Enabling Data Scientist Use Cases with Discoverability and Metadata

Greg Majcher
Principal Application Engineer
Rockwell Automation

Presented at the ODVA
2023 Industry Conference & 22nd Annual Meeting
October 18, 2023
El Vendrell, Spain

Abstract

Data science uses statistics and algorithms to extract or extrapolate knowledge and insights from noisy, structured, and unstructured data. Data scientists often don't know what they are looking for until they see patterns or associations. For this reason, all data can be valuable in analyzing and optimizing a process or system.

Many CIP-enabled devices possess a rich collection of data that never gets used in a user's control program. Furthermore, some of that data is buried in the device and not readily exposed. The CIP specifications provide some mechanisms to make that data discoverable, but more could be defined. This paper explores options using currently specified techniques as well as some new proposals for making device data more discoverable and understandable thereby enabling its use in data scientist use cases.

Keywords

Data, Data Scientist, Discoverability, Metadata, Vertical Communication, Data Acquisition

Definition of terms

CIP:	Common Industrial Protocol defined in [1]
Data Scientist:	IT professional that uses data to understand and solve problems
ListIdentity:	Command used by clients to locate and identify potential CIP communication targets
LLDP:	Link Layer Discovery Protocol defined in [4]
Metadata:	Data that provides information about other data

Introduction

Data scientists use data to answer questions, make predictions, and solve problems. They collect, clean, organize, and analyze data before drawing their conclusions. But what data are they looking for? It depends.

First the data scientist must understand the problem space. Some industrial possibilities are:

- Identify areas for energy savings.
- Predict when a component will fail.
- Increase the efficiency of a process.
- Identify deteriorating quality in a process or a product being produced.
- Diagnose performance or quality differences between similar production lines or facilities.

In some of these cases it would be easy to identify the data needed. In others, the selection of data is less clear. Let's look at a few use cases from above.

User Story: As a plant manager, I want to identify opportunities for energy savings.

The data scientist could start by collecting the power consumption of all products in a plant and ranking them by their power usage. Studying when and how the highest consumers are used could yield energy saving ideas. In this case identifying the needed data is straightforward, but it does rely on knowing where that data is located and how to retrieve it.

User Story: As a plant manager, I want to predict when equipment will fail so that I can proactively schedule maintenance downtime to replace it.

Data scientists can collect data directly related to a component's usage such as hours of operation or a cycle count related to movement. If devices provide predicted lifetimes, they can be used to compare against the actual usage. However, predicted lifetime data might be based on idealized environmental conditions. The actual environment may be accelerating wear. Data that reflects the surrounding temperature or humidity might be of interest. The turbidity or viscosity of any fluids involved may be important. Discovering these data items will require some investigation and knowledge of the application. Additionally, this use case may also require taking baseline measurements (e.g., vibration monitoring measurements or power consumption) and then tracking changes over time.

User Story: As a regional production manager, I want to understand performance and quality differences between two similar plants.

Depending on how well the production facilities and processes are understood, it may or may not be clear what data to use in this case. The data scientist may need to collect a large sample of many different sources of data and simply look for differences. The differences between the data from the facilities may point to areas for further investigation. The plants being compared in this example could be using equipment from different vendors or even different network technologies. To compare the data correctly, the data scientist would need to know that the data collected from the differing equipment was equivalent.

User Story: As a plant manager, I want to detect deteriorating production quality and identify the cause early so that remediation can occur with minimal product loss.

To diagnose decreasing quality, data scientists may need to collect a lot of random, unrelated data over a period of time and look for associations and trends in the historical data. Because the associations are unknown, the data scientist may be looking for any kind of data available. The ability to browse devices in order to see all the data they have available would be very helpful in this case.

The examples above are intended to highlight problem scope ranging from well-defined to highly unstructured. In each case, the data needed, and the steps taken to collect it might be different. Industrial automation products have data that is intended to be exchanged in real time connections with a PLC or DCS. Many products also have additional data available that never gets used in a control program. Things like diagnostics, statistical counters, event logs, and status variables are collected by field devices and are waiting to be used. It is likely that some of this data will be of high value to a data scientist. To better enable data scientists, data needs to be discovered, understood, and delivered in an efficient manner.

Products should efficiently expose the data they possess. Exposed data should be presented along with meaningful metadata, so the meaning of the data is understood. Finally, data should be made available using transport mechanisms that are efficient for the type, amount, or frequency of data. The CIP specifications already define some techniques to present, define, and transport data. The rest of this paper will document some of those along with proposals for how we can do better.

Discovery

Discovering Devices

Devices can easily be found on an ethernet network. The EtherNet/IP Adaptation of CIP (Volume 2 of the CIP Networks Library) recently mandated support for the Link Layer Discovery Protocol (LLDP) in all products. Devices supporting LLDP advertise information to stations attached to the same LAN for the purpose of populating a physical topology. Identifying information (i.e., a device's CIP identity) is returned in responses.

Volume 2 also defines the ListIdentity command sent using an Encapsulation packet over TCP or UDP. Responses to this message include data from all the required attributes of the Identity object as well as the current state of the device. Clients can send this message as a broadcast and quickly discover all EtherNet/IP devices on a network.

The combination of these two mechanisms provides for an effective way to construct a topology tree and identify all the EtherNet/IP devices. The discovery process could be improved if clients were able to discover important capabilities or features that were supported by devices. Volume 8 contains one example of this. Devices that support CIP Security are required to include their supported CIP Security Profiles in response to the ListIdentity request. This same mechanism could be used to indicate support for other important capabilities.

Discovering Data Online

To request data from a CIP-based product, you must construct a path to that data. Paths are addressed to objects. They can be directed to the object class, or to a specific instance of the object. Attributes of the class or individual instances represent the data.

If nothing was known about a product, you may be tempted to use a brute force technique for discovery, sending many requests to discover what a product had to offer. The first step would be to send a request to every possible class code (65,535 possible) to see which classes were supported. For any objects that responded, you would then need to discover the instances (4,294,967,295 possible) of that object that existed. Finally, for each instance you could send a request to every possible attribute id (65,535 possible). As you can see, this technique would result in trillions of requests and take too long to be useful.

The good news is that several techniques already exist to limit the number of messages needed to perform this type of discovery. The bad news is that most products do not support this functionality because it has always been optional.

The Message Router Object has an instance attribute (Attribute 1) that enumerates all the supported objects in an implementation. If this attribute is supported, a client can avoid the brute force method of discovering the supported objects.

Table 1 - Message Router's Object List Attribute

Number	Need in implementation	Access Rule	Name	Data Type	Description of Attribute	Semantics of Values
1	Optional	Get	Object_list	STRUCT of	A list of supported objects	Structure with an array of object class codes supported by the device
			Number	UINT	Number of supported classes in the classes array	The number of class codes in the classes array
			Classes	ARRAY of UINT	List of supported class codes	The class codes supported by the device

This attribute provides a very efficient response indicating all the objects contained in a device. The next step would be to discover the instances of each object class.

Volume 1, Chapter 4 of the CIP Networks Library documents the CIP Object Model. As part of that model some Class Attributes are defined as common to all CIP objects. Attributes 2 and 3 help you to discover which instances are currently created in an object. Attributes 200 and 201 are used if a device supports instances greater than 65,535.

Table 2 - Reserved Class Attributes 2, 3, 200, and 201

Number	Need in implementation	Access Rule	Name	Data Type	Description of Attribute	Semantics of Values
2	Conditional[2]	Get	Max Instance	UINT	Maximum instance number of an object currently created in this class level of the device.	The largest instance number of a created object at this class hierarchy level.
3	Conditional[2]	Get	Number of Instances	UINT	Number of object instances currently created at this class level of the device.	The number of object instances at this class hierarchy level.
200	Conditional[2]	Get	Max Instance	UDINT	Maximum instance number of an object currently created in this class level of the device.	The largest instance number of a created object at this class hierarchy level.
201	Conditional[2]	Get	Number of Instances	UDINT	Number of object instances currently created at this class level of the device.	The number of object instances at this class hierarchy level.
Table Footnotes: 2. These attributes are optional. If the device chooses to implement either Max Instance or Number of Instances attribute, it shall implement the UDINT version if it supports instances greater than 65.535, else it shall implement the UINT version.						

These attributes would be all you needed if the value of Max Instance equaled the Number of Instances. However, if the two do not match, you must try all the instances between 1 and Max Instance to discover the instances that the device currently has instantiated. There are no rules about which instances can be created. In a worst-case scenario, a device may have created instances from the top of the range and worked down or has some dynamic instantiation method that results in a sparsely populated list.

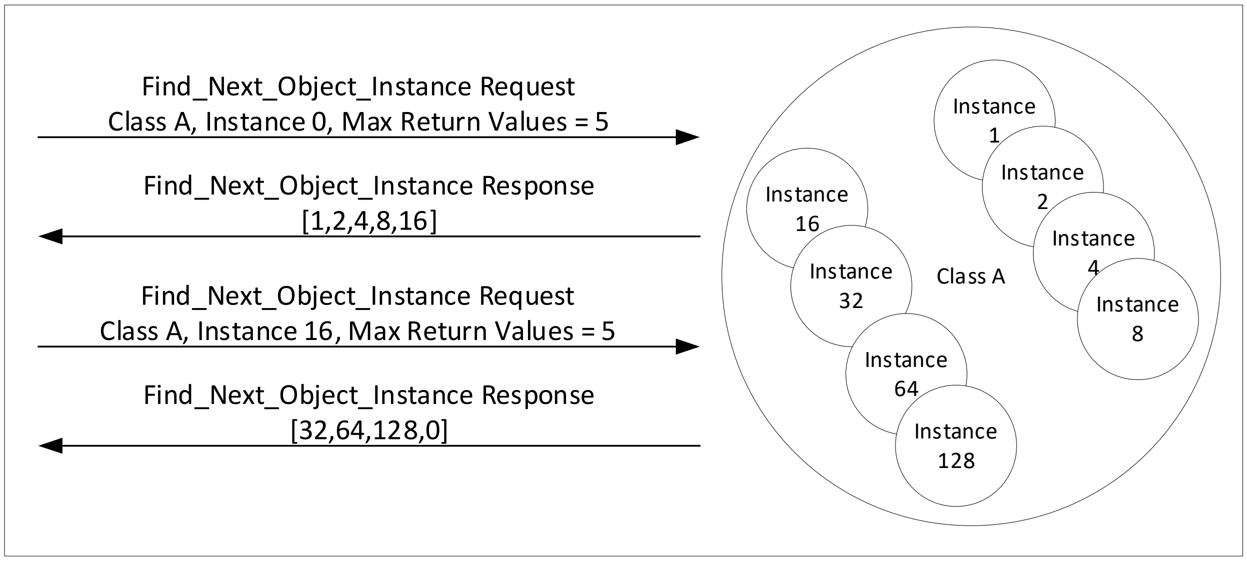
A better solution might be to provide an attribute similar to the Message Router's Object_List that would return an array of the instances that currently exist.

Table 3 - Proposed Instance List Attribute

Number	Need in implementation	Access Rule	Name	Data Type	Description of Attribute	Semantics of Values
N	Optional	Get	Instance_list	STRUCT of	A list of created instances	Structure with an array of currently created instances of this object class
			Number	UINT	Number of instances in the Instances array	The number of instances in the Instances array
			Instance Data Type	UINT	The data type of members in the instances array	0 = USINT 1 = UINT 2 = UDINT
			Instances	ARRAY of Instance Data Type	List of currently created instances	The instances that currently exist in this object class

This attribute would be efficient for most devices. For implementations where the presence of many instances would not allow the response to fit in one packet, CIP provides the Find_Next_Object_Instance service. This service is directed to a class specifying an instance to start with and a maximum number of return values. The class will return a list of existing instances starting with the next instance greater than what was passed in. Returning 0 indicates that the end of the list of created instances has been found. The example in Figure 1 shows a collection of sparsely created instances returned in only two exchanges.

Figure 1 - Example of Find_Next_Object_Instance Service


Find_Next_Object_Instance Request
Class A, Instance 0, Max Return Values = 5
Find_Next_Object_Instance Response
[1,2,4,8,16]
Find_Next_Object_Instance Request
Class A, Instance 16, Max Return Values = 5
Find_Next_Object_Instance Response
[32,64,128,0]
Class A
Instance 1
Instance 2
Instance 4
Instance 8
Instance 16
Instance 32
Instance 64
Instance 128

This service works very well if the implementation does not support instances greater than 65,535. The Find_Next_Object_Instance service was defined before CIP was extended beyond UINT instances. To find UDINT instances, the client would need to use the existing attributes along with the brute force method. Alternatively, the service could be redefined (or a new service could be developed) that supported UDINT instance numbers.

Once the objects and their instances are discovered, finding the supported attributes can be performed. The CIP Object Model currently has the following Class attributes defined to help with this process.

Table 4 - Reserved Class Attributes 4, 6, and 7

Number	Need in implementation	Access Rule	Name	Data Type	Description of Attribute	Semantics of Values
4	Optional	Get	Optional attribute list	STRUCT of	List of optional instance attributes utilized in an object class implementation.	A list of attribute numbers specifying the optional attributes implemented in the device for this class.
			number of attributes	UINT	Number of attributes in the optional attribute list.	The number of attribute numbers in the list.
			optional attributes	ARRAY of UINT	List of optional attribute numbers.	The optional attribute numbers.
6	Optional	Get	Maximum ID Number Class Attributes	UINT	The attribute ID number of the last class attribute of the class definition implemented in the device.	
7	Optional	Get	Maximum ID Number Instance Attributes	UINT	The attribute ID number of the last instance attribute of the class definition implemented in the device.	

If the client has knowledge of the Class definition, attribute 4 can be helpful. If the Class definition is vendor specific, or the client is not CIP-aware, attribute 4 is of no value. Attributes 6 and 7 are of some use, however just like with the Max Instance attribute, the value is diminished when large numbers of attributes exist. Therefore, new class attributes that simply return a list of all supported attributes make the most sense.

Table 5 - Proposed Attribute List Attributes

Number	Need in implementation	Access Rule	Name	Data Type	Description of Attribute	Semantics of Values
N	Optional	Get	Class Attribute list	STRUCT of	A list of supported Class attributes	Structure with an array of supported Class attributes
			Number	UINT	Number of attributes in the Attributes array	The number of attributes in the Attributes array
			Attribute Data Type	UINT	The data type of members in the Attributes array	0 = USINT 1 = UINT 2 = UDINT
			Attributes	ARRAY of Attribute Data Type	List of supported attributes	The Class attributes that are supported in this object class
N	Optional	Get	Instance Attribute list	STRUCT of	A list of supported Instance attributes	Structure with an array of supported Instance attributes
			Number	UINT	Number of attributes in the Attributes array	The number of attributes in the Attributes array
			Attribute Data Type	UINT	The data type of members in the Attributes array	0 = USINT 1 = UINT 2 = UDINT

			Attributes	ARRAY of Attribute Data Type	List of supported attributes	The Instance attributes that are supported in this object class

For implementations where many attributes are supported and the response would not fit in one packet, the existing attributes (4, 6, and 7) could be used along with the brute force method, or a new service could be developed. We could specify a Find_Next_Object_Attribute service patterned after the Find_Next_Object_Instance service.

Discovering Data Offline

There are always arguments against introducing more required functionality which would negatively impact constrained devices, or because some devices are already very complicated. To respond to these objections, much of the discovery information could be exposed using offline device description files (e.g., EDS).

The Public Object Class and Vendor Specific Object Class sections of the EDS define a way to expose a product's supported objects, instances, and attributes. An example is shown below for the Discrete Input Point object.

```
[Discrete Input Class]
        Revision = 2;                       $ Revision 2 of the object is implemented
        MaxInst = 8;                        $ The highest instance number that exists in the product is 8
        Number_Of_Static_Instances = 8;     $ There are 8 static instances present
        Number_Of_Dynamic_Instances = 0;    $ There are no dynamic instances
        Class_Attributes = 1;               $ Class attribute 1 is supported
        Instance_Attributes = 3, 4, 5, 6;   $ Instance attributes 3, 4, 5, and 6 are supported
        Class_Services = 0x14;              $ Get_Attribute_Single is supported for class attributes
        Instance_Services = 0x14, 0x10;     $ Get_ and Set_Attribute_Single are supported for instance attributes
        Object_Name = "Discrete Input Point Object"
        Object_Class_Code = 0x08;
```

As you can see, it would be possible to fully describe the objects in a device's implementation using these EDS keywords. Provisions were also made to describe vendor specific objects in the same way. The only thing that would need to be discovered online would be any dynamically created instances of an object. This is an example of a powerful EDS feature that can be used to enable the discoverability and understandability of device data.

Understanding Data Online

CIP's information model (metadata) is documented in its object definitions. For publicly defined objects an ODVA member can know the details of any object from the CIP specifications. However, vendors are free to extend publicly defined objects or even create their own. Those vendor specific additions would not generally be known to other vendors. And non-member actors, specifically end users, have no access to the CIP specifications. This can make it challenging for a data scientist to have the context of the available data.

Currently there is no online mechanism to communicate a CIP object's metadata. The Parameter Object comes close, but it was written for configuration parameters. The object allows for "Stub" or "Full" definition of parameters. Stub parameters fall short of what is needed for the data scientist, and full parameters might be too heavyweight for many products. A simple and efficient mechanism could be introduced to access object metadata by extending our use of logical segments for paths.

CIP uses encoded items, called segments, to reference or describe elements within a device's information model. Those segments can be used to specify a path indicating relationships among different objects. For our purposes we will be talking about paths of logical segments. These are commonly used to reference an object, its instances, or an instance's attributes. A new logical segment could be defined to provide object metadata.

The CIP object model presents class and instance attribute data using tables as shown below. Each of the eight columns specifies some property of the attribute. You could say an attribute of the attribute, but that might get clumsy. These eight columns are essentially the metadata properties for the attributes.

Table 6 - Attribute Properties

Attribute ID	Need in Implementation	Access Rule	NV	Name	Data Type	Description of Attribute	Semantics of Values
1	2	3	4	5	6	7	8

Columns 2-7 could be standardized metadata properties for all attributes. In other words, the name of any attribute could be addressed by referring to property 5 of that attribute or the data type as property 6.

A new Extended Logical segment (0x3C) could be defined to represent these new metadata properties. See the CIP Networks Library Volume 1, Appendix C, Section C-1.4.2 for a complete definition of Logical segment types.

An example is shown below.

Table 7 - Example Using Logical Segments for Attribute Properties

Segment Contents	Notes
[20][01][24][01][30][03][3C 07][05]	Segment Type = Logical Segment. 20 01 indicates class 1 (Identity Object) 24 01 indicates instance 1 30 03 indicates attribute 3 (Product Code) 3C 07 05 indicates metadata property 5 (Name)

If a Get_Attribute_Single request was sent with this path, the response should be a string equal to "Product Code" which is the value of the Name property of the Identity Object's third attribute.
More work would need to be done on this idea. The items in columns 2-5, and column 7 are straightforward and could be exposed with minimal specification work. Column 6, Data Type, would require some investigation for things like structures and arrays. There are constructs defined in Volume 1, Appendix C that may be used. Standardizing the information in column 8, Semantics of Values, would require significant changes from how it is currently used but is possible using some form of constraint language. If we only accomplished Columns 2-5 and 7, that would provide meaningful metadata to someone without access to the object definition. Exposing column 6 is particularly important and within our reach. Column 8 would be powerful and complete the model for online metadata access.

Understanding Data Offline

Param, Assem, and Variant entries in an EDS file can be used to describe any data. Param entries provide a very comprehensive set of fields to fully describe data including fields like Parameter Name, Data Type, Units String, Help String, etc. The example below exposes details about the Connection Manager Object's Percent I/O Utilization attribute.

```
[Params]
Param1 = 0,
6,"20 06 24 01 30 F0",     $ Link Path Size, Link Path to Connection Manager Object
0x0002,                    $ Descriptor
0xC7, 2,                   $ UINT Data Type, Data Size
"Percent I/O Utilization", $ Name
"%",                       $ Units
"Indicates what percentage of the I/O communications resources are in use in this
device in units of 0.1%",  $ Help string
0,1000,0,                  $ Min/Max/Default
,,,,                       $ Unused fields
,,,,
;
```

More complex data such as structures can be represented in Assem entries. Vendors could describe any, or every supported attribute in their products' EDS files. With this information any client could present a very complete picture of the data to a human actor.

Delivering Data

Finally, getting all this data takes time, especially when it needs to travel from the edge to the cloud. Providing efficient means to retrieve this information will make its collection more practical and have less impact on the high priority traffic in the system. Highly granular, grouped, and bulk transfer mechanisms should be available to cover any amount of data and the variety of data science use cases.

Request, response messaging whether unconnected or over a Class 3 connection provides a highly granular approach to getting the data. These exchanges are best suited for small amounts of specific data but could be expensive in terms of network bandwidth when many requests are made.

For multiple small requests, the Message Router Object provides the Multiple_Service_Packet service. Using this service allows you to specify an array of CIP service requests in one packet. This is a more efficient mechanism than sending a single request and then blocking while waiting for a response. However, this service is still subject to packet size limitations. For much larger requests, the Message Router Object provides the Send_Receive_Fragment service. This service is used when the request, response, or both exceed the size of a single packet.

We have good options for granular and grouped data collection, but new transports should be defined that would facilitate the exchange of large amounts of data. Volume 1 reserved Transport Class 4 as Non-blocking, Class 5 as Non-blocking, fragmenting, and Class 6 as Multicast, fragmenting all without definition. Now would be an appropriate time to revisit these and define an effective bulk transport. This would not only benefit the Data Scientist but may also support faster firmware update or device configuration times.

Conclusion

CIP provides a rich collection of optional functionalities. This paper serves several intended purposes: first to shine some light on these lesser-known functionalities, second to spark a conversation and participation in improving what we already have, and third to build support in the ODVA vendor community for incorporation of these features into products.

Our end users will benefit from:

- Online discovery aids like the Message Router's Object_List attribute and every object's class attributes (Max Instance, Number of Instances, Optional attribute list, etc.).
- Online metadata (once defined)
- Offline descriptions of all valuable data within a product (i.e., rich EDS files with Object Class sections and Param entries to define attributes)
- Support for alternative transport mechanisms like the Multiple_Service_Packet and Send_Recieve_Fragment services.

Data science is still a young field, but it is being incorporated into our systems more each day. As our end users discover the data that we have been providing, they will come to value our products even more. We want to ensure that we maximize that value by enabling data scientists as much as possible and not frustrating them with limited capabilities.

References

[1] The CIP Networks Library Volume 1, Common Industrial Protocol (CIP™)
[2] The CIP Networks Library Volume 2, EtherNet/IP Adaptation of CIP
[3] The CIP Networks Library Volume 8, CIP Security™
[4] IEEE Std 802.1AB-2016 – IEEE Standard for Local and metropolitan area networks – Station and Media Access Control Connectivity Discovery