

WHITE PAPER

IMPORTING FILES WITH METAEDIT+ GENERATORS

MetaCase

Ylistönmäentie 31

FI-40500 Jyväskylä, Finland

Phone +358 400 648 606

E-mail: info@metacase.com

WWW: <http://www.metacase.com>

IMPORTING FILES WITH METAEDIT+ GENERATORS

Abstract

This paper describes how MetaEdit+ generator system can be used to reverse engineer existing data from files into models. While MetaEdit+ Reporting Language (MERL) is made for defining generators it also provides functionality for reverse engineering.

1 DOMAIN-SPECIFIC MODELING

Domain-Specific Modeling (DSM) raises the level of abstraction beyond programming by specifying the solution directly using domain concepts. Rather than creating all models by applying the modelling language, it can be sometimes feasible to import part of those models from external sources. These could be requirements, parameters, interface definitions or other library elements used in modelling.

For importing MetaEdit+ provides an XML format and a programmatic API, but the generator system's MERL language can also be used to parse and import data from external files. MERL is made for generating (turning models into text) rather than for parsing and importing (turning text into models). For complex textual input languages a proper parser would work better, but MERL works fine on simpler formats like CSV files or consistently formatted text or code. In this paper we describe how code stored in files can be imported into MetaEdit+ as models.

2 METAEDIT+ MERL FOR IMPORTING

MERL, the MetaEdit+ Reporting Language, provides functionalities to read external files, perform simple parsing on them, and output XML model files to be imported into MetaEdit+. The full MERL language is described in the Workbench User's Guide, and here we describe the main steps via an example. The steps are:

- Choosing files to be imported
- Parsing them to select data to be imported
- Creating MXM file(s) including the imported data
- Reading the MXM file into MetaEdit+

3 PHASES OF IMPORTING

To avoid learning a particular domain and input format we use a wider known example: Importing Java classes to UML Class Diagrams. You can find the same example from the demo repository's UML project. Open a Generator Editor for Class Diagrams and see the generator called 'Reverse engineer Java' for details. You may also use it by running it on some example Java files.

3.1 Choosing files to be imported

Selecting files to be imported can be done either by:

- 1) Setting a certain location and then reading all files from this location (variable 'dd'). This is detailed for Windows platform in a subgenerator '_Reverse engineer Java dir' available in the demo repository.

```
/* By default, reverse engineer the default reports directory */
variable 'dd' write
  subreport '_default directory' run
close
```

- 2) Explicitly asking for the file to be imported with the askFilename command.

```
/* Ask file to be imported */
local 'importedFile' write
  prompt 'Choose file to be imported' askFilename
close

/* Read file into a variable called 'file' */
variable 'file' write
  filename @importedFile encoding 'UTF-8' read
close
```

In the case of selecting a single file, other files can also be read based on the data given in the imported file(s), e.g. by recognizing `include` statements in the file and parsing those files too.

Before reading the files, it is also good to specify housekeeping like what to do if they are not available or which encoding to use when reading them.

3.2 Parsing files to select data to be imported

MERL-based parsing generally works by analyzing files per line. If the imported file follows a comma separated format or similar, then reading it into variables is straightforward with a translator. Several useful predefined translators can be found from the '_translators' generator, which is defined for 'Graph' itself and thus can be applied in all modeling languages (graph types) in MetaEdit+. You can also define your

own translators for more specific parsing. Below, an example translator to translate semicolons into newlines is defined (note the \ escape before the newline) and applied.

```
/* translates semicolon into newline */
to '%semColToNewline
; \
'
endto

@i = '1' /* local variable name, initial value '1' */
do id%semColToNewline {
    local @i++ write id close
}
```

If the file follows another format, then MERL parser code needs to take account of that format. In the case of parsing Java, the format via sample content is shown below:

```
// Stores one of the top scores and the player who achieved it.
public class TopScore {
    public String playerName;    // Name of the player.
    public int score;
}
```

Listing 1. Sample Java code to be imported

The example below shows MERL parser to identify values for attributes from a Java class. Line 01 translates the white spaces and = signs in the line into newlines, giving us a token per line, and these are then individually analyzed. If the line is a comment that is recognized (line 02) and the rest of that line is not analyzed further (line 03).

```
01 do id%spacesAndEquals
02 { if id =~ '/*' then $comment++%null endif
03   if (id and not $comment) then
04     if id =/ '(public|protected|private|static|final|transient|volatile)' then
05       if id =/ '(public|private|protected)' then
06         $visibility=id
07       endif
08       if id='static' then $scope='class' endif
09       if id='final' then $eaccess='readonly' endif
10     else
11       $token++%null
12       /* Sets the attribute type as type variable */
13       if $token='1' then
14         $type = id
15       endif
16       /* Sets the attribute name as element variable */
17       if $token='2' then
18         $element = id%strip
19         subreport '_c_element' run
21       endif
22     endif
23   endif
24 }
25 $comment=''
```

At line 04 a keyword from the regular expression is recognized, and then line 05 finds the visibility and stores it to a variable in line 06. In a similar way, variables are used to store scope (line 08) and access (line 09) if these have been set in that line of Java.

If the line did not include any of the expected keywords, then \$token, which stores the phase of class parsing, is incremented (line 11). If \$token is 1 (line 13), we have the datatype of the attribute and this value is stored to variable \$type (line 14). If \$token is 2, we have an attribute name which is stored to variable \$element (line 18). Once the attribute name is found a sub-generator ‘_c_element’ is called.

The full code for parsing the class can be found from the generator called ‘_class’ in the demo repository.

3.3 Creating MXM file(s) including the imported data

The sub-generator ‘_c_element’ outputs the values we saved in MERL variables in the previous parsing step, into the MetaEdit+ XML format for Models (MXM). This format is detailed in the Workbench User’s Guide.

A good practice to identify the names and format that the created MXM file should follow is to create an equivalent model manually in MetaEdit+ (see figure below) and then export it as an MXM file. This file then serves as a reference and test case for creating MXM during import.

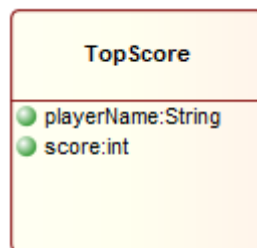


Figure 1. Reference example in MetaEdit+ (see listing 1)

How the ‘_c_element’ generator outputs a UML Attribute to the MXM file as a model object is shown below. The Attribute definition starts in line 02 and ends in line 38. Its content is the five property values that were extracted and saved to variables during the previous parsing phase. Each property value follows the same structure: the slot definition defines the property’s name followed by its details. Here each property consists of a single string value. The name and data type of the Attribute are always written to the MXM, but the others are only written if a value for them was saved to the respective variable during parsing. (When omitted, that property will take the default value defined for it in the modeling language.)

```

01 Report '_c_element'
02 '<object type="Attribute [UML]">
03     <slot name="Name">
04         <value>
05             <string>' $element$xml '</string>
06         </value>
07     </slot>
08     <slot name="Data type">
09         <value>
10             <string>' $type$xml '</string>
11         </value>
12     </slot>
13 '
14 if $evisibility then
15 '     <slot name="Visibility">
16         <value>
17             <string>' $evisibility '</string>
18         </value>
19     </slot>
20 '
21 endif
22 if $escope then
23 '     <slot name="Scope">
24         <value>
25             <string>' $escope '</string>
26         </value>
27     </slot>
28 '
29 endif
30 if $eaccess then
31 '     <slot name="Access">
32         <value>
33             <string>' $eaccess '</string>
34         </value>
35     </slot>
36 '
37 endif
38 '</object>
39 '
40 endreport

```

By default, a generator outputs its result to an output stream inside MetaEdit+, but for importing it must be stored to an external file in MXM format. This can be done by creating files and storing the content to there. Line 01 sets the file name and encoding then opens the file for writing. Lines 02, 03 and 05 are boilerplate, and line 04 calls the main generator that parses the input files and outputs the result in MXM format.

```

01 filename @importedFile '.mxm' encoding 'UTF-8' write
02 '<?xml version="1.0" encoding="UTF-8"?>' newline
03 '<gxl xmlns="http://www.metacase.com/gxlGOPRR">' newline
04 _Import_file()
05 '</gxl>'
06 close

```

3.4 Reading the MXM file into MetaEdit+

The final phase of the import is to read the resulting MXM file(s) into MetaEdit+. For this purpose, MetaEdit+ offers the `fileInPatch:` command, and this is called from MERL with an `internal...execute` command. The result is that the models in the MXM file are created in MetaEdit+.

```
internal 'fileInPatch: "' @importedFile '.mxm"' execute
```

4 CONCLUDING REMARKS

Models can be imported into MetaEdit+ from various sources. We described here the approach of using MERL to import models from textual files. In the Java case the imported model included only the conceptual data, but the MXM file can also include representation information like size or location of the visual diagram elements. If these can be identified from the imported files or created otherwise during import, the layout information can be included in the import. If not, opening the imported data creates a default representation, and the auto-layout commands in MetaEdit+ can be used.

If you plan to implement your own import with MERL, please see the MetaEdit+ MERL and XML sections in the Workbench User's Guide at: <https://metacase.com/support/55/manuals/>. The Java import case used here can be found from the demo repository and its UML project. Open a Generator Editor for Class Diagrams and see the generator called 'Reverse engineer Java'.