:03 UTC.

or through other bandwidth restricted channels.

Using DCCL, we have developed a set of messages to support our operations, using MOOS as the autonomy architecture, the WHOI Micro-Modem as our communications hardware, and a variety of different subsea and surface robots. From these case studies, we hope that others will find inspiration to use DCCL, and we encourage those who are interested in using or improving DCCL to contact us or visit the goby-acomms project website at <http://launchpad.net/goby>.

APPENDIX

For reference and concrete examples of DCCL messages that have been used in field experiments, we present two of our more commonly used messages. In Fig. 5a, we give the LAMSS_STATUS message, sent by a vehicle to provide the operators and other vehicles with the current position and pose (i.e. speed and Euler angles). Also, a simplified version of the LAMSS_DEPLOY message, sent by the operators to command the vehicle into another autonomy state, is presented in Fig. 5b.

ACKNOWLEDGMENT

This work was funded by the Office of Naval Research (ONR) under projects N00014-08-1-0011 and N00014-08-1-0013.

We thank Roger Stokey for CCL, which formed the basis for this work. Also, we appreciate the support of Lee Freitag and the rest of the WHOI Acoustic Communications group on using the WHOI Micro-Modem. Finally, we thank all those who made the field trials possible, from the crew to the science team to the cooks.

REFERENCES

[1] A. Baggeroer, "Acoustic telemetry–An overview," *IEEE J. Ocean. Eng.*, vol. 9, no. 4, pp. 229–235, 1984.
[2] D. Kilfoyle and A. Baggeroer, "The state of the art in underwater acoustic telemetry," *IEEE J. Ocean. Eng.*, vol. 25, no. 1, pp. 4–27, 2000.
[3] J. Partan, J. Kurose, and B. N. Levine, "A survey of practical issues in underwater networks," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 11, no. 4, pp. 23–33, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1347372
[4] M. Stojanovic, "Recent advances in high-speed underwater acoustic communications," *IEEE J. Ocean. Eng.*, vol. 21, no. 2, pp. 125–136, 1996.
[5] L. Freitag, M. Grund, S. Singh, J. Partan, P. Koski, and K. Ball, "The WHOI Micro-Modem: an acoustic communications and navigation system for multiple platforms," in *IEEE Oceans Conference*, 2005.
[6] E. M. Sozer, M. Stojanovic, and J. G. Proakis, "Underwater acoustic networks," *IEEE J. Ocean. Eng.*, vol. 25, no. 1, p. 7283, 2000.
[7] R. P. Stokey, L. E. Freitag, and M. D. Grund, "A compact control language for AUV acoustic communication," *Oceans 2005-Europe*, vol. 2, p. 11331137, 2005.
[8] R. P. Stokey, "A compact control language for autonomous underwater vehicles," WHOI, Tech. Rep. Public Release 1.0, 2005. [Online]. Available: http://acomms.whoi.edu/ccl/
[9] M. Chitre, S. Shahabudeen, and M. Stojanovic, "Underwater acoustic communications and networking: Recent advances and future challenges," *The State of Technology in 2008*, vol. 42, no. 1, pp. 103–114, 2008.
[10] T. Welch, "Technique for high-performance data compression." *Computer*, vol. 17, no. 6, pp. 8–19, 1984.
[11] D. Huffman, "A method for the construction of minimum-redundancy codes," *Resonance*, vol. 11, no. 2, pp. 91–99, 2006.
[12] J. Larmouth, *ASN.1 Complete*. Elsevier, 2000. [Online]. Available: http://www.oss.com/asn1/larmouth.html
[13] Apache, "Xerces-C++ XML parser." [Online]. Available: http://xerces.apache.org/xerces-c/
[14] J. Daemen and V. Rijmen, "AES proposal: Rijndael," 1999. [Online]. Available: http://www.cryptosoft.de/docs/Rijndael.pdf
[15] "Secure hash signature standard," NIST, Tech. Rep. FIPS PUB 180-2. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
[16] W. Dai, "Crypto++ library 5.6.0." [Online]. Available: http://www.cryptopp.com/
[17] M. R. Benjamin, J. J. Leonard, H. Schmidt, and P. M. Newman, "An overview of moos-ivp and a brief users guide to the ivp helm autonomy software," MIT, Tech. Rep. MIT-CSAIL-TR-2009-028, 2009. [Online]. Available: http://hdl.handle.net/1721.1/45569
[18] S. Krempp, "The boost format library." [Online]. Available: http://www.boost.org/doc/libs/1_34_0/libs/format
[19] NIMA, "Department of defense world geodetic system 1984: Its definition and relationships with local geodetic systems. second edition, amendment 1," NIMA, Tech. Rep. TR8350.2, 2000. [Online]. Available: http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf

```xml
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>LAMSS_STATUS</name>
    <id>20</id>
    <size>32</size>
    <header>
      <src_id algorithm="to_lower,name2modem_id">
        <name>Node</name>
        <src_var>VEHICLE_NAME</src_var>
      </src_id>
      <time>
        <name>Timestamp</name>
      </time>
    </header>
    <layout>
      <enum algorithm="to_lower">
        <name>Type</name>
        <src_var>VEHICLE_TYPE</src_var>
        <value>kayak</value>
        <value>asc</value>
        <value>auv</value>
        <value>ship</value>
        <value>buoy</value>
        <value>glider</value>
        <value>usv</value>
        <value>unknown</value>
      </enum>
      <int>
        <name>nav_x</name>
        <src_var>NAV_X</src_var>
        <max>100000</max>
        <min>-100000</min>
      </int>
      <int>
        <name>nav_y</name>
        <src_var>NAV_Y</src_var>
        <max>100000</max>
        <min>-100000</min>
      </int>
      <float>
        <name>Speed</name>
        <src_var>NAV_SPEED</src_var>
        <max>20</max>
        <min>-2</min>
        <precision>1</precision>
      </float>
      <float algorithm="angle_0_360">
        <name>Heading</name>
        <src_var>NAV_HEADING</src_var>
        <max>360</max>
        <min>0</min>
        <precision>2</precision>
      </float>
      <float>
        <name>Depth</name>
        <src_var>NAV_DEPTH</src_var>
        <max>5000</max>
        <min>-10</min>
        <precision>1</precision>
      </float>
      <float>
        <name>Altitude</name>
        <src_var>NAV_ALTITUDE</src_var>
        <max>5000</max>
        <min>-10</min>
        <precision>1</precision>
      </float>
      <float>
        <name>Pitch</name>
        <src_var>NAV_PITCH</src_var>
        <max>1.57</max>
        <min>-1.57</min>
        <precision>2</precision>
      </float>
      <float>
        <name>Roll</name>
        <src_var>NAV_ROLL</src_var>
        <max>1.57</max>
        <min>-1.57</min>
        <precision>2</precision>
      </float>
    </layout>
    <on_receipt>
      <publish>
        <publish_var>
          STATUS_REPORT_IN
        </publish_var>
        <all />
      </publish>
    </on_receipt>
  </message>
</message_set>
```

(a) `LAMSS_STATUS` message used to report vehicle position and pose.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>LAMSS_DEPLOY</name>
    <id>18</id>
    <trigger>publish</trigger>
    <trigger_var
     mandatory_content="MessageType=LAMSS_DEPLOY">
      OUTGOING_COMMAND
    </trigger_var>
    <size>32</size>
    <header>
      <time>
        <name>Timestamp</name>
      </time>
      <src_id>
        <name>SourcePlatformId</name>
      </src_id>
      <dest_id>
        <name>DestinationPlatformId</name>
      </dest_id>
    </header>
    <layout>
      <enum>
        <name>Deploy_Mode</name>
        <value>LOWPOWER</value>
        <value>RETURN</value>
        <value>RACETRACK</value>
        <value>ZIGZAG</value>
        <value>BISTATIC</value>
        <value>TRAIL</value>
      </enum>
      <enum>
        <name>Depth_Mode</name>
        <value>SINGLE</value>
        <value>DUAL</value>
        <value>YOYO</value>
        <value>ADAPTIVE_YOYO</value>
      </enum>
      <enum>
        <name>Sonar_Control</name>
        <value>ON</value>
        <value>OFF</value>
      </enum>
      <float>
        <name>Deploy_Duration</name>
        <max>604800</max>
        <min>0</min>
        <precision>-1</precision>
      </float>
      <int>
        <name>Deploy_X</name>
        <max>100000</max>
        <min>-100000</min>
      </int>
      <int>
        <name>Deploy_Y</name>
        <max>100000</max>
        <min>-100000</min>
      </int>
      <int>
        <name>Deploy_Depth</name>
        <max>255</max>
        <min>0</min>
      </int>
      <int>
        <name>Alternate_Depth</name>
        <max>255</max>
        <min>0</min>
      </int>
      <int>
        <name>Operation_Radius</name>
        <max>262000</max>
        <min>0</min>
      </int>
      <int>
        <name>Radius_Period</name>
        <max>1023</max>
        <min>0</min>
      </int>
      <int>
        <name>Segments</name>
        <max>31</max>
        <min>0</min>
      </int>
      <int algorithm="angle_0_360">
        <name>Survey_Heading</name>
        <max>360</max>
        <min>0</min>
      </int>
      <int>
        <name>Survey_Length</name>
        <max>4095</max>
        <min>0</min>
      </int>
      <int>
        <name>Survey_Width</name>
        <max>4095</max>
        <min>0</min>
      </int>
      <bool>
        <name>Clockwise</name>
      </bool>
      <int>
        <name>Trail_Range</name>
        <max>1023</max>
        <min>0</min>
      </int>
      <int algorithm="angle_0_360">
        <name>Trail_Angle</name>
        <max>359</max>
        <min>0</min>
      </int>
      <int>
        <name>GPS_Interval</name>
        <max>4095</max>
        <min>256</min>
      </int>
    </layout>
    <on_receipt>
      <publish>
        <publish_var>INCOMING_COMMAND</publish_var>
        <all />
      </publish>
      <publish>
        <publish_var>SONAR_CONTROL</publish_var>
        <message_var>Sonar_Control</message_var>
      </publish>
      <publish>
        <publish_var>DEPLOY_STATION</publish_var>
        <format>
          points=%1%,%2%
        </format>
        <message_var>Deploy_X</message_var>
        <message_var>Deploy_Y</message_var>
      </publish>
      <publish>
        <publish_var type="double">
          DEPLOY_RADIUS
        </publish_var>
        <message_var>Operation_Radius</message_var>
      </publish>
      <publish>
        <publish_var type="double">
          DEPLOY_DURATION
        </publish_var>
        <message_var>Deploy_Duration</message_var>
      </publish>
      <publish>
        <publish_var>ORBIT_RADIUS</publish_var>
        <message_var>Radius_Period</message_var>
      </publish>
      <publish>
        <publish_var>SENSOR_DEPLOY</publish_var>
        <format>
          polygon=radial:%1%,%2%,%3%,%4%#clockwise=%5%
        </format>
        <message_var>Deploy_X</message_var>
        <message_var>Deploy_Y</message_var>
        <message_var>Radius_Period</message_var>
        <message_var>Segments</message_var>
        <message_var>Clockwise</message_var>
      </publish>
      <publish>
        <publish_var>
          SENSOR_DEPTH_DEPLOY
        </publish_var>
        <format>depth=%1%</format>
        <message_var>Deploy_Depth</message_var>
      </publish>
      <publish>
        <publish_var>
          SENSOR_DEPTH_ALTERNATE
        </publish_var>
        <format>depth=%1%</format>
        <message_var>Alternate_Depth</message_var>
      </publish>
      <publish>
        <publish_var>GPS_CONFIGURATION</publish_var>
        <format>period=%1%</format>
        <message_var>GPS_Interval</message_var>
      </publish>
      <publish>
        <publish_var>SENSOR_YOYO</publish_var>
        <format>min_depth=%1%#max_depth=%2%</format>
        <message_var>Deploy_Depth</message_var>
        <message_var>Alternate_Depth</message_var>
      </publish>
      <publish>
        <publish_var>SURVEY_HEADING</publish_var>
        <format>heading=%1%</format>
        <message_var>Survey_Heading</message_var>
      </publish>
      <publish>
        <publish_var>ZIGZAG_CONFIG</publish_var>
        <format>
          heading=%1%#period=%2%#amplitude=%3%
        </format>
        <message_var>Survey_Heading</message_var>
        <message_var>Radius_Period</message_var>
        <message_var>Survey_Width</message_var>
      </publish>
      <publish>
        <publish_var>ZIGZAG_STATUS</publish_var>
        <format>
          points=zigzag:%1%,%2%,%3%,%4%,%5%,%6%
        </format>
        <message_var>Deploy_X</message_var>
        <message_var>Deploy_Y</message_var>
        <message_var>Survey_Heading</message_var>
        <message_var>Survey_Length</message_var>
        <message_var>Radius_Period</message_var>
        <message_var>Survey_Width</message_var>
      </publish>
      <publish>
        <publish_var>TRAIL_CONFIG</publish_var>
        <format>trail_range=%1%#trail_angle=%2%</format>
        <message_var>Trail_Range</message_var>
        <message_var>Trail_Angle</message_var>
      </publish>
      <publish>
        <publish_var>BISTATIC_CONFIG</publish_var>
        <format>desired_bistatic_angle=%1%</format>
        <message_var>Trail_Angle</message_var>
      </publish>
    </on_receipt>
  </message>
</message_set>
```

(b) `LAMSS_DEPLOY` message used to command underwater vehicles.

Fig. 5: XML structure of two messages used extensively in our field exercises.