The Dynamic Compact Control Language Version 3

Toby Schneider, Stephanie Petillo
GobySoft, LLC
North Falmouth, MA
Emails: toby@gobysoft.org,
stephanie@gobysoft.org

Henrik Schmidt
Massachusetts Institute of Technology
Center for Ocean Engineering
Cambridge, MA
Email: henrik@mit.edu

Christopher Murphy
Bluefin Robotics
Quincy, MA
Email: cmurphy@bluefinrobotics.com

***Abstract*—The Dynamic Compact Control Language (DCCL) provides a flexible and efficient way to marshall object-messages into very small datagrams. It is well suited to transmission over very low throughput links with small maximum transmission units, such as those commonly used in underwater (acoustic modem) and sea-surface (satellite) applications. DCCL provides a interface description language (IDL) and an extensible set of encoding/decoding algorithms. For example, these messages could be sensor data samples, autonomous underwater vehicle positions, or command and control messages for a fleet of vehicles or instruments.**

The DCCL IDL allows the message designer to bound message fields based on the physical origin of the data sample, including integrated support for static (compile-time) dimensions and units. The default encoders provide reasonable performance for a variety of applications; where more control is desired the DCCL library user can provide custom encoders/decoders for one or more of a given message's fields.

I. Introduction

Inter-vehicle and vehicle-to-operator digital communication is an essential component of collaborative autonomous vehicle missions. However, the available links (such as acoustic modems, satellite radio, and ground-wave radio) tend to have very low throughput (often less than 100 bits per second) due to the physical limitations of the carrier and the power constraints of the autonomous platforms. See Fig. 1 for a comparison of the nominal latencies and bandwidth for these "slow" links and typical terrestrial links. Thus, the information throughput available for collaborative underwater vehicle tasks is low unless significant source encoding is performed.

Due to this need for efficient source encoding, existing data marshalling schemes for marine vehicle communication are often tailored specifically to a given application. As the number of fielded systems grow, the need for interoperability has also grown. The Dynamic Compact Control Language (DCCL) provides an interface description language (IDL) for marshalling and encoding object-based messages for transmission over very slow links. This IDL also includes optional support for static (compile-time) dimensional analysis using common (e.g. SI) or user-defined systems of units.

In addition to the IDL, the open source DCCL reference library (*libdccl3*) provides a default set of numeric encoders that provide acceptable performance for many applications. However, many applications require specific encoders for optimal performance, so DCCL provides an easily extensible shared library plugin mechanism for user-provided encoders.

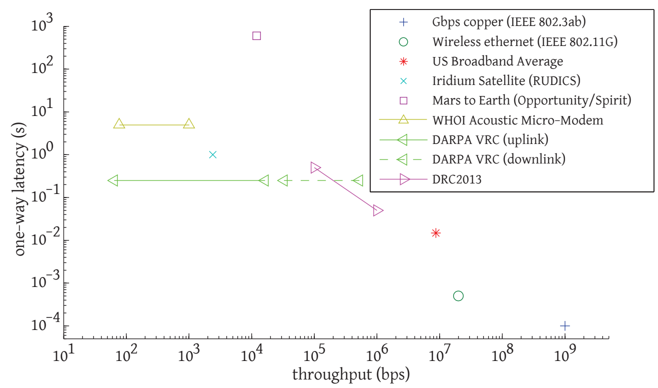


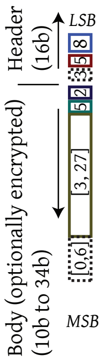
Fig. 1. Logarithmic comparison of nominal latency and bandwidth values for commonly encountered network links (first three quantities) with links used in sea, space, and disaster-relief (as envisioned by the DARPA Robotics Challenge [DRC]) robotics. DCCL was originally designed for the acoustic modem regime represented by the WHOI acoustic Micro-Modem and has been adopted for use on the Iridium and disaster relief (DARPA Virtual Robotics Challenge [VRC] and DRC2013 [1]) links.

This design aims to provide enough generality to be standardizable, but with the flexibility to tackle specific encoding problems when necessary.

This paper presents the design and several applications of DCCL version 3 (DCCL3), the first version released as a standalone project (prior versions are bundled with the Goby Underwater Autonomy Project [2]). The aim of the standalone DCCL3 release is to expand the use of the project into other robotics domains with similar physical link constraints, such as degraded terrestrial links in the event of natural or man-made disasters. While preserving backwards-compatibility with DCCL2, DCCL3 adds compile-time dimensions and units support and improves the performance of the default encoders. The DCCL3 source code and technical documentation are freely available from http://libdccl.org.

II. DCCL Interface Description Language

The DCCL IDL uses the Google Protocol Buffers (GPB) language [3] as the framework for defining messages; an example is given in Fig. 2a. GPB provides a language-neutral way to define object-based messages and a well-documented way to extend its language. Each message is defined as one or more fields, where the fields can be any of a number of typical primitive types (various floating point and integer




```
import "dccl/protobuf/option_extensions.proto";

message CommandMessage
{
  option (dccl.msg) = { id: 125  max_bytes: 32  codec_version: 3 }

  required int32 destination = 1
            [(dccl.field)  = { max: 31 min: 0 in_head: true }];
  optional string description = 2
            [(dccl.field).omit = true];
  enum SonarPower { NOMINAL = 10; LOW = 5; OFF = 0; }
  optional SonarPower sonar_power = 10;
  required double speed = 11
            [(dccl.field) = { units { base_dimensions: "LT^-1" }
                              max: 2.0  min: -0.5 precision: 1 }];
  repeated int32 waypoint_depth = 12
            [(dccl.field) = { units { base_dimensions: "L" }
                              max: 40  min: 0 max_repeat: 4 }];
}
```

(a) Message definition using DCCL. This message definition is compiled into an analogous C++ class using the standard GPB compiler *protoc* (with the DCCL plugin if static units support is desired). On the left is the size of each field in bits.

x_{enc} (bin)	x_{enc} (dec)	x
11111010	250	id: 125 (CommandMessage)
00011	3	destination: 3
000	(padding)	
10	2	sonar_power: LOW (i = 1)
10001	17	speed: 1.2
100	4 [10 15 10 12]	waypoint_depth: [10, 15, 10, 12]
[001010		
001111		
001010		
001100]		
000000	(padding)	

LSB hex: fa 03 46 2a 8f c2 00 MSB
bin: 11111010 00000011 01000110 00101010 10001111 11000010 00000000

(b) Example of encoding the DCCL message in (a) for a representative set of values. The table gives the unencoded x and encoded x_{enc} values; below the table is the encoded message in little endian format (both in hexadecimal and binary notation).

Fig. 2. Definition and encoding example of a basic DCCL message for commanding an underwater vehicle to perform several depth maneuvers while running a sonar. The same message is 21 bytes using the GPB default encoder, compared to 7 bytes using DCCL.

types, booleans, strings, etc.). Fields can also be instantiations of a child message, also referred to as an embedded message field. DCCL extends the GPB language to add additional metadata which provides a framework for more efficient default encoders and user-defined custom encoders (discussed in detail in Section III).

The additional metadata offered by the DCCL IDL is in two categories: message extensions which modify the entire DCCL message, and field extensions which modify a given field of the message. Table I provides the set of available DCCL message and field extensions. The DCCL message extensions provide a numeric identification tag ("id") for the message which is sent to allow the decoder to know which message is to be decoded. In addition, the message extensions allow the message designer to optionally control which codecs are used to decode a given message. The field extensions provide additional bounds for the value of that field, as well a means of controlling the codec to be used on a field-by-field basis.

A. Static Units of Measure Support

Since the DCCL field bounds (min, max, and precision) are often based off the physical origins of the data, it is important to define the units of measure of those fields. The DCCL IDL has support for defining the units of a numeric field's quantity. When using the DCCL C++ library, this support is directly connected to the Boost Units C++ library [4]. The units of a given field are given by two parameters: the physical dimension (e.g. length, force, mass, etc.), and the unit system which defaults to the International System of Units (SI) [5]. The units of the field can also be specified directly, outside of a canonical system (e.g. nautical mile, fathom, yard, knot, etc.).

The fields defined with units generate additional C++ methods using the DCCL plugin to the GPB compiler (*protoc*). These additional methods provide accessors and mutators for the dimensioned Boost Units quantities, with full static "unit safety"[1], and correct conversions between different units of the same dimensions (e.g. feet to meters).

The Units field extension has the following options:

- `base_dimensions` (string): Specifies the dimensions of the field as a combination of powers of the base dimensions given in Table II. For example, acceleration would be defined as `"LT^-2"`.
- `derived_dimensions` (string): As a convenience alternative to the `base_dimensions` specification, any of the Boost Units "derived dimensions" can be used. For example instead of `base_dimensions: "L^-1 M T^-2"` for pressure, one can use `derived_dimensions: "pressure"`. Multiplication and division of derived dimensions is also supported using the "*" and "/" operators.
- `system` (string, defaults to "si"): A boost::units or user-defined system of units to use for this field. Defaults to the SI system with base units of kelvin (temperature), second (time), meter (length), kilogram (mass), candela (luminous intensity), mole (amount of substance) and ampere (electric current).
- `relative_temperature` (bool, defaults to false): A special extension only used for temperature fields. Setting this to true means that the temperature is relative (i.e. a difference of absolute temperatures) instead of an absolute temperature. This matters to support correct unit conversions between different temperature systems. For example, relative degrees Kelvin are the same as relative degrees Celsius, but the absolute scales differ by 273.15 degrees.

[1]We define unit safety as static (compiler-checked) dimensional analysis. The term is a blending of the (computer science) notion of type safety with (physical) dimensional analysis. For example, in a unit-safe system, the compiler will not allow the user to set a field with dimensions of length to a quantity of hours.

TABLE I
DEFINITION OF THE DCCL INTERFACE DESCRIPTION LANGUAGE

Message Extensions[a]			
Extension Name	Extension Type	Explanation	Default
(dccl.msg).id	int32	Unique identifying integer for this message	-
(dccl.msg).max_bytes	uint32	Enforced upper bound for the encoded message length	-
(dccl.msg).codec_version	int32	Default codec set to use (corresponds to DCCL major version)	2
(dccl.msg).codec	string	Name of the codec to use for encoding the base message.	dccl.default2
(dccl.msg).codec_group	string	Group of codecs to be used for encoding the fields.	dccl.default2

Field Extensions[b]					
Extension Name	Extension Type	Explanation	Applicable Fields	Symbol	Default
(dccl.field).precision	int32	Decimal digits to preserve; can be negative.	double, float	p	0
(dccl.field).min	double	Minimum value that this field can contain (inclusive)	(u)intN[c], double, float	x_m	-
(dccl.field).max	double	Maximum value value that this field can contain (inclusive)	(u)intN, double, float	x_M	-
(dccl.field).max_length	uint32	Maximum length (in bytes) that can be encoded	string, bytes	L_M	-
(dccl.field).max_repeat	uint32	Maximum number of repeated values.	all *repeated*	r_M	-
(dccl.field).codec	string	Codec to use for this field (if omitted, the defaults given in Table III are used)	all	-	-
(dccl.field).omit	bool	Do not include field in encoded message (default = false)	all	-	False
(dccl.field).units	Units[d]	Physical dimensions and units system information	(u)intN, double, float	-	-

[a] Extensions of `google.protobuf.MessageOptions`
[b] Extensions of `google.protobuf.FieldOptions`
[c] (u)intN refers to any of the integer types: int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64
[d] See Section II-A for definition of the Units class.

TABLE II
BASE DIMENSIONS IN DCCL

Physical dimension	Symbol character
length	L
time	T
mass	M
plane angle	A
solid angle	S
current	I
temperature	K
amount	N
luminous intensity	J
information	B
dimensionless	-

- `unit` (string): As an alternative to the `dimensions` and `system` specification, the field can be set to use particular (typically non-SI) units. A few examples of such units that are still often encountered in the marine domain are `unit: "metric::nautical_mile"`, `unit: "metric::bar"`, and `unit: "us::yard"`.

For example, to set an AUVStatus message's `x` and `y` fields to meters (the default for the base dimension of length, since the default system is SI), and then later access them as nautical miles, one can use this C++ example:

```
using namespace boost::units;
typedef metric::nautical_mile_base_unit::unit_type
   NauticalMile;

AUVStatus status;
status.set_x_with_units(1000*si::meters);
status.set_y_with_units(500*si::meters);

quantity<NauticalMile> x_nm(status.x_with_units());
quantity<NauticalMile> y_nm(status.y_with_units());
```

The value of `x_nm` is 0.54 nautical miles and `y_nm` is 0.27 nautical miles.

III. DCCL ENCODERS/DECODERS

From the DCCL IDL, the user can instantiate a message in C++ and then encode it using the DCCL reference library. Unless otherwise specified, fields are encoded using the DCCL3 defaults. For special applications, user-defined codecs can be used in place of some or all of the default encoders.

A. Defaults

The DCCL default field encoders/decoders ("codecs") achieves lossless compression for all numeric fields through bounded types with customizable ranges and decimal precisions. For example, an integer (perhaps representing vehicle depth in meters) with minimum value of 0 and maximum value of 5000 takes 13 bits instead of the 32 or 64 bits typically used for an integer type. As a complete example, Fig. 2b gives the encoding for a realization of the message defined in Fig. 2a.

The precise mathematical formulations of the default field encoders are given in Table III (the decoders are exactly the inverse operation, so they are omitted to save space). Intuitively, the codecs are split into two groups: numeric (integers, floats, enumerations[2], booleans[3]) and others (strings, bytes).

Numeric values are all encoded essentially the same way. Integers are treated as floating point values with zero precision, where precision is defined as the number of (base 10) decimal places to preserve (e.g. precision = 3 means round to the

[2] Enumerations can be considered integers with bounds based on the size of the defined set of values.

[3] Booleans can be considered integers with only two possible values: 0 or 1.

closest thousandth, precision = -1 means round to the closest tens). Thus, integer fields can also have negative precision, if desired. Fields are bounded by a minimum and maximum allowable value, based on the underlying source of the data.

To encode, the numeric value is rounded to the desired precision, and then multiplied by the appropriate power of ten to make it an integer. Then it is increased or decreased so that zero (0) represents the minimum encodable value. At this point, it is simply an unsigned integer. To encode the optional field's "not set" state, an additional value (not an additional bit) is reserved. To allow "not set" to be the zero (0) encoded value, all other values are incremented by one.

This default encoder assumes unset fields are rare. If a message commonly has unset optional fields, it would be more efficient to implement a "presence bit" encoder that uses a separate bit to indicate if a field is set or not. These are two extremes of the more general purpose idea of an entropy encoder, such as the arithmetic encoder. In that case, "not set" is simply another symbol that has a probability mass relative to the actual values to capture the frequency with which fields are set or not set.

B. *Custom Encoders*

Along with the default encoder reference implementation, the DCCL library includes two sets of custom encoders: a set that provides WHOI Compact Control Language [6] compatibility, and a set that implements an arithmetic encoder for a user-provided data probability model. This latter set can be used to provide highly compact encoding of data streams with low entropy. Such data sources are those that can be modeled well *a priori* (e.g. physical oceanographic measurements from a Conductivity-Temperature-Depth (CTD) sensor, navigation trajectory of the vehicle, or target track predictions) and thus only the difference between the model and the data needs to be sent (which is therefore inexpensive to send with a properly designed arithmetic encoder model). An application of the DCCL arithmetic encoder for minimally encoding the position of an autonomous underwater vehicle is given in [7].

Any DCCL3 library user can define their own set of codecs by creating a shared library that subclasses the appropriate DCCL field codec classes. Custom codecs can use any algorithm they wish as long as they conform to two requirements: 1) codecs must always by able to produce a maximum and minimum encoded size based on the message's description only, and 2) the decoder must consume the exact number of bits that the encoder produced.

C. *Encoding Algorithm*

The encoded DCCL message is split into three conceptual sections: the DCCL id (which identifies to the decoder which message is to be decoded), the header, and the body as shown in Fig. 2a. Either the header or the body may be empty (zero bytes). The main purpose of the header is to provide a nonce for encrypting the body of the message, if desired. For best results, this assumes the header includes a constantly varying value, such as a timestamp.

DCCL messages are always encoded and decoded from the least significant bit to the most significant bit, where **appending** new bits to an existing Bitset means concatenating the new bits with the existing Bitset starting at the most significant bit of the existing Bitset. The field Codec encode functions are given in Table III. Given that, the DCCL encoding process is defined by the following encode algorithm:

function ENCODE(DCCL Message m)
    Bitset $b, b_{id}, b_{head}, b_{body} \leftarrow \varnothing$
    $b_{id} \leftarrow$ EncodeId(m.id)
    append b_{id} to b
    Fields $f_{head} \leftarrow m$.fields where in_head is True
    $b_{head} \leftarrow$ EncodeFields(f_{head})
    append b_{head} to b
    Fields $f_{body} \leftarrow m$.fields where in_head is False
    $b_{body} \leftarrow$ EncodeFields(f_{body})
    optionally **encrypt** b_{body} using b_{head} as a nonce.
    append b_{body} to b
return b
function ENCODEID(Id i)
    Bitset $b_{id} \leftarrow \varnothing$
    Codec $c \leftarrow$ default (dccl.msg).id or user-defined codec.
    $b_{id} \leftarrow c$.encode(i)
return b_{id}
function ENCODEFIELDS(Fields fields)
    Bitset $b_{fields} \leftarrow \varnothing$
    for all fields **as** f **do**
        Codec $c \leftarrow$ FindCodec(f)
        Bitset $b_f \leftarrow \varnothing$
        if f is an child message **then**
            $b_f \leftarrow$ EncodeFields(f.fields)
        else
            $b_f \leftarrow c$.encode(f)
        append b_f to b_{fields}
    while b_f **mod** 8 **is not** 0 **do**
        append 0 to b_{fields}
    return b_{fields}
function FINDCODEC(Field f)
    if (dccl.field).codec is set **then return** that codec.
    else if f is a child message and (dccl.msg).codec is set in the child message definition **then return** that codec.
    else if (dccl.msg).codec_group is set in the parent message **then return** that codec.
    else
        return the codec for (dccl.msg).codec_version

IV. EXAMPLE MESSAGES AND PERFORMANCE

DCCL can send any type of data that can be defined as an object-oriented message. However, it is often valuable to have several examples for commonly used problems. In the marine sensors and vehicles domain, we can often split data into three categories:

TABLE III
DEFAULT DCCL FORMULAS FOR ENCODING THE FIELDS FOR DIFFERENT DATA TYPES.

GPB Type	Size (bits) (q)	Encode[a]
	(dccl.msg).id header (varint)	
int32	$\begin{array}{ll} 8 & \text{if } x \in [0, 128) \\ 16 & \text{if } x \in [128, 32768) \end{array}$	$x_{enc} = \begin{cases} x \cdot 2 & \text{if } x \in [0, 128) \\ x \cdot 2 + 1 & \text{if } x \in [128, 32768) \end{cases}$
	required fields	
bool	1	$x_{enc} = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{if } x \text{ is false} \end{cases}$
enum	$\lceil \log_2(\sum \epsilon_i) \rceil$	$x_{enc} = i$
(u)intN	$\lceil \log_2(x_M - x_m + 1) \rceil$	$x_{enc} = \begin{cases} x - x_m & \text{if } x \in [x_m, x_M] \\ 0 & \text{otherwise} \end{cases}$
double, float	$\lceil \log_2((x_M - x_m) \cdot 10^p + 1) \rceil$	$x_{enc} = \begin{cases} \text{nint}((x - x_m) \cdot 10^p) & \text{if } x \in [x_m, x_M] \\ 0 & \text{otherwise} \end{cases}$
string (of length L)	$\lceil \log_2(L_M + 1) \rceil + \min(L, L_M) \cdot 8$	$x_{enc} = L + \sum_{n=0}^{\min(L, L_M)} x[n] \cdot 2^{8n + \lceil \log_2(L_M+1) \rceil}$
bytes	$L_M \cdot 8$	$x_{enc} = x$
Message	$\sum q_{subfields}$	x_{enc} for each field of *Message* appended to the previous (recursive encoding).
	optional fields	
bool	2	$x_{enc} = \begin{cases} 2 & \text{if } x \text{ is true} \\ 1 & \text{if } x \text{ is false} \\ 0 & \text{if } x \text{ is not set} \end{cases}$
enum	$\lceil \log_2(1 + \sum \epsilon_i) \rceil$	$x_{enc} = \begin{cases} i + 1 & \text{if } x \in \{\epsilon_i\} \\ 0 & \text{otherwise} \end{cases}$
(u)intN	$\lceil \log_2(x_M - x_m + 2) \rceil$	$x_{enc} = \begin{cases} x - x_m + 1 & \text{if } x \in [x_m, x_M] \\ 0 & \text{otherwise} \end{cases}$
double, float	$\lceil \log_2((x_M - x_m) \cdot 10^p + 2) \rceil$	$x_{enc} = \begin{cases} \text{nint}((x - x_m) \cdot 10^p) + 1 & \text{if } x \in [x_m, x_M] \\ 0 & \text{otherwise} \end{cases}$
string	same as *required*; empty string treated as "not set"	
bytes	$\begin{array}{ll} 1 + L_M \cdot 8 & \text{if } x \text{ is set} \\ 1 & \text{if } x \text{ is not set} \end{array}$	$x_{enc} = \begin{cases} x \cdot 2 + 1 & \text{if } x \text{ is set} \\ 0 & \text{if } x \text{ is not set} \end{cases}$
Message	$\begin{array}{ll} 1 + \sum q_{subfields} & \text{if } x \text{ is set} \\ 1 & \text{if } x \text{ is not set} \end{array}$	$x_{enc} = \begin{cases} \textit{required } x_{enc} \text{ appended to } 1 & \text{if } x \text{ is set} \\ 0 & \text{if } x \text{ is not set} \end{cases}$
	repeated fields (of size r)	
all	$\lceil \log_2(r_M + 1) \rceil + r_M \cdot q_{required}$	From LSB to MSB: 1. Size r is encoded using the *required* (u)intN encoder (with $x_m = 0, x_M = r_M$). 2. *required* x_{enc} is calculated for each repeated element then appended to the previous encoded element.

Symbols (in addition to those defined in Table I):
· x is the original (and decoded) value.
· $x[n]$ is the ASCII value of the nth character of the string.
· x_{enc} is the encoded value.
· ϵ_i is the ith child of the enumeration definition (where $i = 0, 1, 2, \ldots$), *not* the value assigned to the enum (which need not be sequential).
· nint(x) means round x to the nearest integer.

[a] If data are out of range (e.g. $x > max$ or $x < min$), for *optional* fields they are encoded as zero ($x_{enc} = 0$) and decoded as not set; for *required* fields, they are encoded as the min value. In the case of strings whose length exceeds L_M, the string is truncated to L_M before encoding. Thus, care should be taken not to exceed the min and max values to ensure the message is losslessly decodable.

1) Command and control messages: messages to be sent to reconfigure an AUV mission or sensor settings. Fig. 2 provides a small complete example of a message that could be sent to command a vehicle to traverse a set of waypoints at a given speed. Clearly, one could expand this 7-byte message to include much more information while still fitting in the O(10-100) byte maximum transmission units seen on marine data links (e.g. the WHOI Micro-Modem [8] uses 32 to 256 bytes; Iridium Short-Burst Data is 270-1960 bytes).
2) Vehicle navigation report messages: AUVs typically provide a navigation estimate (position, depth) and orientation angles that can be used to monitor missions and geolocate sensor samples. Fig. 3 gives a possible DCCL message for such a use (which is modeled off a similar message that the MIT Laboratory for Autonomous Marine Sensing Systems has used for all vehicle missions since 2009).
3) Sensor data messages: A Conductivity-Temperature-Depth (CTD) sensor is a widely used oceanographic instrument that measures conductivity of the seawater (from which salinity is computed), temperature, and pressure (from which depth is computed). These values can also be used to empirically compute the compressional speed of sound and the density of the water. Fig. 4 provides a means to transmit a sample with bounds that would work in much of the world's oceans shallower than 6000 meters. Increasing the bounds would make

```
import "dccl/protobuf/option_extensions.proto";

message AUVStatus {
  option (dccl.msg) = { id: 122
                        max_bytes: 32
                        codec_version: 3 };

  // Header
  required double timestamp = 1  [(dccl.field) = { codec: "_time"   in_head: true }];
  required int32 source = 2          [(dccl.field) = { min: 0  max: 31  in_head: true }];
  required int32 destination = 3   [(dccl.field) = { min: 0  max: 31  in_head: true }];

  // Body
  required double x = 4 [(dccl.field) = { units { base_dimensions: "L" }
                                                    min: -10000  max: 10000  precision: 1 }];
  required double y = 5 [(dccl.field) = { units { base_dimensions: "L" }
                                                    min: -10000  max: 10000  precision: 1 }];

  required double speed = 6 [(dccl.field) = { units { base_dimensions: "LT^-1" }
                                                         min: 0   max: 20.0  precision: 1 }];
  required double heading = 7 [(dccl.field) = { units { derived_dimensions: "plane_angle"
                                                                  system: "angle::degree" }
                                                           min: 0  max: 360.0  precision: 1 }];

  optional double depth = 8 [(dccl.field) = { units { base_dimensions: "L" }
                                                         min: 0  max: 6500  precision: 0 }];
  optional double altitude = 9 [(dccl.field) = { units { base_dimensions: "L" }
                                                          min: 0  max: 500  precision: 1 }];
  optional double pitch = 10 [(dccl.field) = { units { derived_dimensions: "plane_angle" }
                                                          min: -1.57  max: 1.57 precision: 2 }];
  optional double roll = 11 [(dccl.field) = { units { derived_dimensions: "plane_angle" }
                                                       min: -1.57  max: 1.57 precision: 2 }];
  optional MissionState mission_state = 12;
  enum MissionState { IDLE = 0;  SEARCH = 1; CLASSIFY = 2; WAYPOINT = 3; }

  optional DepthMode depth_mode = 13;
  enum DepthMode { DEPTH_SINGLE = 0;  DEPTH_YOYO = 1;
                              DEPTH_BOTTOM_FOLLOWING = 2; }
}
```

Fig. 3. Example DCCL message definition for reporting the status (position, pose, and basic mission state) of an autonomous underwater vehicle. x and y are assumed to be offsets from an operation datum using a local cartesian or Universal Transverse Mercator coordinate system.

```
import "dccl/protobuf/option_extensions.proto";

message CTDMessage
{
  option (dccl.msg) = { id: 123  max_bytes: 32  codec_version: 3 };

  required double temperature = 1 [(dccl.field) = { units { derived_dimensions: "temperature"
                                                                     system: "celsius" }
                                                                     min: 0  max: 30  precision: 1 }];
  required int32 depth = 2 [(dccl.field) = { units { derived_dimensions: "length" }
                                                       min: 0  max: 6000 }];
  required double salinity = 4 [(dccl.field) = { min: 10  max: 40  precision: 1 }];
  required double sound_speed = 5 [(dccl.field) = { units { base_dimensions:  "LT^-1" }
                                                              min: 1450  max: 1550  precision: 1 }];
}
```

Fig. 4. Example DCCL message definition for reporting a sample from a Conductivity-Temperature-Depth sensor (where salinity and sound speed are pre-computed using the appropriate formulas).

it applicable for more environments; decreasing them would make smaller messages for more targeted applications (e.g. for use in shallow Mediterranean waters, one could increase the salinity minimum and decrease the depth maximum).

DCCL provides several qualitative benefits: for example, type safety and consistent applications of units with direct C++ support. In addition, it is possible to quantitatively evaluate the default encoder performance in terms of the size of the serialized messages. For this, we chose two baselines: the GPB built-in encoder, and the Python packed binary data class (`struct`) which is similar to the various ad-hoc techniques used for packing binary data that the authors have seen used in the field. Table IV gives the encoded message size in bytes for these two baselines and DCCL on the three example messages included in this paper. In these cases, DCCL gives an increased compression ratio of about 50 to 80% over these other marshalling schemes. This can be the difference between sending useful sensor data from the vehicle during the mission run and only sending navigation updates.

V. Conclusion

The Dynamic Compact Control Language version 3 provides a concise type-safe and unit-safe interface description language for object-based messages to be transmitted between ocean robots, “smart” sensors (e.g. data buoys, Argo floats) and human on ships or shore.

From the DCCL description of the message, the default or user-defined encoders can be used to achieve highly compressed datagrams suitable for transmission over the very low throughput links present in this domain.

We believe DCCL has suitability for compression of many types of data in other domains (e.g. outer space, human-restricted disaster sites) where highly compact messages are of importance due to the limitations of the physical links.

Acknowledgment

The authors would like to thank the many ship crews and sea-going organizations that have allowed this work to be tested and refined outside of an office, without which it would undoubtedly be less useful. We would also like to thank all our science teachers and professors who taught us that a value without a unit is meaningless. Finally, thanks to the MIT DARPA Robotics Challenge (DRC) team for allowing us an opportunity to use and refine DCCL in a non-marine domain.

References

[1] M. Fallon, S. Kuindersma, S. Karumanchi, M. Antone, T. Schneider, H. Dai, C. P. D'Arpino, R. Deits, M. DiCicco, D. Fourie *et al.*, “An architecture for online affordance-based perception and whole-body planning,” *Journal of Field Robotics*, 2014.

[2] T. Schneider and H. Schmidt, “Goby-Acomms version 2: extensible marshalling, queuing, and link layer interfacing for acoustic telemetry,” in *9th IFAC Conference on Manoeuvring and Control of Marine Craft, Arenzano, Italy*, 2012.

[3] Google, “Protocol buffers.” [Online]. Available: https://developers.google.com/protocol-buffers/

[4] M. C. Schabel and S. Watanabe, “Boost units.” [Online]. Available: http://www.boost.org/doc/libs/1_57_0/doc/html/boost_units.html

TABLE IV
DCCL PERFORMANCE

Message	Example Instantiation	GPB Size (bytes)	Python `struct` [9] Size (bytes)	DCCL Size (bytes)	DCCL Compression
CommandMessage (Fig. 2)	destination: 3, sonar_power: LOW, speed: 1.2, waypoint_depth: [10, 15, 10, 12]	21	15	7	53-67%
AUVStatus (Fig. 3)	timestamp: 1427316658, source: 1, destination: 2, x: 2326, y: 1100, speed: 1.1, heading: 152.4, depth: 2150, altitude: 100, pitch: 0.01, roll: -0.02, mission_state: SEARCH, depth_mode: DEPTH_BOTTOM_FOLLOWING	90	48	19	60-79%
CTDMessage (Fig. 4)	temperature: 10, depth: 50, salinity: 32, sound_speed: 1485	29	18	7	61-76%

[5] I. B. of Weights, Measures, B. N. Taylor, and A. Thompson, "The international system of units (SI)," 2001.

[6] R. P. Stokey, L. E. Freitag, and M. D. Grund, "A compact control language for AUV acoustic communication," in *Oceans 2005-Europe*, vol. 2. IEEE, 2005, pp. 1133–1137.

[7] T. Schneider and H. Schmidt, "A state observation technique for highly compressed source coding of autonomous underwater vehicle position," *Oceanic Engineering, IEEE Journal of*, vol. 38, no. 4, pp. 796–808, 2013.

[8] E. Gallimore, J. Partan, I. Vaughn, S. Singh, J. Shusta, and L. Freitag, "The WHOI MicroModem-2: A scalable system for acoustic communications and networking," in *OCEANS 2010*. IEEE, 2010, pp. 1–7.

[9] Python Software Foundation, "The Python Standard Library: struct – interpret strings as packed binary data." [Online]. Available: https://docs.python.org/2/library/struct.html