by experts
for experts

Transformation Manager
Version 5.8

User Guide

Transformation
Manager

Copyright Notice

Table of Contents

User Guide

Introduction

The Transformation Manager User guide is arranged in chapters for each of the component parts of the application suite. Each chapter presents useful user information about the functionality contained therein with advice or guidance on how to use each feature of the application or language. The links below will take you to the relevant chapters in this guide.

TM Designer

Introduction

TM Designer is used to create transform Projects and is the most important application for developing integration or migration projects. This chapter describes the features available in TM Designer and provides advice and information on how to use each feature.

Data Models

Introduction

A data model in Transformation Manager is the description of a data store including its elements, attributes and relationships but not its data. It is a meta-data model that is created from an existing data store or created using functionality within TM Designer.

Sometimes the representation of a data store in a TM Designer data model is not sufficient for an integration or migration project. In this event a user can enhance the model in TM Designer. The most common enhancement is to add or modify relationships. This is particularly true for model types like flat files and Excel spreadsheets. The import process cannot determine relationships automatically, so you add them, for example between two worksheets in an Excel workbook.

TM Designer can load data models from a wide range of data stores. For Transformation Manager these data stores are classified into five main groups which are listed below.

- Databases accessed with a JDBC driver
- XML based data stores including XSD, XML instance and DTD
- Java data stores

- Flat files both fixed and variable width
- Excel spreadsheets

Loading database data models is only limited by the availability of a JDBC driver for the database. TM Designer can generically access data models on Windows systems using the ODBC layer but an appropriate ODBC driver must be installed.

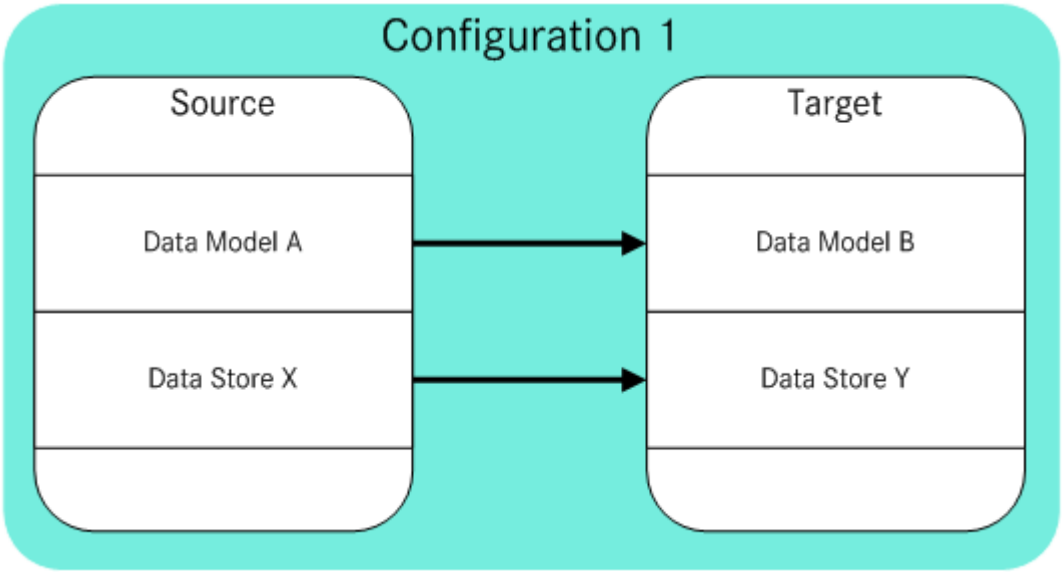
Data models are stored in a repository and can be exported and imported for use in more than one repository if required. The data models are used by projects. A project has the function of managing transforms using data models. A project, when executed, reads and writes from and to source and target data stores using the data models. There may be several data stores with the same applicable data model for the source, target or both. Reading and writing data only occurs when a project is executed using the run or launch functionality in Transformation Manager.

Multiple projects can reference any given model in their repository. For example one project could be writing to a database, while another project might read from it. A user can enhance and modify a model within TM Designer; these changes are visible to all projects that reference the model.

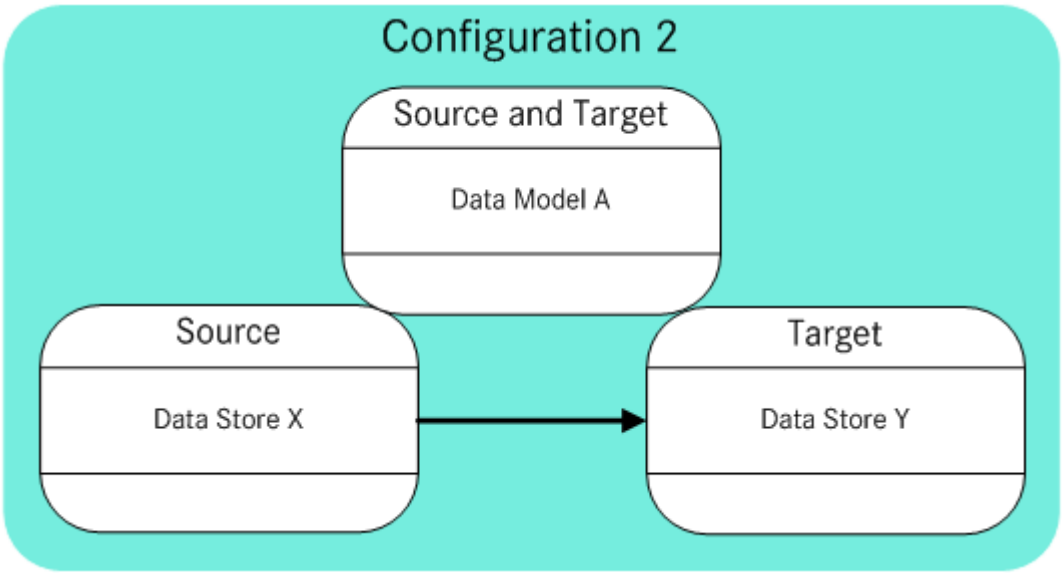
Data stores may change over time with new elements, relationships and attributes being added. In some instances a user may wish to update the affected data model to reflect the changes but on other occasions it may be easier to re-load a data store and create a new updated data model which can then replace the older version in the users projects.

Data Model and Project Configuration Options

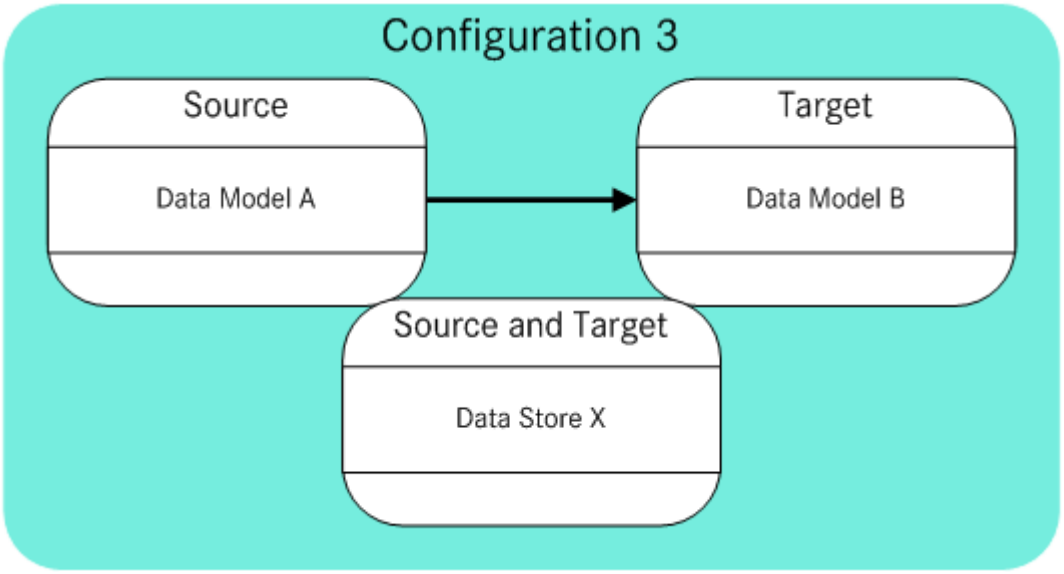
Projects can be configured in four distinct ways depending on the task to be performed. The first configuration assumes that there are two different systems each with their own data models and data stores. In this case a project would have two data models, A and B. The execution of the project would move data from data store X to data store Y using the transforms in the project.



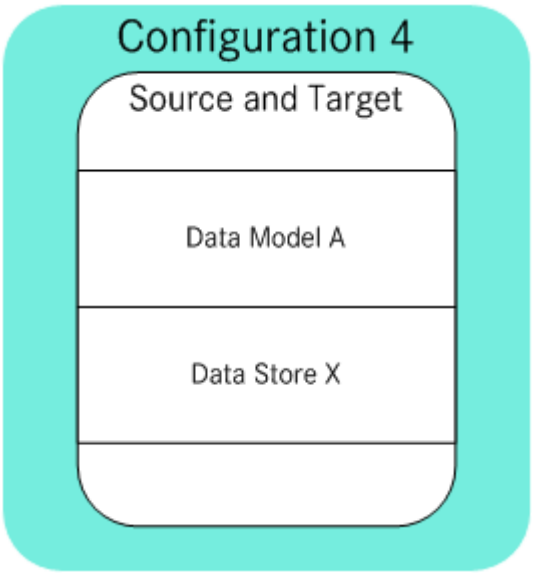
Configuration 2 shows how Transformation Manager can be used to move data between two data stores which have the same data model. In this case the source and target data model is the same but the source and target data stores are different. So data model A will be used to move data between data store X and data store Y and assumes that the structure of the data stores is the same for the purposes of the extract, transform and load task.



Configuration 3 shows a project where the source and target data models are different but the data store is the same. An example of this may be where the structure of the data store has been updated or added to and there is a requirement for data manipulation within the same data store. Care is required in this situation to ensure that the project and its transforms are not attempting to read and write to the same elements and attributes at the same time. This can be managed by executing individual transforms in an appropriate order.



Configuration 4 is similar to configuration 3 as it uses the same data store but the source and target data model is also the same. In this case the user is updating the data store and there is no difference in the data model. As with configuration 3, care is required in this situation to ensure that the project and its transforms are not attempting to read and write to the same elements and attributes at the same time.



See Also: Data Model Constituents (p. 5)
Data Model Terminology (p. 7)
Loading Data Models (p. 7)
Managing Models (p. 8)
Generic Model Properties (p. 9)
Model Packs (p. 9)
Using Pseudo-attributes (p. 10)

Data Model Constituents

Transformation Manager uses a common classification for the constituent parts of all types of data store. For example, a common type of data store is a relational database. A relational database at its most basic level contains tables, columns and relationships. Transformation Manager lets a user refer to the constituents of a relational database as elements, attributes and relationships. So, it will interpret a relational database using the loaded data model and display it to the user using Transformation Managers common classification.

When the data store type changes, for example the target data store is XML; Transformation Manager will use the data model and common classification for that data store. This approach provides a simple method of presenting data models from different data store types consistently.

Each data model will have elements and attributes and may have relationships depending on the data store it was generated from. For example a flat file may have related data but there is no explicit relationship that can be extracted into a data model whereas a relational database will have relationships that can be loaded into a data model.

Element

An element is an identifiable object in a data store normally representing something real that you want to integrate or migrate to or from. It may contain text and/or child elements or be empty. In an XML data model an element is actually called an element while an element in a relational database would be called a table. A flat file would probably be an element while a sheet in an Excel file might be an element. Elements contain attributes and have relationships.

There are two types of elements: standard elements and pseudo elements. Pseudo elements are automatically added to a data model by Transformation Manager and are used for specific purposes. Standard elements are normally loaded from an external data store when a data model is first created. A user can manually add a standard element as required where a data model needs to be modified. Most elements are imported from an external model.

Pseudo elements are used in transform code in the same way as standard elements but they have a special meaning. The most common pseudo element used is $document, which corresponds roughly to the document or schema. It is often used to create a $document transform which contains initialisation logic, and a series of one or more calls to other transforms between standard elements to implement the project requirements.

Elements are used in three places: as the source and target of a transform, as the origin and destination of a relationship and as the source and or target of a procedure. An element is only referenced directly in the code inside a transform when it is required or used as a parameter for a function. When you see an element name in transform code it will refer to a relationship.

Attribute

An attribute is used as a generic term to describe the parts or characteristics of an element. For example, an attribute in a relational database table corresponds to a column. In an XML data store an XML attribute is an element parameter that modifies or refines the meaning of an element, and consists of a name and a value. For example, the attribute called `title` is part of the element called `BOOK`, `<BOOK title='fred'>`. Note that in an XML model the pseudo attribute $Content is also available and represents the immediate node content which in this example would be the textual content of the XML node i.e. 'fred'.

An attribute typically represents a value with a simple type, such as a string (text), a number (decimal, integer, etc), a date or time, or a boolean true/false value. An attribute can be required or optional. To construct an element, you usually need to set at least all of the mandatory attributes. For example, a column in a relational database table defined with a NOT NULL constraint is imported as a mandatory attribute. If it is not set, a runtime constraint violation will ensue. With XML, you can produce a document that omits mandatory attributes, but it may not be schema valid. A transform cannot create, update or delete a source attribute.

Transforms cannot write to a source attribute, they can only be read. A target attribute can be read from and written to. To read a target attribute requires the keyword THAT placed before the target attribute name. In addition the target element must be identified or located.

Relationship

Relationships relate one or more instances of an origin element to one or more instances of a destination element. They are more fully described in the section titled Introduction to Relationships. They are displayed as part of an element in a data model and are a key part of the data model. The They have a specific set of icons that distinguish them from attributes and other elements.

See Also: Introduction to Relationships (p. 32)
Data Model Terminology (p. 7)

Data Model Terminology

Each type of data store has its own model constituents and terminology but Transformation Managers common approach to all data stores lets a user easily develop their projects irrespective of the terms used by a specific data store. Transformation Manager abstracts different data models into a common format and uses common terms which are specifically element, attribute and relationship.

From a logical data model perspective data stores share similar terminology in terms of attributes and relationships and there is an equivalence between the term element and entity. At a physical data model level there are differences in the terminology, all of which can be accommodated within Transformation Manager.

The table below shows how each of these vocabularies equate to Transformation Manager terminology. It describes the constituent components of different data stores and how these relate to that data store and its data model. For example, if the data store is XML, then the terminology applies as is but if the data store is a relational database then where the term element is used it means a table, where the term attribute is used it means a column and where the term relationship is used it means a relationship.

A point to notice is that neither flat files or Excel spreadsheets are loaded into a repository with relationships set which is the reason that they have the term Not Applicable used in the relationship row. However, TM Designer does let a user manually add relationships to data models loaded from these types of data store.

Transformation Manager Terminology	XML	Database	Java	Flat File	Excel
Element	Element	Table	Class	File	Sheet
Attribute	Attribute	Column	Simple Field	Column	Column
Relationship	Relationship	Relationship	Class Field	Not Applicable	Not Applicable
Instance	XML Instance	Row	Object	Row	Row

Loading Data Models

Data models are part of a repository and are loaded or created in a repository using specific Load Model dialogs in TM Designer. A newly created repository will only have an automatically created TMError data model but no user data models so loading data models is a necessary TM Designer task. Each load model dialog is described in detail in our reference guide. The list below shows the available dialogs including the options available within the **Generic Model Loaders** window.

- Load Database Model dialog
- Load Excel Spreadsheet Model dialog
- Load Flat File Model dialog
- Generic Model Loaders window

    - Generic JDBC Loader dialog
    - MySQL JDBC Loader dialog
    - Oracle JDBC Loader dialog
    - PostgreSQL JDBC Loader dialog
- Load Java Model dialog
- Model Packs window
- Load XML Model dialog

For TM Designer to load a data model, it needs to know where the data store it will use is. TM Designer will then import that model information for you. The following table shows where model information is usually located.

Data Store	Model Information Found
XML	within XSDs, DTDs or the XML file itself
Relational Database	within your database data dictionary
Java	within the underlying classes
Flat Files	within the file
Excel Spreadsheet	within an Excel file
Generic Models	within a database data dictionary
Model Pack Models	from Model Pack jar files

Valid Model Names

Imported models must have valid names. The model name must start and end with a letter and contain only letters, digits and underscores.

When importing XML, if you choose the option to **Use the File Name, which is** and the filename is not a valid name, then a valid name is created automatically:

- If the name does not start with a letter, an 'm' is put at the front
- If the name does not end with a letter, an 'm' is put at the end
- All illegal characters in between - i.e. anything that is not a letter, digit or underscore - are removed.

A valid name is also created automatically if, while importing XML, you select **Take Name from Root Element** and the root element name is invalid.

Managing Data Models

The Models pane of TM Designer displays a list of the data models in a repository. Each model has an associated context menu providing access to functionality that lets you change or update characteristics of that model.

Models can be renamed, copied, deleted and have new versions created. This lets a user ensure that they have only the models needed with clear names and appropriate versions for their project. The context menu lets a user sort and view different items in their data models. For example the user can sort elements, attributes and relationships alphabetically and view inverse, sub and super relationships and terminal nodes.

The context menu also provides access to other useful functionality that a user may require. Examples are connecting and disconnecting from data stores and exporting data models for use in other repositories.

See Also: Generic Model Properties (p. 9)

Generic Model Properties

A feature of Transformation Manager is its ability to make a data model generic. Transformation Manager creates a data model using its loader functionality. Part of the process involves connecting to a data store so it can be translated into the required data model. The connection takes place using an appropriate adapter for that type of data store. The details of the adapter used are then stored with the data model.

When a project is created the user provides the information for a source and target data model and therefore the details of the adapter needed for each; this occurs in the New or Edit Project Configuration window. When the project is executed this information is used to prompt the user to provide the necessary information to connect the data model to an actual data store.

However, there are occasions when a user may wish or need to use a different adapter which is more specialised and relevant to connect to a specific data store. In this circumstance it is possible to use the Generic Model Properties... functionality to specify a different adapter to use.

Model Packs

Model packs are supplied direct from ETL Solutions. A Model Pack is a jar file containing a predefined set of models from which one or more models can be imported into a repository.

They are installed and made available by manually creating a model packs directory in one of the two locations below and placing the jar file in the directory.

- `[TM Home]/modelpacks`
- `[TM Installation directory]/modelpacks`

TM Designer must be restarted after a Model Pack is placed in either of these locations. If a Model Pack of the same name exists in both locations the one from the [TM Home] directory takes precedence.

When TM Designer is started details of available valid and invalid Model Packs are displayed in the About window which is available from the Help menu.

The models in a Model Pack are loaded into a repository by using the Model Packs window. The window is reached from the File > Load Model > Model Packs... menu item.

Using Pseudo-attributes

Pseudo-attributes are special attributes that provide attribute-like access to non-attribute data. For example, `$Content` represents an XML element's immediate text content and `$Name` represents the local name of the element. Pseudo-attributes are used in assignment statements to simplify the copying of data. For example, to reference all text content in an XML node and its sub-nodes the pseudo-attribute `$BigContent` can be used in a single assignment statement instead of writing several separate statements.

Data Stores Types

Introduction

As described in the data models (p. 2) section, Transformation Manager can be used with a variety of different data store types. Specific functionality has been provided for some of these data stores and this will help with any data integration or migration project.

See Also: Database (p. 11)
XML (p. 11)
Flat Files (p. 11)

Database Data Models

Transformation Manager has functionality specific to databases in general and some types of database specifically. This functionality can be very helpful in any project involving a database. When developing any relational database transformation, ensure that all databases are correctly installed according to the documentation supplied with the database.

The CLASSPATH environment variable should contain the JDBC driver required when using Transformation Manager.

A database schema contains tables, which contain columns. Some columns form the primary key constraint for the table. Other columns are used in foreign keys which reference other tables.

The schema equates to the Transformation Manager model. The tables in the schema become the elements of the data model. The columns in each table become attributes in the corresponding element. Primary key columns are indicated with a different icon in TM Designer, and by default take the role of the identification for that table.

Foreign key constraints become relationships from one element to another, as indicated by the constraint. The columns underlying a foreign key constraint form the relationship sub-condition. A sub-condition defines the rules by which the relationship is navigated or constructed.

See Also: Prefetch and Batching Settings

XML Data Models

An XML model can be loaded from a DTD, XML Schema or XML document. The elements and attributes defined in for example, a DTD, become elements and attributes in the data model. An XML document is hierarchical; in the DTD, you define amongst other things the parent/child relationships between elements. These relationships are imported into the data model.

For XML models, the source or target elements are specified using XPath.

Flat Files

The term Flat File refers to a text file that contains one record per line. Within such a record the single fields can be separated by delimiters, e.g. commas, or have a fixed length. TM Designer can load a data model from a flat file and create a flat file data store.

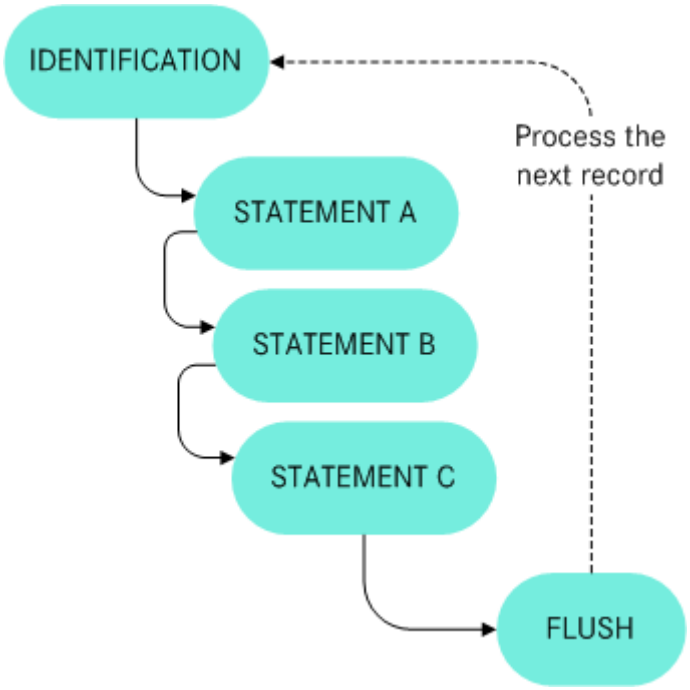
Debugging

Debugging is an integrated feature of TM Designer. It lets a user step through a given project testing and checking each step. The debugger suspends the running of a given line or part of a line of code just before each action is taken. At each suspension point the user can find out relevant information about the project state before that action occurs.

Several features are provided to support a user wishing to debug a project. These can be grouped into the debugging session, user feedback and debugging control.

The Debugging Process

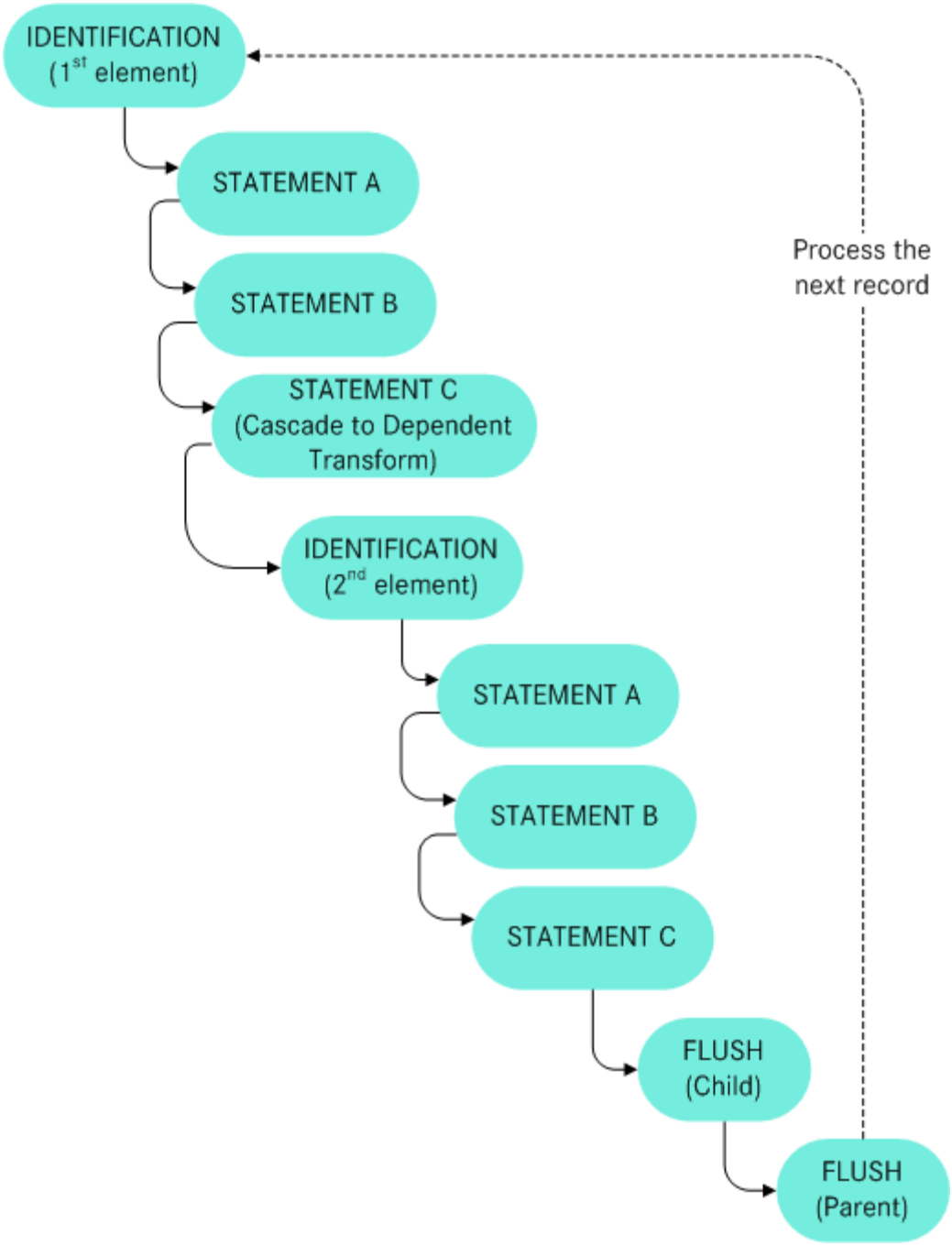
Debugging a project at its most simple level is relatively straightforward but can become very complex depending on the nature and extent of the project. In its simplest form, the process is described below.



The identification step sets up and identifies the key for a particular target element record. The exact nature of this step depends on the how the transform code is written but this is a useful starting point. This is then followed by various statements that are prepared and executed in order to build up the transformed record. When the record is fully prepared it is then flushed to the adapter which is being used by the project. The next record from the source is then processed in the same manner as the first and so on until all the source records have been transformed and handed to the adapter.

The suspension point on each statement may infer an individual line of code in a transform. However, that is not necessarily the case. An individual line of code may have more than one statement event but this will be shown in the Debugger Console.

The diagram below shows the process in a scenario where two elements are being transformed to a target using two transforms. The second transform is dependent on the first.



Debugging sessions

TM Designer lets a user invoke debugging when required. The debug session starts the process of stepping through each action of the project and its transforms. The user has access to common debugging functionality such as step over, step into and step out for any given action point.

It is possible to run more than one debug session at a time. These can be accessed from the Sessions pane.

Debug Feedback

TM Designer provides feedback using the following components, the Output pane and associated Debug and Debug Console features, the Variables pane, the Call Stack pane and the Breakpoints pane. Each has a specific purpose.

Debug Component	Description
Output Pane	In a debug session the output pane provides its normal function of informing the user of the success or not of the build process plus any appropriate warnings. In addition to this two further features are added. The Debug pane reports a summary of the project execution. This is the same as the output when launching a project in TM Migrator. The Debug Console displays a record of all the actions that have occurred while executing the project.
Variables Pane	This pane provides the state of the data at a given point in the debug process. It lets a user inspect the source and target data plus any variables. The user can ensure that the project is transforming the data between data stores as expected.
Call Stack Pane	This pane lets a user identify which transform the debug process is in at a given step in a project.
Breakpoints Pane	The Breakpoints pane provides an overview of where a user has placed their breakpoints in the project describing the type of breakpoint and its exact location including the transform and line number.
Sessions Pane	This pane lets a user inspect the currently running debug sessions and swap between sessions to view the related debug process.

Debug Events

The Debugger Console displays a log of the debug session. There are three main parts to the log. The first set of messages describe the current state of the session and includes the session started and finished messages and the project running and project stopped messages. The second set of messages are the event messages which are described in the table below. The third and last set of messages describe other features of the debug session such as breakpoints.

Event Type	Description

STATEMENT	A STATEMENT event corresponding to the highlighted line of transform code is about to be executed. This event type is typically an assignment. The transform code below assigns the value `7` to the attribute called `variable`. In this instance there is only one STATEMENT event for the line of code.
	`variable := 7;`
	However, some lines of code have many events. In the example below there may be more than one event depending on the rest of the transform and the relationship , `rel1`.
	`rel1.attr := 12;`
	It will have a STATEMENT event that assigns `12` to the `attr` attribute but it may also have a STATEMENT event that links up a relationship assigning keys on either side of the `rel1` relationship.
	Each of the events would suspend the debug session with the STATEMENT event.

IDENTIFICATION	An IDENTIFICATION event identifying the target element. This event may highlight more than one line of code. This is because the highlighted lines are those which could be involved in the identification. Typically they will be the lines in the transform which assign to the primary keys of the target. This could be through a relationship though, not necessarily an attribute assignment.
FLUSH	FLUSH is called at the end of a fully populated transform target record.

CONDITIONAL	A CONDITIONAL event occurs when the next step in a transform is about to evaluate a conditional expression. The example below uses an `if` statement.
	`if (x > 8) then`
	`  a := b;`
	`end_if;`
	A CONDITIONAL event will be generated for the line `if (x < 8) then` and the debug session will suspend at that line.

ERROR	This occurs when the debug process reaches an action in the process that causes an error to be generated.

Debugging control

Breakpoints let the user define control points in the flow of the debug process. They let the user suspend the execution of a project in a predefined way. TM Designer lets you define four types of breakpoint called Error, Identification, Line (Procedure) and Line (Transform). The breakpoint types are described below.

Breakpoint Type	Description
Error	This breakpoint lets a user suspend the debug session when an error occurs. A user can decide on the level of error to break on having three options available, FATAL, ERROR and NOTIFICATION.
Identification	This breakpoint lets a user suspend the debug session when an identification event for a specific data model and element occurs.
Line (Procedure)	This lets a user set a breakpoint within a procedure causing the debug session to stop on that line in the procedure.
Line (Transform)	This lets a user set a breakpoint on a line in a transform causing the debug session to stop on that line in the transform.

Error Handling

This section describes the behaviour of the Transformation Manager runtime environment when errors are thrown. Errors can occur at runtime for a variety of reasons. Below, there are some examples.

- The adapter configuration at start-up is illegal.
- A User Defined Function or Lookup may fail.
- An adapter may throw an exception, for example, when a database constraint is violated.
- Bad source data was encountered.

- An exception is forced from the SML using one of the built-in functions THROWERROR, NOTEERROR and FATALERROR.
- A system error occurs, for example failure to connect to a database.

SML conditional logic can be used to detect errors in source data. For example, an attribute called ATTRIBUTE_A of a source element called ELEMENT_A must be 12 to 14 characters long.

```
IF LENGTH(ATTRIBUTE_A) < 12 OR LENGTH(ATTRIBUTE_A) > 14 THEN
  PRINTLN('Invalid Value: ' & ATTRIBUTE_A);
ELSE
  CONSTRUCT;
  -- assignment statements ..
END_IF;
```

While identifying errors is useful it is not always sufficient and a user may want to write errors into a database table, an XML document, or a text file. If a project is moving data into a data store and the user wants to write errors to a different location then a cascade statement or other SML code cannot be used to write the error details to a different target data store. They can only be written to elements within the current target data store.

Transformation Manager provides Error Handling Projects which allow a project to write errors to any target supported by Transformation Manager. The Error Handling Project is created in the same way as any other project. The source model for an error handling project is the automatically created TMError data model which exists in every repository. The target model can be any which the user wishes. The TMError data model contains attributes which describe the error. Some attributes are set automatically by Transformation Manager and some can be set manually. In the example above, the user may want to save the invalid ATTRIBUTE_A value so that it can appear in an error report. Transformation Manager identifies or defines an error project as one where the source model is TMError.

An alternative to using an error handling project is to write an external function and pass the error details to that. The disadvantage is that the user must switch to low level Java to implement error handling – connecting to a database and issuing SQL insert statements, or creating an XML DOM and serialising it.

This section assumes that users are familiar with the design-time aspects of error handling which are listed below.

- TMError.
- CATCH_BLOCKS.
- Error handling projects.
- SETSTOPONERROR.
- The functions NOTEERROR, THROWERROR, FATALERROR.

The behaviour described in this section is that of the default when running in short lived mode.

Note that if an internal error occurs in the code this is treated differently. The system will exit immediately with an appropriate error message.

Execution Order

SML (p. 59) is a semi-declarative language. This means it is generally not possible to specify or determine the order in which transforms will be executed, with the Code Generator and Execution System deciding how to perform transformations described in SML.

SML contains some procedural constructs, such as variables, `IF`, `CASE`, `FOR_EACH`, and so on. These can be used when the order of statements within a transform is significant. Transformation Manager may not always be able to guarantee correct results from such use. For example, re-use of a local variable may produce unexpected results.

SML supports three distinct transform styles. The actual transform solution used for a specific project depends upon the type of transform problem to be solved and the structure of the models involved. For example a stack of dependent transforms are generally convenient where the target requires order as in an XML document but in some cases a few independent transforms will do the work of the stack with much less effort.

See Also: Transform Styles or Approaches (p. 37)
Priority (p. 40)
Relationship Assignments (Cascades) (p. 66)

Lookups

Lookups provide a simple method of extracting values held in an external database, or in the source or target data models of the current project, and entering those values into a transform. Lookups are created and stored on a project basis. Lookups are used in the Editor pane either by typing the Lookup name directly or using the code completion window invoked using the Ctrl+space key combination and selecting the lookup from the list. Lookups can be called in the same way as a built-in function or procedure. The project context menu provides access to the New Lookup..., Edit Lookup... and Delete Lookup... functions. A lookup function returns a single value. A lookup value can be searched for based on any number of attributes in the relevant table.

They are a way to read data from a data source that is not the same as the source model of the project. They can be used to let a project read reference data keys from a target database. In the second PPDM tutorial a lookup is used to read the primary key for the foot length measurement from the unit of measure table, PPDM_UNIT_OF_MEASURE. This is saved in a global variable so that the unit of measurement for each length attribute set can quickly be referenced.

Monitoring the affect of a lookup and the lookup cache when a given project is executed is supported using Error Handling projects and a built-in function called ENABLEADVANCEDSQLSTATS(). A user can use an error handling project to capture any errors that occur during the execution. These are managed using the functionality enabled within the error project and the CATCH_ERROR block. The built-in function, ENABLEADVANCEDSQLSTATS(), lets a user gather additional statistics in the output displayed in the Output pane of both TM Designer and TM Migrator.

Valid Names

Naming a Lookup has certain restrictions which are listed below.

- The name must start with a letter and contain only alphanumeric characters.
- The name must not be a current Lookup name.

- The name must not be a current user-defined function name.
- The name must not be a built-in function name.

Lookup Components

Each lookup in a project is made up of several sections of information as a part of its implementation. The sections below relate to the individual panes in the **New Lookup** and **Edit Lookup** wizards and describe relevant information relating to each.

Connection Details

This pane is only available when an External Database is being used for the Lookup source. So, in this circumstance a connection is required for the external database source. Each connection alias has an associated JNDI connection name and each JNDI connection name can be used by more than one connection alias.

The connection alias refers to a logical database connection that will be used by the lookup. At run-time, a database connection for the alias will be requested by this name. The first use will obtain a physical connection, and subsequent requests for a connection of that name will most likely return the same physical connection. Therefore, the same connection alias name cannot be reused at run-time to refer to a different database.

Typically, when using the lookup wizard, you connect to the database to read the meta-data to assist in creating the lookup.

The 

button opens the Manage Connection Aliases window which lets a user create, update and remove connection aliases.

Table and Entity

This step in the creation of a lookup will list the tables in the database that a user has connected to. This only occurs if the **Connect Now (this enables the wizard to read the database meta-data)** tick box has been selected. If this has not been selected then the Table and Column must be typed into their respective fields.

Default Value

A user can specify default values to be used in their lookup in the event that a value cannot be found. The default can be an empty string. A user can specify that the default value is returned in the event of a lookup generating an exception - for example, if there is a problem connecting to an external database.

Conditions

The Conditions pane lets a user add conditional statements which become the 'WHERE' clause in the SQL that the Lookup executes. A conditional statement consists of a column name and either a parameter or constant. The conditions can be added and removed from the list as required using 

and 

buttons.

To pass null values into the WHERE clause, select the radio button field below the parameter radio button field and leave the value blank. This results in generated SQL containing a null, e.g. WHERE 'countryCode' = null.

Advanced

The Advanced pane has two modes depending on whether the lookup uses an external database or one of the source or target data stores. If the lookup uses an external database then the user can if necessary edit the SQL that is used to access the lookup value. Care must however be taken to not edit any parameters that have been added to the SQL via the Conditions pane. The user can define the quote identifier for the database in this mode as well. If the lookup uses the source or target data store then the user can only decide whether to use cached values or not by placing a tick in the **Build Cache of Values** tick box. This tick box is available for an external database as well.

Gathering Lookup Statistics

A user can obtain statistics about the use of a lookup using the ENABLEADVANCEDSQLSTATS() function. This will provide details of the use of a lookup in the Output pane of both TM Designer and TM Migrator. The function takes a number of parameters.

Basic statistics regarding the number of lookups will be sent to the output when the following code is added to a transform.

```
ENABLEADVANCEDSQLSTATS(true);
```

The basic output will be similar to the output shown below.

```
JDBC Lookups Total Usage:    1
    Source: 1
    Target: 0
    External JDBC: 0
    External JDBC (Hand Edited SQL Target): 0
```

More detail can be requested using the following code.

```
ENABLEADVANCEDSQLSTATS(true,true);
```

This more explicit output will be similar to the output shown below.

```
Named Lookups:
    otherFromPer:    total = 1, cached = 0, successful = 1, found null = 0
```

The output gives more detailed information about each defined lookup:

- `total` is the total number of times the lookup was made.
- `cached` gives the number of those lookups that were satisfied with cached data.
- `successful` gives the number of lookups that were not in the cache but found their data by doing the actual lookup.
- `found null` gives the number of lookups that were not in the cache and found no data with the actual lookup.

Models, Transforms and SML

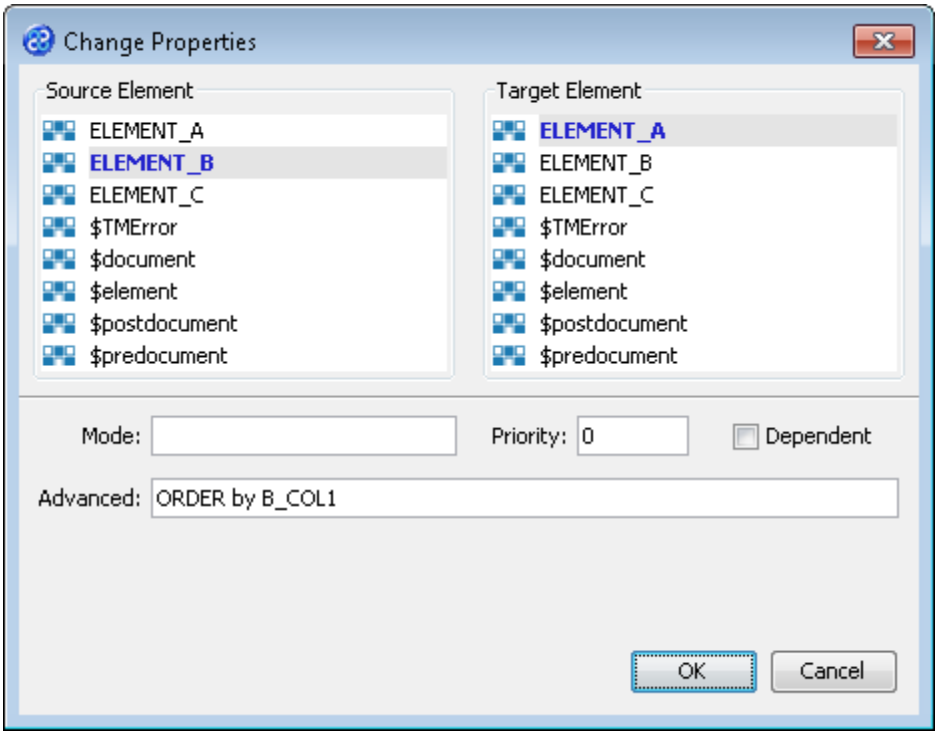
TM Designer provides functionality for managing the transformation or conversion between data models, SML and transforms all of which are managed within projects. Source and target data models

are not part of SML although the meaning of the SML code is partly determined by the models. For the purpose of illustration, SML code may refer to an attribute, a one-to-one relationship or a one-to-many relationship and can describe whether an attribute or relationship plays a role in instance identification. Less obvious is TM Designers ability to enhance data models to achieve functionality which is difficult to write directly in SML. This can improve the clarity and meaning of transform code. The topics below describe three useful approaches available to a user.

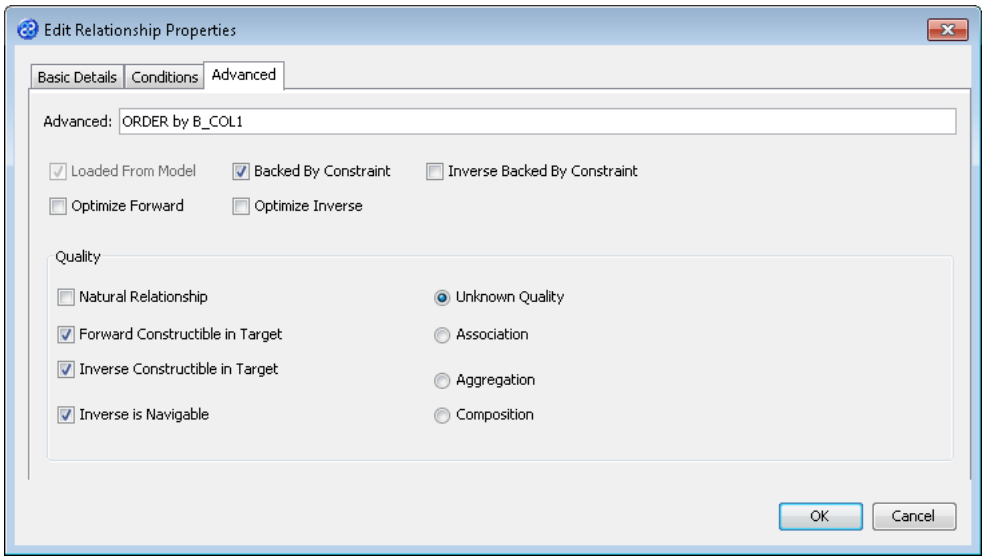
Ordering

The order in which instances are processed is normally determined by the data store itself. Flat files and XML documents are generally processed in the order in which they appear in the data store document and this is the case for Transformation Managers built-in adapters; for a relational database data store it is determined by the data store itself.

For a relational database data model there may be occasions when a specific task requires a specific order, for example an ascending order based on some attribute value. If the transform is independent an ORDER BY clause can be added to the **Advanced** property of the transform as in the image below. This does not apply to any other type of data store.



This approach can be extended to ordering a dependent transform as a result of a cascade. A relationship has properties and one of these is also the **Advanced** property. This property provides the same functionality for a relationship as it does for an element in a transform. The image below illustrates the same ORDER BY clause that was applied to the transform above but used in a relationship. The same restrictions on usage apply.

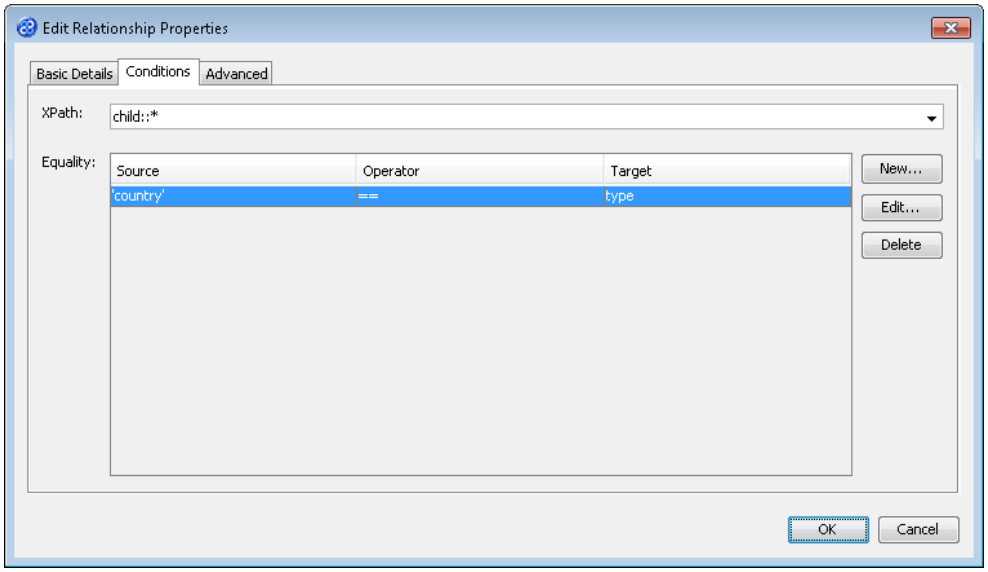


A point to consider is whether to modify an existing relationship or create a copy of the relationship and modify the copy giving it a different name. The latter option has the advantage that you are not modifying relationships loaded from the original model. However, if that original relationship is not useful or relevant in its current form then it may be advantageous to change the existing one.

Filtering

The Advanced property for both a transform and a relationship can also accept a WHERE clause with the same restriction on usage i.e. it only applies to a relational database data store. An alternative solution is to use a relationship condition which has the advantage of working with all types of data store.

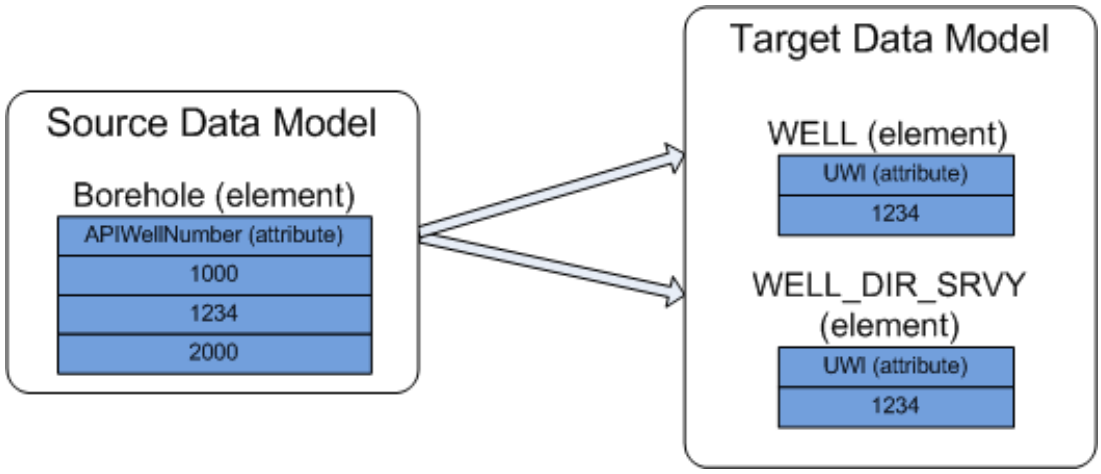
Conditions filter element instances for those which are matched by a relationship. For example, if there is a relationship to many AREA instances of varying types but country instances need processing, then a user can clone the existing relationship, name it country and add a condition to filter the required instances. The image below shows that condition as entered in the Edit Relationship Properties window.



Filtering can also be implemented using an IF statement and the CONSTRUCT keyword.

self Relationship

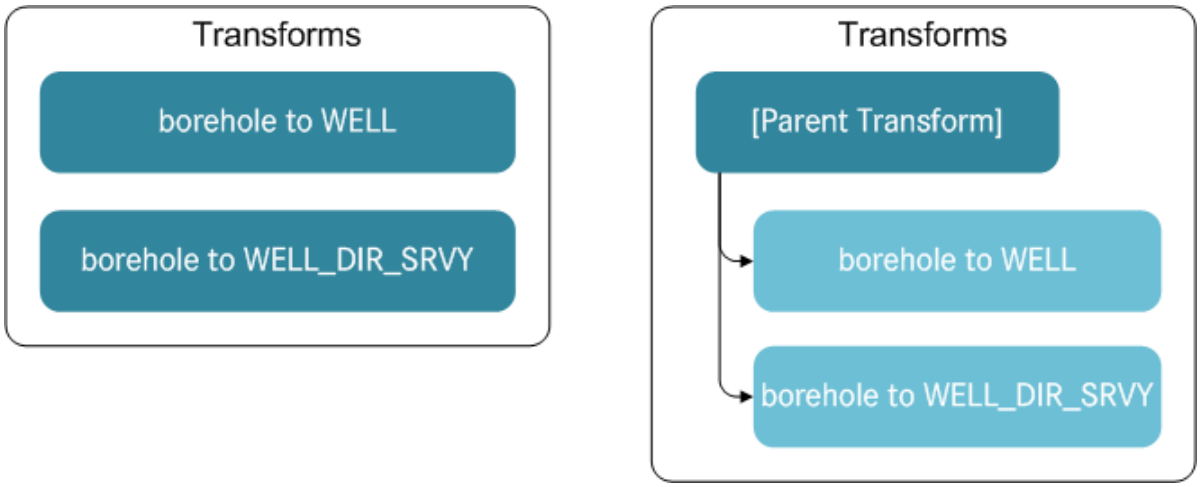
A frequent requirement is to populate two or more target elements from a single element in the source. The following example is based on moving information in one element and splitting it into two elements using a self relationship. The example takes a borehole set of data stored in the **borehole** element and splits this into two elements called **WELL** and **WELL_DIR_SRVY**. The data is presumed to be structured on the basis of one WELL instance having a one to many relationship with WELL_DIR_SRVY so that any WELL may have any number of related WELL_DIR_SRVY instances. So, the **borehole** element holds both well and directional survey data which must be split into the **WELL** and **WELL_DIR_SRVY** elements of the data model as shown in the diagram below.



The assumption made at this point is that separate rules will be used to create the WELL and WELL_DIR_SRVY instances and therefore two transforms will be required to implement the solution. The transforms are listed below.

- **borehole to WELL**
- **borehole to WELL_DIR_SRVY**

An initial approach puts the transforms at the same level in the cascade structure, either as two independent transforms, or called from the same parent transform as shown in the diagram below.



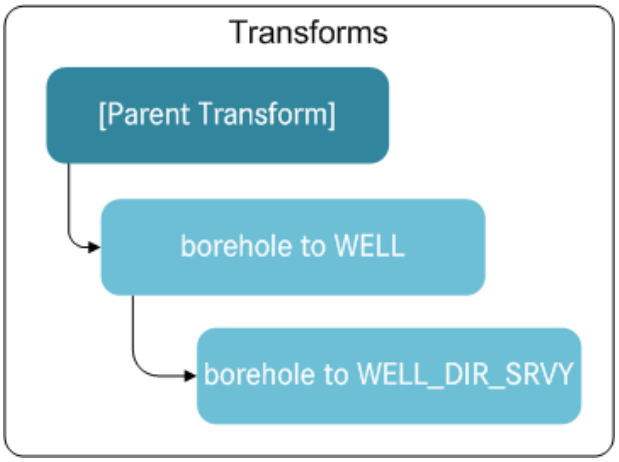
If a parent transform is used then each transform must be cascaded to and the calls to each will look like the following code in the parent transform.

```
WELL := borehole;
WELL_DIR_SRVY := borehole;
```

This means that the borehole element will be processed first by the **borehole to WELL** transform and then by the **borehole to WELL_DIR_SRVY** transform. So, **WELL** instances will be created first followed by the **WELL_DIR_SRVY** instances. In this case, the approach works reasonably well as long as the UWI attribute is correctly set for both WELL and WELL_DIR_SRVY instances when the relationship between WELL and WELL_DIR_SRVY will be correctly set up.

A potential problem with this approach however is that all the boreholes in the WELL transform may not be processed perhaps because some instances don't match all the required business rules. The solution to this problem would be to apply the same logic in the second transform ensuring that WELL_DIR_SRVY instances for which there is potentially no matching WELL are not created.

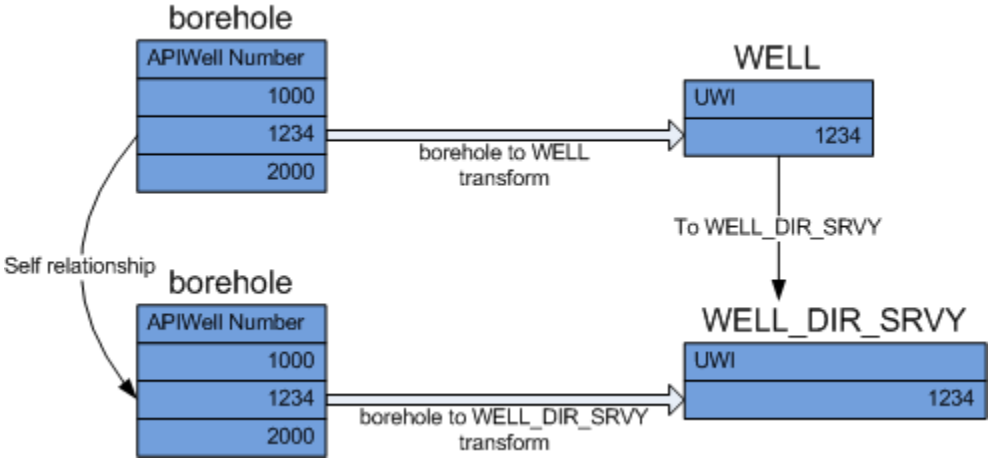
A second approach to this would be to implement the following transform structure which steps through the transforms only creating WELL_DIR_SRVY instances when a WELL instance exists.



This is achieved by adding a self relationship to the borehole element in the source data model. The **borehole to WELL** transform will now be able to cascade to itself with the following transform code where self is using the inverse relationship for WELL_DIR_SRVY.

```
iWELL_DIR_SRVY := self;
```

When the project is executed the self relationship lets the process leave the read position of the source data at that point. The target context can then change using the read position and start populating the **WELL_DIR_SRVY** element as in the diagram below where the borehole instance **1234** is processed to our two elements in the target called **WELL** and **WELL_DIR_SRVY**.



The self relationship can be used in the target data model and allows the population of one target element from several source elements using cascades. The self relationships must be added to elements in the target data model manually in the same way as they are for the source data model.

As above, the following code leaves the target write position it is in and changes the context from where it is reading from to whatever element instance is found by navigating the `sourceRelationship` and then calls the cascade transform.

```
self := sourceRelationship;
```

The self relationship can only be a one-to-one relationship as it relates the current instance to itself. This means that `sourceRelationship` must also have a maximum cardinality of one or a specific instance must be referred to using an index.

Procedures

Introduction

Procedures are a way to re-use commonly required code. Many Transformation Manager projects need to reuse the same logic in different transforms. For example, evaluating a complex equation or implementing a standardised mapping. Procedures are designed to help with this code reuse.

A Procedure contains SML statements, can be passed parameters and can optionally return a value. A Procedure can be associated with a source data model, a target data model, both of these or neither. For a given repository, any procedure can be used by all the projects in that repository.

Procedures are invoked using an identical syntax to built-in functions. They do not need to be built as this occurs when a calling project is built.

Source and Target Element

The New Procedure dialogue by default assumes that both source and target data models will be required for a procedure. The dialogue lets a user select the data model and element required for the procedure. However, if these are not required the **No Source** and **No Target** tick boxes let a user flag the procedure as not using a source data model, a target data model or either data models.

The source or target element for a procedure is defined by the source or target model combo box and their associated element list pane. The models shown in the model combo box reflect the data models available in the repository. The choices for the source element are the elements from the chosen source model. This also applies to the target model.

If a procedure does have a source element specified, it can only be called from a transform which uses the same source element, or one where the source element attached to the procedure can act as a substitute. This means that if the original source element for the procedure has the attributes COLA_1, A_KEY and a relationship toB, then the source element of the calling transform must also have them, and they must be compatible types. The calling transform can have more attributes and relationships, which are not accessible in the procedure, so it need not be identical, just compatible. This also applies if the procedure has a target element specified.

If a procedure does not need to access a source or target data model then there is no reason to specify either. For example a procedure may be used to display a message and so requires neither a target or source data model so there is no need to select either data model or element.

Parameters

Procedures can use parameters which provide a user with the ability to apply a parameter to a procedure for example limiting the processing of instances in a given source data store. Any valid SML type is acceptable.

Return Type

The return type can be specified in the Type drop down list. It can be any valid SML type.

Procedure Global Variables

Procedures can use global variables. The global variables that are referenced within a procedure can be forward-declared in the procedure definition. This ensures that the procedure parses successfully.

Valid SML in procedures

Within a procedure most SML code is valid. There are some exceptions however.

1) A procedure cannot call a transform using a relationship to relationship assignment at runtime. If this happens an error will be generated.
2) A procedure cannot identify a source or target element. These will be identified by the calling transform before the procedure code is called. For example, if a procedure has a target element specified then the target of the calling transform must have been already constructed before the procedure is invoked.
3) User defined functions (UDFs) cannot be scoped at a procedure level. If a UDF is to be referenced from a procedure it must have repository level scope.

Procedures can call other procedures.

Returning Values

The RETURN() function should be used to return a value from a procedure.

See Also: Calling Procedures (p. 25)
Matching Source and Target Elements (p. 26)
Procedures Tutorial (p. 24)

Calling Procedures

Procedures can be called in the same way as external or internal functions. So, if you have a procedure called `convertToKg` which takes a decimal and returns a decimal you can call it in the transform as shown below.

```
mass := convertToKg(12.0);
```

In the example below, the first procedure called `LogTransform` uses one parameter and does not return a value. The procedure called `GetNextSequence` takes one parameter and does return a value to the attribute called ATTRIBUTE_KEY in the target.

```
LogTransform('A_KEY = ' & ATTRIBUTE_KEY);
ATTRIBUTE_KEY := GetNextSequence('MY_ORACLE_SEQ');
```

Source and Target Instances

If a procedure refers to a source or target element then the default situation is to use instances of the source element, target element or both for the calling transform. If the transform is from AUTHOR to WRITER then the current instance of AUTHOR will be used as the source of the procedure and the current instance of WRITER will be used as the target.

When a user builds a project the definition of the procedure is checked to ensure the transform's elements are compatible with those required by the procedure. See below for an explanation of how the compatibility is determined.

However, you do not need to use the default instances. You can call a procedure with an instance of a compatible element that is identified via a relationship from the calling transform. The following syntax is used for this.

```
transferAddress|targetRelationship, sourceRelationship|();
```

In this call the source instance is determined by the relationship given as `sourceRelationship`. This relationship must define a single identified instance in the source. Similarly, the `targetRelationship` defines a single identified instance in the target. Any valid relationship path can be used.

Neither path is required in the call. The following examples are also valid and will result in the default instance being used in place of the missing definition.

```
transferAddress|,sourceRelationship|();
transferAddress|targetRelationship,|();
```

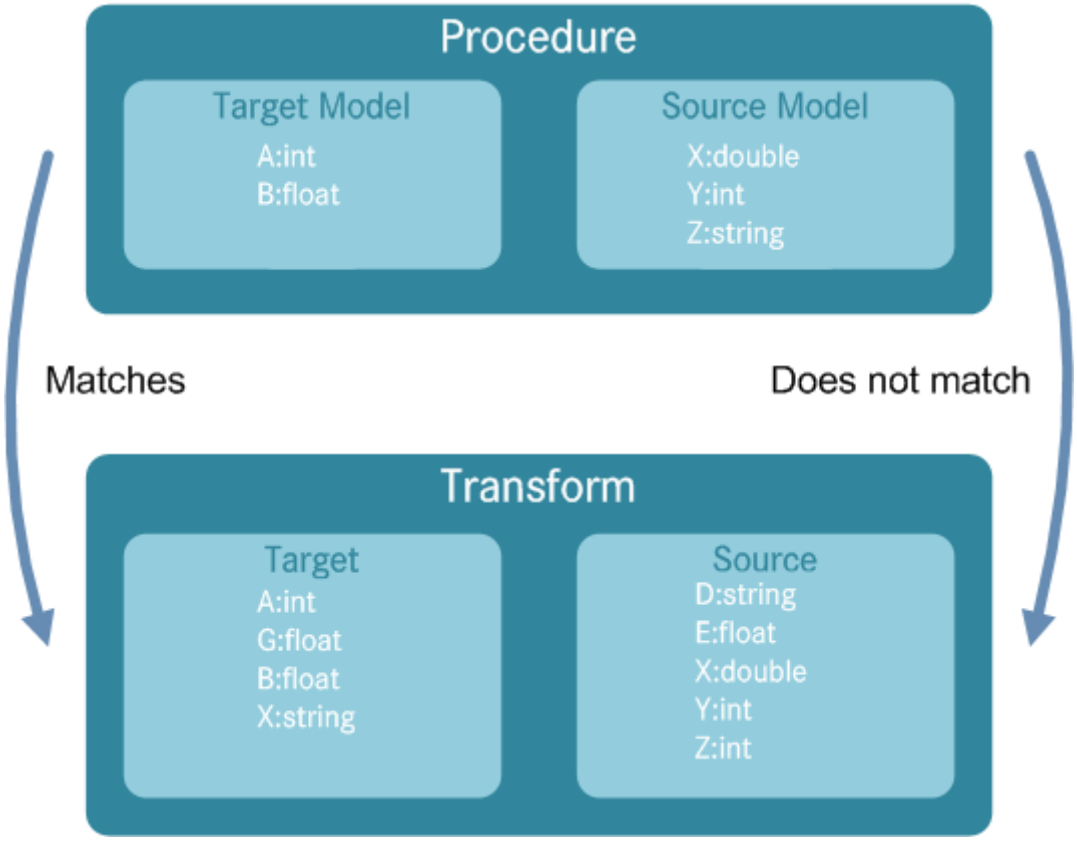
This approach means a procedure can be used in any appropriate transform where it may be required without the need to create a cascade transform in order to get the result. An appropriate transform is one that follows the rules for matching elements correctly.

Matching Source and Target Elements

Procedures know the elements they use as a source and as a target. When a procedure is called from a transform, the elements given to the procedure must be compatible with those the procedure is expecting. The elements will often be the same, but do not have to be.

A called element is considered to be compatible with a procedures element if it contains all attributes in the procedures element and they have the same types. Any relationships in the procedures element must also be present in the called element and the elements referenced by the relationships must be similarly compatible.

The diagram below shows how elements do or do not match. Notice that on the target side the procedure requires attributes `A` and `B` which are of type `int` and `float` respectively. Checking the target transform shows that the target element has both of these attributes and the type for each is the same. However, the source side has a problem. The attribute `Z` in the procedure source model is of type `string` but in the transform is of type `int`.



Generally elements can be re-used in the main model. However, elements can be used from any model in the repository. Elements used to define a procedure source and target are used only to assign a type to the source and target. They do not allow you to access any data sources other than those used by the calling transform. This means that you can create a new element which represents a "supertype" of element in your model and contains all of the common fields which the procedure will operate upon. This element can then be used as a source or target for a procedure.

Examples of usage

Simple calculation

As part of updating a legacy database a conversion routine is required to convert from inches to cm. This conversion can be encoded in a simple procedure with no source or target. The procedure, called `convertInchesToCm`, would take a single parameter, `inches`, of type `double` and has a return type of `double`. The code is shown below.

```
return(inches * 2.54);
```

An example of the procedure being called is shown below.

```
length := convertInchesToCm(length);
```

Target Update

Here is the scenario, a new parts system comes with a poorly normalised database which contains several tables for different types of parts, one for tools, one for airframe components, and another for miscellaneous items. Each of these tables contains a repeated set of identifying columns i.e name, stock number, description and key. The key is generated by concatenating the name and the stock

number. Since the code to write this part of each table is identical it should be performed by a procedure.

In this case each table is represented by a different element in the model, so it will be necessary to create a smaller element which contains the common columns of the tables. This element, called `PartsDescriptor`, will not relate to any actual database table but acts like a superclass of the different parts tables.

The procedure, called `writePartsDescriptor`, can be defined to use the new target element and takes the parameters `theName`, `theStockNumber` and `theDescription`. It has no return value.

The code is shown below.

```
name := theName;
<stock number> := theStockNumber;
description := theDescription;
key := name & <stock number>;
```

The procedure can then be called in each of the different transforms as follows.

```
writePartsDescriptor(name, <stock number>, description);
```

Automatic Generation of Assignments - Procedures

The editor pane used for creating a procedure provides access to the **Auto generation of assignments** window as used in the transform editor. This lets a user quickly create assignments between a source and target element by matching attributes (and specifically terminal nodes in XML data models) to each other. The window provides three matching strategies to help match attributes correctly and these are **Exact match**, **Case insensitive** and **Regular expression**. It is only available in the procedure editor if a source and target element are defined for the procedure.

The example below shows the result of using this functionality for the **BOOK to BookList** transform in tutorial 1. The matching strategy used here is **Exact match**.

```
-- NO MATCH FOUND FOR CURRENCY
-- NO MATCH FOUND FOR <ID>
-- NO MATCH FOUND FOR PRICE
-- NO MATCH FOUND FOR EDITOR_ID
-- NO MATCH FOUND FOR IS_FICTION
-- NO MATCH FOUND FOR PUBLISHED_DATE
-- NO MATCH FOUND FOR TITLE
```

Below is the result of the auto generation process using the **Case insensitive** matching strategy. In this example there are four attributes that have matched now that the matching strategy is case insensitive.

```
currency := CURRENCY;
<id> := <ID>;
price := PRICE;
-- NO MATCH FOUND FOR EDITOR_ID
-- NO MATCH FOUND FOR IS_FICTION
-- NO MATCH FOUND FOR PUBLISHED_DATE
title := TITLE;
```

A matching strategy can also make use of regular expressions in order to disregard common parts of attribute names. For example, a data store may prefix attribute names with a common term such as PERSON in a customer relationship management environment or WELL in an oil and gas enterprise. The user can then use a regular expression to limit the matching strategy to just that part of the attribute name required, for example, just NAME from the PERSON_NAME attribute and TITLE from

the PERSON_TITLE attribute. In a PPDM project this could be used to disregard a prefix such as WELL_ from the matching strategy.

See Also: Auto generation of assignments window

Inlining Procedures

Transformation Manager provides the option to inline procedures. Inline procedures have their code incorporated into the calling transform as though it was written there in the first place. This gives the user two benefits. The first is a performance benefit. The second benefit is that inline procedures can be used in identification which is not allowed if the procedure is not inlined. However limitations do apply which may prevent a function from being inlined.

TM Designer will attempt to place all procedure calls inline but some are not suited. This applies to procedure calls where there is a specified source and or target. For example a procedure such as this using the following `proc()¦rel1,rel2¦`.

Some procedures cannot be inlined at all.

1) If the function contains a CATCH_BLOCK section of code.
2) If the function contains any of the following functions.
   a) GETCALLITERATION()
   b) GETCALLSTACKSOURCE()
   c) GETCALLSTACKTARGET()
   d) GETCASCADENAME()
   e) GETMAPLEVEL()

If the **Generate for Debug** setting is ticked in either the Repository Settings... or Project Settings window, as may be the case when using the **Debug** context menu item for a project, then procedures will not be inlined, irrespective of the inline setting.

See Also: Repository Settings... window
Project Settings window

Projects

Introduction

A Project is the complete specification of a data transformation or migration process using SML. It transforms data from the source model to the target model. A project comprises of references to the source and target models, plus one or more transforms, and may include global declarations and Java functions. All projects should include at least one independent transform that processes data. A project can only reference data models in its own repository. Projects are stored in the Transformation Manager Repository.

A project can have an associated error handling project. The error handling project is used to process errors that occur during the execution of a project. This functionality lets a user write errors to any data store supported by Transformation Manager. Error handling projects are similar in

characteristics to a standard project allowing a user to write specific error handling code in the Editor pane.

Valid Project Names

Projects must have valid names. The project name must start and end with a letter and contain only alphanumeric characters.

See also: Types of Project (p. 30)
Building Projects (p. 31)
Running Projects (p. 31)
How do I open a project? (p. 32)

Types of Project

There are two types of Project.

Primary Projects

Primary Project is the term used to describe the majority of projects defined in Transformation Manager. A primary project includes all the transforms necessary to achieve the transformation of a single source to a single target.

Error Handling Projects

A project can include any number of error handlers. The source of an error handling project is always the standard pre-loaded model called $TMError. The target can be any target supported by Transformation Manager (e.g. XML, Java, RDBMS). All properties of the transformation system are available in an error handler (including the ability to catch and transform an error in the error handler itself).

An error handling project contains independent transformations. Each error handling project is expected to include one or more independent transforms defined from $TMError to items chosen in the target. All error handling projects included in the main project will be invoked in response to any error occurring in the main project. Errors may also occur inside the error handling projects. If this happens then the error is only handled by the other defined error handling projects. If no other error handling projects are available then the error will be regarded as unhandled and appear on a standard log output. User errors are also referred to as forced errors.

Notes

A single error handling project may be used by more than one primary project where projects do not include:

- NOTEERROR, THROWERROR and FATALERROR
- CATCH_ERROR
- Error Handling Projects

Text errors are directed to StdOut and StdErr. Where a CATCH_ERROR block and Error Handlers are included, the error is first processed by the CATCH_ERROR statement and then by the error handler(s).

Building Projects

A project needs to be translated into executable Java programs before the transformation can take place. This is achieved by building the project. Building a project can be executed using the application menu option Run > Build or using the context menu available from the project title in the **Projects** pane. When the project builds successfully a project jar is created.

The Output pane displays messages that indicate whether program generation has completed successfully. It is during this phase that all of the transformation statements in the project are checked for syntactic correctness, and transforms are checked for dependencies.

Transformation Manager is capable of detecting many different kinds of errors, and it displays relevant messages in the Output pane. Some kinds of errors will prevent program generation from starting, whereas others may only cause a warning message to be displayed. Depending on the messages displayed, you may need or want to edit the transforms before you can run the project.

For example, an assignment such as

```
<ID> := $UNIQUEKEY;
```

will often cause a warning like this to be displayed:

```
Insufficient identity may have been set ... A new row will be created for every
input
Some configurations and database systems may not allow insertion of these rows
```

If you *do* want a new row in the target database for every row in the source database, then the warning can be ignored (the generated programs will run). However, if you really want only one new row then you will have to edit the statement so that the destination becomes identifying:

```
*<ID> := $UNIQUEKEY;
```

Repeat the Build operation and then Run (p. 31) the transformation.

Notes

Transform Projects can be built programmatically using a command line interface.

Running Projects

TM Designer can only execute a project after it has been built and has generated a jar file for the project. A project can be run in four distinct ways. These are from the menu bar, the project context menu, from TM Migrator and using the Run Transform tool. If a project is executed using the menu bar or project context menu options then TM Designer will automatically build the project as a part of the process. The first time you launch TM Migrator for a project that has not been built it will open but the project will not be available as the jar file does not exist. If the project has been built and changed but not re-built then TM Migrator may either run and display errors or just display errors. The Run Transform tool will behave in the same manner as TM Migrator. In TM Designer a project is automatically built on the first occasion that the project is run or debugged. Subsequently, a project will only be rebuilt if it has been changed within TM Designer.

How do I open a Project?

Projects are listed in the Projects pane. All projects in a given repository are available from this location. If the projects pane is not open then this can be opened for display in the following way.

1) Click on the Window menu bar option.
2) Click on the Projects option. This will display the Projects pane.

The same action can be invoked using the keyboard. The key stroke is Ctrl 1.

Relationships

Introduction

Relationships relate an instance of an origin element to one or more instances of a destination element described in a forward or inverse direction. Relationships have many associated properties including the cardinality between the origin and destination and whether a relationship is optional or mandatory, the optionality. The existence of a destination element can be optional as it may be created when a project is executed. Relationships have a name and a specific set of properties that determine their meaning. These can be positional (an XPath defines the route from source to destination), rely on value equivalence (a combination of values in the source has the same combination of values in the destination), or be intrinsic (i.e. in Java object models). Relationships are part of an element in a data model and often have the same name as the element but a relationship is always referred to by name when writing data integration logic. It is the properties of the relationship that define its runtime behaviour. Relationships can be created by the user manually or by copying existing ones.

Transformation Manager uses relationships to reach out to other elements in a data model so it can construct appropriate relationships and data instances. This is a requirement for an appropriately constructed XML document and is a more useful approach for databases although a database target could be populated table by table if foreign key values were set correctly and in the right order.

Transformation Managers runtime mechanism navigates and constructs relationships correctly. If you use relationships on the target, Transformation Manager will ensure they are correctly set up. For example, referenced instances are inserted before they are referenced; the element instance containing the primary key is inserted before the related foreign key.

Forward and Inverse Relationships

A relationship between two elements can be viewed from both a forward and inverse perspective. A forward relationship defines the relationship derived from the origin element to the destination element in the data model, whereas an inverse relationship defines the relationship derived from the destination element to the origin element. For example, in XML, a forward relationship might exist from the parent to the child element, whereas an inverse relationship would exist from the child element back to the parent element. Similarly, in a relational database, a forward relationship might exist from a Foreign Key in one table to a Primary Key in another table, and an inverse relationship from the Primary Key back to the Foreign Key in the first table. TM Designer by default displays forward relationships.

In a PPDM data model forward relationships start at the foreign key element and point towards the element with the primary key. All elements have a forward relationship to R_SOURCE, for example, because each table has a foreign key constraint which references R_SOURCE. The maximum cardinality of the forward relationship is 1 because the primary key is unique.

A second example is the forward relationship between WELL and WELL_DIR_SRVY which uses a foreign key constraint in WELL_DIR_SRVY. Looking in the WELL_DIR_SRVY element there will be a relationship called WELL which refers to the WELL element. The inverse view of the WELL relationship in the element WELL_DIR_SRVY has a maximum cardinality of many (*). So the element instances at the end of the inverse relationship are the collection of directional survey headers belonging to the current WELL.

Transformation Manager allows the definition of user created relationships as part of a data model. In these circumstances the user can define the origin and destination elements. These may reflect the forward or inverse relationships between the parent or child element depending on the purpose of the transform.

Types of Relationship

The characteristics of a relationship are vital to how transformations are carried out. The types of relationships are:

- positional: one set of instances is related to another by the relative position of the elements (for example, child, parent, and sibling relationships). This is relevant to hierarchical models such as XML. In XML models positional relationships are specified using Xpath.
- derived: the relationship is by attribute value. Derived relationships occur in both XML and relational data models.
- hybrid: a combination of positional and derived relationships (for example, the child element whose name attribute is "Anna").

Transformation Manager automatically creates parent-child relationships when importing XML models. Foreign key/primary key constraints in relational models are also loaded automatically as derived relationships. Other types of relationships can be added manually.

Relationship Properties

The properties of a relationship include its cardinality and advanced clauses. The minimum cardinality defines whether the relationship is optional – 0 means optional, 1 means mandatory. The maximum cardinality specifies how many instances the relationship can point to – 0, 1 or many.

When constructing an instance of an element, it is likely that all the mandatory relationships must be set up to produce a valid target. The maximum cardinality can influence how transforms are written; if it is one, then the transform will reach at most one row in the destination element, so attributes can be directly read from that row. If it is many, it refers potentially to a collection of more than 1 instance of the destination element. All the properties of a relationship are editable.

See Also: Self Relationship (p. 33)

Self Relationship

All systems (RDBMS, XML and Java) allow the introduction of relationships based on self. The value of the Self relationship is that it can be used to modularise transforms and encourage code re-use. If you find yourself trying to transform from `THIS` or to `THAT` then the Self relationship should be considered.

Notes

- Self is a reference to the current instance in the source or target data store as though it were another instance. ($instance is not; it is a reference to the current transform handle and should only be used in the triangular problem.)
- Self uses the key word `self::*` as the relationship XPath even in non-XML systems.
- To add a self relationship, right click on an element in the source or target pane and select the New Self Relationship option from the context menu.
- The Self relationship is available on both target and source sides and when added to an element is identified in the data model with the name **self**.
- There are no restrictions on the use of Self on the source side.
- Using self on the target side requires care. This is because Transformation Manager will construct a transform target in the first part of the transform and then use the navigation by the self-relationship to revisit that target node and add further properties. Nevertheless this feature is invaluable in writing some types of transforms with a modular approach.

See Also: Using the Self Relationship in SML (p. 75)

Parameterised Relationships

Relationships can be managed using conditions and a condition can include parameters. A parameter or parameters let a user manage the data that is evaluated and transformed to the target. For example a user may wish to only transform and load a subset of data from a given element and its related elements. In this case parameters could be used to limit the records transformed for example to only those WELLS of a specific type or types.

Parameterised relationships can be checked using the Navigate... context menu option in the View Data pane. All user entered values are entered as strings and converted to the required type when the sub-condition is evaluated.

See Also: Parameter values window
Select Relationship window
View Data Context Menu
Tutorial 10 - Adding and Parameterising Relationships

Repository

Introduction

The repository holds all the information about data models and the logic of the transforms you write. It includes source and target data models, projects, transforms, local and global variables and

User Defined Functions. It is file based and allows the files to be stored in a version control system and there is no limit on the number of repositories that can be created. An instance of TM Designer can connect to only one repository at a time.

The structure of a repository is important because this will help decide how to organise projects. Each project can only reference one source and one target model and can only reference models in its repository although you can have many data models in the repository.

Data models are added to the repository using load processes. Each load process creates a copy of the metadata for that data store. No link is maintained to the original source of the model and changes made in one repository are not reflected in other repositories or in the original source. For example, if you have two repositories, A and B, containing data model version 1 and changes are made in TM Designer to one of those models in repository A then those changes are not applied to the original source of the data model or to the model in repository B.

A repository is created in the Repositories pane in the main TM Designer application interface and is identified by its connection name. Physically it will be located in the users `\TM\repositories\MyRepository`, which is the concatenation of the Parent Directory and the Repository Directory.

When running Transformation Manager for the first time, you will be given the option of installing a "samples" repository which contains the projects described in the installed application documentation. If you wish to install the "samples" repository after installation then this is provided from our web site customer area.

Multiple projects can reference any given model in the repository. For example, one project could be writing to data model A, while another might read from data model A. A user can enhance and modify a model within TM Designer and these changes are visible to all projects that reference that model. Procedures contain common business logic and can be used by any project in the repository. External functions are not part of the repository, although a user can choose to place them inside the repository data structure. They are external libraries of Java code which can be used to extend Transformation Manager functionality. The repository does contain configuration settings for External functions allowing their use in data integration projects. A user can set some preferences at the repository level and these are the default values used by all projects in the repository.

The repository's files and directories are used for backup and can be placed under version control etc. If a user needs to send someone a repository then they should use the Export and Import functionality provided within TM Designer and then send the file generated. A user can compress or zip the exported file for ease of data transfer.

The repository must not be edited outside of the application.

Directory Structure

A repository contains a number of sub directories under the specified root. These are CORE, PROJECT and MODEL. The CORE directory contains data that is repository-wide. The MODEL directory contains a subdirectory for each imported model plus 2 additional models – one empty, and one used for error handling projects. The PROJECT directory contains a subdirectory for each project. The transforms for each project are held in the MAPS sub-directory within the project's directory.

Backing Up Repositories

You are advised to backup your repositories frequently and certainly before installing a new version of Transformation Manager. If you are installing a new version of Transformation Manager then you should go to Upgrading repositories for use with New Versions of TM.

See Also: Working with Version Control (p. 36)

Version Control

Transformation Manager supports two approaches to version control. The first is where a user explicitly takes control of a file locking other users out before changes are made and the second is where files are scanned for changes. The second approach requires no special action from Transformation Manager. Users scan for changed files after the termination of each Transformation Manager editing session.

In systems that lock files, the files should be unlocked before changes are made. If a change is attempted on a locked file (i.e. a read only file), Transformation Manager will write that change in a new file with the extension '.etl1' (rather than '.etl'). Transformation Manager will warn a user that it has been unable to save the change. The '.etl1' files will not be read back by Transformation Manager. In these cases the locked file must be unlocked and the contents of the '.etl1' file copied to the '.etl' file.

Model files will change when editing parts of a model. Typically adding a relationship will cause the 'relation.etl' file and the 'relationship_sub_condition.etl' file to be updated. Other model alterations, for example addition of an element or attribute, may cause other files to change. Once this happens any changes made to the models or projects will be detected by the user's version control software. The files may be checked in to the central version control repository as normal.

Users should not update files while Transformation Manager is running. If a file is changed while Transformation Manager is running then the change will be detected and warn that a third party alteration has been made. However, Transformation Manager will not merge in third party changes - that is the task of the user's version control system.

If a version control system is being used then some operations which may appear automatic in Transformation Manager will require the user to maintain consistency in their version control system. In particular if a user deletes or adds a transform then the user must also delete or add the corresponding transform file in their version control system. If the user fails for example, to delete a transform from their version control system then a future version control checkout will reinstate the deleted transform. The same provisos also apply to changes in the model. Note that occasionally these changes result in additional files, or in files disappearing that have previously appeared. This is because Transformation Manager never stores an empty file.

Transforms

Introduction

A transform specifies how an element in the source data store is transformed to an element or elements in the target data store using the source and target data models and transform code written using SML (p. 59). A transform can contain simple assignments, or it can include processing

instructions to change a format or a value. For example, a number of litres in the source may need to be converted to gallons in the target. It describes how a specific part of the source model will transform to a specific part of the target model. Transforms will typically be written using two different data models, the source and the target which represent the source and target data stores. The source and target data models can be the same if required.

Each transform comprises of one or more lines of transform code, most of which are assignment statements. Assignment statements can simply copy attributes from the source or target or specify sophisticated paths through relationships. Transforms are written in the Editor pane of TM Designer. An assignment transform is written from left to right with the Transform Target on the left of the assignment and the Transform Source on the right of the assignment.

```
TransformTarget := TransformSource;
```

Transforms can be executed independently of each other or can be dependent on an independent transform. A group of dependent transforms can be called one after another but will always begin their execution from an independent transform. This is described as a stack where an independent transform calls a dependent transform and that dependent transform calls a further transform until all the transforms in the stack have been called, of course there may only be one dependent transform if that is all that is required. Transformation Manager uses the overall term cascade to describe the process of executing each transform in the stack.

A cascade is implemented in a transform using transform code that assigns a source relationship to a target relationship. This means that for each element instance that the source relationship relates to, create an element instance for the target relationship.

TM Designer has built-in cycle detection to avoid infinite loops when two or more transforms call each other in a repeated sequence. This is controlled using the SML Generation tab in the Repository Settings... or Project Settings window.

See Also: Transformation Approaches (p. 37)
Editing Transform Code (p. 38)
Mode (p. 39)
Priority (p. 40)
Advanced Property (p. 40)
How do I Create a Transform? (p. 42)
How do I Call a Dependent Transform? (p. 41)
Automatic Generation of Assignments - Transforms (p. 41)

Transform Approaches

TM Designer supports three fairly distinct transform approaches:

1) many independent transforms standing alone where execution order does not matter
2) many independent transforms standing alone where execution order does matter
3) one independent transform and a called tree of dependent transforms

The first style simply takes data from the source and places it within the target data store. There is no requirement to move the data in a controlled sequence and once that data has been moved any relationships will be preserved in the target without further processing.

The second style moves element data but in an ordered sequence. Each element is moved in total before the next independent transform and its element data is executed. This implies that a data element needs to be present and populated before the next one can be logically processed into the data store. An example could be where an element i.e. a table in a relational database must be populated before a child table, i.e. element, is populated; this may occur where constraints are in place in the target relational database.

The last uses dependent transforms where an independent transform starts a cascade of dependent transforms as part of the processing of data from the source data store to the target data store. This is different from a sequence of independent transforms because it processes each instance or record and all of its related data before moving to the next record or instance.

Transformation Manager also supports any combination of the above. The actual solution you use depends upon the type of transform problem you are trying to solve (combinations of RDBMS, XML and class) and the actual structure of the models. Trees of dependent transforms are generally convenient where the target requires order (for example, XML), but in some cases a few independent transforms will do the work of the entire tree with much less effort.

Editing Transform Code

The TM Designer Editor pane is where the details of a transform are entered. The statements here specify precisely what must be done to data from the source data store in order for it to be inserted into the target data store. In many cases, this will be a simple assignment, just copying a value such as a name or a reference number from one to the other. In other cases, a calculation or other operation will have to be performed before the data from the source data store can be added to the target data store. For example, a date in the format MM/DD/YY may need to be reformatted as DD/MM/YYYY to be valid in the target.

Each transform must include at least one assignment statement. Dragging and dropping an attribute from the source pane to the appropriate attribute in the target pane can create this very quickly.

If the transform requires more than a simple assignment, then the necessary statements must be entered into the transform editor pane either by typing the details, or by using the available quick methods of building statements. Listed below are some of the most frequently used methods for building transformation statements.

- Type in SML (p. 59) statements which describe the required transformation
- Drag and drop attributes and/or relationships from:
    - either the source or the target into the middle pane, or
    - the source to the target tree, or
    - the target to the source tree

The editor pane highlights syntax elements to assist with entry of correct statements, but full syntax checking is not performed until Build (p. 31) is chosen from the Run menu. Any error and warning messages arising from the check are displayed in the Output pane.

Assignment statements created using the drag-and-drop method will always be syntactically correct. For example, if any element or attribute identifiers in either the source or target models are reserved words in SML, then those identifiers are automatically enclosed in angle brackets. The following is a syntax error because `name` is a reserved word:

```
name := $Content;
```

If angle brackets are used, then the statement becomes valid:

```
<name> := $Content;
```

It is inevitable that there will be elements in source and target models that have names that conflict with reserved words in SML, so this mechanism is provided to avoid potential ambiguities. However, it is recommended that you avoid using SML keywords in identifiers for variables or functions that you define in order to reduce the chances of introducing such ambiguities.

There is a Find function on the Edit menu that can be used to search for particular strings within transforms.

The transforms you create are saved in the repository (p. 34) as you go. Transforms are not saved until you choose either Save or Save As from the File menu, or build the project.

Mode

Transforms have a property called Mode. Specifying a Mode permits more than one transform between the same two elements. A Mode is specified in the **New Transform...** dialog, which is displayed when a new transform is created, or launched from the specific transforms context menu from the **Change Properties...** option which opens the **Change Properties** dialog.

Transformation Manager has a series of functions that are used to move parameter values between cascading transforms. The calling transform will "push" the parameter values onto the cascade stack. A subsequent transform can then retrieve those values. The PUSHMODE() built-in function specifies the transform Mode for a given cascade call and will "push" the chosen Mode value onto the cascade stack.

The classic example that demonstrates the purpose of Mode is a transformation that involves a document. Initially, for example, transforms can be invoked in Mode 'toc', to produce the Table of Contents only, and subsequently in Mode 'body' to produce the main body of text.

Example

Assume we have two elements: a Source element known as `Document` and a Target element called `Divider`. We wish, initially, to produce the table of contents only from the `Document`, and subsequently to produce the main body of text. This will involve two transforms, both between `Document` and `Divider`. The assignment statements within each transform will, however, be different. The two transforms will be distinguished by the Mode property.

Within the Transform Properties Dialog, 'toc' must be entered in the Mode text box for the first transform, and 'body' must be entered in the box for the second.

Assuming Divider and Document are relationships to the elements Divider and Document respectively, the statement below will search for and execute the transform Document to Divider whose Mode property is ‘toc’, to produce the Table of Contents only.

```
Divider := Document + PUSHMODE('toc');
```

Subsequently, the statement below will search for and execute the transform Document to Divider whose Mode property is ‘body’, to produce the body of text.

```
Divider := Document + PUSHMODE('body');
```

Note

When specifying the Mode for a transform, only letters and numbers are allowed.

Priority

Every transform in a project can be given a priority. The maximum priority that may be assigned is 100,000 the minimum priority is -100,000. Do not use values outside this range.

In SML the transform priority has two purposes:

- It indicates to Transformation Manager which independent transforms should be run before which other transforms.
- It provides a mechanism for excluding certain transforms from generation.

Setting the run order

In general Transformation Manager assumes that independent transforms may be run in any order - and will choose an arbitrary order of its own. However by setting priorities it is possible to specify which transforms are run first. This can be very useful when you are developing transforms as you can specify that the transform you are currently developing is to be run first simply by giving it a priority higher than the other transforms. Note that Transformation Manager displays transforms in transform priority order, where this is defined. Transforms are executed from the highest value priority first to the lowest value priority last.

It is important to notice that transform priority plays no role in the cascade calling sequences between transforms. These are simply called in the order in which they are displayed in the transform navigator. Therefore setting a priority for a dependent transform is generally not very useful however setting a low priority may be used to exclude any transform, even a dependent transform, from the build see below.

Transforms which run between $document, $predocument and $postdocument are not directly affected by priority. It is not possible to specify that a given conventional transform is to be run before $predocument. However where more than one transform exists from $document the priority can be used to specify which of these is run first.

The second use of priority is to exclude certain transforms from generation. To do this you will need to set a value for the setting called **Lowest Priority to Generate** which is found in the **Project Settings** window on the **SML Generation** tab. This is accessed by using the project context menu and selecting the Settings... option. By default this is set to -1,000,000. By setting it to a higher value , for example '1', you can ensure that only those transforms with a priority above '0' will be generated. By setting it to '0' you can ensure that only those transforms with none negative priority will be generated. If you are developing a set of transforms this technique can rapidly remove all but the transforms you wish to run.

Note finally that by default all transforms have a priority of zero.

Advanced Property

All transforms have a property called Advanced which is used only with the source data store. This field provides functionality that lets a user add a modifier to the transform. An example of this would be for a database data store where a value here modifies the SQL statement as in the addition of an `order by` clause.

Automatic Generation of Assignments - Transforms

TM Designers editor pane lets a user quickly create assignments between a source and target element by matching attributes (and specifically terminal nodes in XML data models) to each other. This uses the **Auto generation of assignments** window available from the context menu of the editor pane. The window provides three matching strategies to help match attributes correctly and these are **Exact match**, **Case insensitive** and **Regular expression**.

The example below shows the result of using this functionality for the **BOOK to BookList** transform in tutorial 1. The matching strategy used here is **Exact match**.

```
-- NO MATCH FOUND FOR CURRENCY
-- NO MATCH FOUND FOR <ID>
-- NO MATCH FOUND FOR PRICE
-- NO MATCH FOUND FOR EDITOR_ID
-- NO MATCH FOUND FOR IS_FICTION
-- NO MATCH FOUND FOR PUBLISHED_DATE
-- NO MATCH FOUND FOR TITLE
```

Below is the result of the auto generation process using the **Case insensitive** matching strategy. In this example there are four attributes that have matched now that the matching strategy is case insensitive.

```
currency := CURRENCY;
<id> := <ID>;
price := PRICE;
-- NO MATCH FOUND FOR EDITOR_ID
-- NO MATCH FOUND FOR IS_FICTION
-- NO MATCH FOUND FOR PUBLISHED_DATE
title := TITLE;
```

A matching strategy can also make use of regular expressions in order to disregard common parts of attribute names. For example, a data store may prefix attribute names with a common term such as PERSON in a customer relationship management environment or WELL in an oil and gas enterprise. The user can then use a regular expression to limit the matching strategy to just that part of the attribute name required, for example, just NAME from the PERSON_NAME attribute and TITLE from the PERSON_TITLE attribute. In a PPDM project this could be used to disregard a prefix such as WELL_ from the matching strategy.

See Also: Auto generation of assignments window

How do I Call a Dependent Transform?

In this example details of people and their qualification are being transferred between a personnel and a training system. Both systems support the model of a person having many qualifications.

In the source data model, the element EMPLOYEE is related to the element QUAL. The relationship has the name relToQuals.

In the target data model, the element PERSON is related to the element QUALIFICATION. The relationship has the name relToQualifications.

Two transforms are required.

- The transform of EMPLOYEE to PERSON will be independent.
- The transform of QUAL to QUALIFICATION will be dependent, and called from the transform EMPLOYEE to PERSON.

The transforms will be similar to those shown below.

TRANSFORM to PERSON from EMPLOYEE

```
PersonCode          := Employee_Code;
Name                := Firstname & ' ' & Surname;
relToQualifications := relToQuals ;
```

TRANSFORM to QUALIFICATION from QUAL

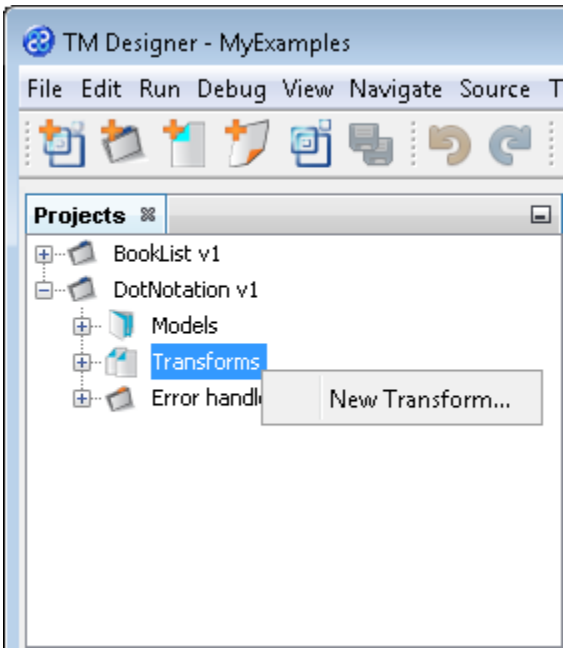
```
Abbreviation := Qual_Abbreviation ;
FulLDescription := Desciption ;
DateAwarded := AwardDate ;
```

Notes

- You make the second transform dependent by making sure that the Dependent tick box on the `Transform | Properties` dialog is ticked.
- For transforms to XML targets, make sure the target XPath is relative (`Transform | Properties` dialog).
- `relToQualifications := relToQuals` is the statement that calls the dependent transform. It is a mapping between two relationships.
    - `relToQualifications` tells Transformation Manager to cascade to a transform where QUALIFICATIONS is the target element.
    - `relToQuals` tells Transformation Manager to cascade to a transform where QUALS is the source element.
- Transformation Manager will transform the EMPLOYEE details one at a time. As it comes to cascade call, it will transform just QUAL details for the current EMPLOYEE.

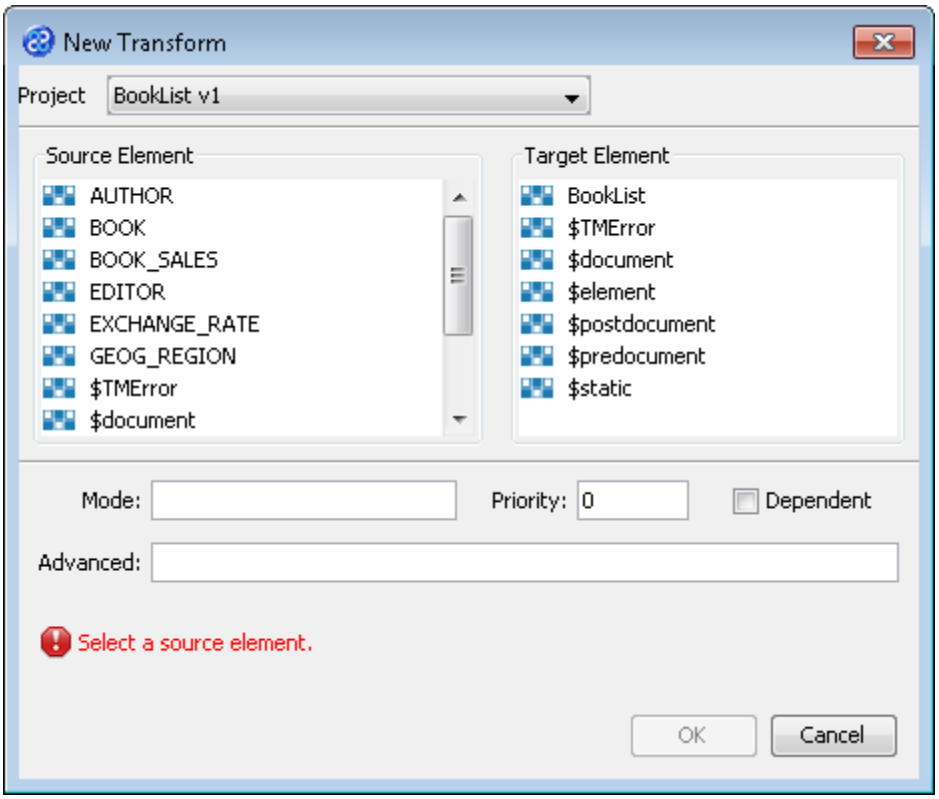
How do I Create a Transform?

The process below describes how to create a transform in Transformation Manager and is taken from Tutorial 1. This example uses the application menu bar but a shortcut is to use the context menu available from the Transforms branch within the Project pane of the application and selecting New Transform... as shown in the image below which is taken from Tutorial 3.

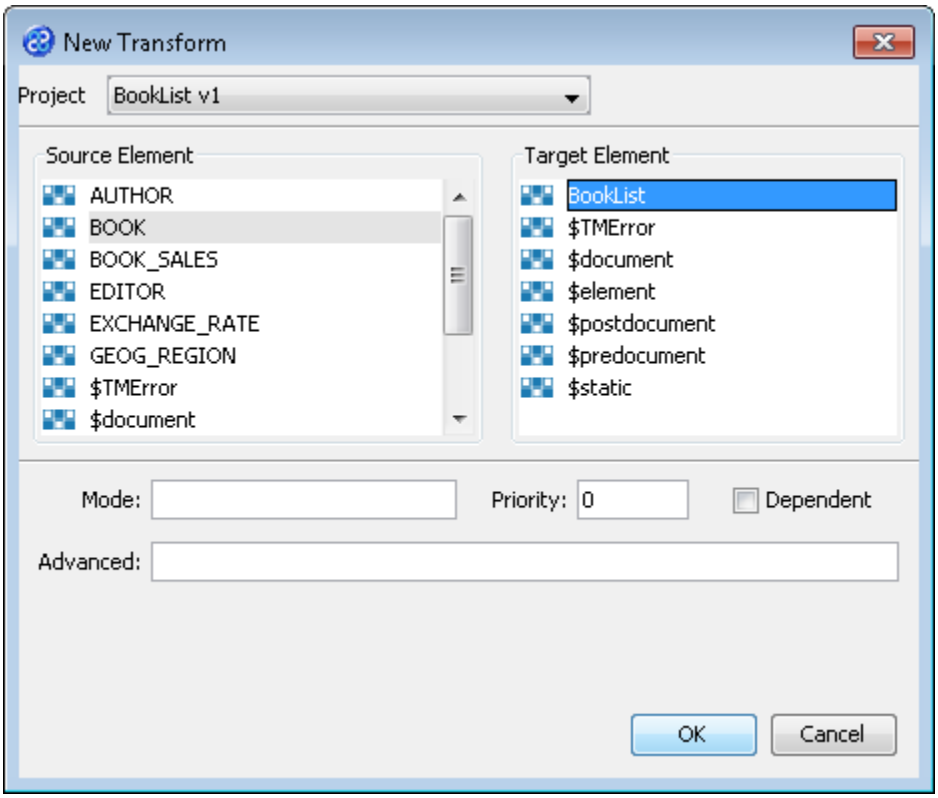


Follow the instructions below to add a new transform to a project.

1) Click once on the File menu bar option.
2) Click once on the New Transform... option from the sub-menu.
3) This will open the **New Transform** window. Ensure that you have the correct project selected in the **Project** drop down list, in the example below this is **BookList v1**.

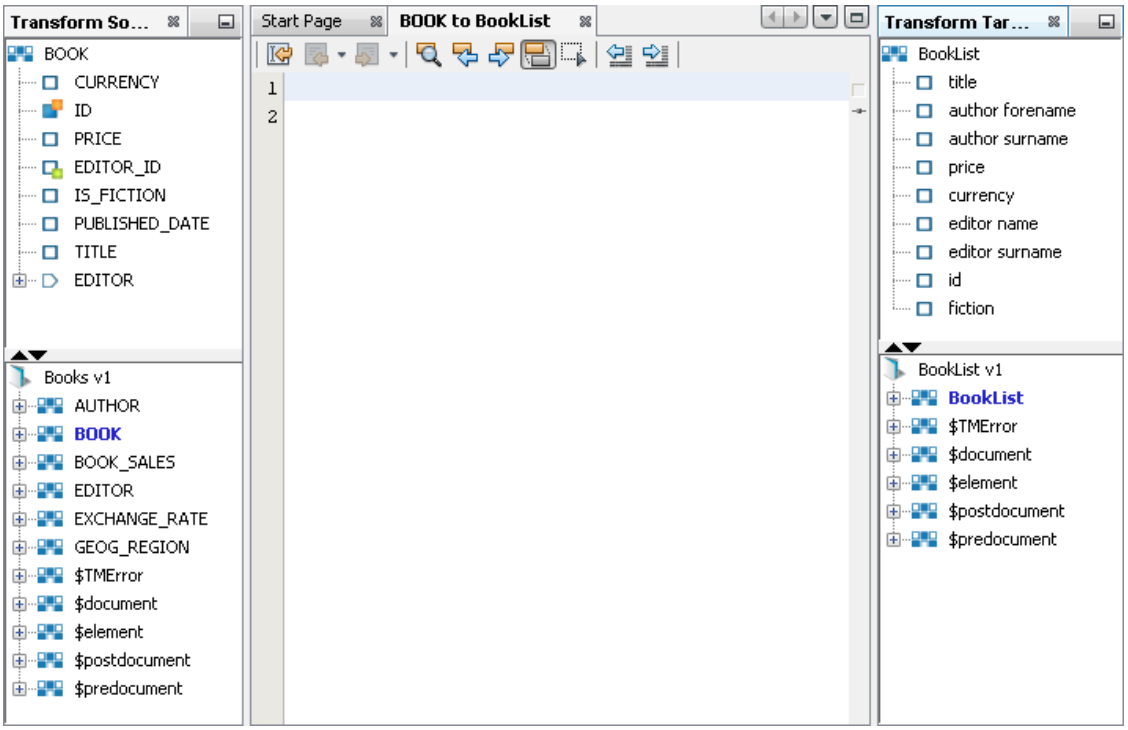


4) With the **New Transform** window open,select the elements for the source and target. In this case the source element will be **BOOK** and the target element will be **BookList**.



5) Click once on the OK button to create a new transform.

6) The Editor pane will open ready for transform code to be written Note that the **Transform Source** and **Transform Target** panes will open populated with the relevant content.



7) Once the transform has been completed click on the File menu bar option.

8) Click on the Save menu bar option to save the transform. Make sure that the cursor focus is the Editor pane otherwise the Save option will not be activated.

User Defined Functions

Introduction

The functionality of SML (p. 59) can be extended through the addition of User Defined Functions (UDFs). Transformation Manager has a large range of built in functions however there may be occasions when these do not meet the needs of a given transformation task. In this circumstance Transformation Manager provides the ability to add new UDFs when required. In PPDM a commonly found requirement is co-ordinate conversion for which a user may already have a system or library and this functionality lets a user make use of their own co-ordinate conversion functions.

The functions can be created within TM Designer or outside of TM Designer. Externally provided functions must be written in Java and are implemented using the **Externally Defined Functions...** window accessed from the File menu bar option. This provides access from TM Designer to any Java compatible object including standard Java libraries, Web Services, .NET or native code. Functions created within TM Designer can be created in Java or SML. Java functions can be written using the **New** and **Edit Function** windows accessed from the project context menu. SML functions are created as procedures using the context menu in the **Procedures** pane.

Transformation Managers facility to access externally developed functions is useful for users when there are existing libraries of functions or for users who want to develop their functions using advanced Java editing facilities available in third party development environments. Externally developed function libraries can be configured for each repository. The configuration lets a user make the functions available to all projects in the repository, the default condition, or specific projects as required.

TM Designer procedures are available to all projects within a given repository and are therefore useful when a procedure is likely to be needed in more than one project.

Functions created using the New Function... window are only available in the project they were created in so their scope is that project only. They can be edited and deleted as necessary using the Edit Function... and Delete Function... options from the project context menu.

UDF Characteristics

- The function can have none, one or more input parameters.
- The function must always return a single value.
- Calls to User Defined functions follow the form below.

```
targetelement := function_name(param1[, param2] ... )
```

Definition

- `targetelement` is the attribute or variable that will contain the result of the function
- `function_name` is the name of the function
- `param1, param2` and so on are the input parameters

SML parameters are unnamed; they are simply retrieved in the order they are given. If you add, delete, or change the order of any parameters accepted by the function, you may need to make corresponding changes to the body of the function.

See Also: Internal User Defined Functions (p. 46)
Accessing External Functions (p. 46)

Internal User Defined Functions

Writing internally-defined UDFs is often best done in SML using procedures. SML procedures may themselves call Java UDFs which can be both internally or externally defined. Procedures (p. 24) are UDFs that contain SML transforms that are called from other transforms.

TM Designer internally-defined Java functions also have their own benefits.

1) There is no need to add any Java files or jars to the Transformation Manager Class Path.
2) The functions are part of the project and not located in external files.

In general, internally managed functions and procedures are useful when the number of functions is relatively small and the bodies are simple. Typically good uses of an internally managed UDF are listed below.

1) Calling out to some existing customer specified mechanism.
2) Establishing a source of surrogate key values in a target database.
3) Reading a configuration file.

Accessing External Functions

Transformation Manager provides access to externally created libraries of functions using the **External Defined Functions...** window. This is useful if you have existing libraries of functions you wish to utilise.

Function Types

In order to provide the maximum flexibility we have provided three mechanisms, of which any one, two or all three mechanisms can be used simultaneously. If you are new to this topic please read this section carefully because using an inappropriate mechanism will limit the use of your functions. It is also important to choose the right abbreviation convention for your project(s) as early as possible, this improves the readability and long term maintenance of a project.

Each of the importation mechanisms or **Default load setting** options has its own name and characteristics.

- Static Explicit.
- Static Implicit.
- Public functions.

The 'Static Explicit' load setting refers to functions that are specifically flagged as Static Explicit in the Property Editor window. These functions have a first parameter of type 'TMContext'. See the notes below on singleton functions (p. 54). All other functions in the classes are ignored.

The 'Static Implicit' load setting refers to functions that are specifically flagged as Static Implicit. These functions have no requirement for the function to have a first parameter of type 'TMContext'

but any explicitly marked functions will also be imported. All other functions in the classes are ignored.

The 'All Public' load setting refers to all public functions and it also imports 'implicitly marked' and 'explicitly marked' functions found in the name classes. All other functions in the classes are ignored.

In general, imported functions will appear in TM Designer in the same manner as the source Java code. Parameter types will be converted to the equivalent parameter types in SML. However, there are two exceptions. Firstly, the 'TMContext' parameter never appears in SML, it is filled in automatically by the run-time system to provide access to the mapping context. Secondly, non-static functions appear in SML with an extra parameter of type jObject. At run-time this must be an SML jObject of the same class. For example, if a class 'pkg.MyClass' declares a function 'functionX' then in SML the first parameter to the function must be pkg.MyClass jObject.

Using Externally Referenced Functions

Externally referenced functions can be accessed and used in your transform in the standard way within the transform Editor pane. All instance functions (not static or singleton functions) will have an additional first parameter. This will be described in SML as a jObject parameter and should be an instance of the class from where the function is referenced. This may have to be obtained as the return value from a second UDF.

It is possible to mix externally managed functions and internally managed functions within the same project. If a function appears to exist both externally and internally then the internally managed function will take precedence. However, this is an error and will be reported as such.

Abbreviation Convention

Functions are called from within SML using <underscore separated fully qualified class name | abbreviation > : <functionName>. Below are examples.

```
result := myCo:doSomething('true');
result := com_myCo_project_utilities:doSomething('true');
```

We strongly recommend the use of a consistent set of abbreviations.

Polymorphic loading of UDFs

This section deals mostly with the polymorphic loading of UDFs. UDFs are loaded polymorphically either because more than one method of the same name exists in the Java source code or because you have chosen the option to load all UDFs as though polymorphic.

The usage of UDFs loaded polymorphically is different because the actual parameters have to match the function you have defined. In contrast to when Transformation Manager is dealing with UDFs which have been loaded conventionally, Transformation Manager itself will perform all the necessary conversions. A complexity is that Transformation Manager is based on a XSD-type system, not on the Java type system. Hence, when dealing with UDFs it may be necessary to know the correspondence between a Transformation Manager type and a Java type. The standard conversion table is shown below. Note that any user defined types, or model loaded types, will always be based on one of these types. When Transformation Manager decides to extract a Java object and send it to a function it will always establish the base type which must be in this list and will then extract the relevant type.

TMType	JavaType
anyURI	java.lang.String
base64Binary	java.lang.String
Binary	java.lang.String
binaryLargeObject	java.io.Reader
boolean	java.lang.Boolean
byte	java.lang.Byte
character	java.lang.Character
characterLargeObject	java.io.Stream
date	java.lang.Date
dateTime	java.lang.DateTime
decimal	java.lang.BigDecimal
double	java.lang.Double
duration	java.lang.String
ENTITIES	java.lang.String
ENTITY	java.lang.String
float	java.langfloat
gDay	java.lang.String
gMonth	java.lang.String
gMonthDay	java.lang.String
gYear	java.lang.String
gYearMonth	java.lang.String
hexBinary	java.lang.String
ID	java.lang.String
IDREF	java.lang.String
IDREFS	java.lang.String
instance	[cannot be used as function parameter in version 4.xx].
int	java.lang.Integer
integer	java.lang.Long
jobject	java type is that placed into the object, this may be cast in SML.
language	java.lang.String
long	java.lang.Long
LongVarBinary	java.io.Reader
LongVarChar	java.io.Stream

Name	java.lang.String
NCName	java.lang.String
negativeInteger	java.lang.Long
NMTOKEN	java.lang.String
NMTOKENS	java.lang.String
nonNegativeInteger	java.lang.Long
nonPositiveInteger	java.lang.Long
normalizedString	java.lang.String
NOTATION	java.lang.String
positiveInteger	java.lang.Long
QName	java.lang.String
short	java.langshort
sqlArray	java.lang.Object
sqlDistinct	java.lang.Object
sqlJava	java.lang.Object
sqlNull	[cannot be used as a function parameter].
sqlRef	java.lang.Object
sqlStruct	java.lang.Object
string	java.lang.String
time	java.lang.Time
token	java.lang.String
unsignedByte	java.langunsignedByte
unsignedInt	java.lang.Integer
unsignedLong	java.lang.Long
unsignedShort	java.lang.Short
userDefinedType	java.lang.Object

Note that Transformation Manager does not require any distinction between scalar Java primitives (int, char, long etc) and wrapped Java primitives (Integer, Character, Long). A function which accepts 'Character' for example will automatically also accept 'char'. The only issue is that TM Designer will not load functions which provide signatures for both types, in other words you cannot have Function1(Long x) and Function1(long x) although TM Designer will allow you to have Function1(Long x) and Function1(float x).

It can be seen from the above table that the following SML code is acceptable.

```
T1:doubleNonPrimitive( <Double>);
T1:doublePrimitive(<Double>);
T1:shortNonPrimitive(<Short>);
T1:shortPrimitive(<Short>);
T1:longPrimitive(<Long>);
T1:longNonPrimitive(<Long>);
T1:intPrimitive(<Int>);
T1:intNonPrimitive(<Int>);
T1:floatPrimitive(<float>);
T1:floatNonPrimitive(<float>);
T1:bytePrimitive(<Byte>);
T1:byteNonPrimitive(<Byte>);
T1:booleanPrimitive(<Boolean>);
T1:booleanNonPrimitive(<Boolean>);
T1:BigDecimalNonPrimitive(<Decimal>);
```

Here `<Double>`, `<Short>` are attributes of Transformation Manager type 'Double' 'Short' etc and functions doubleNonPrimitive, doublePrimitive etc are defined in Java as 'String doubleNonPrimitive(Double d)' and 'String doublePrimitive(double d)'.

Note that an attribute of Transformation Manager type <Decimal> will automatically match to a BigDecimal.

Casting may also be used to produce SML code that is acceptable. For example while Transformation Manager will refuse to compile `T1:doublePrimitive(<float>)`, it will compile `T1:doublePrimitive({<java.lang.Double>}<float>)`. This code will also execute successfully at run time as Transformation Manager will internally convert the 'Double' to a Float before passing it into the function.

Note that the type used in the cast must either be a type known to the system from other external UDFs, or must be a non-primitive Java class name; the fully qualified name is required. The rule about using nonprimitive Java class name continues to apply even if the function itself accepts the corresponding primitive. If the cast is missing or unknown then Transformation Manager will produce an error message telling you which function signatures were available, and why it has failed to match. For example the following function will produce the error beneath it.

Function

```
T1:doubleNonPrimitive({<java.lang.double>}<float>);
```

Error

```
ERROR:For Function 'T1_doubleNonPrimitive' the supplied parameter type(s)
['java.lang.double'] does not match the defined signature, which is
'java.lang.Double.'
```

The same rules about conversion also apply whenever a local or global variable is passed to a procedure or function. The variable must either match the type of the function directly, according to the rules given above or must be cast in the same way.

When a number of polymorphic functions exist then casting may be used to discriminate between a number of implementations. For example, look at the following Java definitions.

```
  public static String primitiveDiscriminationTest(TMContext theTMContext, byte
b)  {
    return "byte=" + b;
 }

  public static String primitiveDiscriminationTest(TMContext theTMContext, long
b)  {
    return "long=" + b;
  }

  public static String primitiveDiscriminationTest(TMContext theTMContext, int
b)  {
    return "int=" + b;
  }

  public static String primitiveDiscriminationTest(TMContext theTMContext,
short b)  {
    return "short=" + b;
  }

  public static String primitiveDiscriminationTest(TMContext theTMContext,
double b)  {
    return "double=" + b;
  }

  public static String primitiveDiscriminationTest(TMContext theTMContext,
float b)  {
    return "float=" + b;
  }

  public static String primitiveDiscriminationTest(TMContext theTMContext, char
b)  {
    return "char=" + b;
  }
```

In these functions in SML, it is the cast which defines which function is actually called. The example below demonstrates this.

```
ans1 := T1:primitiveDiscriminationTest({<java.lang.Integer>}<Float>);
```

This will call the `primitiveDiscriminationTest(TMContext theTMContext, int b)` method , not the method on float.

It is possible to call array functions via SML. In the following SML example, 2 items are added to a vector. The contents of the vector are then converted into a native int array and the array is then passed into a function expecting a native array. In contrast with the examples above, the function call will discriminate between array primitives (int[ ] etc) and object arrays (Integer[ ] etc).

```
LOCAL
   AJobject1 : jobject;
   AJobject2 : jobject;
END_LOCAL;

 AJobject1 := MAKEJOBJECT('java.util.Vector');
 -- Jobject constains a Vector

 ADDTOARRAYLIST(AJobject1, {<java.lang.int>}<attr1>);
 -- add an object to the list

 ADDTOARRAYLIST(AJobject1, {<java.lang.int>}<Attr2>);
 -- add an object to the list

 AJobject2 := GETNATIVEARRAY(AJobject1,'java.lang.int',-1);
 -- convert to a native array of 'int'

ans1 := T1:primitiveDiscriminationArrayTest({<int[]>}AJobject2);
 -- type it to int[] and pass it through to UDF
```

An alternative is to define the type for the SML local jObject and then the cast is not required.

```
LOCAL
   AJobject1 : jobject;
   AJobject2 : jobject;
END_LOCAL;

 AJobject1 := MAKEJOBJECT('java.util.Vector');
 ADDTOARRAYLIST(AJobject1, {<java.lang.int>}<Int>);
 ADDTOARRAYLIST(AJobject1, {<java.lang.int>}<Short>);
 AJobject2 := GETNATIVEARRAY(AJobject1,'java.lang.int',-1);
ans1 := T1:primitiveDiscriminationArrayTest(AJobject2);
```

Examples

Simple examples are distributed in the package samples.UDFExamples.

Static Explicit Functions

Example 1

```
public static integer functionX(TMContext theTMContext ,int x, String y)
{
  . . .
}
```

The example above will appear in SML as shown below.

```
integer functionX(integer x, string y)
```

Example 2

```
public static Ztype functionY(TMContext theTMContext , Xtype x , Ytype x)
{
  . . .
}
```

The example above will appear in SML as shown below provided that the Java definition of Xtype , Ytype and Ztype all implement net.etltm.TMType.

```
Ztype functionY(Xtype x , Ytype x)
```

Static Implicit Functions

Example 1

```
public static integer functionX(int x, String y)
{
  . . .
}
```

The example above will appear in SML as shown below.

```
integer functionX(integer x, string y)
```

Example 2

```
public static Ztype functionY(TMContext theTMContext , Xtype x , Ytype x)
{
  . . .
}
```

The example above will appear in SML as shown below provided that the Java definition of Xtype , Ytype and Ztype all implement net.etltm.TMType.

```
Ztype functionY(TMContext theTMContext , Xtype x , Ytype x)
```

Public Functions

```
public class Fred
{
  public String getDateAsString(SomeDateClass date)
  {
    return "..";
  }

  public static String getStaticSomething(String x)
  {
    return "dd";
  }
}
```

The example above will appear in SML as shown below.

```
string getDateAsString(jobject instance, jobject date)
string getStaticSomething(string)
```

Singleton Functions

A singleton function is an instance method of a Singleton class as in the example below.

```
package com.myco.db;
public class DBLookup
{
  private static DBLookup instance;
  private DBLookup() {}
  public static synchronized DBLookup getInstance()
{
    if (instance == null)
{
      instance = new DBLookup();
    }
    return instance;
  }
  public String translateBusinessCentre(String businessCentre)
{
    return "translate:" + businessCentre";
  }
}
```

The `translateBusinessCentre(String)` method would be invoked using the following code.

```
DBLookup.getInstance().translateBusinessCentre("bc1");
```

In the settings below, this would be specified by `com.myco.db.DBLookup#getInstance`. The static method getInstance(), taking no parameters and returning the singleton instance of DBLookup, is specified after the class name separated by a #. The generated method invocation is constructed according to the following rules.

- Class name (to the left of the #)
- Append a '.'
- Append the String following the #
- If this does not already end with a ")", append ")"
- Append a '.'
- Append the singleton method name (eg translateBusinessCentre).

Singleton functions are configured in the static rather than the instance settings because they appear and are used in Transformation Manager as static functions. Although translateBusinessCentre is an instance method, as DBLookup#getInstance is specified in the static settings, it does not gain an instance parameter as if it were specified in the Classes for all Public Functions setting. This is because Transformation Manager obtains the instance to invoke the method on by calling DBLookup.getInstance().

Note

All classes that are imported as UDFs must be part of a package.

External UDF Management

The functionality of SML can be extended through User Defined Functions (UDFs) (p. 45) written in Java. The UDF manager allows the user to add jars and standalone Java classes to the search path via

a file browser interface. The user can then select which types of methods from the classes will be recognised by Transformation Manager as internal functions.

Global abbreviations

When UDFs are loaded into TM Designer each class name can be abbreviated for use in SML. Three default options for this abbreviation are supported; however, one can override the abbreviation for a specific class in the Properties Dialog. The default options are:

- Class name – Just abbreviate the class to its name, dropping all package information
- Package name – Use the last part of the package name and the class name itself
- Package substitute – Substitute the package part of the class name with the specified string and then add on the class name

The table below shows how these abbreviations work when applied to two classes:

Abbreviation type	Class names	
	org.my.utilities.StringUtils	org.my.custom.Configuration
Class name	StringUtils	Configuration
Package name	utilities_StringUtils	custom_Configuration
Package substitute (with the string UDFS)	UDFS_StringUtils	UDFS_Configuration

Advanced Settings

The advanced settings apply to all files unless overridden. The advanced settings are:

- Include standard methods – Normally standard methods of the Object class are not loaded as UDFs (for example, toString()). If this is selected then these methods are loaded.
- Load all as polymorphic - In its default mode TM Designer will load all overloaded functions polymorphically and will load all functions which are not overloaded using its non-polymorphic approach. Functions loaded non-polymorphically may be called in SML with parameters that do not match and TM Designer will make the conversions for you. Polymorphic functions on the other hand require that the function parameters are either matching or have been cast to a matching type. Selecting this option will load all the functions with the Polymorphic approach.

See Also: Accessing External Functions (p. 46)

Pure Classpath Entries

Sometimes it is necessary to add jars to the classpath so that the UDF methods can be correctly resolved. When this is the case it is necessary to add the jars using this dialog. Mark the default load type as **None**; this will stop TM Designer from trying to load any UDFs from the jar.

Variables

Introduction

Variables are temporary storage locations that can be used to hold intermediate results. Transformation Manager typically uses simple variables of type string or integer to contain text or whole numbers respectively. Variables can be used for more sophisticated purposes, for example as handles to objects in external code. SML (p. 59) has two distinct types of variable.

Variable Type	Description
Local variables (p. 57)	These are declared by the user for use within a specific Transform.
Global variables (p. 56)	These are declared by the user, and are available for use within all Transforms in a Project.

Variables are set as though they were target attributes on the left hand side of an assignment as in the example below where the variable called `myVar` is being set as the value from `COLX_1` plus `3`.

```
myVar := COLX_1 + 3;
```

Variables can be read or accessed by name as in the following code where the attribute `COLA_1` in the target data store will be set with the value from the variable `myVar`.

```
COLA_1 := myVar;
```

Variables can be used in an index where they need to be preceded with a $ sign. The example below shows a relationship to relationship assignment but with the addition of the `$myVar` variable being used as the index.

```
toB := iToZ[$myVar];
```

If at the point of execution, `myVar` has the value 3, then the code above has the same meaning as that shown below.

```
toB := iToZ[3];
```

Variable Names

The rules for naming variables are simple and must follow the rules below.

- Must not begin with a reserved word
- Must begin with an alphabetic character
- After the first character may contain any alphanumeric character and the underscore character

Variable names are case-sensitive. There is no practical limit to the length of a variable name.

See Also: Global Variables (p. 56)
Local Variables (p. 57)

Global Variables

Transformation Manager let's a user declare global variables. The scope of global variables is a project so any global variable created can be used in all transforms in that project. Transformation Manager provides two ways in which global variables can be added to the system. The first uses the Global Variables window which is available from the project context menu and the second is by directly typing the global variable into a transform, this would normally be the **$document to $document** transform. A specific project may require both global and local variables and where this is the case local variables should not have the same name as a global variable.

Numeric global variables are initialised to zero when the transformation starts and string variables are initialised to the empty string.

The example below shows how to declare global variables in a **$document to $document** transform. The symbol == means immutable final constant. Transformation Manager will not let you change it later.

```
GLOBAL
        X : boolean := true;
        Y : int == 34;
        Copyright : string == 'Copyright by ETL solutions 2013';
        gblAreaId: int;
        gblMessages: string;
        gblCompany: string == 'ETL';
        gblSeparator: string := '//';
END_GLOBAL;
```

See Also: Variables (p. 56)
Local Variables (p. 57)

Local Variables

A LOCAL code block is used to declare variables for use within a specific transform; there can be only one LOCAL block in a transform and it must appear at the top of the transform before any other executable SML code. Local variables are initialised each time the transform is executed – for each source instance processed by the transform. Variables with the same name can be declared in the LOCAL code blocks of different transforms. A variable declared within a LOCAL block should not have the same name as a global variable (p. 56) but it is possible. Transformation Manager will warn a user and by default stop the code generation of a project. This can however be allowed by ticking the setting called **Local variables may mask global variables** which can be found in the **SML Generation** tab of the **Repository Setting...** window available from the File menu option.

Local Variables may be declared in 3 ways.

1) With no initial value

```
LOCAL
        str   : string;
        rel   : float;
        anint : integer;
        count : integer;
END_LOCAL;
```

2) With an initial value

```
LOCAL
        aVar : string := '\'';
        separator: string := '.';
END_LOCAL;
```

3) With an initial, constant value as below, where the == notation indicates that this is a final constant, and its value cannot be changed elsewhere in the transform.

```
LOCAL
        aVar2  : string == 'hhhh';
END_LOCAL;
```

A LOCAL block may contain several variables, which may be declared using a combination of the ways described above.

The data types allowed in a local variable are:

- boolean
- date
- datetime
- decimal
- double
- float
- integer
- JObject
- long
- string
- time

These data types are also applicable to global variables.

See Also: INCVAR

Simple Mapping Language

Introduction

Transformation Manager uses SML code in transforms. SML displays transform code in a simple, humanly readable, high-level transformation syntax which can be used as a domain specific language for describing data integration logic. SML simplifies the process of describing the transformation of data from a variety of source to target data stores. It masks the many differences between different types of data store and makes it possible to concentrate on the data without worrying about the details of how the data is stored.

There are three concepts behind SML.

- It is a Semi Declarative Language. (p. 60)
- It is Extensible (p. 61).
- It is a Common Language (p. 7).

SML is a fully featured programming language. This allows a user to easily perform simple transformations (such as copying attributes), while retaining the power to perform more complex transformations (such as those involving relationships between data items or complex mathematical operations). SML has a range of built built-in functions to assist with this.

SML permits the use of User Defined Functions (p. 45) (UDF) to extend Transformation Managers capabilities for specific ETL tasks. For example, Java functions can be used to enhance SMLs ability to transform a set of spatial co-ordinates from Cartesian to polar values which requires trigonometric functions not available in SML.

Transforms are semi-declarative and include coding patterns to control program flow and perform iterations such as `BEGIN END`, `CASE`, `FOR EACH`, `IF THEN ELSE` and `REPEAT`.

Over 280 built-in functions are provided for a wide variety of purposes including Aggregate data, Data Quality, Database Specific, Error Handling, Error Recording, Internationalisation, Logging, Testing, Maths, String, Transaction Handling, XML plus many others.

SML code can also be re-used with the help of procedures which are supported within TM Designer.

Local variables (p. 57), accessible in the transform, and global variables (p. 56), accessible by all transforms in a given project, may be used.

An Example of a Transform Written in SML

```
LOCAL
        NoOfParts : string;
        AttName   : string;
        TradeID   : string;
END_LOCAL;

--Code to get the attribute name from the full path node
NoOfParts    := STARTPARSER(<name>, '.');
AttName      := GETNEXTBYINDEX(<name>, NoOfParts);
NoOfParts    := STARTPARSER(AttName, '[');
AttName      := GETNEXT();

IF (AttName = 'amt' and value > 500000000)

THEN
        NOTEERROR('Authorisation Limit Exceeded')
END_IF;

--Mapping Statements
<ID>         := $uniqueKey;
PATH         := <name>;
VALUE        := value;

FK_KNOWNFIELDS.!SHORTNAME := AttName;

CATCH_ERROR
        IF ($TMError.detailMessage = 'Authorisation Limit Exceeded') THEN
                $TMError.contextInfo1 := value;
                $TMError.contextInfo2 := <name>;
                $TMError.action       := 'C';
        END_IF
END_CATCH
```

Drag and drop functionality in TM Designer automatically generates SML code for simple transforms. For more complex transformations it is possible to write SML code directly into a transform.

This chapter contains the following sections:

SML is Semi Declarative

Broadly speaking, SML (p. 59) is a declarative language: it generally does not matter in what order a given set of statements are processed. (Indeed, in some cases the *logical* order will be quite different from the order in which the statements appear in the project.)

Statements in SML describe the way that the source model transforms to the target model. This allows you to focus on describing the transformation and not on how it is executed. The problems of executing a declarative transformation specification are handled by Transformation Manager's code generator and execution system.

For maximum flexibility, however, SML also supports some conventional procedural constructs, such as iterative loops and variables.

Transformation Manager also includes stack based calling between transforms similar to those you find in conventional low level procedural languages like XSLT or Java.

The combination of :

- A declarative language
- Some procedural constructs
- Stack based calling

means that you can create transforms of almost any degree of complexity to correctly handle extreme differences in structure between source and target data stores.

See Also: Execution Order (p. 17)

SML is Extensible

Transformation Manager has the ability to create its own or use externally created User Defined functions (p. 45). User Defined functions can be used to enhance the scope of SML letting a user cope with the specifics of a given ETL task; for example, advanced statistical analysis or check sum calculations which require functions that are not available in SML. Transformation Manager lets you build your own internally generated Java functions, or use externally available libraries of functions.

Demonstrating SML using Relational Database Models

This section is an introduction to using SML in TM Designer and gives a brief overview of how to use SML. Fictitious source and target data models, consisting of three elements in each, will be used and these are based on relational database data stores. For simplicity, the structure of both data models is equivalent. The major features of SML will be demonstrated using example transforms.

The demonstration transforms may not be legal in the context of an integration project although the individual transform code statements are valid SML. The reason for this is that, for example, they will contain conflicting assignments to the same attribute so that differences between transforms can be discussed.

There are four transforms involved, two of which are independent, **$document to $document** and **X to A** and two dependent, **Y to B** and **Z to C**. Some initialisation statements are only valid in a **$document to $document** transform. The transform **X to A** starts a call stack where **X to A** cascades to, or calls dependent transform **Y to B** and then calls **Z to C**.

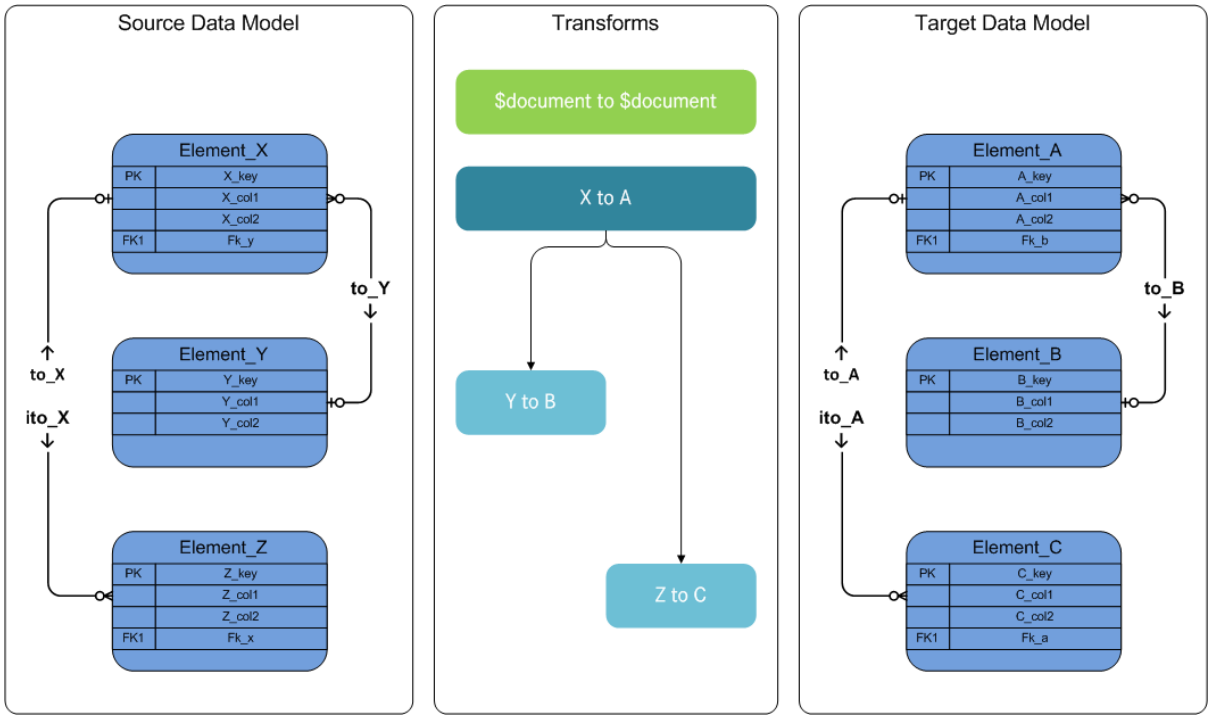
The basic idea is that the source data is copied to the target. Transformation Manager can perform sophisticated structural change but the intent here is to demonstrate SML.

The relationships between elements shown as **to_A**, **to_Y** and so on, have the same name in the diagrams as you will see in the SML code. In the PPDM models the relationship name will often be the

same as the element name but there is no ambiguity as only the relationship name appears in SML code, not the element name.

In the target data model the element called **Element_A** has a foreign key reference to **Element_B** i.e. **Fk_b** and the element called **Element_C** has a foreign key reference to the element called **Element_A**, **Fk_a**. We look at both relationships from the perspective of **Element_A**; each instance in **Element_A** can reference at most one instance in **Element_B** but many instances in **Element_C**. As described in Forward_and_Inverse_Relationships (p. 32), forward relationships are from the FK end to the PK end in a typical relational database model loaded into Transformation Manager. So the forward relationship from **Element_A** to **Element_B** has the name **to_B** in the diagram below. To get from **Element_A** to **Element_C** we need the inverse relationship of **to_A**, which has the name **ito_A** in the diagram below. It is common to name inverse relationships with a preceding lower case "i". This is the case for PPDM data models, but it is not universal.

The source model has an equivalent set of relationships, **to_Y**, **ito_X** and so on, as shown in the diagram.



A different way to understand this in Transformation Manager is to look at specific values in the source and target data stores. In the diagram below, each instance or row in **Element_X** can reference one instance or row in **Element_Y**. The foreign key attribute **Fk_y** in **Element_X** references the primary key column, **Y_key**, of **Element_Y**. This is how the **to_Y** relationship above is implemented.

You can view the relationship starting from **Element_Y**. Each instance in **Element_Y** can reference multiple instances or rows in **Element_X** via the **ito_X** inverse relationship.

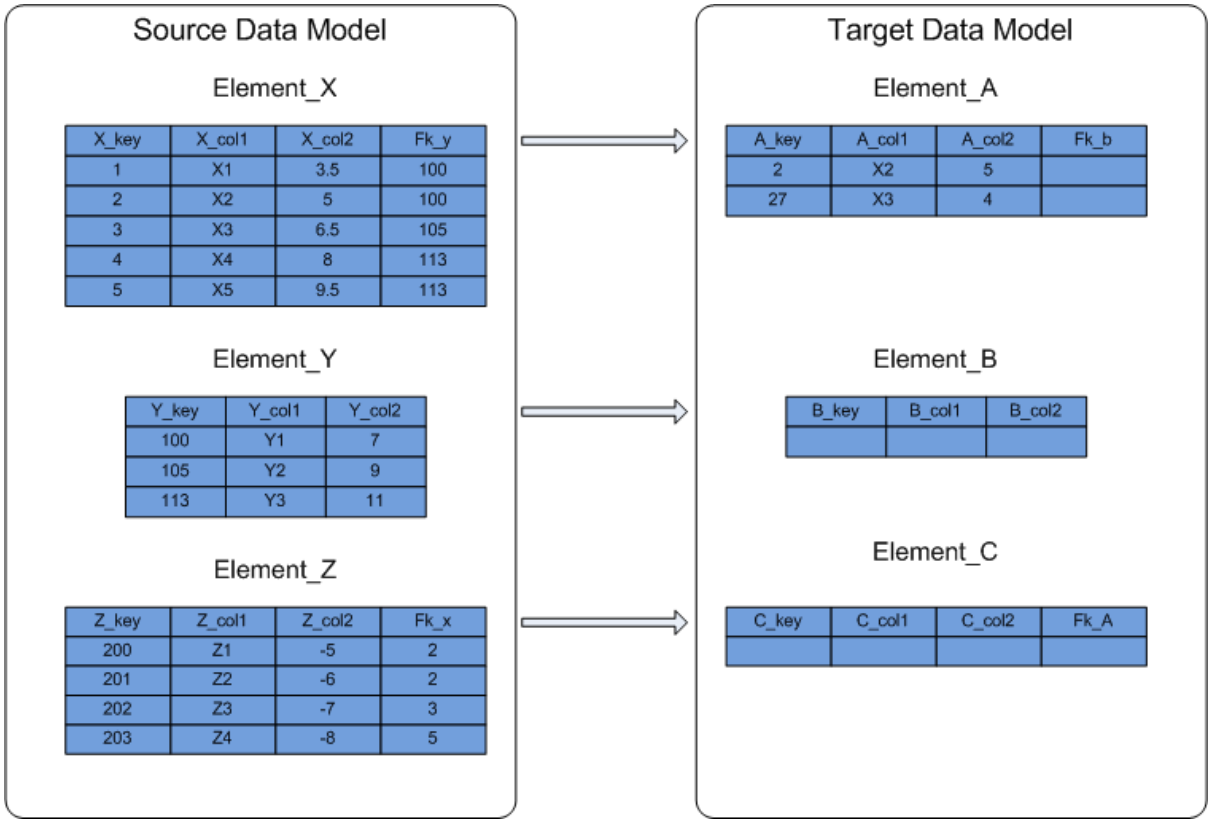
Element_X and **Element_Z** are similarly related.

In the transform **X to A**, you can use the **to_Y** relationship to reference the single related instance in **Element_Y**. For example, if you are processing the **Element_X** instance whose **X_key** is 5, the relationship **to_Y** refers to the instance in **Element_Y** whose **Y_key** is 113.

You can use the inverse relationship **ito_X** to reference the related **Element_Z** instances, but there may be many of them. The second instance of **Element_X** where **X_key** is 2 references two instances of **Element_Z** with the **Z_key** attribute values of 200 and 201.

This difference is exposed when you use the relationships in SML, as shown in the Assignments section below.

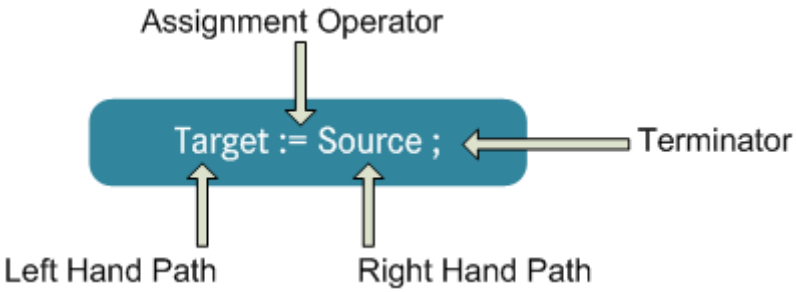
If the target is a database, it may already contain some data. This could be reference data, and therefore a user needs to ensure it is correctly referred to by the instances inserted or updated. It might be data that represents the same object as one of the items in the source – a WELL that is already known about for example. In this case the user may need to define how to detect the source and target instances which might refer to the same object, and take appropriate action – updating the values rather than inserting, or taking no action at all. This is the subject of identification (p. 67).



SML provides features to implement complex transform logic such as filtering source data, merging data, arithmetic, string manipulation, etc. The majority of transforms simply describe copying parts of the source data model to equivalent parts of the target data model. It is far removed from low level code that only a software developer can understand although it is probably true that the involvement of a software developer will be beneficial to successfully complete a data integration project with Transformation Manager. A well written project can be discussed and understood by both software developer and business or domain expert.

Assignments

Assignments are created in transforms. The image below shows a simple schematic for an assignment with the component parts of an assignment labelled, the source, the target, assignment operator and terminator.



The left hand path generally refers to some object in the target model, for example an attribute, relative to the target element of the current transform. The right hand path refers to an object in the source and is the item that will be assigned to the target. In addition to this is the assignment operator, `:=`, and the terminator for the line of code, `;`. The assignment operator indicates the copying of an item from the source to the target.

Simple Attribute Copy

The example below is a simple copy from a source attribute to a target attribute and assumes that the type is the same for both attributes. Simply stated the target attribute COLA_1 takes its value from the source attribute COLX_1.

```
COLA_1 := COLX_1;
```

If the target attribute, COLA_1 has a different type to the source attribute then the assignment is a little more complicated. Transformation Manager will attempt to convert the value from the source into an appropriate one for the target. If the source attribute is of type string, with the value '12.34', then a target attribute of type decimal will be correctly set. If the conversion fails, for example a source string of 'XYZ' being assigned to a number, the target attribute value becomes, within Transformation Manager, a value which represents the fact that the source value cannot be represented: in this case it is not a number and carries the internal value of NaN.

If the target is a database, this will become NULL. If you need to guard against this conversion, you can test the source value using functions like ISNAN() and take appropriate action.

Assign to a Primary Key

This is like any other assignment in terms of the value the primary key column receives. Because A_KEY is the (only) primary key column for A, at runtime Transformation Manager will do a find or create based on the value assigned to it.

```
A_KEY := X_KEY + 1000000;
```

If there is an existing row whose A_KEY attribute has the value (X_KEY + 1000000), it will be updated, otherwise it will be inserted. This behaviour is the default, and can be changed. If the primary key is a surrogate key then a common requirement occurs where a user does not want to find or create an instance using a uniquely generated identifier, such as a numeric value returned from an Oracle sequence, because such a row will never be found.

See Also: Identification (p. 67)
Surrogate Database Keys (p. 73)

Expressions

A user can compute values using an expression on the right hand side of an assignment. SML supports numeric, string, date and other expression types.

Arithmetic

```
COLA_1 := COLX_1 + 3 * COLX_2;
```

String concatenation

If COLX_1 has the value 'H' and COLX_2 has the value 'P', COLA_1 will contain 'H Some Text P'.

```
COLA_1 := COLX_1 & ' Some Text ' & COLX_2;
```

Read Attributes from Related Element

Each X has at most one related instance in Y

In this example an instance in X reaches the instance in Y by navigating the toY relationship. This sets the attribute called COLA_1 to the value of COLY_1 in element Y.

```
COLA_1 := toY.COLY_1;
```

Each X can have many related rows in Z

In this example an instance in element X can have many instances in Z which are reached via the iToZ inverse relationship. Copying an attribute requires the user to specify which one is wanted with an index. This reads the COLZ_1 attribute of the first item in the collection of related Z instances.

```
COLA_1 := iToZ[1].COLZ_1;
```

If the source is a database then the answer is nondeterministic and therefore depends on the order the database returns the collection. To make this deterministic an ORDER BY clause can be placed on the relationship iToZ. This technique could be used to identify the most recent instance by ordering in descending order on a date/time column.

Combined with Expressions

This example demonstrates an assignment using expressions.

```
COLA_1 := toY.COLY_1 & ' and ' & toY.COLY_2;
```

Paths can be longer than just one relationship and an attribute. The code below is taken from a transform called **Z to C**. The right hand side assignment path navigates from Z to X and on to Y to read the COLY_1 attribute.

```
COLC_1 := toX.toY.COLY_1
```

Write Attributes in a Related Element

Assignments let a user construct related elements. In the example below there are three assignments using the relationship toB.

```
toB.B_KEY := X_KEY;
toB.COLB_1 := COLX_1;
toB.COLB_2 := 'UPDATED';
```

Because the maximum cardinality of toB is 1, all the assignments refer to the same related row in B and because B_KEY is the primary key for B, a find or create is performed to find or insert an instance in B where the B_KEY column has the value of the source X_KEY.

If it is not found and an instance is inserted into B then COLB_1 and COLB_2 are inserted along with the B_KEY. If it is found, then COLB_1 and COLB_2 are updated with the values specified in the assignments above.

If the instance of a one to many relationship, a named index can be used to denote which assignments refer to the same row.

```
iToC[row1].C_KEY := X_KEY + 1000;
iToC[row1].COLC_1 := COLX_1;

iToC[row2].C_KEY := X_KEY + 2000;
iToC[row2].COLC_1 := COLX_2;
```

The first two assignments refer to the same row in C, which is found or created based on the primary key assignment. They are tied because the index name, row1, is the same on both lines. The last two assignments are also tied, and refer to a different row to the first two. If the assignments are in a loop, the index name ties assignments together per iteration. See loops for more information.

As with right hand source paths, it is possible to have more than one relationship in a left hand target path. It is not common since the normal task is to assign attributes to each related element.

Relationship Assignments (Cascades)

SML lets a user assign a relationship to a relationship.

```
toB := toY;
```

When both sides of an assignment are relationships, it is necessary to provide more information to describe how to perform the assignment. In the example above the toB relationship refers to the related B element in the target model, and toY the related Y element in the source. So the above statement means make an instance of B from the related Y. Then ensure the relationship from A to B in the target data store relationship is correctly set up. If the target is a database, for example, make sure the primary key and foreign key columns contain the correct values.

The requirement to "make a B from a Y" is the key. A transform is the mechanism used in Transformation Manager to describe how to copy elements. So a transform Y to B must be provided. Transformation Manager will call it, passing in the related Y to construct the B instance, and then set up the relationship from the current A to the constructed B.

toB and toY both have a maximum cardinality of 1, so the transform will be called at most once, and not at all if there is no related Y.

If the maximum cardinality is many, the cascade transform is called once for each related source instance. The target relationship is set up for each constructed instance.

```
iToC := iToZ;
```

Again, if there are no related rows to process, the assignment does nothing. If the source has a maximum cardinality of 1 and the target is many, then the cascade call is legal. At most one related element will be constructed.

```
iToC := toY;
```

If the source has a maximum cardinality of many and the target is one, a direct assignment is illegal (by default).

```
toB := iToZ;
```

The problem described is where a transform is putting many related elements into the target where only one can go. This is resolved by specifying which source element is wanted with an index.

```
toB := iToZ[2];
```

Identification

In general, identification is relatively simple. Transformation Manager has a number of built-in facilities which control the finding or creation of objects in the target. These include operators on individual attributes or relationships together with the identification information present in the data model itself.

Operators Used in Identification

The table below summarises the key operators used in identification. This method of using operators is used because the left hand target side of an assignment path can consist of multiple components. A user can annotate any element on that path to change its identifying behaviour.

There is no ambiguity with the use of these operators, for example, multiplication. The find or create operator, *(asterisk), precedes a target attribute or relationship, where multiplication is not allowed and has no logical role.

Operator	Meaning
*(asterisk)	Find or Create identification means that an attribute or a relationship will form part of the identification of the target instance. It helps control whether an insert or update is used.
-(minus)	The attribute/relationship does not form part of the identification of the instance, even if it is the primary key. Use this to remove the identifying role for the primary key attribute, or an identifying relationship.
!(exclamation)	Find only identification. A find will occur, but if the item is not found, it will not be created.
+(plus)	Update only identification. Allows a row to be updated but not for a new one to be created.

In the example transform below are two assignments from a source element called Account to a target element called Account, in which there are identifying attributes on the target side i.e. customer number identifies 'Account', `*customerNum`.

```
*customerNum := customerNum;
balance      := 100 + balance;
```

At run-time this will find or create 'Account' elements using `customerNum` as the identifying key and will then set the balance from the source side, having added 100. However various different systems, broadly categorised into hierarchical and non-hierarchical in type, impose different limitations on the kind of identification which can be meaningfully carried out.

Non-hierarchical systems

Non-hierarchical systems include relational databases, JDBC using Transformation Managers built-in Java bean adaptor and non-hierarchical generic systems, for example a target adaptor writing to comma separated variable (CSV) format. In these systems it is usually possible to create 'Account' items entirely in accordance with the identification information - the only limitations tend to occur in JDBC.

For the example above each instance of an 'Account' is represented by a row in a database. Insertion or update of the row may not be possible under certain circumstances. For example, where the table has primary key or foreign key constraints which have been ignored in the transform because the transform author has used the -(minus) symbol to cancel built in identification or because the metadata loaded into TM Designer for the model is out of step with the real target and therefore does not include all the constraints which are present in the database itself.

Hierarchical systems

Hierarchical systems can be more complex and the hierarchical structure of the data - which in XML is visible in a text editor - may conflict with or modify the behaviour of the identification. The issue to consider is the extent to which identification is within context in the hierarchy. This can be demonstrated in the example below where the target data store already has the following structure built with two bank branches and customer accounts created in both banks.

```
<Bank>
        <Branch ref='456712'>
                <Account customerNum = '125678' balance ='233'/>
                <Account customerNum = '445678' balance ='133'/>
        </Branch>
        <Branch ref='456713'>
                <Account customerNum = '676768' balance ='233'/>
                <Account customerNum = '565566' balance ='133'/>
        </Branch>
<Bank>
```

In this example the next `customerNum` to arrive is `676768` which already exists in bank branch `456713`. There are now two reasonable ways to interpret the identification.

1) `customerNum` is a universal identifier. If this is the case then the existing Account node will be updated.

2) `customerNum` only identifies accounts within the particular branch. In this case a new customer may need to be added to the first branch.

Transformation Manager can allow either interpretation, both being potentially valid and applicable. The way this is resolved in practice is as follows.

1) In the case that the transform **Account to Account** is called as a result of a cascade operation to its natural XML parent i.e. in this case Branch, then the identification is assumed to be in the context of its parent. In other words there may well be more than one Account in the resulting file with the same customer number.

To obtain the effect of being within the context of the parent, an independent and dependent transform will be used with the following transform code. The first transform code block is the independent transform.

```
Account := Account;
```

The second transform code block, shown below, is the dependent transform.

```
*customerNum := customerNum ;
balance := 100 + balance ;
```

2) Otherwise Transformation Manager assumes that identification is universal or that there should only be one Account with each customer number.

The following example obtains universal identification as an independent transform in its own right and is the second transform code block from above.

```
*customerNum := customerNum ;
balance := 100 + balance ;
```

The example below shows how to complete the transform as an independent transform ensuring that the bank branch transform also creates the account records.

```
FOR_EACH Individual_Account over Account;
BEGIN
        Account[idx].*customerNum := Individual_Account.customerNum;
        Account[idx].balance      := 100 + Individual_Account.balance;
END;
```

Finally, if identification within context using an iterator is required in the second type of map, then use the special function `GETTARGETREF` to indicate that Account is identified within the context of the Branch. This is shown in the transform below.

```
FOR_EACH Individual_Account over Account;
BEGIN
        Account[idx].*customerNum := Individual_Account.customerNum;
        Account[idx].*iBranch     := GETTARGETREF();
        Account[idx].balance      := 100 + Individual_Account.balance;
END;
```

The function `ISSEENBEFORE` is a useful supplement or replacement for identification especially for XML targets. This `ISSEENBEFORE` function enables the transform to act differently for each new item. The special attribute `$id` can be used to provide identification, and hence control over the number of items created, for any node which in reality has no identifying attributes as in the example below. The `$id` attribute will not appear in the final output as it is a pseudo-attribute (p. 10).

```
*$id := CustomerNum;
```

Mixing identification and non-identification within the same transform is not a supported design pattern and will lead to errors as in the example below.

```
IF ( some condition) THEN
        *customerNum := customerNum;
ELSE
        customerNum  := newNumberFunction();
END_IF;
        balance      := 100 + balance ;
```

To achieve this effect the mode (p. 39) property of a transform should be used to divide the transform route into two components. In the example below an independent transform will use two dependent transforms to process the source data depending on whether the record is a new record or an update to an existing record. The dependent transforms will have mode labels set. These will be **ID** and **New** and the required dependent transform will be called depending on the result of the initial independent transform which is shown in the example below. Note the use of the PushMode function.

```
IF (some condition) THEN
        Account  := Account + PushMode('ID');
ELSE
        Account  := Account + PushMode('New');
END;
```

The two dependent transforms, shown below, will be called depending on the calling transform above and the resulting PushMode value, **ID** or **New** which have also been set as the **Mode** values. The first transform is called for the **ID** **Mode** and the second transform is called for the **New** **Mode**.

```
*customerNum := customerNum;
balance      := 100 + balance;
```

```
customerNum  := newNumberFunction();
balance      := 100 + balance ;
```

This assumes that the condition can be evaluated for each branch. If the condition is to be evaluated for each account then the pattern in the Branch transform will be similar to that shown below.

```
FOR_EACH Individual_Account over Account;
BEGIN
IF  (Individual_Account. some condition) THEN
        Account := Individual_Account + PushMode('ID');
ELSE
        Account := Individual_Account + PushMode('New');
END;
```

Data Model Properties Used in Identification

Data models will have default identification schemes. For example, in a relational database which is a non-hierarchical system, the default identification scheme uses primary key values while in XML which is a hierarchical system, the document structure defines the scheme. Some data models may not have an intrinsic identification scheme at all – an Excel spreadsheet, for example.

Relational Database Data Models

In a relational database model there may be occasions when the primary key needs to be excluded from the identification process. For example, the primary key is a surrogate key with its value taken from a sequence or some other unique value generator. Identifying an instance in an element based on a guaranteed freshly obtained unique value is destined for failure as an instance creation will always ensue. On such occasions, it is necessary to indicate which attributes are required for locating existing instances.

When a relational database model is loaded the primary key property of an attribute is automatically detected in the load process. A user may be tempted to think that editing the primary keys in a data model is a good way to indicate which attributes are identifying. However ETL Solutions does not advise this approach for specific projects or transforms. The reason is that the identifying attributes for one project or transform may be different to those used in other circumstances. Because data models are shared between all projects and their transforms within a repository this can give rise to unexpected results when other projects using the same data model are executed later.

Identifying Attributes

In the representation below, element A has two columns, A_KEY and COLA_1. The A_KEY attribute is a primary key and COLA_1 has no identification role in the data model. The first scenario will make COLA_1 identifying using the find or create *(asterisk) operator which means the transform code will search for an instance where the attribute COLA_1 has the value TEST and the attribute A_KEY has a value of 34. A_KEY remains identifying because it is the primary key and it has no modifying operator.

```
A_KEY := 34;
*COLA_1 := 'TEST';
```

This transform code will insert a new instance in element A if there is no record that satisfies both of these assignments. If a record is found then other attributes that are part of the transform will be updated.

The second scenario describes a situation where the primary key attribute A_KEY is modified with the -(minus) operator so it is not an identifying attribute.

```
-A_KEY := 34;
COLA_1 := 'TEST';
```

As there are no identifying attributes, an attempt is always made to create or insert a new item. If A_KEY is a primary key in the target data store then a constraint violation will occur when the instance with a duplicate value of 34 is inserted.

The following scenario only makes sense in a dependent transform where the identification performed is only finding an instance with the !(exclamation) operator.

```
!B_KEY := 34;
```

This assignment will attempt to find an instance of element B where the primary key is 34, but if none is found no insert will be found. If it is found, the calling transform will set up the relationship so that the located row is referenced correctly using the equality of the value 34.

The last scenario in this section sets the identification of a related item.

```
toB.B_KEY := 34;
toB.*COLB_1 := 'TEST';
```

The transform code here means find or create a B where the B_KEY is 34 and COLB_1 is TEST, and then set up the relationship to it according to the rules of the toB relationship. In our case, this means setting the FK_B foreign key attribute to 34 – the value of the primary key column it references.

Identifying Relationships

Transformation Managers runtime environment ensures a relationship is correctly constructed when a relationship is used in the left hand target side of an assignment so that the integrity of a relational target data store is maintained. For a database, or similar model type, this may mean setting the underlying foreign key attributes, which in turn means the foreign key attributes are not directly assigned in a transform. The following assignment causes a transform to cascade to another transform.

```
toB := toY;
```

When the specific project is executed an instance of `B` is found or created according to the rules in transform **Y to B**. Relating the instance of `A` then being constructed to the instance of B correctly needs the underlying attribute, FK_B, set so it matches the primary key of the B instance we just found or created as a result of the cascade call.

If `FK_B` was to form part of the identification of `A` then the following transform code would assign it directly.

```
*FK_B := toY.Y_KEY;
```

However, the assignment can be written using only the relationship assignment by making the relationship identifying as shown below.

```
*toB := toY;
```

The relationship assignment will cascade to the **Y to B** transform as usual. The transform will create or find an instance of `B` and set up the relationship to it by making `FK_B` match `B`'s primary key. This value is used as part of the identification of `A` as though it had been modified directly onto `FK_B` using the *(asterisk) operator..

Relationships based on multiple attributes being equal will have identification roles applied to all the attributes that identify the relationship. For example in PPDM a relationship to AREA references the AREA_ID and AREA_TYPE attributes, because the primary key of AREA is these two columns.

Identification for non Database Targets

Transformation Managers runtime creates a new data store for most model types other than database data stores. For XML, Excel or files, it creates a new document.

Identification is still used and is important. As a project executes element instances are constructed and identification can be used to avoid duplicates in the same way that they can for a database data store. The target data is still queried for existing instances based on the identifying attributes and relationships. The only difference is that the projects transforms will only find previously created instances for this run of the project.

The database adapter copes with data that exists before the project starts, and with instances that are constructed as the transform runs. The identification specified in the transform is affected but the potential for duplication does arise.

Built-in Functions

Transformation Manager includes built-in functions to perform most common operations, including text manipulation, mathematical functions, date/time handling and logging.

More specialized requirements are also covered, for example XML specific functions and cascade functions to read properties of the currently executing transform.

To use a function, type its name, and a comma separated list of parameters enclosed in brackets.

For example, the SUBSTRING function takes 3 parameters, and could be used as follows:

```
countyCode := SUBSTRING(APIWellNumber, 1, 3);
```

Functions which return no value are typically seen in a stand-alone statement.

```
PRINTLN('Processing well '& UWI);
```

Functions can be nested. You can extract from the 3rd character to all but the last two characters using the SUBSTRING function combined with the LENGTH function:

```
part := SUBSTRING(APIWellNumber,3,LENGTH(APIWellNumber) - 1);
```

Access the Built-in Functions using the Code Completion CTRL+SPACE option within the editor pane as mentioned in Chapter 4.1.3 – Auto Completion for Functions, Lookups and Coding Patterns above.

Functions do not change parameters. The SUBSTRING function returns a new String; it does not alter the original String.

Comments

Comments can be added to transform code as single commented lines or as comment blocks.

Surrogate Database Keys

Most advanced database systems use artificial or surrogate key values for both primary and foreign keys. Such key values are only meaningful inside the database instance and have no meaning in business terms. Surrogate keys are used to provide a unique value, for example so a foreign key can easily reference it. Invariably each database system supplies some mechanism for producing such keys. Database systems generally use two types of mechanism to provide surrogate key values although there are others available.

- A dedicated sequence
- A dedicated table

SML (p. 59) provides special facilities to cover all options including a simple default mechanism for writing into a database that will only be used by Transformation Manager itself. The only difference between the systems is in the reserved words used. The default system uses $UniqueKey. The more complex system may be invoked using the UniqueKey function.

```
UniqueKey('Model',0,'getkeyNext')
```

`$UniqueKey` is a pseudonym for `UniqueKey()`.

PPDM and Surrogate Keys

A common method to obtain surrogate key values in an Oracle database is to use an Oracle sequence although Transformation Manager does not have direct support for Oracle sequences. PPDM by default does not include Oracle sequences but they can be added by the user if required. An Oracle sequence can be used to provide a surrogate key value with the use of an external function or a direct SQL query. If a SQL query is used it can be placed in a procedure for re-use.

When a target element has a surrogate key, it often has either implicitly or explicitly a natural key too. For example, PPDM_GUID can be used as a surrogate key taking its value from an Oracle sequence. The code below initially shows how this may be expressed in a transform.

```
*PPDM_GUID := GetOracleSequenceNumber();
*UWI := SOURCE_WELL_NUMBER;
WELL_NAME := SOURCE_NAME; -- we want this to be updated if the row already
exists.
```

The required behaviour is as follows, if the row does not already exist, insert it, then set the PPDM_GUID value to the unique sequence number calculated and the UWI attribute to the value in the SOURCE_WELL_NUMBER attribute. However, if the row already exists, update the WELL_NAME. Do not set the PPDM_GUID – as a surrogate key it is likely to be used in relationships. Either a constraint violation will occur, or if no constraint has been applied, the relationships will simply break.

There is however a problem with the code above. It will always perform a create action. It will never find an existing element instance based on the identification using PPDM_GUID and UWI because PPDM_GUID is always assigned a unique value. This is addressed by calling the SURROGATEKEY function in the assignment statement. The corrected code is shown below with a procedure called GetOracleSequenceNumber() added to the first line of code.

```
*PPDM_GUID := SURROGATEKEY(GetOracleSequenceNumber());
*UWI := SOURCE_WELL_NUMBER;
WELL_NAME := SOURCE_NAME;
```

It is a requirement that the attribute PPDM_GUID is an identifying attribute by using the *(asterisk) which will correctly implement the desired behaviour. The UWI attribute, if part of the primary key, does not strictly need the *(asterisk). It is shown because both the surrogate and natural keys need to be identifying and often only one of them forms the primary key.

In the PPDM tutorials three methods are used to generate "unique" values; only one of them uses SURROGATEKEY. In the first tutorial a global variable counter and the $UniqueKey pseudo attribute are used to generate values but they do not generate genuine guaranteed unique values – they start at the same value each time you run the project. In the second tutorial SURROGATEKEY is demonstrated together with an Oracle sequence.

Interactions between Primary and Secondary Keys

TM Designer supports many different styles of interaction with primary and foreign keys. Below are some general recommendations although specific circumstances may render these recommendations inappropriate.

When reading from a database do not read the values of primary or foreign keys. Allow Transformation Manager to navigate the implied relationships directly. It is unlikely that the actual instance values in primary surrogate keys should be copied to the target.

When writing to a database there are two strategies. The first allows TM Designer to automatically supply primary key values invoking the repository setting called **Automatically Supply RDBMS PKeys** with a tick. The second explicitly sets all primary key values. Irrespective of which of these two approaches is used do not write directly to foreign key values. If the database contains any foreign key constraints then such an approach will almost always fail. Even if the database does not contain foreign key constraints writing to foreign key values directly will duplicate TM Designers internal logic for setting relationships and probably impact performance and lead to subtle conflicts between TM Designer's logic and the explicit SML (p. 59) code in your transform which may cause fatal errors at run-time.

SML provides a special keyword, $UniqueKey and a special function, UniqueKey for providing artificial values. Be aware that such values must be assigned directly to attributes in the database. It is quite incorrect to use UniqueKey values for any other purpose, for example to assign a value to a local variable. This is because the relational database target adapter does not use $UniqueKey to actually find a row in the database, only to re-find the same row.

Using the Self Relationship in SML

The Self relationship is used in the same way as other relationships are in an assignment.

In the example below the starting point is a transform called **Y to X** with the code `self to z;`. Here the self relationship is used on the target side of the assignment with an element called Z on the source side. The self relationship refers back to the X element which is the source element of the transform. The result is that a transform with the elements X and Z will be called as in **Z to X**.

```
self := Z;
```

The reverse of this code as shown below has an alternative affect. In this case the **Y to X** transform shows the self relationship referring to the **Y** element. The result is that a transform with the elements Y and Z will be called as in **Y to Z**.

```
Z := Self;
```

- The cascade target of a self transform as used in the first example above in the transform **Z to X** must not try to identify **X** because this must have already happened in the **Y to X** transform. This is a logical consequence of the re-use of **X**.
- The current release of Transformation Manager will issue a warning message for such transform telling you it is not identified – this warning may be safely ignored.

If you want the transform **Z to X** to be called under other circumstances then use the following approach.

```
IF GETCALLSTACKTARGET(0)='X' THEN
-- The transform was called from self
ELSE
-- The transform was called from somewhere else
END_IF;
```

See Also: Self Relationship (p. 33)

TM Migrator

Introduction

TM Migrator provides an environment in which generated transforms can be executed. It has a Graphical User Interface (GUI) and it can also be invoked from a command line. The GUI is provided primarily for the purpose of testing transform projects created with TM Designer. The transformation programs produced would typically be deployed either as 'stand-alone' utilities with the use of TM Migrator and short-lived deployment or as components in some larger system in long-lived mode although the flexibility of Transformation Manager can provide many variations on these basic approaches depending on specific requirements. TM Migrator can be started from a command prompt with the filename of a transform pack passed as its argument. This will open the transform pack within TM Migrator allowing you to edit its properties and/or execute it.

The following sections describe how TM Migrator is used to transform data using programs developed with TM Designer. It includes instructions for the following areas.

- Starting and exiting TM Migrator
- Running a project
- Restoring a project
- Opening a project
- Opening a deployment pack
- User Settings
- Exporting as a deployment pack
- Working with Multiple Sources and targets

Projects can be deployed via transform packs, the command line interface or a Java API.

TM Designer Menu Reference

This section includes descriptions of all of the options on all of the menus in the TM Designer GUI. Each option is described in left-to-right, top-to-bottom order.

- Project Menu
- View Menu
- Run Menu
- Source Menu

Note that the menus are context-sensitive and therefore the options available depend on what you are doing. For example, options on the Source menu are only available when an XML document has been edited in the Source pane.