101-2007

SAS® Data Integration Studio and SAS® Macro – Working in Harmony for Data Warehousing Projects

Tatyana Kovtun, Bayer Healthcare Pharmaceuticals, Montville, NJ

ABSTRACT

SAS DI Studio is gaining popularity as a Data Warehousing tool. As a gateway to the SAS® Metadata Server, it enables users to design and run metadata driven processes. This paper explores two techniques we developed based on the power and flexibility of DI Studio and SAS/Macro. First, we will present a method for integrating external business metadata containing complex derivation logic. Second, we will present a scheme for building modular DI Studio job segments, which facilitates creation of reusable processes capable of handling diverse data. To demonstrate these techniques, we will profile a DI Studio based application that converts clinical trial data to FDA required format.

Anyone who has tried to use DI to 'beat' raw data to an arbitrary Data Warehouse storage structure knows that derivation logic can sometimes be more extensive than is practical to enter in DI Studio Expression field. Our alternative solution uses SAS/Macro inside custom transforms to read external tables of mapping metadata, which enables these tables to drive transformation processes inside DI Studio.

Our second technique is built with SAS code capable of querying a Metadata Server for attributes and associations. The response can feed macro parameters to our jobs 'on the fly'. Thus, a single macro enclosed in DI custom transformation can generate different code depending on Metadata Server response, making it adaptive to its context. Populating macro parameters from the Metadata Server to generate a code inside DI Studio brings a powerful new approach to designing modular data transformation processes.

INTRODUCTION

Many companies are turning to SAS as their Data Warehousing storage format, and until recently the tool of choice was Base SAS enabled with SAS Macro. However, with the emergence of the SAS®9 Open Metadata Architecture, Data Warehouse load jobs can now be built as generic, metadata driven processes from 'building blocks' available through the Data Integration Studio GUI. Such metadata driven components significantly facilitate job development and maintenance. Other benefits of SAS9 Business Intelligence Platform include the availability of the centralized metadata for the variety of business needs, and more efficient Data Warehouse refresh procedures that can now be turned to Stored Processes with convenient scheduling features.

In this paper we will demonstrate how we design and use metadata driven processes to convert our collected data to FDA required format to be stored in JANUS Data Warehouse. Although the examples provided deal with pharmaceutical data, the proposed approach is not industry specific and can be expanded well beyond our application.

Our strategy for mapping data to Data Warehouse format is supported by two techniques, **Externalized Metadata**, and **Modular Job Construction.** These techniques enable the following:

Metadata Driven Jobs All load jobs are designed within DI Studio and are metadata driven.

Division of Business Metadata vs. Data and Process Metadata Not all metadata is registered with the Metadata Server. The paper will explain what *Business Metadata* is, and why we store it in external table, called Mapping Control Table.

DI Studio Extensibility Plain use of DI Studio 'out-of-box' features does not fully address our Data Warehousing needs. We took advantage of the tool's flexibility and expandability, enhancing it with custom Java plugins and custom transformations.

Our Mapping Transformation One of our custom transformations, the Mapping Transformation, converts Business Metadata stored in Mapping Control Table to SAS code. This SAS code creates and populates variables in the Data Warehouse.

Job Modularity We construct our jobs out of two segments. The first segment is load specific and is subject to modifications if there are changes in the source data. The second segment is standardized across the load jobs and is flexible enough to 'self-adapt' to the changes.

Metadata Server Communication Custom transformations contain parameterized macros that get resolved 'on the fly' by two-way communication with Metadata Server. The communication with Metadata Server is implemented through both DI Studio and SAS API.

Job Standardization The dynamic communication with Metadata Server provision for creation of standardized job segments that can become part of any load job without modifications.

Dynamic Macros The use of dynamic macros inside DI Studio transformations improves reusability and simplify the maintenance of Data Warehouse load jobs.

We will use term "**Source** datasets" for our collected data in SAS format, and the term "**Target** Datasets" for Data Warehouse tables – the load process output.

EXTERNALIZED METADATA

BUSINESS METADATA

Designing a load job typically involves the following tasks:

1. Creation of a Data Dictionary - a document that describes all table and variable attributes.
2. Creation of a Technical, or Mapping Specification document. This document should provide details about variable transformation process, including derivation logic behind converting Source fields to Target variables.
3. Development of a SAS process that implements the specified derivation logic.
4. Testing, debugging, and validation of the results.

The Mapping Specification described above is what we call **Business Metadata**. Sometimes the mapping logic (which includes derivation logic) can be very complex and extensive. Consider the following cases:

- Several Source variables are to be conditionally mapped to a single Target variable
- Numeric codes in Source variables are to be decoded to text strings
- Some Target variables are not directly derived from Source variables, but rather are populated by string literals based on certain conditions, including the Source dataset from which the given record came.
- Some standardized outputs require transposition from horizontal to vertical structure
- Sometimes the derivation logic to be applied is itself conditional. In FDA submission preparations, this often occurs when source variables coming from different experimental protocols require different derivation logic.
- All date/time values must be presented in ISO format, etc.
- Multi-level nested conditional logic is to be applied.

CODING WITH BASE SAS

Obviously, with very extensive coding, Base SAS can handle all the above cases. However, these programs would be difficult to maintain as their reusability for changing data structure would be limited. The use of SAS macros could improve program reusability, but still changes in mapping logic, or Source and Target datasets structure, would lead not only to changes in macro parameters, but also changes in macro code itself. Re-coding, as we all know, will inevitably necessitate re-testing, re-debugging, re-validation, and re-installation. We consider it quite inefficient in terms of business handling and resource utilization.

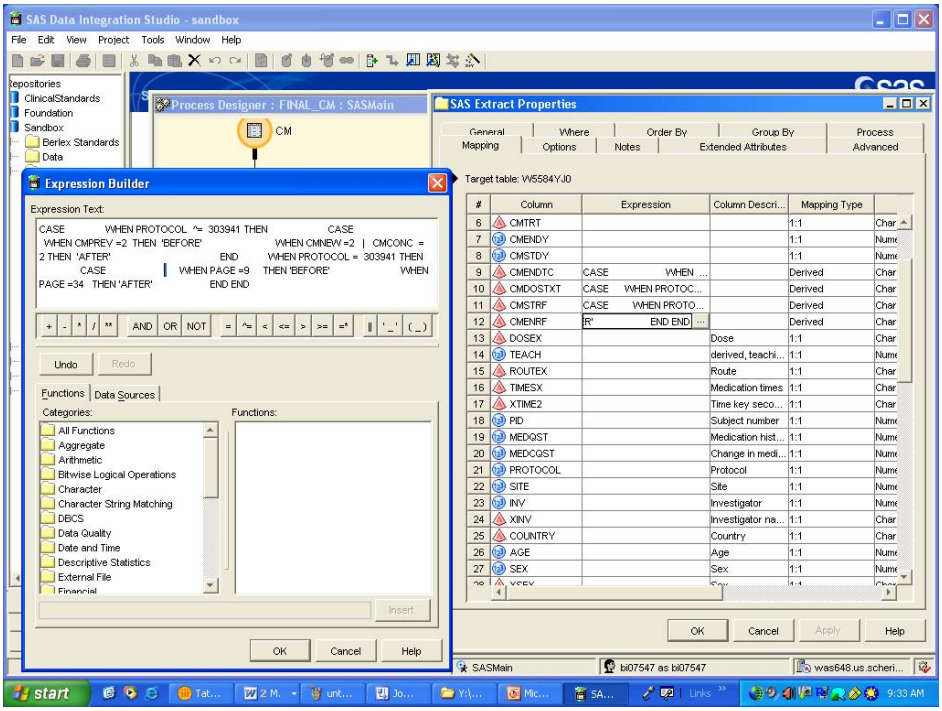
BUILDING JOBS WITH DI STUDIO

DI Studio is a code generator that 'brews' SAS code out of different kinds of metadata stored within Metadata Server repositories. This metadata includes, but is not limited to

- **Data Metadata** includes tables and variables attributes – everything you see in the output of the SAS CONTENTS PROCEDURE.
- **Process Metadata** are the job building blocks called DI Studio Transformations. Process metadata includes transformation properties and options along with underlying pieces of built-in SAS code.
- **Business Metadata**, as mentioned above, is in fact the 'mapping' part of the Process metadata. We separate it from the rest of Process metadata because it forms foundation for our methodology.

DI Studio is a great tool for operating with Data metadata and Process metadata. Unfortunately, the DI Studio GUI does not fully address the complexity of Business metadata, especially when it contains nested conditional derivation logic, requiring many lines of coding. The standard use of DI Studio requires manual entry of these lines into the relatively small Expression Builder window. There is process in place to validate these hard-coded expressions, and the window doesn't have any of the code-coloring features available at the Enhanced Editor.

Fig. 1 DI Studio windows for defining mapping expressions.



Similar to the traditional SAS programming, this hard-coded derivation logic needs to be modified, debugged, and retested any time a change in Business Metadata takes place. Fortunately, DI Studio is flexible enough to accommodate custom macro processes, which we use to internalize our *externally* stored mapping logic.

SAVING BUSINESS METADATA TO THE MAPPING CONTROL TABLE

An alternative to entering complex derivation logic to Expression Window is to store Business metadata in an external SAS dataset. We refer to this SAS dataset as our **Mapping Control Table**, or just **Control Table.** Our **Mapping Control Table** is 3-in-1 entity:

- ✓ It is an extended Data Dictionary with all variable attributes populated directly from DI Studio metadata.
- ✓ It is Technical specification providing details on variable derivation logic, and finally
- ✓ It is an input 'driving force' behind the data transformation process.

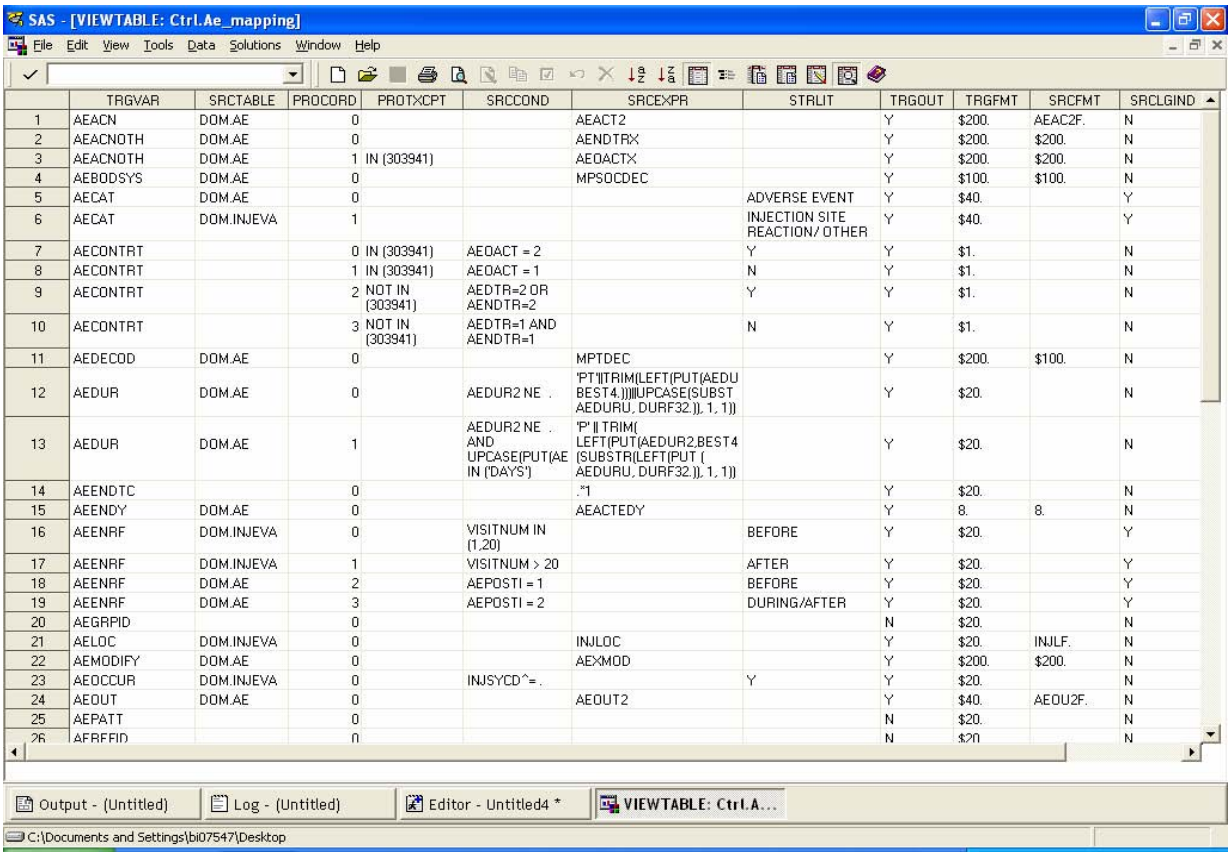
The logic in the Mapping Control Table is incorporated into the processing of our job (built in DI Studio) with a custom transformation containing SAS Macro code. The creation of the Mapping Control Table is facilitated by a **Mapping Plugin**, which offers a much more friendly interface for entering SAS expressions. You can find more information on how to build custom Java Plugins in Application Development section paper entitled Extending SAS® Data Integration Studio with Java: Custom GUIs and SAS Server Interaction by Christopher Treglio.

Key points regarding the Mapping Control Tables:

- They are a **SAS datasets**, which can be easily read and processed .
- They are **Target focused**, i.e. any one Control Table contains the logic to create one Target dataset.
- They are **load job specific**, i.e. one Control Table is used by one load job
- They follow a **naming convention –** concatenation of Target name (e.g. AE) and string 'mapping', e.g. AE_mapping. This naming convention is important for generic processes controlled by Mapping Tables.
- They all have an **identical structure**, therefore they can be **represented by the same metadata object** in any number of DI processes (jobs). The proper table is accessed by the process through parameter resolution in the table name.
- They have **at least one record for each Target variable**.
- Their structure **breaks multi-tier nested conditional statements** to several records, with one record per unique combined condition. This implies that there could be **multiple records** per one Target variable.

- Some of their fields (such as variable names and attributes) are **populated directly from Metadata Server**.
- Some of their fields are **populated by users through a Mapping Plugin GUI,** specifically built to facilitate the creation of the control tables. The editable fields are either menu driven (like Source Table column), or require typing into plugin expression windows (e.g. Source Condition, Source Expression).
- The **date sensitivity** can be either incorporated through Source Condition Logic, or through another special field (in our case it is Protocol Exception field)
- Their structure enables content to be inserted in IF.. THEN … statements to **drive internal data transformation process.**
- They get **validated** through the GUI with the results shown in a validation report that points to discrepancies, conflicting values, or syntax errors in each control record at the time point of saving the table.
- Having the same structures, Mapping Control Tables for all DW loads can be concatenated into one 'mega dataset' that will provide the most complete metadata on the entire Data Warehouse.
- This Mega Data Dictionary is easily searchable for any business and/or IS analytical needs.

Fig. 2 Example of Mapping Control Table shown in SAS Viewtable window.



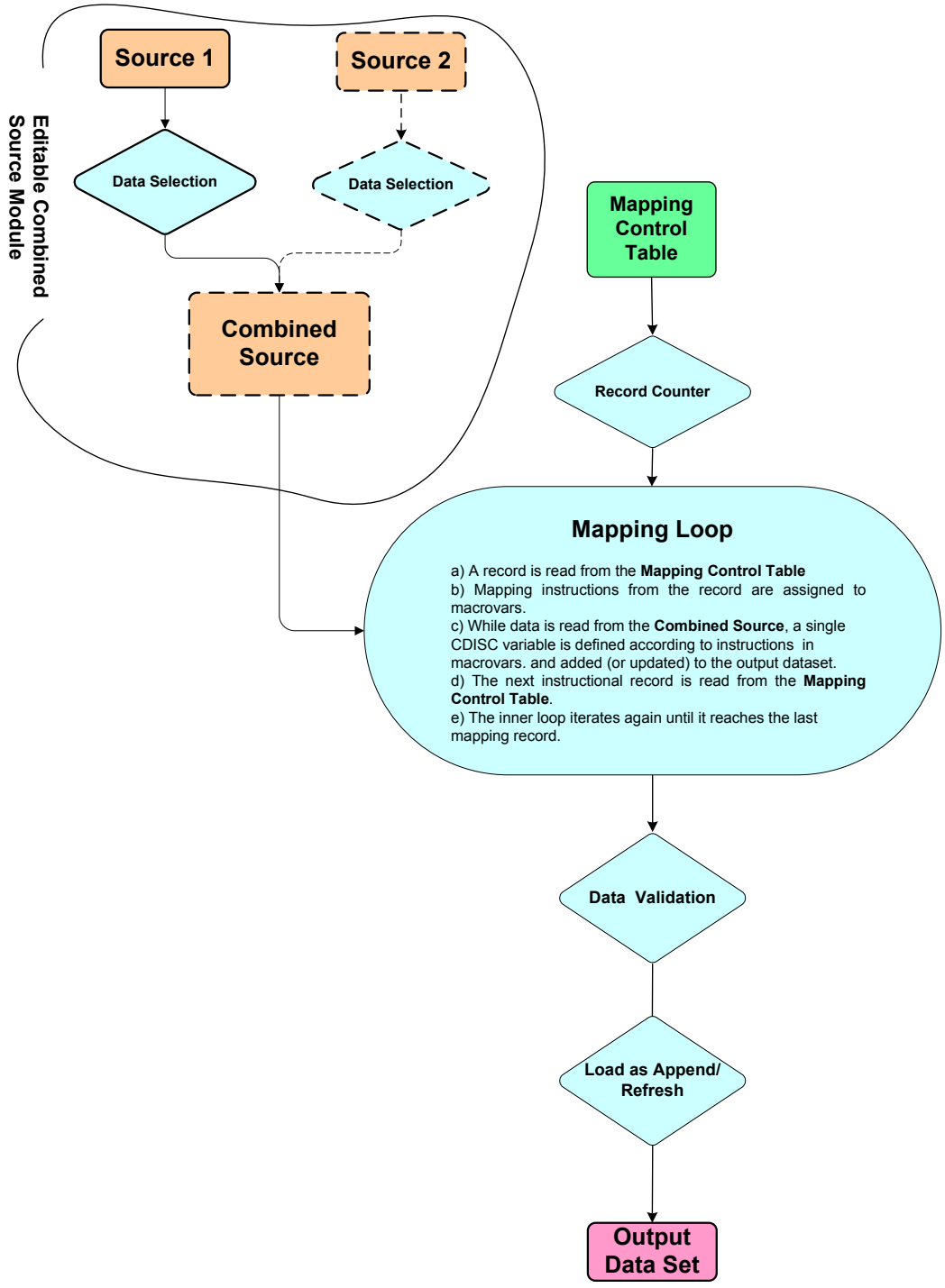
	TRGVAR	SRCTABLE	PROCORD	PROTXCPT	SRCCOND	SRCEXPR	STRLIT	TRGOUT	TRGFMT	SRCFMT	SRCLGIND
1	AEACN	DOM.AE	0			AEACT2		Y	$200.	AEAC2F.	N
2	AEACNOTH	DOM.AE	0			AENDTRX		Y	$200.	$200.	N
3	AEACNOTH	DOM.AE	1	IN (303941)		AEOACTX		Y	$200.	$200.	N
4	AEBODSYS	DOM.AE	0			MPSOCDEC		Y	$100.	$100.	N
5	AECAT	DOM.AE	0				ADVERSE EVENT	Y	$40.		Y
6	AECAT	DOM.INJEVA	1				INJECTION SITE REACTION/ OTHER	Y	$40.		Y
7	AECONTRT		0	IN (303941)	AEOACT = 2		Y	Y	$1.		N
8	AECONTRT		1	IN (303941)	AEOACT = 1		N	Y	$1.		N
9	AECONTRT		2	NOT IN (303941)	AEDTR=2 OR AENDTR=2		Y	Y	$1.		N
10	AECONTRT		3	NOT IN (303941)	AEDTR=1 AND AENDTR=1		N	Y	$1.		N
11	AEDECOD	DOM.AE	0			MPTDEC		Y	$200.	$100.	N
12	AEDUR	DOM.AE	0		AEDUR2 NE .	'PT'||TRIM(LEFT(PUT(AEDUBEST4.))||UPCASE(SUBSTAEDURU, DURF32.)), 1, 1))		Y	$20.		N
13	AEDUR	DOM.AE	1		AEDUR2 NE . AND UPCASE(PUT(AEIN ('DAYS')	'P' || TRIM(LEFT(PUT(AEDUR2,BEST4 (SUBSTR(LEFT(PUT (AEDURU, DURF32.)), 1, 1))		Y	$20.		N
14	AEENDTC		0			.*1		Y	$20.		N
15	AEENDY	DOM.AE	0			AEACTEDY		Y	8.	8.	N
16	AEENRF	DOM.INJEVA	0		VISITNUM IN (1,20)		BEFORE	Y	$20.		Y
17	AEENRF	DOM.INJEVA	1		VISITNUM > 20		AFTER	Y	$20.		Y
18	AEENRF	DOM.AE	2		AEPOSTI = 1		BEFORE	Y	$20.		Y
19	AEENRF	DOM.AE	3		AEPOSTI = 2		DURING/AFTER	Y	$20.		Y
20	AEGRPID		0					N	$20.		N
21	AELOC	DOM.INJEVA	0			INJLOC		Y	$20.	INJLF.	N
22	AEMODIFY	DOM.AE	0			AEXMOD		Y	$200.	$200.	N
23	AEOCCUR	DOM.INJEVA	0		INJSYCD^=.		Y	Y	$20.		N
24	AEOUT	DOM.AE	0			AEOUT2		Y	$40.	AEOU2F.	N
25	AEPATT		0					N	$20.		N
26	AEREFID		0					N	$20		N

The labels for the columns (from left to right) are:

Target Variable, Source Table, Processing Order, Protocol Exception, If Condition, Source Expression, String Literal, Output Flag, Target Variable Format, Single Source Variable Format, Source Logic Indicator.

THE MAPPING CONTROL TABLE IN ACTION

Fig. 3 provides an example of a job flow that utilizes a Mapping Control Table for Target variables derivation.



The core of the process (represented by a blue oval) is a **Mapping Loop** that iterates through each record of the Mapping Control Table, passes its values to macro variables, which in part generate a SAS data transformation code.

The Mapping Loop consists of a simple Extract step that creates a work copy of the Control table and a custom Mapping Transformation step.

Mapping Transformation Highlights:

1) The custom Mapping Transformation contains a macro loop that iterates through each control table record.

2) As a control table record is read, the values of certain fields get assigned to macro variables.

3) DATA STEP reads Combined Source (an input dataset), record by record, resolving macro variables into an IF… THEN… statement that drives a conditional assignment of a single Target variable. Thus, the process creates and populates the Target variable according to the derivation logic in the control record.

4) After the last row in the Combined Source is reached, the newly defined/updated target variable gets added to the output dataset through an UPDATE statement.

5) The process reads the next record in the control table to pass its values to macro variables.

6) The process starts iterating through Combined Source dataset again, etc.

7) Mapping Transformation is a custom transformation created using DI Studio **Transformation Generator Wizard**

After the loop processes the last control table record, the Target dataset is formed. At this point, all output variable attributes conform to Target Metadata, and these variables will have been populated according to mapping instructions from the Control Table.

MODULARIZED JOB CONSTRUCTION

Having discussed above our first broad technique, **Externalized Metadata**, we now move to our **Modularized Job Construction**.

SOURCE SPECIFIC AND GENERIC JOB MODULES FOR DATA WAREHOUSE LOADS

Our Fig. 3 example presents a case where data from primary sources first has to be pre-selected, then joined together into the Combined Source dataset. After being combined (vertically, or horizontally – depending on business needs) the data passes through the Mapping Loop for a required transformation. After all target variables get defined according to Control Table instructions, the final data just needs to be validated and loaded. Some additional tasks may also include sorting and indexing.

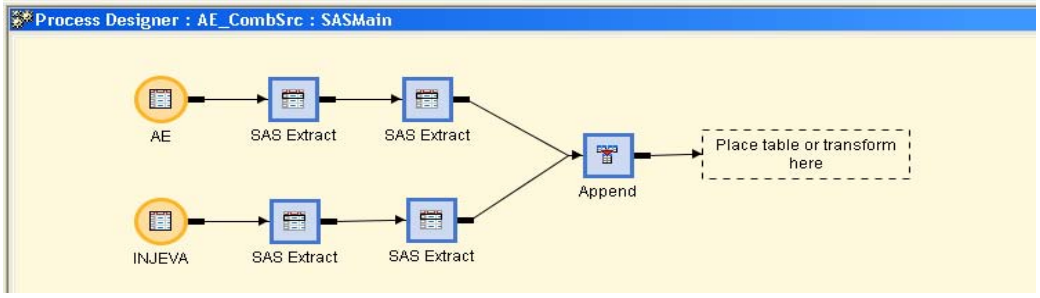
One can easily see that the main module of the presented job flow can be considered *generic* across many (if not all) load jobs. This basically means, that the same macro (or in our setting DI process) can be inserted as a component to any load job without modifications. The necessary degree of flexibility on *variable* level is implemented through the Mapping Control Table. Specifics of the output dataset , such as output library, sort order, indexes, etc, can be supplied to the macro/DI as parameter values. The subsequent sections will provide an insight on how we feed these parameters on-the-fly through the direct communication with the Metadata Server.

However, the first part of the process, where the input data get pre-selected and joined together, can not be easily standardized across the load jobs. This is what we consider a *load-specific* module. Such modules need to be individually constructed for each load job.

The next three diagrams display DI Studio implementation of modular job construction.

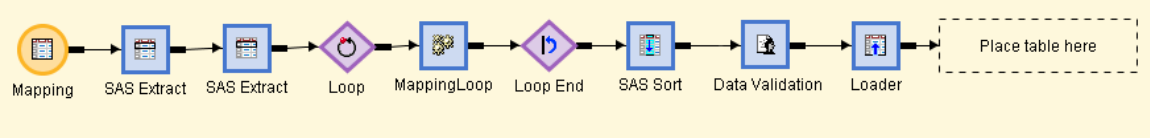
The screen shot below presents an example of a *load-specific* module, constructed from DI Studio standard transformation blocks.

Fig. 4 Load-specific job module.



The next screen shot shows our *generic* module, constructed partially out of DI standard transforms (SAS Extract, Loop, SAS Sort, Loader) and partially out of our custom designed transforms with SAS macro running 'behind the scene' (Mapping Loop, Validation). The additional outer Loop represented by diamond-shaped icons utilizes DI standard Loop transformation. This transformation brings another dimension to the flexibility of our method and takes care of data transposition process where applicable. This generic module can be inserted in any load job 'as is' without any modifications.

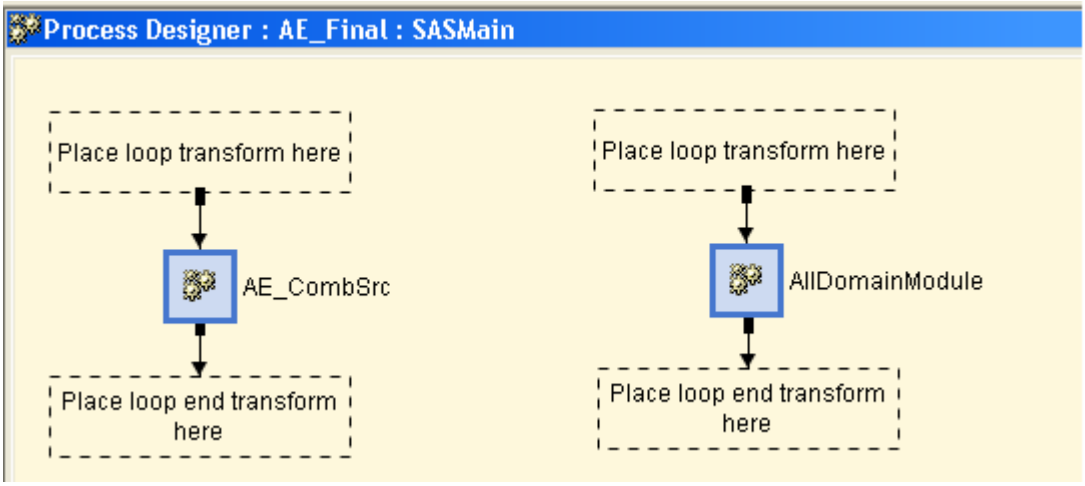
Fig. 5 Generic job module.



As we have already mentioned, the Mapping table object in the beginning of the generic module represents *any* Mapping Control table. But how does the process know which concrete Control Table it has to access for this particular load job? And what is the BY group in the SORT step? What target dataset is this empty placeholder at the end of the process supposed to represent? In most general terms: all these 'unknowns' (macro parameters) will be resolved on-the-fly during job execution by direct communication with Metadata Server implemented through requests submitted from SAS API and XML responses mapped back to SAS. The next section will provide more details regarding this process.

The last screen shot displays an example of a shell job that puts these two modules together. The output of the first *load-specific* module becomes an input dataset for the second *generic* module.

Fig. 6 Shell job – a job that 'forces' two modules into one process



QUERYING METADATA SERVER FROM SAS INTERFACE

Key to our ability to make our job module generic is to have our job steps run differently under different circumstances. This context-dependent logic essentially requires self-configuring SAS code.

This is how it all happens:

The pre-process (custom SAS code) built into the shell job utilizes a set of job specific optional macro variables supplied by DI Studio. One of these macro variables holds the metadata ID of the current job. This triggers two-way communication with Metadata Server:

Pre-process: Hello Metadata Server, my Id is A5NFKGV.AF00001M. What is my *Name*?

Metadata Server: The job with this ID is registered under the name **AE_Final**.

Pre-process: Thank you very much. Now I know that my process will need to retrieve **AE_Mapping** control table and build **OUTLIB.AE** target dataset. However, I have some more questions to ask:

- What is the *physical directory* of the OUTLIB output library?

- What are the *attributes*, including *Extended Attributes* of the OUTLIB.AE dataset? (We have sort order string stored as an extended attribute)

- Also please give me the list of target variables and their attributes, including lengths, formats, and labels.

SAS and Metadata Server communicate in XML, which means that every requests must be filed in XML (using FILE and PUT statements) and every response must be mapped back to SAS dataset using pre-built map and XML

LIBNAME engine. Once became available as SAS values, these responses can be parsed to macro parameters that will be passed on to the AllDomainModule. Macro resolution will transform the code from generic to a load specific.

This dynamic communication with the Metadata Server makes AllDomainModule highly adaptable and reusable.

The code samples of such dialog are provided in the Appendices 1 and 2. Both code samples query Metadata Server for directory path to our output library (named CDISC). Appendix 1 shows explicit XML interaction with the Metadata Server. The XML response from the server is then translated to SAS using XML LIBNAME engine and XML map. Appendix 2 demonstrates another solution to the same task. This time the communication to Metadata Server is implemented in a DATA STEP through METADATA_GETNASN and METADATA_GETATTR SAS metadata functions.

In both examples, after the directory path is retrieved, a macro variable holding the path string is defined. Finally, SAS resolves this macro variable in a LIBNAME statement. The code is built upon the sample provided by SAS Technical Support. More code examples, including building association paths for **XMLSelect search=** option can be found in 'Using the Interface to Query Metadata' chapter of SAS Open Metadata Interface: User's Guide.

CONCLUSION

This paper discusses two techniques we've used to improve reusability of our Data Warehousing load jobs within DI Studio environment: **Externalized Metadata** and **Modularized Job Construction**. By externalizing our business metadata, we enable greater reusability of our logic and also simpler, more user-friendly interface. Modularizing our load jobs furthers the goal of reusability by splitting the job modules that need to be modified for each load target from the ones that can be used generically across load jobs. Furthermore, interaction with the SAS Metadata Server allows jobs to "self-configure" their processes, promoting even greater job standardization.

These techniques combine to provide an innovative design for Data Warehouse load jobs, leveraging the advantages of SAS®9 Business Intelligence Platform with tried and true SAS Macro techniques. The isolated use of DI Studio as the only gateway to the SAS Metadata Server doesn't always bring desired results in terms of job efficiency and reusability. However, customized interaction with the SAS Metadata Server through SAS macro makes these jobs generic and dynamic, simplifying their reusability and maintenance.

REFERENCES

SAS Institute Inc. 2004, *SAS 9.1.3 Open Metadata Interface: User's Guide.* Cary, NC:SAS Institute Inc.

Christopher Treglio, "Extending SAS® Data Integration Studio with Java: Custom GUIs and SAS Server Interaction" *SAS Global Forum 2007 Proceedings,* April 2007.

ACKNOWLEDGEMENTS

I would like to thank John Markle for providing encouragement and advice for this paper and Chris Treglio for technical and linguistic help.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. For a copy of the code shown in Appendix 1, please contact me at:

Tatyana Kovtun
Bayer HealthCare Pharmaceuticals
Montville, NJ USA
E-mail: tatyana_kovtun@berlex.com

APPENDIX 1. Using SAS API to Retrieve Directory Path via explicit XML Interaction with Metadata Server

```
 *
 * macro: tempFile
 * description: defines a temporary filename for xml
 *
 *;

%macro tempFile( fname );
```

```
  %if %superq(SYSSCPL) eq %str(z/OS) or %superq(SYSSCPL) eq %str(OS/390) %then
    %let recfm=recfm=vb;
  %else
    %let recfm=;
  filename &fname TEMP lrecl=2048 &recfm;
%mend;

 *
 * macro: getMetadataID
 * description: searches repository and returns metadata ID
 *
 *;

%macro getMetadataID(SEARCH);

  /*  Build the GetMetadata request. */

  %tempFile(request);
  data _null_;
    file request;
    put "<GetMetadataObjects>";
    put "<ReposId>$METAREPOSITORY</ReposId>";
    put "<Type>&OBJTYPE</Type>";
    put "<Objects/>";
    put "<ns>SAS</ns>";
    put "<Flags>392</Flags>";
    put "<Options>";
    put "%bquote(<XMLSelect search="&SEARCH"/>)";
    put "</Options>";
    put "</GetMetadataObjects>";
  run;

  /*  Issue the request. */

  %tempFile(response);

  proc metadata
    in=request
    out=response;
  run;

  /*  Build the XML Map file to parse the response. */

  %tempFile(map);
%mend getMetadataID;


%let DirPath = ;
%let objtype=Directory;
%getMetadataID(Directory[UsedByPackages/SASLibrary[@Name='CDISC']]);

/* Mapping reponse to SAS dataset */
  data _null_;
    file map encoding="utf-8";
    put '<?xml version="1.0" encoding="utf-8"?>';
    put '<SXLEMAP version="1.0">';
    put '<TABLE name="respid">';
    put "<TABLE_XPATH>//&OBJTYPE.</TABLE_XPATH>";

    put '<COLUMN name="DirectoryId">';
    put "<XPATH>//&OBJTYPE.@Id</XPATH>";
    put '<TYPE>character</TYPE>';
    put '<DATATYPE>STRING</DATATYPE>';
    put '<LENGTH>17</LENGTH>';
    put '</COLUMN>';
```

```
    put '<COLUMN name="DirectoryPath">';
    put "<XPATH>//&OBJTYPE.@DirectoryName</XPATH>";
    put '<TYPE>character</TYPE>';
    put '<DATATYPE>STRING</DATATYPE>';
    put '<LENGTH>200</LENGTH>';
    put '</COLUMN>';
    put '</TABLE>';
    put '</SXLEMAP>';

  run;


/*  Parse the response with the XML library engine. */

  libname response xml xmlmap=map;

  data respid;
       set response.respid;
       call symput('DirPath', trim(left(DirectoryPath)));
  run;

/* Issue library assignment statement. */

libname cdisc "&DirPath";
```

APPENDIX 2. Using SAS Metadata Functions to Query Metadata Server from SAS API

```
%let DirPath = ;
data _null_;
  length uri $256
         Dirpath $256
         Text $256;

  rc=1;
  arc=0;
  n=1;

  do while(rc>0);   /* First get URI for SAS Library assotiation */
   rc=metadata_getnasn("omsobj:SASLibrary?@Name='CDISC'", "UsingPackages", n,   uri);
   arc=1; /* Next get the value of the attribute 'DirectoryName' using returned URI.
   */
   if (rc>0) then arc=metadata_getattr(uri,"DirectoryName", Dirpath);
   if (arc=0) then call symput ('Dirpath', left(trim(Dirpath)));
   n=n+1;
  end;
run;


libname cdisc "&DirPath";
```