DOI 10.5162/ETTC2024/B2.1

Automating Data Conversion

Fatih Hacıömeroğlu, Raşit Uzun, Mehmet Kekeç, Ali Tahsin Kaymak
Turkish Aerospace, Ankara
fhaciomeroglu@tai.com.tr, rasit.uzun@tai.com.tr, mehmet.kekec@tai.com.tr, alitahsin.kaymak@tai.com.tr

Abstract

Data Acquisition Units provide raw data which needs to undergo a data conversion process, named as either “Engineering-Unit Conversion” for sensor parameters or “Avionic Data Decoding” for avionic parameters. Albeit trivial, this step is crucial, since a simple typo suffices to jeopardize all of the instrumentation efforts. Considering the enormity of the instrumentation, the increased frequency of flights accompanied with changes in sensor and avionic configurations, need for helper tools emerged, to prevent us from relying solely on the attention of the Instrumentation Engineers. In this paper, which is in the same spectrum as the works submitted in the two previous conferences, we present some of the developed utility tools. First one is a tool to generate conversion formulas of all sensor parameters (from Parameter List file and instrumentation definition file (xidml file)) as well as to perform some sanity checks of the Parameter List aiming to minimize human errors. Besides, another tool to compare formulas is presented. This second tool is valuable because it enables to compare formula sets residing in different locations (i.e. hangar computer, lab computer and telemetry room), and to ensure the equivalency of the formulas.

Key words Data Management Applications

Introduction

Starting with the T625 Gökbey Project, the volume of data request increased considerably. It became conventional to use a structured spreadsheet named “Numbered Parameter List” (NPL) for management of the data requests.

Instrumentation architecture of the data acquisition system is Ethernet-based [1, 2].

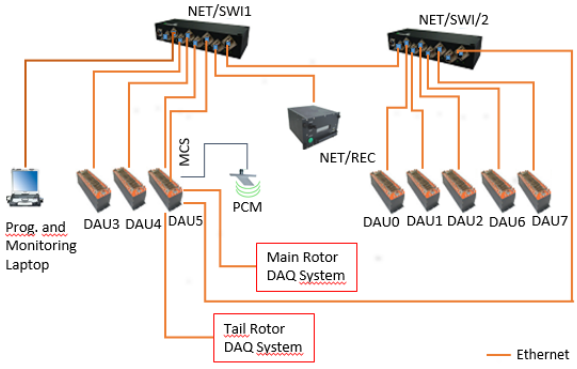

Fig. 1 Instrumentation Architecture in T625

In dealing with numerous parameters, the formula creation becomes an important and critical task, better not performed manually. Relying solely on the attention of the Instrumentation Engineer would not be wise since a simple typo can have severe outcomes. In case of any human error, such as taking wrong calibration coefficient, or wrong sensitivity for a sensor, or using wrong voltage levels than the ones, which the measurement channel’s gain setting imposes, the output turns out to be erroneous. It is unfortunate that such errors cause not only flight test interruptions, but also doubts in data requesters mind about the instrumentation system, which made them think that any abnormal data is a result of such an error. Although in practice, an abnormal data is normally the result of an anomaly or malfunction in the system of interest. Hence, data analysis process suffers because people starts to shortcut their reasoning and question the instrumentation system rather than the measured system. To prevent such unfortunate interpretations, we cannot emphasize enough the importance of a tool for automating sensor related formula generation.

On the other hand, another difficulty lies in the fact that there are formula sets in different locations. The computers in Lab, in hangar and in telemetry room have their own formula sets. We worked on a tool to ease checking their equivalency (both textually and computationally).

Manual Formula Creation

EU formula converts the raw data obtained from a DAU (Data Acquisition Unit) channel to an Engineering Unit data, amenable to data analysis. As an example, consider a formula for a strain gage, named STR153c.

Here, STR is an acronym for strain measurement, “1” means it is from the location coded as 1 (cabin area), “53” is a number uniquely identifying the sensor (we can infer that there are at least 52 gages defined in cabin area other than this one), finally “c” means that it is a rosette leg (other legs are named as STR153a and STR153b).

```
STR153c
=
D2_J6_ADC_135_B_ch3/65535*(0/009765625-0/0048828125)/2.5*4/(1*2.14)*1e6
```

Fig. 2 An example formula

This formula is entered in our Data Conversion and Monitoring tool (IADS [4]) as a field entry of a table. In the tool, the table to hold Conversion formulas is named as “*ParameterDefaults*” table, and the field, which contains the text of the formula is named as “*DataSourceArguement*” field. Descriptions of the terms, based on their order in the formula are as follows:

- **D2_J6_ADC_135_B_ch3** is the raw parameter that DAU is programmed to output. (has to be fetched from NPL and xidml)

- **65535** comes from 16 bit sampling done by DAU

- **0.009765625** and **0.0048828125** come from the channel gain settings (from maximum and minimum voltages) (they have to be fetched from xidml file, based on the channel information in NPL)

- **2.5** is the voltage excitation of the bridge (has to be fetched from xidml file)

- **4** is a constant term in strain formula

- **1** is the Bridge Factor. It is 1 for “STR” parameters (quarter bridges having one active leg)

- **2.14** is the Gage factor (has to be fetched from NPL)

- **1e6** is a coefficient to obtain EU parameter in terms of microstrain

This structure is used for all of the strain (STR) measurements.

There are various reasons why formula generation can become errorenous: First, there are different types of formulas for each sensor type. Other than STR, there are 24 different analog data types, such as ACC for Accelerometers, POT for Potentiometer sensors, PRE for Pressure Sensors, etc. Generally the structure is similar and can be reduced to the form given in Equation 1:

$$EU\ Parameter = \frac{Measured\ Voltage}{Sensitivity} + Offset$$

Eq. 1 Generic EU conversion formula for a sensor

where “*Sensitivity*” is in terms of mV/EU and “*Offset*” is EU value when sensor output is 0 mV. However not all of the sensor types can be written in this manner. For example the analog parameters obtained from the 3rd Party Rotating Data Acquisition systems has system specific formulas. Moreover, within a single data source type, we may have more than one formula structure. For instance the structure of the formula for an RTD sensor depends on whether its excitation is done by instrumentation system or excitation is handled by the system in Platform.

Second, the sheer number of parameters is challenging to handle. One can very likely enter a wrong information in one of the formulas if he/she is supposed to enter hundreds of formulas.

Third, there are cases where the structure can have slight changes. For instance, the triaxial vibration sensor parameters may require orientation correction. Similarly; if a triaxial sensor is installed for *VIB101x*, *VIB101y*, *VIB101z* measurement and x/y/z conventions of the Platform are not as the orientation of the sensor as it is installed, then we will need to convert the orientation. For example, we may have a case where we are measuring z axis of the platform from the x channel of the sensor. Oftentimes such conversions are error-prone.

Finally, the IADS’s ParameterDefaults table also contains many other fields than the “*DataSourceArguement*” field. For instance; the *Group* and *SubGroup* fields are useful in sensor grouping. Moreover *SubGroups* can be used for easy grouping of parameters for data export groups. More than rarely, when entering such fields, there are inconsistencies that cause confusion. It would be much better to have a standardization, which enforces conventions on the *ParameterDefaults* table.

As a result, for all of the above potential error sources, it is of utmost importance to generate formulas automatically in the process, where severe consequences might occur as discussed in the Introduction section.

Master Formula Generator

We have used Python to implement the tool. Main libraries which are used extensively are:

- **pandas** : for NPL reading, for merging NPL and xidml, for dumping results
- **minidom** : for xidml reading
- **re** : for NPL checks, for xidml and NPL sheet binding in merging process

The tool performs the tasks depicted in the following diagram:

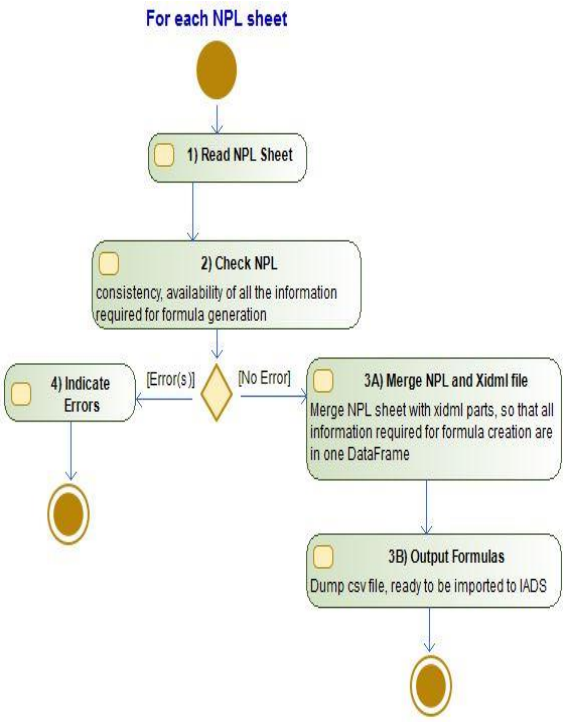

Fig. 3 Activity Diagram of the operations for a single sheet in NPL

STEP 1) Read NPL Sheet

All sheets are read separately, it is logical for the tool to loop each sheet because structure of the formulas depends mainly on the sensor types (hence sheets in NPL's).

STEP 2) Check NPL

We implemented an architecture to easily define checks on NPL so that we are in a better position to generate formulas. Checks are regular expressions (regex's) imposed on the content of specific columns of NPL. For instance, the below code snippet shows the checks imposed on the "POT" sheet of NPL (Potentiometer measurements).

```
# POT sheet checks ##############################################################
CHECKs['ONE_COLUMN']['POT'] = {(1, 'Unit') : 'ONLY_mm',
                               (2, 'ParameterName') : 'POT_PARAMNAME',
                               (3, 'ParameterNo') : 'POT_PARAMNO',
                               (4, 'RealTimeFrequencyHz') : 'IS_WHOLE_NUMBER',
                               (5, 'Sensitivity') : 'IS_NUMBER_WITH_OR_WITHOUT_mVpmm',
                               (6, 'ZeroOffset') : 'IS_NUMBER_WITH_OR_WITHOUT_mm'}
CHECKs['TWO_COLUMN_SUBSTRING']['POT'] = {(1, 'ParameterNo', 'Zone')
                                         :
                                         ('GET_ZONE_FROM_PARAMETERNO', 'GET_ZONE_FROM_ZONE')}
```

Fig. 4 Example of NPL checks defined for POT sheet

Each element imposes a rule to check. For instance:

- The "*Unit*" column must obey the regular expression saved as "*ONLY_mm*" (which is **'^\s*(?-i:mm)\s*$**'). If it is empty or something else, the check will fail.
- "*ZeroOffset*" column must obey regex named as "*IS_NUMBER_WITH_OR_WITHOUT_mm*" (which is **'(^\s*(-?\d+|-?\d*\.\d+)\s*$|^\s*(-?\d+|-?\d*\.\d+)\s*mm\s*$)**'). If zero offset is empty or is filled something else, the check will fail.

The above examples are so-called "one-column" checks. Also "two-column" checks can be defined. For POT sheet, one such "two-column" check is defined as below.

- The "*ParameterNo*" column must obey a regex named "*GET_ZONE_FROM_PARAMETERNO*" (which is **'^\s*\w{3}(\d)(?:\d{2}|\d{1}[a-zA-Z][a-zA-Z]{2})[xyzXYZabcABC]*\s*$**')
- The "*Zone*" column must obey a regex named as "*GET_ZONE_FROM_ZONE*" (which is **'^([0-9])[\.0]*'**). Both must give the same output. For instance if parameter is POT304 (3 being the location (zone) number), the zone column must be 3.

A sheet may have as many checks as desired. For instance, the checks defined for RTD sheet are more than the ones defined for POT sheet. In RTD case, there are "two-column" checks such as checking if "*Excitation Current*" field is entered in case RTD is an externally excited one (i.e. excited by the system, not by DAU). Note that checks for informations that should not be entered are also defined, for instance if an RTD excitation is done by DAU, its column named "*Alpha*" should be empty (as the DAU provide the linearized output and we do not need alpha coefficient of the RTD inside the formula for an RTD excited by DAU).

```
# RTD sheet checks ##############################################################
CHECKs['ONE_COLUMN']['RTD'] = {(1, 'Unit') : 'ONLY_C',
                               (2, 'ParameterName') : 'RTD_PARAMNAME',
                               (3, 'ParameterNo') : 'RTD_PARAMNO',
                               (4, 'RealTimeFrequencyHz') : 'IS_WHOLE_NUMBER',
                               (5, 'ExcitationSource') : 'IS_PROPER_RTD_EXCITATION_SOURCE'}
CHECKs['TWO_COLUMN_SUBSTRING']['RTD'] = {(1, 'ParameterNo', 'Zone')
                                         : ('GET_ZONE_FROM_PARAMETERNO', 'GET_ZONE_FROM_ZONE')}
CHECKs['TWO_COLUMN']['RTD'] = {(1, 'ExcitationSource', 'ExternalExcitation')
                               : ('IS_KAM500_EXCITATION', 'IS_EMPTY'),
                               (2, 'ExcitationSource', 'ExternalExcitation')
                               : ('IS_EXTERNAL_EXCITATION', 'IS_NUMBER_WITH_OR_WITHOUT_mA'),
                               (3, 'ExcitationSource', 'Alpha')
                               : ('IS_KAM500_EXCITATION', 'IS_EMPTY'),
                               (4, 'ExcitationSource', 'Alpha')
                               : ('IS_EXTERNAL_EXCITATION', 'IS_NUMBER_WITH_OR_WITHOUT_OhmpC'),
                               (5, 'ExcitationSource', 'Rref')
                               : ('IS_KAM500_EXCITATION', 'IS_EMPTY'),
                               (6, 'ExcitationSource', 'Rref')
                               : ('IS_EXTERNAL_EXCITATION', 'IS_NUMBER_WITH_OR_WITHOUT_Ohm')}
```

Fig. 5 Checks defined for RTD sheet of NPL

As a by-product of this work, more involved checks of some columns of NPL (other than the ones for formula generation) are implemented. For instance we check if serial numbers of sensors are added, whether they are unique,

whether the parameter names follows our convention etc. Note that having lacked time at the beginning of the Project, we contented to use a spreadsheet named NPL for handling parameter requests. Implementing such auxiliary checks is a useful band-aid to make this spreadsheet based Parameter List handling method more robust.

NPL is read as a pandas DataFrame and the checks are actually created using a class named "*DataFrameChecker*", which first transforms all the DataFrame into strings. The defined rules can be added into a DataFrameChecker and the results can be obtained. Using such a method leads to flexibility in adding/removing/applying checks.

STEP 3A) Merge NPL and Xidml File

Below diagram shows the tasks inside the Merge task.

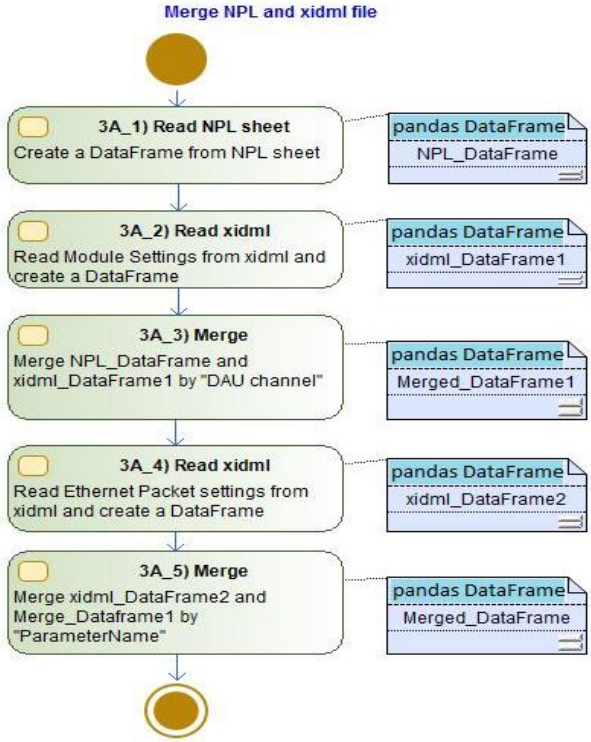

Fig 6 Steps of Collecting Information to use in automatic creation of the IADS's ParameterDefaults table

To create a formula, we need to gather information. For example to generate formula given in Figure 2, we need to collect the parameter name, gage factor, the channel number on which the gage is connected to, the channel settings (maximum and minimum voltages of ADC), excitation setting, etc. IADS's parameter default contains not only the formula string but also other fields that we want to fill automatically. For instance we will set the *SubGroup* as "FTI_STR_1024Hz" for STR153b, because this parameter is sampled at 1024Hz, which can be infered from the xidml file. This will be used for grouping for data exporting purposes, so that all parameters having same *SubGroups* will be exported together.

Figure 7 to Figure 10 are screenshots illustrating what the DataFrames in Figure 6 look like (for demonstration purposes, DataFrames for only two accelerometers in ACC sheet of NPL are considered).

Once NPL and xidml are read into DataFrame's, we make use of the pandas' built-in merge capability according to any common content in DataFrames, similar to table joining operations in database tables. We have to read xidml twice, first for channel settings and second for ethernet packet rate settings (from which we calculate parameter recording rates to fill the SubGroup field). We also merge twice, first by channel and then by parameter name.

STEP 3B) Create and Output Formulas

Having assembled all the required parameters in a single DataFrame, creating formulas for a specific sheet is a straight-forward task of string concatenations. Figure 13 shows the output for this demo case. As can be seen, two formulas are created for a single channel. For instance for ACC101x channel, we have one formula for EU conversion (ACC101_g) and one formula for the millivolt representation (ACC101_mV) (It is useful to generate such middle parameters, particularly for troubleshooting).

Note that all information are combined either from NPL or xidml file. For instance SubGroup columns are different for ACC101 and ACC201 since they are sampled at different rates as deduced from xidml file. Moreover, ACC101 channels are not oriented but the ACC201 ones are. (Note that small blank gaps left in the formula of ACC201 provides a visual hint to indicate that orientation was performed).

STEP 4) Indicate Errors

If there is an error or an inconsistency, the tool indicates the specific parameters where information is not properly entered in NPL. Full reliance to the defined rules is expected. Since the rules can be created per each measurement type, in a flexible fashion by using regex's, this process results in a clarified parameter list from which reliable EU conversion formulas are obtained.

The output provides references to where are the errors as well as what they are. An example for errors and inconsistencies in Parameter List are provided in Figure 11.

-	Parameter No	Zone	Requester	Parameter Name	Unit	Recording Frequency (Hz)	Real Time Frequency (Hz)	Sensor Name	Serial Number	Sensitivity	Orientation
D4_J6_Ch0	ACC101x	1	ATA0330	ACC101x_CG_g	g	256	256	DC Accelerometer (Dytran 7503D4 25g)	11001	159,2	
D4_J6_Ch1	ACC101y	1	ATA0330	ACC101y_CG_g	g	256	256	DC Accelerometer (Dytran 7503D4 25g)	11001	159,75	
D4_J6_Ch2	ACC101z	1	ATA0330	ACC101z_CG_g	g	256	256	DC Accelerometer (Dytran 7503D4 25g)	11001	159,5	
D1_J9_Ch16	ACC201x	2	ATA0320	ACC201x_AHRS2_g	g	4096	256	DC Accelerometer (Dytran 7503D3)	10876	401,3	CableLEFT StudDOWN
D1_J9_Ch17	ACC201y	2	ATA0320	ACC201y_AHRS2_g	g	4096	256	DC Accelerometer (Dytran 7503D3)	10876	404,4	CableLEFT StudDOWN
D1_J9_Ch18	ACC201z	2	ATA0320	ACC201z_AHRS2_g	g	4096	256	DC Accelerometer (Dytran 7503D3)	10876	402,1	CableLEFT StudDOWN

Fig.7 ACC sheet of NPL for this example case

df_module_ - DataFrame

DAU_Slot_Ch	ParameterName	Filter Cutoff	Unit	RangeMax	RangeMin
4_6_0	D4_J6_ADC_112_10V_ch0	0.5	Volt	10	-10
4_6_1	D4_J6_ADC_112_10V_ch1	0.5	Volt	10	-10
4_6_2	D4_J6_ADC_112_10V_ch2	0.5	Volt	10	-10
4_6_3	D4_J6_ADC_112_10V_ch3	0.5	Volt	10	-10
4_6_4	D4_J6_ADC_112_10V_ch4	0.5	Volt	10	-10
4_6_5	D4_J6_ADC_112_10V_ch5	0.5	Volt	10	-10

Fig. 8 DataFrame created from xidml (only ACC101 related part is shown)

df_merge_ - DataFrame

Index	-	ParameterNo	MeasurementName	Unit_NPL	RecordingFrequencyHz	Sensitivity	Orientation	arameterName_XIDML_PARAM	RangeMax	RangeMin	Stream_ID	inet_Name	Parameter_Name	Occurrences	SampleRate
0	D4_J6_Ch0	ACC101x	Body Accelerat…	g	256	159.2	NOT ORIENTABLE	D4_J6_ADC_112_10V_ch0	10	-10	60	D4_Voltage _256Hz	D4_J6_ADC_112_10V_ch0	4	256
1	D4_J6_Ch1	ACC101y	Body Accelerat…	g	256	159.75	NOT ORIENTABLE	D4_J6_ADC_112_10V_ch1	10	-10	60	D4_Voltage _256Hz	D4_J6_ADC_112_10V_ch1	4	256
2	D4_J6_Ch2	ACC101z	Body Accelerat…	g	256	159.5	NOT ORIENTABLE	D4_J6_ADC_112_10V_ch2	10	-10	60	D4_Voltage _256Hz	D4_J6_ADC_112_10V_ch2	4	256
3	D1_J9_Ch16	ACC201x	AHRS X	g	4096	401.3	CableLEFT StudDOWN	D1_J9_ADC_112_10V_ch16	10	-10	49	D1_Voltage_4096Hz	D1_J9_ADC_112_10V_ch16	16	4096
4	D1_J9_Ch17	ACC201y	AHRS Y	g	4096	404.4	CableLEFT StudDOWN	D1_J9_ADC_112_10V_ch17	10	-10	49	D1_Voltage_4096Hz	D1_J9_ADC_112_10V_ch17	16	4096
5	D1_J9_Ch18	ACC201z	AHRS Z	g	4096	402.1	CableLEFT StudDOWN	D1_J9_ADC_112_10V_ch18	10	-10	49	D1_Voltage_4096Hz	D1_J9_ADC_112_10V_ch18	16	4096

Fig.9 Merged DataFrame

ParameterDefaults	Parameter	ParamType	ParamGroup	ParamSubGroup	ShortName	LongName	Units	Color	Width	DataSourceType	DataSourceArguement	UpdateRate
FTI_ACC	ACC101x_CG_g	float	FTI_ACC	FTI_ACC_256Hz	ACC101x_g	Body Acceleration from CG X axis	g			Derived	(D4_J6_ADC_112_10V_ch0/65536*20-10)*1000 / 159.2	256
FTI_ACC	ACC101y_CG_g	float	FTI_ACC	FTI_ACC_256Hz	ACC101y_g	Body Acceleration from CG Y axis	g			Derived	(D4_J6_ADC_112_10V_ch1/65536*20-10)*1000 / 159.75	256
FTI_ACC	ACC101z_CG_g	float	FTI_ACC	FTI_ACC_256Hz	ACC101z_g	Body Acceleration from CG Z axis	g			Derived	(D4_J6_ADC_112_10V_ch2/65536*20-10)*1000 / 159.5	256
FTI_ACC	ACC201x_AHRS2_g	float	FTI_ACC	FTI_ACC_4096Hz	ACC201x_g	AHRS X	g			Derived	(D1_J9_ADC_112_10V_ch17/65536*20-10)*1000 / 404.4	4096
FTI_ACC	ACC201y_AHRS2_g	float	FTI_ACC	FTI_ACC_4096Hz	ACC201y_g	AHRS Y	g			Derived	(D1_J9_ADC_112_10V_ch16/65536*20-10)*1000 / 401.3	4096
FTI_ACC	ACC201z_AHRS2_g	float	FTI_ACC	FTI_ACC_4096Hz	ACC201z_g	AHRS Z	g			Derived	(D1_J9_ADC_112_10V_ch18/65536*20-10)*1000 / 402.1*(-1.0)	4096
FTI_ACC	ACC101x_CG_mV	float	FTI_ACC	FTI_ACC_256Hz	ACC101x_mV	Body Acceleration from CG X axis	mV			Derived	(D4_J6_ADC_112_10V_ch0/65536*20-10)*1000	256
FTI_ACC	ACC101y_CG_mV	float	FTI_ACC	FTI_ACC_256Hz	ACC101y_mV	Body Acceleration from CG Y axis	mV			Derived	(D4_J6_ADC_112_10V_ch1/65536*20-10)*1000	256
FTI_ACC	ACC101z_CG_mV	float	FTI_ACC	FTI_ACC_256Hz	ACC101z_mV	Body Acceleration from CG Z axis	mV			Derived	(D4_J6_ADC_112_10V_ch2/65536*20-10)*1000	256
FTI_ACC	ACC201x_AHRS2_mV	float	FTI_ACC	FTI_ACC_4096Hz	ACC201x_mV	AHRS X	mV			Derived	(D1_J9_ADC_112_10V_ch17/65536*20-10)*1000	4096
FTI_ACC	ACC201y_AHRS2_mV	float	FTI_ACC	FTI_ACC_4096Hz	ACC201y_mV	AHRS Y	mV			Derived	(D1_J9_ADC_112_10V_ch16/65536*20-10)*1000	4096
FTI_ACC	ACC201z_AHRS2_mV	float	FTI_ACC	FTI_ACC_4096Hz	ACC201z_mV	AHRS Z	mV			Derived	(D1_J9_ADC_112_10V_ch18/65536*20-10)*1000	4096

Fig. 10 IADS ParameterDefaults Table containing ACC101 and ACC201 formulas

```
VIB  sheet needs correction:
Column(s)       :  Sensitivity
Check Description  :  IS_NUMBER_WITH_OR_WITHOUT_mVpg_or_pCpg
Failing Rows       :  ['D4_J14_Ch3', 'D4_J14_Ch4', 'D4_J14_Ch5', 'D3_J15_Ch9',
'D3_J15_Ch10', 'D3_J15_Ch11', 'D4_J14_Ch8', 'D4_J14_Ch9', 'D4_J14_Ch10']


Column(s)       :  ParameterNo
Check Description  :  VIB_PARAMNO
Failing Rows       :  ['D7_J15_Ch11']


Column(s)       :  ('ParameterNo', 'Zone')
Check Description  :  ('GET_ZONE_FROM_PARAMETERNO', 'GET_ZONE_FROM_ZONE')
Failing Rows       :  ['D7_J15_Ch11']
```

Fig. 11 Example for indication of errors in VIB sheet

Formula Comparing

Normally, we have IADS sessions in 3 computers. First one is in the Laptop in Hangar, second one is in the Data Monitoring Room where engine tests on ground are performed, and a final one in the Telemetry Room. We are to ensure that all the formulas in these 3 places are same. The created Formula Generation tool is named as “Master” Formula Creation tool, (“Master” word is used in the sense that these formulas are the “original” or the “reference” ones.) A simple text based compare is by itself very helpful but we also did a computation wise comparison using synthetic value injection in places of raw data and middle parameters. In other words, we intend to compare formulas not only textually but also by their results. Basically **“b+a+2”** and **“2+a+b”** are considered as equivalent, although not “same” when compared textually.

Inspired from [5], lexical analysers, parsers and interpreters are implemented to evaluate formulas. We scan the formula character by character and match with defined tokens to form a tree, composed of nodes. The nodes can be either a “*Leaf Node*” (such as a constant of a variable) or a *“Tree Node”* (which are either operators (like addition and substraction) or *“Function Nodes”* (as defined in IADS (such as *pow*(), *byteswap16*(), etc.)). Then, upon assigning values for variables (if present), a “Parser” evaluates the tree. Recursive nature of the process handles calculation of nested formulas as well as conditional formulas which are common in the present EU conversion formulas. An example formula and its Tree are given below, where leaf nodes and tree nodes are outlined orange and black respectively. Note that function nodes are also shaded. If it existed, names not known as functions would have been taken as variable names (like raw data names), to which user could assign/inject values and calculate output of the formula.

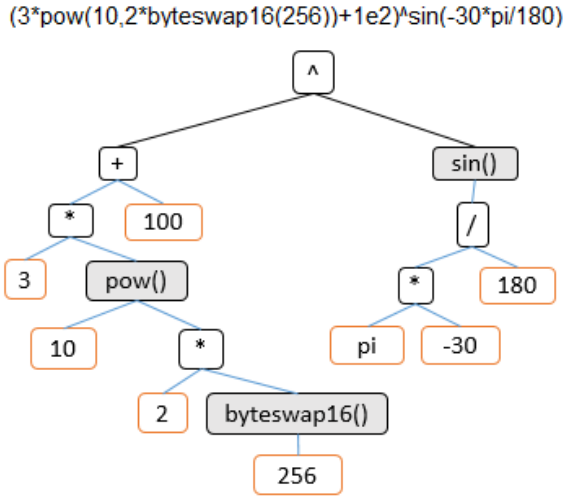

Fig. 12 Example formula and its Tree (outputs 0.05)

Conclusion

An error in an EU conversion formula could have devastating effects on the credibility of the instrumentation system and consequently shift the focus of data analyzers from measured system to the instrumentation system which is traditionally the easy target, the usual suspect, the first place where fingers are pointed in delicate flight test campaign processes. Therefore, it is wise to generate analog parameter formulas in an automatic fashion. We developed a software tool which is built to fetch required information from parameter list (NPL) and from the instrumentation configuration file (xidml file). The tool also performs some consistency checks during the process to enforce instrumentation engineers to prepare NPL and xidml in a structured fashion. The benefit of the tool can better be appreciated by realizing that even the provided minimal ACC examples contain tricky parts (such as orientation correction, subgroup differentiation, etc.). On the other hand a formula comparison tool by means of raw data injection is also developed to ensure that raw data is not subjected to different conversion in the data monitoring locations.

References

[1] Automated PCM Placement, Fatih Haciomeroglu et al. ETTC 2023

[2] Evolution of Data Exporting, Fatih Haciomeroglu et al. ETTC 2022

[3] www.xidml.org

[4] www.curtisswrightds.com

[5] Chapter 19 of Programming Python 4th Ed. O’Reilly, Mark Lutz