A Practical Tutorial For FermaT and WSL Transformations

DONI PRACNER and ZORAN BUDIMAC, University of Novi Sad

The transformation system FermaT and the language WSL can be very powerfull tools and are successfully used event in some industrial applications. In this paper we try to illustrate a process of making a new transformation and some intermediate steps and ways to test the new functionalities in a simple way.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**] Distribution, Maintenance, and Enhancement

General Terms: Theory, Experimentation

Additional Key Words and Phrases: software evolution, FermaT, WSL, transformations

1. INTRODUCTION

Software evolution is mainly about how program code changes in its environment. A very important part of these changes is the re-engineering of software that can be done very efficiently and more importantly reliably by using tools that offer formally provable transformations.

FermaT is the current implementation of the language WSL (short for *Wide Spectrum Language*) and the surrounding code transformation libraries. It is available under the GPL v3 software licence and works on most computer platforms, including Linux, Windows and Mac OS. Early versions of this tool were developed as “The Maintainer’s assistant”[Ward 1989] and it has been developed and reimplemented since then. It has been used in several industrial projects of converting legacy assembly code to human understandable and maintainable C and COBOL[Ward 1999][Ward 2004][Ward et al. 2004][Ward 2013]. It also has support for program slicing[Ward and Zedan 2017] and can be used to derive program code from abstract specifications[Ward and Zedan 2014]. A companion graphical application FermaT Maintenance Environment (FME)[Ladkau 2007] is also available and can be very useful especially for initial experiment with the transformation system. This tutorial will give just a brief overview of WSL and some of the needed ideas and will not go into depths with the syntax of the language that is available in the official manual[Ward et al. 2008].

This paper is organised as follows: Section 2 shows some aspects of FermaT and how to use it and expand it, mainly through the expression and condition simplifiers (Section 2.1. A working example is introduced in Section 2.2, which is then developed into a full transformation that can be added to the system in Section 2.3. Finally Section 3 gives a brief conclusion to the paper. Along the way Some

This work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support";
Author’s address: Doni Pracner, Zoran Budimac, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: doni.pracner@dmi.uns.ac.rs, zjb@dmi.uns.ac.rs

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, 11-13.9.2017, Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

ways of checking the results of transformations will also be presented, as well as ways to find more information on the structure of the programs, both of which are very important for beginners.

2. WORKING WITH META-WSL

WSL has the ability to work with programs written in WSL. This part of the language is called *MetaWSL*. A piece of code under inspection is represented by an abstract syntax tree (AST), or more precisely in the current implementation by a list that can have lists as its elements. Meta-WSL procedures know how to handle these lists and what to expect in them when they represent a valid program. Items in the lists have associated general and specific types. For instance a while loop will be of type `T_While` and will contain in its list a `T_Condition` and `T_Statements`. An assignment is of type `T_Assignment` and holds one or more items of type `T_Assign` (it is separated because it is used in other places as well, and there can be multiple simultaneous assignments) which in turn holds an `T_Lvalue` and a `T_Expression`. Tables with these relations can be found in the reference manual for WSL[Ward et al. 2008], or can be found by directly extracting the information from the source code with a command like shown in Figure 1.

```
> grep "\[T_Assign\]" $Fermat/src/adt/WSL-init.wsl

Syntax_Name[T_Assign]             := "Assign";
Syntax_Comps[T_Assign]             := <T_Lvalue, T_Expression>;
```

Fig. 1. Grepping component types in WSL

To assist many operations, especially transformations, there is also a concept of the current program that is being worked on (accessible via the `@Program` procedure), the current position in it (represented by a list of indexes in the lists starting from the top one, obtained by `@Posn`) and the current item (`@I`) in it. There are many built in procedures to move around in the program and to change it. The current program can be specified with `@New_Program` and to understand better what are the components of particular statements, the `@Print_WSL` procedure can be used, as shown in Figure 2. To print the code of an item there is the pretty print `@PP_Item` procedure, that accepts three parameters: the item, the width of the maximum line, and the file to print out the code to. If the filename is an empty string it will be printed out to the standard output. When working with the whole program, a convenient shorthand is the `@Checkpoint` procedure which accepts only one parameter, the file name, and always prints the whole program in 80 characters width.

2.1 Item simplification

One of common things done in transforming programs is to detect specific patterns and simplify them. WSL has matching constructs for this type of work. To demonstrate a simple example we'll try to make a matcher for calling absolute values on negated values and replace them with just the value (of course this is already present in FermaT). To quickly test our matcher, we can create a small program that will define an entry to test on, use the checkpoint command to display the code before and after the changes, and use the `FOREACH` construct to apply our matcher to all expressions. This is demonstrated in Figure 3. Alternatively we could use commands to move the current position (such as `@Down`, `@Right`, etc) in the program to an appropriate expression and apply it just there.

Displaying the changed program, or saving it in a file gives a good indication whether the changes were successful, but additional checks can and should be applied especially when prototyping. This can be done with `@Syntax_OK?`, as shown in the end of Figure 3. This can catch some subtle errors.

Program:

```
@New_Program(FILL Statements
a := 5;
b := 10;
PRINT(a + b)
ENDFILL);

@Print_WSL(@Program, "")
```

Output:

```
Statements
   Assignment
   :  Assign
   :     Var_Lvalue a
   :     Number 5
   Assignment
   :  Assign
   :     Var_Lvalue b
   :     Number 10
   Print
   :  Expressions
   :     Plus
   :        Variable a
   :        Variable b
```

Fig. 2. Creating a new program and viewing its structure

```
@New_Program(FILL Statements
a:=5; b:=-5; PRINT(ABS(-a));PRINT(ABS(-b))
ENDFILL);

@Checkpoint("");

PRINT("transforming");

FOREACH Expression DO
    IFMATCH Expression ABS(- ~?x)
    THEN @Paste_Over(FILL Expression ABS(~?x) ENDFILL) ENDMATCH;
OD;

@Checkpoint("");
IF @Syntax_OK?(@Program) THEN
        PRINT("Syntax OK")
ELSE
        ERROR("Bad syntax") FI
```

Fig. 3. Absolute value expression matching

For instance if in our paste over command we used “Expressions” instead of “Expression”, the output would look appropriate and the saved file would actually be valid, but the syntax tree would have a problem, and further transformations would likely fail. Calling `@Syntax_OK` on this version of the code would result in a report shown in Figure 4.

FermaT has a built in maths simplifier and several procedures that rely on it. Procedures such as `Simplify`, `Simplify_Cond` and `Simplify_Expn` return a new simplified item, while others like `@Or`, `@And` will first combine conditions and try to simplify them together. More information about the simplifier and adding patterns to it can be found in the manual and the documentation that comes with FermaT, more precisely the document `doc/adding-patterns.txt`.

```
Expressions
   Expressions
      Abs
         Variable a
Bad type at (3 1 1)
Gen type is: Expressions(10)  Should be: Expression(2)
```

Fig. 4. Example of a bad syntax report

2.2 Example: Converting Numeric Codes to Strings

As a working example we will work on a relatively simple and understandable problem. One of the tools that generates WSL code works on bytecode that has numeric codes for characters and handles that by creating `@List_To_String` calls with the appropriate numbers[Pracner and Budimac 2017]. To make the code more understandable to humans and more compact in general, we will try to transform those into strings.

The first version (Figure 5) assumes that everything in a `@List_To_String` is a number literal and will not do anything if this is not the case. If all the items in the list are numbers, we can apply the `MAP` function to the list, and convert the items to their values (`@V`), convert those to a string with `@List_To_String` and finally make a valid string item with `@Make`. This is then pasted over the initial expression. As can be seen from the output shown in Figure 5 this means that if a numeric variable is in the list, the code will not transform it at all. In this example a constant propagation transformation executed before this code would actually solve the problem, but in a general case, the variable "a" could be a user input, or a procedure parameter, and might not be replaceable.

```
@New_Program(FILL Statements a := 92;
   PRINT(@List_To_String(<65>));
   PRINT(@List_To_String(<65, 66>));
   PRINT(-a);
   PRINT(@List_To_String(<a>));
   PRINT(@List_To_String(<a, 67>))
              ENDFILL);
@Checkpoint("");
PRINT("transforming ----------");
FOREACH Expression DO VAR < x := < >, str := "" >:
  IFMATCH Expression @List_To_String(<~*x>)
    THEN IS_OK := 1;
    FOR elt IN x DO
      IF @ST(elt) <> T_Number THEN IS_OK := 0 FI OD;
    IF IS_OK = 1
      THEN str := @Make(T_String,
                @List_To_String(MAP("@V", x)), < >);
           @Paste_Over(str) FI
    ELSE SKIP ENDMATCH ENDVAR OD;
PRINT("result ----------");
@Checkpoint("")
```

Output:

```
....
result ----------
a := 92;
PRINT("A");
PRINT("AB");
PRINT(-a);
PRINT(@List_To_String(<a>)
    );
PRINT(@List_To_String(<a,
    67>))
```

Fig. 5. Program that converts lists with only numbers to strings

To have a more fine grained handling of individual items, we need to go through them one by one and store the changed versions in a list (called “res” in the code given in Figure 6). If a numeric code is found it should be converted to a character and added to the list. If the previous item in the list was a string, it should be added to it, otherwise it is added as a new item in the list. In this version we will also include anything that is not a number as a separate entry in the resulting list. The final step, when the list is completed is to paste it over the original expression, but it is important to consider that multiple entries should be concatenated, while a single entry is directly pasted over.

```
FOREACH Expression DO
VAR < x := < >, res := < > >:
  IFMATCH Expression @List_To_String(< ~*x >)
    THEN
      FOR elt IN x DO
       IF @ST(elt) = T_Number
         THEN
            IF NOT EMPTY?(res) AND @ST(HEAD(res)) = T_String THEN
                t := HEAD(res);
                t := @Make(T_String, @V(t) ++ @List_To_String(< @V(elt) >), < >);
                res := TAIL(res);
                res := < t > ++ res
            ELSE
                s := @Make(T_String, @List_To_String(< @V(elt) >), < >);
                res := < s > ++ res
            FI
         ELSE
            res := < FILL Expression @List_To_String(< ~?elt >) ENDFILL > ++ res
         FI
       OD;

        IF LENGTH(res)>1 THEN
                @Paste_Over(@Make(T_Concat, < >, REVERSE(res)))
        ELSE
                @Paste_Over(HEAD(res))
        FI;
       SKIP
    ENDMATCH ENDVAR OD;
```

Fig. 6. Second version of the string converter

It is important to note that while this insistence on converting numbers to characters increases the readability of the code, it can also result in a more complex program with more expressions, which is often not desirable, and is definitely not something to be included in the general simplifier. The actual code that is included in FermaT’s simplifier is shown in Figure 7 and is much more similar to the first version of the code, just much more compact. It also takes care to not convert number 34, which is the code for double quotes and can not be included in a regular string as such, since it is the string delimiter. Further effort could be made handle this case as well, but it is not very common and it would significantly increase the code complexity.

There are also a few other improvements that will be discussed in the next section when the code from Figure 6 will be converted into a full transformation.

```
IFMATCH Expression @List_To_String(<~*x>)
      THEN VAR < OK := 1 >:
           FOR elt IN x DO
             IF @ST(elt) <> T_Number OR @V(elt) = 34 THEN OK := 0 FI OD;
           IF OK = 1
             THEN @Paste_Over(@Make(T_String,
                                    @List_To_String(MAP("@V", x)),
                                    < >)) FI ENDVAR ENDMATCH OD;
```

Fig. 7. String simplification included in FermaT

2.3 Writing A Transformation

Transformations in WSL can be anything that changes the current program while keeping the semantics of the original, with a few potential exceptional cases. For instance there are transformations that will reverse an IF/ELSE statement (for instance to make the condition evaluation simpler), or unroll the first loop of a FOR statement. Additionally there are applicability tests that will check if the semantics would be affected by the specific transformation. For instance there is a transformation that deleted the current item, but only if it is redundant. Therefore the test for this transformation is checking if the current item is redundant.

The main exception to the preservation of full semantics are the slicing transformations, as these by definition preserve only a part of the original behaviour of the program that is relevant to the slicing criterion. These will not be covered in this paper.

The transformations that are built into FermaT are kept in the `src/trans` folder and new ones can be added there and the whole system recompiled. Alternatively it can be added "on the fly" in the working directory using a `patch.tr` file. More details about this can be found in the manual and the documentation that comes with FermaT.

Transformations themselves are represented by two WSL files. The first one holds the code of the transformation. There are two main entry points that need to be defined in this file. The first one is a test procedure that has no parameters and checks whether the transformation can be applied to the current item in the program. It should raise errors with `@Fail` if the transformation is inapplicable, or call `@Pass` otherwise. The other procedure is the actual transformation that receives a single parameter with any potential additional data, for example a rename transformation will receive the old and the new names. Other than these there can be any number of helper procedures defined, and to comply with the definition of a WSL program the file also needs to contain a body for the main program which can be a single `SKIP` instruction.

The second file should be named the same as the first one with a "`_d`" suffix and it holds the description of the transformation and some meta information as well as the names of the actual procedures to test and to apply the transformation from the first file. Figure 8 shows how this file should look for a new transformation that is based on the code shown in the previous section.

Figure 9 shows how the main file with the new transformation should look like. This transformation has no additional procedures. The test procedure just calls `@Pass` always, since there are no specific pre conditions needed for the transformation to be applied and the semantics are not changed anyway – at worst it will not find anything to change and leave the original program as it is. A more zealous version of the test procedure could check if there are any `@List_To_String` items, and even analyse their content to see if the transformations will change anything. On the other hand a general approach to programs that call transformations is to first call the test and then to call the main transformation, which would result in duplicate checks and loss of efficiency.

```
IF EMPTY?(TR_SimplifyChar ) THEN TR_SimplifyChar := @New_TR_Number()  FI;

TRs_Proc_Name[TR_SimplifyChar] := "SimplifyChar" ;
TRs_Test[TR_SimplifyChar]:=!XF funct(@SimplifyChar_Test);
TRs_Code[TR_SimplifyChar]:=!XF funct(@SimplifyChar_Code);
TRs_Name[TR_SimplifyChar] := "Simplify Char";
TRs_Keywords[TR_SimplifyChar] := < "Simplify" > ;
TRs_Help[TR_SimplifyChar] := "Simplify Char will find expressions like '
    @List_To_String(<97>)' and replace them with chars.";
TRs_Prompt[TR_SimplifyChar] := "";
TRs_Data_Gen_Type[TR_SimplifyChar] := ""
```

Fig. 8. Transformation description file

```
MW_PROC @SimplifyChar_Test() ==
  @Pass END;

MW_PROC @SimplifyChar_Code(Data) ==
  FOREACH Expression DO
  IFMATCH Expression @List_To_String(<~*x>)
      THEN VAR < res := < >, IS_OK := 1 >:
      FOR elt IN x DO
        IF @ST(elt) = T_Number
          THEN IF NOT (EMPTY?(res)) AND @ST(HEAD(res)) = T_String
                 THEN res := <@Make(T_String, @V(HEAD(res))
                      ++ @List_To_String(<@V(elt)>),
                      < >)>  ++ TAIL(res)
                 ELSE res := <@Make(T_String,
                      @List_To_String(<@V(elt)>), < >)> ++ res FI
        ELSIF @ST(elt) = T_Variable
          THEN res := <FILL Expression @List_To_String(<~?elt>) ENDFILL>
                      ++ res
          ELSE IS_OK := 0 FI OD;
      IF IS_OK = 1
        THEN IF EMPTY?(res)
               THEN @Paste_Over(@Make(T_String, "", < >))
             ELSIF LENGTH(res) > 1
               THEN @Paste_Over(@Make(T_Concat, < >, REVERSE(res)))
               ELSE @Paste_Over(HEAD(res)) FI FI ENDVAR
      ELSE SKIP ENDMATCH OD END;

SKIP
```

Fig. 9. Transformation main file

The main procedure is very similar to the earlier developed version shown in Figure 6 with some commands being less verbose and less temporary variables used, which in turn can make it harder to read. There are several functional improvements beside that. In a proper program `@List_To_String` should receive a list of numbers and numeric variables, anything other in the list results in undefined behaviour, therefore it is probably best if our transformation leaves any problematic call as it was. This means that the new version only handles numbers and variables, and if the current item is anything else it sets the error flag which will result in no changes being applied. The other thing that is changed is that the new version correctly handles an empty list being passed to the procedure and replaces it with an empty string (the previous version would crash if this was the case).

3. CONCLUSIONS

FermaT and the WSL language offer a powerful transformation library that has been developed for many years, and has been used in industrial applications of re-engineering legacy software[Ward 2004][Ward 2013].

This paper shows some practical steps in working with FermaT and WSL, especially in early prototyping of new ideas. Some aspects of the maths simplifier and full transformations are presented with a concrete understandable example being developed in the process. Attention was also given to ways to check if the written programs output valid WSL code and how to find more information on the structure of programs, which tends to be a big problem for beginners.

REFERENCES

Matthias Ladkau. 2007. *FermaT Maintenance Environment Tutorial*. Technical Report. Software Technology Research Laboratory, De Montfort University, Leicester.

Doni Pracner and Zoran Budimac. 2017. Enabling code transformations with FermaT on simplified bytecode. *Journal of Software: Evolution and Process* 29, 5 (2017), e1857–n/a. DOI: http://dx.doi.org/10.1002/smr.1857

Martin Ward. 1989. *Proving Program Refinements and Transformations*. Ph.D. Dissertation. Oxford University.

Martin Ward. 1999. Assembler to C Migration using the FermaT Transformation System. In *IEEE International Conference on Software Maintenance (ICSM'99)*. IEEE Computer Society Press, 67–76.

Martin Ward. 2004. Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations. *Science of Computer Programming, Special Issue on Program Transformation* 52/1-3 (2004), 213–255. DOI: http://dx.doi.org/dx.doi.org/10.1016/j.scico.2004.03.007

Martin Ward. 2013. Assembler restructuring in FermaT. In *SCAM*. IEEE, 147–156. DOI: http://dx.doi.org/10.1109/SCAM.2013.6648196

Martin Ward, Tim Hardcastle, and Stefan Natelberg. 2008. *WSL Programmer's Reference Manual*.

Martin Ward and Hussein Zedan. 2014. Provably correct derivation of algorithms using FermaT. *Formal Aspects of Computing* 26, 5 (2014), 993–1031. DOI: http://dx.doi.org/10.1007/s00165-013-0287-2

Martin Ward and Hussein Zedan. 2017. The formal semantics of program slicing for nonterminating computations. *Journal of Software: Evolution and Process* 29, 1 (2017), e1803–n/a. DOI: http://dx.doi.org/10.1002/smr.1803 e1803 smr.1803.

Martin Ward, Hussein Zedan, and Tim Hardcastle. 2004. Legacy Assembler Reengineering and Migration. In *ICSM2004, The 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society.