Argot – Data Encoding Technology

Abstract

This paper introduces Argot – a fully describing Data Encoding Technology that has the potential to change the way Computer Networks and Computer languages are built and evolve. It introduces the concept of a software alphabet, which enables data to be uniquely defined and will enable computer languages to evolve.

Introduction

In all areas of human endeavour, no great progress was made until the basis of the technology was understood and applied:

- In chemistry the discovery and use of the periodical table
- In the biological sciences the discovery that all cells are built from genes
- In ancient Egyptian, the discovery of the Rosetta stone enabled all Egyptian pictograms to be read and understood
- In human language an alphabet is the fundamental element that enables dictionaries to be assembled and from the dictionary, books can be written that describe all forms of human endeavour from novels, philosophy, science etc.

In computer science the fundamentals of computer hardware have never changed. All computer hardware is designed using components that manipulate "1" and "0" which are then formed into bytes and multiple bytes. These are the fundamental alphabet of every computer ever built and because the fundamental alphabets have never changed, progress in Hardware design has been rapid and prolific.

The same cannot be said for computer software. Pick up any book on computer software and we start reading about this or that language (COBOL, Pascal, C++, C#, Java etc.) or method of encoding data, which is also called a language (XML, IDL etc.). There has been no fundamental, unchangeable, fully self-describing alphabet upon which all other programs and data can be built. If you start building languages without a common alphabet, you end up with a progression of languages and data formats that are largely incompatible. This is the state of the computing industry today. All of these different languages and formats have, over time, created an overly complex environment in which ever more complex applications need to be built. If such a software alphabet could be found then software could evolve much more rapidly without the proliferation of formats and standards that exist today.

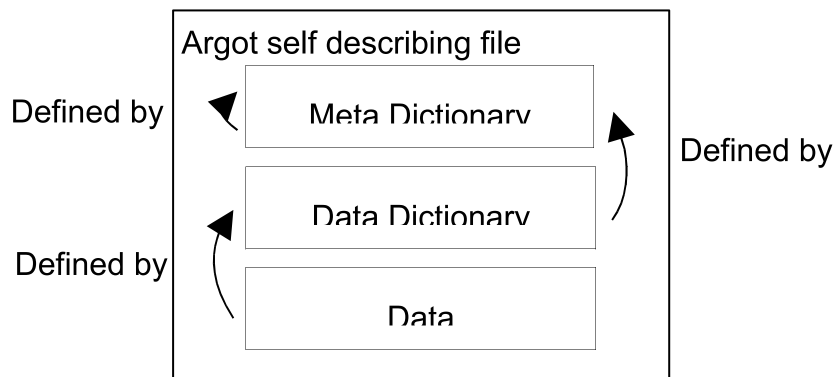
We do know that software and data are built from a small number of binary elements: bytes, double bytes, arrays, sequences etc. If we can find an alphabet that is defined using only these basic binary elements, we could then build dictionaries that uniquely define a field of human endeavour. We could build software and data from these dictionaries, without reference to multiple offline standards that get developed as the next closed software standard.

Argot is a radically new data encoding technology that has been developed by going back to first principles of language evolution.

- **Argot.meta.dictionary** is the fundamental, fully self-describing **alphabet** that all other Argot elements are built upon. Just as the English alphabet has 26 letters “A” to “z” that are unique and unchangeable, the Argot Alphabet has only 21 elements that are absolutely unique and unchangeable. The Argot alphabet is binary and can be fully communicated between computers with only 500 bytes.
- **Argot.Data.Dictionary** is a dictionary that is assembled for an application. It is defined using only the Argot.Alphabet. As with English dictionaries, books can be written with a simple dictionary (children’s stories) through to complex scientific papers. You only need to use the dictionary entries that are appropriate to your application.
- **Argot.Data** is application data defined using only elements defined in the Argot.Dictionary, and because the Argot.Dictionary is described using only the Argot.Alphabet, the data will be uniquely defined.

Argot uses techniques that are a departure from the methods used in data formats available today. It’s totally self-describing nature has many positive implications for how we communicate information in files, web sites, messaging systems and service-oriented architectures.

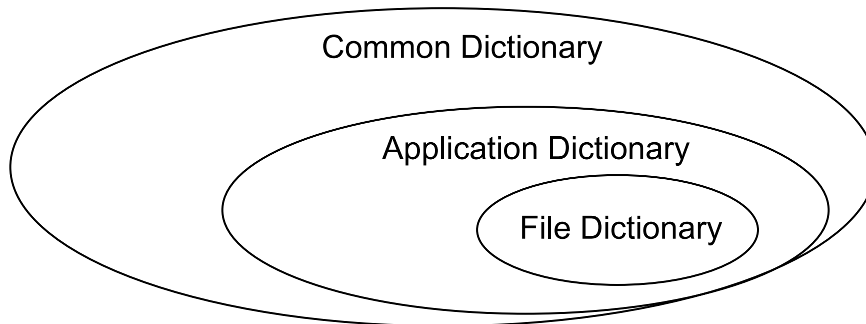
Argot uses a dictionary to give meaning to data. A single Argot file or message can contain data, the data dictionary which describes the format of the data, and a meta dictionary which describes the format of the data dictionary. This creates a purely self-referencing and self-describing data file with no external dependencies.



By removing these external dependencies, Argot allows systems and devices to communicate data in evolving formats. Data formats may be as complex or as simple as an application requires, and may be extended without removing meaning for other applications that share the data. These features make Argot an ideal match for distributed systems, data storage, embedded applications, and proprietary web applications.

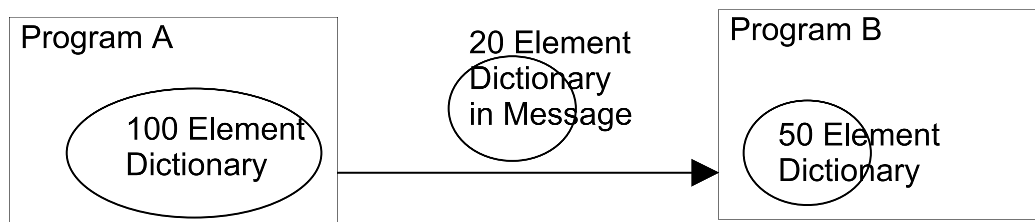
Argot provides the strong typing you receive from a system like XML Schema, but without the constraints of an external static schema. Because all definitions are local to a file, an application is able to confirm whether a file’s data format description conforms to its own. This completely localizes the interaction between application and data, removing the need for systems and files to refer to an external static schema.

An Argot designer uses the concept of a common dictionary to choose the required data types for an application. The common dictionary is able to grow over time and can contain information from a variety of domains. Each concept in the dictionary is defined within the context of other types in the dictionary. After the data types have been chosen from the common dictionary, an application dictionary is created. The application dictionary is a subset of the common dictionary and only includes the data types the application needs to communicate.

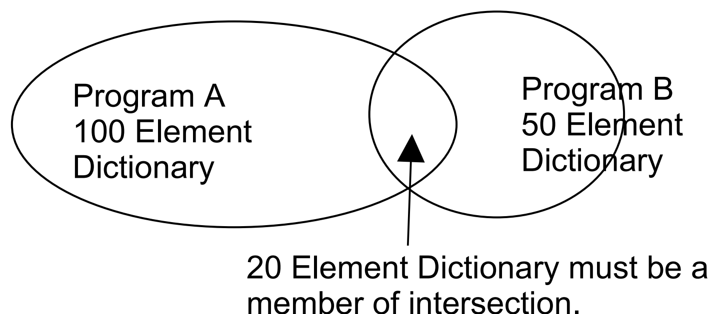


When the application creates an Argot enabled file, the file itself contains the complete description of the data being sent. The file's dictionary is a subset of the data types of the application dictionary. In this way each application and file in a system contains all the knowledge about the information it is able to process or data it contains.

The receiver of an Argot enabled file is able to read the dictionary and compare the data types of its own dictionary with that of the files. Once the types of the file dictionary have been matched with that of the application reading the file, the data can be read. This completely removes the need for a static common domain schema. Each application and file in effect contains its own schema.



This can be re-illustrated using the following set theory:



The ability to compare dictionaries and ensure that the formats are compatible gives Argot its flexibility. The use of a common dictionary allows concepts to be updated and changed without the versioning consequences found with XML or ASN.1.

When a new concept is introduced into a system, the programmer adds the additional new code into the applications that must process the new concept. The Argot library allows the new marshalling code to be added without disrupting other parts of the application.

Argot and its dictionary format are binary. This ensures the most flexibility and only a small overhead for dictionary information during communications. The binary nature of the dictionary and data also allows the Argot library reading the data to be small, making the library and format appropriate for embedded applications.

Argot allows a file or application to contain its own specification. This removes the need for a file to directly adhere to external schemas. As a direct result, two systems, which have previously had no contact, are able to discover dynamically if they have common data types to allow communications between them. This has a number of implications in the way we build future software and communications systems. This section examines how Argot can be applied to solve issues currently facing the software development industry.

Argot Applications

As already discussed, Argot allows a file or application to contain its own specification, removing the need for a file to directly adhere to external schemas. This has a number of implications in the way we build future software and communications systems. This section of the white paper examines how Argot can be applied to solving the current issues facing the software development industry.

Colony

Argot was originally developed as the basis for a client server remote procedure call system called Colony. The same techniques used to compare a file's dictionary with an application's were originally designed to compare and match the data types dynamically between a client and server's dictionary.

In asynchronous communications, such as remote procedure calls, Argot is able to define a virtual common dictionary. The virtual dictionary can be extended over the life of the conversation allowing client and server to discover the common types they share. Once again, this is without any common external static specification as would be used in XML or ASN.1.

In our first implementation of Argot enabled asychronous middleware (**Colony**), we have implemented the concept of a network virtual computer or Drone. The Drone uses the Argot dictionary to define the concept of a Heap, Instructions and code pointer: the base concepts of a computer. Each instruction of the drone is also defined using Argot. This allows the system to implement and change instructions to meet the needs of a specific environment. A client and server that implement a Drone are able to dynamically discover if both systems contain the correct instructions to execute the drone.

This ability to ensure data types are correct we refer to as strong data type binding and is analogous with the strong typing found in programming languages. This type of strong binding serves the same purpose as languages; to speed development and find errors faster. In the near future we plan on developing Colony further to also include interface descriptions providing strong interface binding for method calls.

The Drone combined with Argot allows full objects to be serialized and passed as parameters or returned in results. The Drone also allows a client to request multiple results from a single request/reply interaction. In Internet based applications where the Internet has a slow response time, this can greatly increase the responsiveness of an application.

As the drone extends and more instructions are defined, we envisage that the drone itself may soon become a full dynamic virtual machine like a Java Virtual Machine. The ability to record to a file a set of byte code defined using Argot, allows a virtual machine to discover during loading if it implements the correct byte codes to execute the code. This allows a developer to mix and add new byte codes to a virtual machine dynamically. This may have benefits in scientific or graphics systems where specialized instructions can greatly improve the responsiveness of a system.

Binary XML

XML and XML Schema are already the data format of choice of many industries. It is used because it assists greatly in the development process of applications across the enterprise. However, it is also recognized that XML consumes enormous processing and networking resources.

The W3C has recognized the advantages and disadvantages of XML and XML schemas. They are currently in the process of investigating a solution for binary XML. The XML Binary characterization (XBC) working group is in the early stages of its investigations. At this stage, it is developing use cases and a list of desirable properties for an XML to binary conversion. Their work creates a useful set of property descriptions and criteria to examine the properties of Argot (refer to the XBC properties document for a full description of these properties).

We are developing an XML Schema to Argot dictionary converter and corresponding XML to Argot format converter. These enabling tools bring the benefits of Argot to an already established community by preserving the developmental advantages of XML and XML schemas, while reaping the runtime binary advantages of Argot with very significant reductions in processing and networking resources and simplifying the versioning problems of XML schemas.

Data Storage mechanisms

Allowing a file to completely describe its own format has some great prospects for long-term data storage. Public records, which must last for hundreds or more years can be examined and read without any of the original software.

Argot also offers opportunity in the area of database communications. A database is itself a dictionary of concepts. By layering the dictionary concepts of Argot as the

communications layer for a database, the database is able to negotiate the formats required by the client. A database is also able to dynamically configure tables based on the dictionary information of data. This has the potential to reduce greatly the complexity of object relational mappings and align database communications with other forms of communication.

An application using Argot and the properties of synchronous communications described earlier can simply send an object directly to a database with an instruction to update or insert the object. By applying the dictionary approach to meta descriptions of a database we have also reused the communications format used in other methods of communications. This reduces the amount of code and overall complexity of the software.

Programming Languages

This concept can be further extended to programming languages. Using the Argot dictionary, it is possible to capture the concepts used in languages such as Java, and represent them in a canonical binary format. In effect this simply takes a language like Java's parser definition, and redefines it as part of an Argot dictionary. Now in a canonical binary format, there is opportunity for the language to be extended with new features. The traditional static notion of a language is removed with Argot. A programming language can now be defined dynamically to best suite a specific need.

These concepts of treating programming languages as data is not new. Lisp and languages extending from this have been around a long time. However, using environments like the Eclipse Integrated Development Environment (IDE), Argot would allow the way we interact with languages to be separated from how we represent them. This is a very different approach to environments like Lisp, which is still bound by text representations and a common parser.

There is also an opportunity for the higher level languages to be mixed with byte code and other Argot dictionary definitions. As Argot captures its data description within its file and is dictionary based, it allows mixing of any concepts together. Developers are able to mix both high level language concepts with low level assembly concepts in a single file.

Extending further up the development process, the concepts of Model Driven Architectures or Intentional programming can be applied to Argot. A project is able to be defined using high level concepts appropriate to the application. These can be captured alongside java language concepts and byte code concepts in a single file.

These concepts of mixing language concepts and data so fluidly can only be achieved using a data format such as Argot where the file and application are able to check if they match on individual concepts, not whole external common schemas. The question of how to design compilers and virtual machines to handle this flexibility has not yet been researched.

Browser design

The ability to merge data and code in a single common structure naturally suggests that the concepts we encode into our browsers can also be encoded in a single common way. Currently the concepts presented in HTML, CSS and JavaScript are each encoded and designed separately. This means that their interaction is restricted by their ability to

describe interactions with each other. Re-encoding these concepts using Argot would remove these barriers which are to a large extent an artefact of our inability to create file formats which allow the descriptions of these interactions.

The ability to capture and integrate these different concepts has many implications for how we create, edit, publish and browse Internet documents. Allowing JavaScript to fully interact with components of HTML, and CSS will lead to a richer browser experience for users.

While not a short-term goal, these methods offer real hope for convergence in data and languages. The challenge is to devise editors which allow a developer to interact with the definitions in natural and intuitive way.

Global Dictionaries

The dictionary approach to capturing concepts allows new concepts to grow and be based on previous methods of describing information. The base meta dictionary is itself only described using descriptions from the dictionary. This allows a dictionary to grow as more entries are defined. The result of this growth is that over time common global dictionaries will develop that provide common descriptions for information. Much like a language dictionary, this will change and grow as we learn the best ways to describe various domains.

It is expected that industry dictionaries will also develop, which much like XML Schemas will capture specific domains. In contrast to XML Schema, an industry participant will be able to extend and modify these dictionaries to suite their specific needs. Developers will be able to choose and integrate definitions from both global dictionaries and industry dictionaries to create a solution.

Argot Technical

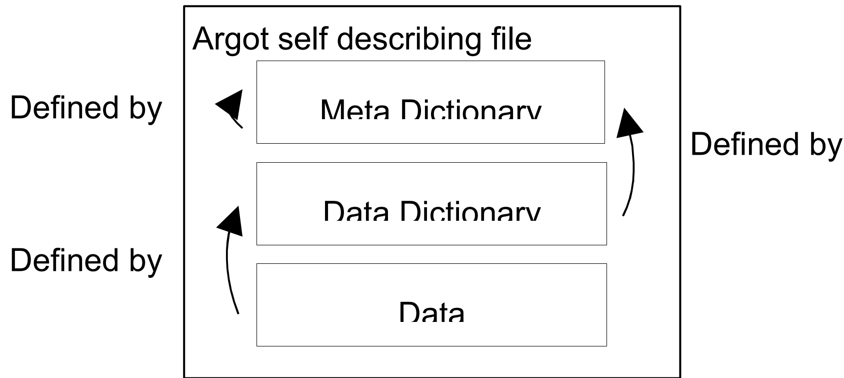
We have already presented many of the features of Argot without describing the technical aspects of how these concepts are achieved. The following section outlines the important aspects of the Argot file format.

The most important aspect of Argot to understand is its self-describing nature. This is the main feature, which differentiates it from ASN.1 and XML. XML and ASN.1 have been described as being self-describing, however we do not believe this is a fair or correct description. We believe that Argot is as close to a self-describing format possible in computer science.

An Argot self-describing file is made up of three parts:

- Meta Dictionary – A set of descriptions which describe the structure of the Data Dictionary. The Meta Dictionary also describes the structure of data used in the Meta dictionary.
- Data Dictionary – A set of descriptions, which describe the various formats, used in the data.
- Data – The file's actual data.

In this way, the File is able to contain a full description of every single element.



One of the design aims of Argot was that the data dictionary descriptions would align as closely as possible with the format of the data. This is in contrast to ASN.1 which is specifically designed to be an "*abstract* syntax notation", meaning that it describes an abstract set of data qualities which are then mapped to an encoding.

To describe this more clearly, a simple example of an Argot description in text might be:

```
first Name: david
last Name: ryan
date Of Birth: 1973/02/01
```

which we want to encode to the following format in hex which will be placed as the data of the file:

```
0x05 0x64 0x61 0x76 0x69 0x64
0x04 0x72 0x79 0x64 0x6E
0x07 0xB5 0x02 0x01
```

This can be described using a textual representation of an Argot data dictionary with the following:

```
person: meta.sequence( [
        meta.reference( #u8ascii, "firstName" ),
        meta.reference( #u8ascii, "lastName" ),
        meta.reference( #basicdate, "dateOfBirth" )
  ] );

basicdate: meta.sequence([
        meta.reference( #s16, "year" ),
        meta.reference( #u8, "month" ),
        meta.reference( #u8, "day" )
]);

u8ascii: meta.encoding(
                meta.array(
                        meta.reference( #u8, "size" ),
                        meta.reference( #u8, "data" )
                   ),
                "ISO646-US" );
```

note: The # symbol denotes the identifier of the type specified.

The above definitions would be contained in the data dictionary portion of the file. It is the meta dictionary part of the file which really creates the self describing nature. Both the meta.reference and meta.sequence elements are described as part of the Meta Dictionary.

```
meta.reference:  meta.sequence( [
                meta.reference( #u16, “type” ),
                meta.reference( #meta.name, “name” )
  ] );

meta.sequence:
        meta.array(
                meta.reference( #u8, "size" ),
                meta.reference( #meta.expression, "type" )
        );
```

A sequence is defined as an array of meta.expressions. The array's size is defined as being a u8 (unsigned eight bit big endian) which is also described. The second part of the array is the type of elements in the array. In a meta.sequence the abstract type meta.expression is used. This allows types such as meta.reference to be mapped to its type and used in its place.

Elements such as u8 and meta.name are also defined in the meta dictionary. In fact every element of the meta dictionary is described using elements from the meta dictionary. It is this fact which allows Argot to create truly self describing file formats. A capability that is very unique to Argot. The complete meta dictionary contains twenty one definitions which contain no external reference.

As mentioned earlier an important difference between Argot and ASN.1 is that Argot always resolves to a single concrete representation. This is because information is described using concrete concepts such as unsigned 8bit big endian number (u8). -

This ability to create a truly self describing file means that a file can always be examined to discover how it was defined. If the definition of the file is modified in anyway, from meta dictionary through to data dictionary, the software reading the file will be able to pick up the change.

Data Type References

To allow such a flexible format, a unique identifier is used for each type in a dictionary. It would be impossible to have each identifier predetermined and requires that identifiers be dynamically assigned. When a file is being read an application must match up its own identifiers with the files through a Type Map.

Each item in the dictionary contains a name, an identifier and description. For an application to match a type both the name and the description must match. Once a match has been established a mapping between the applications internal identifiers to the files external identifier is matched.

File Format

The following Argot descriptions define the overall structure of an Argot file.

```
dictionary.definition:
	meta.envelop(
		meta.reference( #u16, “size” ),
		meta.reference( #meta.definition, “definition” )
	);
```

A dictionary definition is placed in an envelope. An envelope allows a buffer to be read without parsing the information. When the dictionary is initially being read, the identifiers required to read the meta definition is not known.

```
dictionary.entry:
	meta.sequence([
		meta.reference( #u16, “id” ),
		meta.reference( #meta.name, “name” ),
		meta.reference( #dictionary.definition, “definition” )
	]);
```

A dictionary entry provides the type's identifier, name and definition. The identifier is only relevant in the file that the data is defined. The identifier is matched to the applications internal identifier.

```
dictionary.map:
	meta.envelop(
		meta.reference( #u16, “size” ),
		meta.array(
			meta.reference( #u16, “size” ),
			meta.reference( #dictionary.entry , “word” )
		)
	);
```

A dictionary map is an array of data type descriptions placed inside an envelope.

The final argot file format consists of the meta dictionary, data dictionary and data as originally described. The dictionary portions consist of arrays of dictionary maps. This allows a dictionary to define a method of describing data and using that description in following dictionary maps.

```
argot.file:
	meta.sequence([
		meta.array(
			meta.reference( #u8, “size” ),
			meta.reference( #dictionary.map, “meta.dictionary” )
		),
		meta.array(
			meta.reference( #u8, “size” ),
			meta.reference( #dictionary.map, “data.dictionary” )
		),
		meta.reference( #meta.identified, “data” )
```

]);

The process of reading a file involves:

1.Binary compare of core dictionary map. The very first dictionary map of the meta dictionary is the core dictionary. The only way to read this entry is by performing a binary compare. These are the base dictionary items used to describe new items.
2.Build and read Meta dictionary. The rest of the meta dictionary is read and mapped between the application and file.
3.Read the Data dictionary. Using the Type Map produced from entries in the Meta Dictionary the Data dictionary is read. A Data dictionary type map is created based on the types identified.
4.Read the Data. Using the Data dictionary type map the actual data of the file is read.

A similar technique is used when using synchronous communications. However, during synchronous communications the virtual Type Map can be created over time. Each data type can be checked dynamically as required during a communications session.

Reference

The following section provides the full details of the Argot Meta dictionary and how it is encoded.

The Meta Dictionary provides the core concepts from which any new data types must be created. This is much like the schema which defines XML Schemas. In this case every element defined must be defined using a definition contained in the meta dictionary. The following provides the definitions of each type in the meta dictionary.

empty: meta.basic(0, 0);

The empty type is a place holder for a type that has no data associated. This can be useful for identifying specific data which has no additional information associated other than the identifier itself.

u8: meta.basic(8, 0);

This is a basic element which reads an unsigned eight bit big endian number between 0 and 255.

u16: meta.basic(16, 0);

This is a basic element which reads an unsigned sixteen bit big endian number between 0 and 65536.

```
meta.basic:
	{
		@u8["size"],
		@u8["flags"]
	};
```

This defines how the meta data associated with a basic data definition is serialized. Two values of unsigned 8 bit values. The first value is the size if available and the second value is additional flags. Basic data types are atomic values which require specific function to read the values.

```
meta.name:
	{
		meta.encoding(
			meta.array(
				@u8["size"],
				@u8["data"]
			),
			"ISO646-US"
		)
	};
```

A name is a simple ASCII encoded string of maxiumum length 255 characters. This is used to describe the names of each type in the Argot.

```
meta.abstract:
	{
		@empty["abstract"]
	};
```

A type defined as abstract has an empty definition. The abstract data type is an important concept in Argot. It allows a concept to be defined which has no definition associated. A map is used to associate a concrete representation with an abstract data type. This allows the Argot to be expanded by the user without modifying types previously defined.

```
meta.map:
	{
		@u16["abstract"],
		@u16["concrete"]
	};
```

A map is used to map an abstract data type to a concrete data type. Two u16 values are used to specify identifiers of the data types in the dictionary being mapped.

```
meta.expression: meta.abstract();
```

An expression is an abstract type. It allows different expressions to be used to define data types. New expressions can be added to expand Argot's specification language.

```
meta.encoding:
	{
		@meta.expression["data"],
		@meta.name["encoding"]
```

```
        };
```

Encoding specifies the data encoding used on a character string. The data expression must return an array that can have encoding applied.

```
meta.reference:
        {
                @u16["type"],
                @meta.name["name"]
        };
```

A reference declares a usage of another data type in the system. The name data type is used to define a description of usage of that data type. A reference is defined using the @ type "[" name "]" syntax in the definition given.

```
meta.sequence:
        {
                meta.array(
                        @u8["size"],
                        @meta.expression["type"]
                )
        };
```

A sequence defines a set of expressions which are executed in order. In the most normal case, it defines an ordered set of types in a data buffer. A sequence is defined as "{" "}" in the grammar defined to describe argot definitions.

```
meta.array:
        {
                @meta.expression["size"],
                @meta.expression["type"]
        };
```

An Array is used to define any collection of data with a size and a type. The abstract type exprssion is used to define how the size and type is specified.

```
meta.expression#reference: meta.map( #meta.expression, #meta.reference );
meta.expression#sequence: meta.map( #meta.expression, #meta.sequence );
meta.expression#encoding: meta.map( #meta.expression, #meta.encoding );
```

The abstract expression type is mapped to reference, sequence and encoding. Any of the concrete types can be used in place of the expression type.

```
 meta.definition: meta.abstract();
```

The definition of a type is abstract. Specific concrete types are mapped to the abstract type.

```
meta.definition#basic: meta.map( #meta.definition, #meta.basic );
meta.definition#sequence: meta.map( #meta.definition, #meta.sequence );
meta.definition#map: meta.map( #meta.definition, #meta.map );
meta.definition#abstract: meta.map( #meta.definition, #meta.abstract );
```

A definition is defined as being basic, sequence, map or abstract.

Using the above definitions only, it must be possible to serialize each type. This creates a purely self referenced system. An example is given in the following type map to identify each type:

Identifier	Type
1	empty
2	u8
3	u16
4	meta.basic
5	meta.abstract
6	meta.map
7	meta.expression
8	meta.sequence
9	meta.reference
10	meta.name
11	meta.encoding
12	meta.array
13	meta.expression#reference
14	meta.expression#sequence
15	meta.expression#array
16	meta.expression#encoding
17	meta.definition
18	meta.definition#basic
19	meta.definition#map
20	meta.definition#sequence
21	meta.definitino#abstract

The type map used is the same for both identifying the data and for referencing other types in meta.reference type data. We are then able to serialize each of the types as a definition:

empty: meta.basic(0, 0);

```
      (meta.definition#basic)     (size)(flags)
(hex) 00 12
                                  00 00
```

u8: meta.basic(8, 0);

```
      (meta.definition#basic)     (size) (flags)
(hex) 00 12
                                  08 00
```

u16: meta.basic(16, 0);

```
      (meta.definition#basic)     (size)(flags)
(hex) 00 12
                                  10 00
```

meta.basic: { @u8["size"], @u8["flags"] };

```
(meta.definition#sequence)          (type references)
(hex)   00 14 02
                                    00 0D 00 02 04 73 69 7A 65
                                    00 0D 00 02 05 66 6C 61 67 63
```

meta.abstract: { @empty[“abstract”] }

```
(meta.definition#sequence)          (type references)
(hex)   00 14 01
                                    00 0D 00 01 08 61 62 73 74 72 61 63 74
```

meta.map: { @u16["abstract"], @u16["concrete"] };

```
(meta.definition#sequence)          (type references)
(hex)   00 14 02
                                    00 0D 00 03 08 61 62 73 74 72 61 63 74
                                    00 0D 00 03 08 63 6F 6E 63 72 65 74 65
```

meta.expression: meta.abstract();

```
(meta.definition#abstract)
(hex)   00 15
```

meta.sequence: { meta.array(@u8["size"], @meta.expression["type"]) };

```
(meta.definition#sequence)
                        (meta array)
                                    (type reference)
(hex)   00 14 01
                        00 0F
                                    00 0D 00 02 04 73 69 7A 65
                        00 0D 00 07 04 74 79 70 65
```

meta.reference: { @u16["type"], @meta.name["name"] };

```
(meta.definition#sequence)          (type references)
(hex)   00 14 02
                                    00 0D 00 03 04 74 79 70 65
                                    00 0D 00 0A 04 6E 61 6D 65
```

meta.name: { meta.encoding(meta.array(@u8["size"],@u8["data"]),
"ISO646-US") };

```
(meta.definition#sequence)
                        (meta encoding)
                                    (meta array)
(hex)   00 14 01
                        00 10
                                    00 0F
                                         00 0D 00 02 04 73 69 7A 65
                                         00 0D 00 02 04 64 61 74 61
                        09 49 53 4F 36 34 36 2D 55 53
```

meta.encoding: { @meta.expression["data"], @meta.name["encoding"] };

(meta.definition#sequence) (type references)
(hex) 00 14 02
00 0D 00 07 04 64 61 74 61
00 0D 00 0A 08 65 6E 63 6F 64 69 6E 67

meta.array: { @meta.expression["size"], @meta.expression["type"] };

(meta.definition#sequence) (type references)
(hex) 00 14 02
00 0D 00 07 04 73 69 7A 65
00 0D 00 07 04 74 79 70 65

meta.expression#reference: meta.map(#meta.expression, #meta.reference);

(meta.definition#map)
(hex) 00 013 00 07 00 09

meta.expression#sequence: meta.map(#meta.expression, #meta.sequence);

(meta.definition#map)
(hex) 00 013 00 07 00 08

meta.expression#array: meta.map(#meta.expression, #meta.array);

(meta.definition#map)
(hex) 00 013 00 07 00 0C

meta.expression#encoding: meta.map(#meta.expression, #meta.encoding);

(meta.definition#map)
(hex) 00 013 00 07 00 0B

meta.definition: meta.abstract();

(meta.definition#abstract)
(hex) 00 015

meta.definition#basic: meta.map(#meta.definition, #meta.basic);

(meta.definition#map)
(hex) 00 013 00 11 00 04

meta.definition#map: meta.map(#meta.definition, #meta.map);

(meta.definition#map)
(hex) 00 013 00 11 00 06

meta.definition#sequence: meta.map(#meta.definition, #meta.sequence);

(meta.definition#map)
(hex) 00 013 00 11 00 08

meta.definition#abstract: meta.map(#meta.definition, #meta.abstract);

(meta.definition#map)
(hex) 00 013 00 11 00 05

Using these serialized type definitions a client and server are able to confirm the format of metadata before attempting to communicate additional metadata information. Additional elements which fully describe the format of this meta data include:

meta.envelop: { @meta.expression[“size”], @meta.expression[“type”] };
meta.expression#envelop: meta.map(#meta.expression, #meta.envelop);

The meta.envelop type is an extension to the expression type. An envelop is designed to hold the data of another type in a binary buffer. The first argument specifies how the size of the buffer will be specified, while the second identifies the contents of the buffer. The envelop allows the data to be read without knowing how to interpret contained in the envelop.

dictionary.definition: { meta.envelop(@u16[“size”], @meta.definition[“definition”]) };
dictionary.entry: { @u16[“id”], @meta.name[“name”], @dictionary.definition[“definition”] };
dictionary.map: { meta.array(@u16[“size”], @dictionary.entry[“entry”]) };

The dictionary types specify the format of a collection of specifications. The type identifier used in the map, the name of the type, and the full definition are included.

Using these definitions, the self referencing meta definitions specified above would be written to a buffer:

(hex)
00 15, 00 01 05 65 6D 70 74 79 00 04 00 12 00 00, 00 02 02 75 38 00 04 00 12 08 00, 00 03 03 75 31 36 00 04 00 12 10 00, 00 04 0A 6D 65 74 61 2E 62 61 73 69 63 00 16 00 14 02 00 0D 00 02 04 73 69 7A 65 00 0D 00 02 05 66 6C 61 67 73, 00 05 0D 6D 65 74 61 2E 61 62 73 74 72 61 63 74 00 10 00 14 01 00 0D 00 01 08 61 62 73 74 72 61 63 74, 00 06 08 6D 65 74 61 2E 6D 61 70 00 1D 00 14 02 00 0D 00 03 08 61 62 73 74 72 61 63 74 00 0D 00 03 08 63 6F 6E 63 72 65 74 65, 00 07 0F 6D 65 74 61 2E 65 78 70 72 65 73 73 69 6F 6E 00 02 00 15, 00 08 0D 6D 65 74 61 2E 73 65 71 75 65 6E 63 65 00 17 00 14 01 00 0F 00 0D 00 02 04 73 69 7A 65 00 0D 00 07 04 74 79 70 65, 00 09 0E 6D 65 74 61 2E 72 65 66 65 72 65 6E 63 65 00 15 00 14 02 00 0D 00 03 04 74 79 70 65 00 0D 00 0A 04 6E 61 6D 65, 00 0A 09 6D 65 74 61 2E 6E 61 6D 65 00 23 00 14 01 00 10 00 0F 00 0D 00 02 04 73 69 7A 65 00 0D 00 02 04 64 61 74 61 09 49 53 4F 36 34 36 2D 55 53, 00 0B 0D 6D 65 74 61 2E 65 6E 63 6F 64 69 6E 67 00 19 00 14 02 00 0D 00 07 04 64 61 74 61 00 0D 00 0A 08 65 6E 63 6F 64 69 6E 67, 00 0C 0A 6D 65 74 61 2E 61 72 72 61 79 00 15 00 14 02 00 0D 00 07 04 73 69 7A 65 00 0D 00 07 04 74 79 70 65, 00 0D 19 6D 65 74 61 2E 65 78 70 72 65 73 73 69 6F 6E 23 72 65 66 65 72 65 6E 63 65 00 06 00 13 00 07 00 09, 00 0E 18 6D 65 74 61 2E 65 78 70 72 65 73 73 69 6F 6E 23 73 65 71 75 65 6E 63 65 00 06 00 13 00 07 00 08, 00 0F 15 6D 65 74 61 2E 65 78 70 72 65 73 73 69 6F 6E 23 61 72 72 61 79 00 06 00 13 00 07 00 0C, 00 10 18 6D 65 74 61 2E 65 78 70 72 65 73 73 69 6F 6E 23 65 6E 63 6F 64 69 6E 67 00 06 00 13 00 07 00 0B, 00 11 0F 6D 65 74 61 2E 64 65 66 69 6E 69 74 69 6F 6E 00 02 00 015, 00 12 15 6D 65 74 61 2E 64 65 66 69 6E 69 74 69 6F 6E 23 62 61 73 69 63 00 06 00 13 00 11 00 04, 00 13 13 6D 65 74 61 2E 64 65 66 69 6E 69 74 69 6F 6E 23 6D 61 70 00 06 00 13 00 11 00 06, 00 14 18 6D 65 74 61 2E 64 65 66 69 6E 69 74 69 6F 6E 23 73 65 71 75 65 6E 63 65 00 06 00 13 00 11 00 08, 00 15 18 6D 65 74 61 2E 64 65 66 69 6E 69 74 69 6F 6E 23 61 62 73 74 72 61 63 74 00 06 00 13 00 11 00 05
(comas have been used to denote dictionary.entry boundries)

A typical client server communications system would first send this data and the server would confirm that it matches the server side serialized data representation. Until the metadata is compared and matched at a binary level, the server will not be able to confirm that the data can be read successfully.

Conclusion

We have only started to examine the possibilities of using the dictionary-based approach to data representation. The potential is that the combined effect of bringing multiple applications, services and systems into Argot will allow data to move faster and with more flexibility through systems. The overall aim is to remove the cost associated with software communications. Allowing an application to leverage software used to communicate with a server to also communicate with file formats, databases and messaging systems can only lead to software that is more flexible.

The Argot library creates a common marshaling library which is able to reuse one description of information across multiple communication methods. This is in great contrast to the software we build today that requires programmer effort for each and every method we use to communicate.

The most exciting prospect for Argot is not how it can be used in individual types of software or methods of communications. The most exciting prospect is how Argot can be used across all of these domains simultaneously.

The uses for Argot as described here offers far reaching implications for many parts of computer science. By changing the way we are able to store and combine information, we in effect create new ways of how that information can be combined during execution.

To realise the full implications of Argot require that we can use it for problems that face us now. In the short term, we see many benefits for Argot in industries where XML has not been appropriate such as mobile and scientific applications. Argot can be used to compliment XML, mixing compact binary data along side the easy to read XML information. Many of the use cases for Argot have already been identified by the XBC working group, including grid computing, gaming, mobile services and financial systems to name a few.

Argot offers many opportunities for how we build synchronous and asynchronous communications systems, databases, programming languages, web documents and browsers. As Argot is developed, we expect to see these concepts become reality. While Argot is currently a proprietary data format we have the opportunity to ensure that the science behind its design is correct and open to future expansion. In the future, we envisage that Argot can become a free and open standard that can integrate into a wide variety of applications and systems.