FermaT Maintenance Environment Tutorial

FermaT Maintenance Environment Tutorial

Matthias Ladkau (matthias@ladkau.de)

Abstract

This document gives a brief practical oriented introduction to the FermaT Maintenance Environment (FME). It covers the installation and usage of the program via practical examples.

Contents

1 Installation of the FermaT Maintenance Environment

This chapter gives a guidance through the installation process of the FermaT Maintenance Environment. The installation is explained for Unix/Linux and Windows operating systems.

1.1 Installation on Unix / Linux

1.1.1 Requirements

- Perl (version >= 5.6.1)
  http://www.cpan.org/
- Bit::Vector (A perl module for efficient sets of integers by Steffen Beyer)
  http://search.cpan.org/search?module=Bit::Vector
- Set::IntRange (Perl module based on Bit::Vector for sets of integers in a given range by Steffen Beyer)
  http://search.cpan.org/search?module=Set::IntRange
- gcc or a compatible C compiler
  http://www.gnu.org/software/gcc/gcc.html
- A working JAVA environment (version >= 6)
  http://java.sun.com/
- The "make" command
  http://java.sun.com/

1.1.2 Install of Perl and gcc

Perl and gcc are included in every current linux distribution. See the install instructions of your distributions if these components are not already installed in the standard installation.

1.1.3 Installation of a JAVA environment

On order for the FME to work correctly the Java environment of SUN should be used. The environment can directly optained from the SUN Microsystems as a cost-free download.

- Download the Java **JDK** for Unix/Linux from
  http://java.sun.com/javase/downloads/index.jsp

- Install it according to the provided instructions
- A quick solution is to download the self extracting version (without "-rpm" in the filename) and install it into your `/opt` directory. Create symbolic links in your /usr/bin directory to the `java` and `javac` executables in the `/bin` directory of the extracted java distribution.

1.1.4 Installation of Bit::Vector for perl

- Download Bit::Vector from CPAN
  http://search.cpan.org/search?module=Bit::Vector
- Unpack the archive:

```
tar zxvf Bit-Vector-6.4.tar.gz
```

- Change directory to the unpacked files:

```
cd Bit-Vector-6.4
```

- Make and install the binaries:

```
perl Makefile.PL
make
make install
```

1.1.5 Installation of Set-IntRange for perl

- Download Set::IntRange from CPAN Beyer)
  http://search.cpan.org/search?module=Set::IntRange
- Unpack the archive:

```
tar zxvf Set-IntRange-5.1.tar
```

- Change directory to the unpacked files:

```
cd Set-IntRange-5.1
```

- Make and install the binaries:

```
perl Makefile.PL
make
```

```
make install
```

1.1.6 Installation of FermaT Maintenance Environment

- Unpack the archive (note the directory path must not contain any space characters) :

  ```
  tar zxvf fme.tar.gz
  ```

- Change directory to the unpacked files:

  ```
  cd fme
  ```

- The program should run now by executing the `fme.sh` script

1.2 Installation on Windows

1.2.1 Requirements

- Active Perl (version >= 5.6)
  http://www.activestate.com/ActivePerl/
- Bit::Vector (A perl module for efficient sets of integers by Steffen Beyer)
  http://search.cpan.org/search?module=Bit::Vector
- Set::IntRange (Perl module based on Bit::Vector for sets of integers in a given range by Steffen Beyer)
  http://search.cpan.org/search?module=Set::IntRange
- MinGW package
  http://www.mingw.org/
- Windows Installer >2.0 (if using older versions of windows e.g. Win9x/WinME)
  http://downloads.activestate.com/contrib/Microsoft/MSI2.0/
- A working JAVA 6.0 environment
  http://java.sun.com/

1.2.2 Install Windows Installer (for older versions of windows)

- Install InstMsiA.exe when using Win9x/WinME or InstMsiW.exe for WinNT. The setup binaries can be found on the ActiveState website:

http://downloads.activestate.com/contrib/Microsoft/MSI2.0/

1.2.3 Installation of a JAVA environment

- Download the Java JDK for Windows from
  http://java.sun.com/javase/downloads/index.jsp
- Install it according to the provided instructions

1.2.4 Install active perl

- Download Active Perl (version >= 5.6)
  http://www.activestate.com/ActivePerl/
- Install ActivePerl with the installer.
  NOTE: The command "perl" should now work in a DOS box (Start->Run->cmd). To end perl press CTRL+C.

1.2.5 Install of Bit::Vector and Set-IntRange for perl

- Download and Install Bit::Vector and Set::IntRange through the ppm program from ActiveState. Open a DOS box and type at the prompt:

  ```
  ppm: install Bit-Vector
  ppm: install Set-IntRange
  ppm: quit
  ```

  If this doesn't work then the names might have changed. Try to search the "Set" and Bit" modules to get the right name:

  ```
  ppm: search Bit
  ```
  or
  ```
  ppm: search Set
  ```

1.2.6 Install gcc

- Install the MinGW (e.g. MinGW-6.0.2.exe)
- Extent the path variable for the gcc compiler

  In Windows XP and Vista:
  Start->Settings->Control Panel->System

  Pick Advanced tab

Click on "Environment Variables"

Search for "Path" variable in the system variables list

Click on edit

Append to the Variable value string the path of the gcc.exe

(e.g. when MinGW was installed to "`C:\Program Files\MinGW`" then append "`C:\Program Files\MinGW\bin`")

- The command "gcc" should now work in a DOS box

1.2.7 Installation of FermaT Maintenance Environment

- Unpack the archive (with WinZIP or WinRAR) into any directory (note the directory path must not contain any space characters)
- The program should run now by executing the `fme.bat` script

2 Applied Software Evolution With The FermaT toolset

2.1 FermaT Transformation System

The objective of the FermaT transformation system is to enable the migration of large, highly complex legacy systems from Assembler to higher-level language such as C or COBOL. The FermaT transformation system is built on the transformation theory that has the following objectives.

1. Improving the maintainability (in particular, flexibility and reliability, and
2. hence extending the lifetime) of existing mission-critical software systems;
3. Translating programs to modern programming languages;
4. Developing and maintaining safety-critical applications;
5. Extracting reusable components from current systems, deriving their specifications, and storing the specification, implementation, and development strategy in a repository for subsequent reuse;
6. Reverse engineering from existing systems to high-level specifications, followed by subsequent reengineering and evolutionary development;

Once migrated, these systems are substantially easier to maintain and can evolve faster to meet the changing needs of the business they support. Unlike simple line by line language migration technologies, the FermaT transformation's unique semantics preserving code transformations enable the original application to be automatically cleaned-up, simplified and restructured to its optimum state for migration to the chosen new language [4]. This ensures that only functional code is migrated to the new language, helping to ensure that the migrated code is significantly easier to maintain and adapt than the original.

Because of FermaT's use of a unique and formally defined high-level language, Wide Spectrum Language (WSL), and its specifically designed code transformations, the migration process can be automated [6]. The migration process of the FermaT transformation system can be divided into three basic steps:

1. Translation of the assembler to WSL;
2. Translate and restructure data declarations;

3. Apply semantics-preserving WSL to WSL transformations;

4. Translate the high-level WSL to the target language.

2.2 The Wide-Spectrum Language

The core of the FermaT transformation system is the WSL language. It is based on a wide spectrum language, using Morgan's specification statement [2] and Dijkstra's guarded commands [1]. The intention is to form a language which acts as an intermediate language when processing a legacy system [3].

WSL was designed for reengineering tasks and covers:

- Simple, regular and formally defined semantics
- Simple, clear and unambiguous syntax
- A wide range of transformations with simple, mechanically-checkable correctness conditions
- The ability to express low-level programs and high-level abstract specifications

The heart of the WSL language is a very small and mathematically tractable kernel language. This language supports already all necessary operations needed for a programming and specification language. In the context of this tiny kernel language it is relatively easy to prove the correctness of a transformation, but the language is not very expressive for programming. For that reason the language is extended into an expressive programming language by defining new constructs in terms of the kernel. This extension is carried out in a series of layers with each layer building on the previous language level (see [3] for details).

2.3 FermaT Maintenance Environment

The FME is written entirely in the Java language. The choice for the Java language was made because it is a very safe (strong typed) and flexible language with an extensive API and a vast amount of open-source libraries for almost every possible computer task. Primary the FME is a graphical interface to the FermaT transformation engine. It consists of a text editor which is able to express WSL together with an Abstract Syntax Tree (AST) viewer. A maintainer can navigate through the code via the source code or via the AST. The environment provides a console to the FermaT transformation engine which can be used to directly command the engine. The core

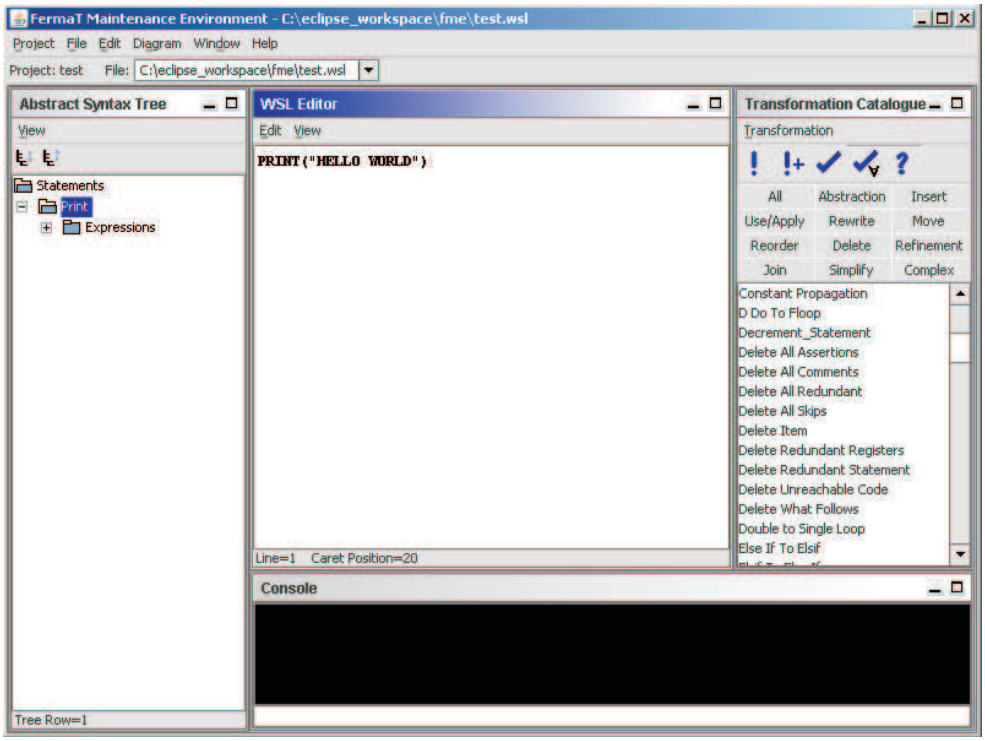

Figure 1: The FermaT Maintenance Environment

of the transformation engine is a set of mathematical proven program transformation to simplify the source code. The transformations can be selected from a transformation catalogue and performed on a chunk of WSL code. The transformations will either produce a semantically equivalent or refined version of the source program construct [3]. The communication between the FermaT Maintenance Environment and the FermaT transformation engine is provided through a communication pipe provided by the underlying operation system.

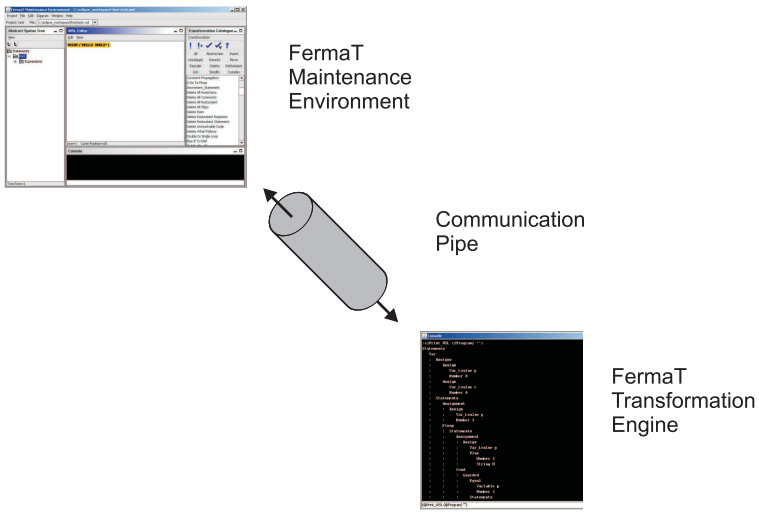

Figure 2: Communication between the Engine and the FME

3 Interface

Following sections give a brief explanation of the main functions of the FME.

3.1 Project

WSL files which are to be used with the FME must be organised in a project. An own directory should be reserved for the project and WSL files. A project can be created or modified with the Project Manager (Project→Show Project Manager). The Project Manager provides the following functions:

- New Project
  Create a new project.
- Open project
  Opens an existing project.
- Save project
  Saves the changes which have been done to the project.
- Close project
  Closes a project
- Open file from project
  Open a file from the project in the FME.
- Create empty file in project
  Creates an empty file and adds it to the project.
- Add file to project
  Add an existing file to the project. The file should be in the same directory as the project file.
- Remove file from project
  Removes a file from the project.
- Show Modification History
  Shows a graph with all applied transformations and saved intermediate WSL files.
- Delete History
  Resets the modification history record.

3.2 File

Every correct WSL file can be interpreted and executed by the underlying Scheme interpreter (File→Run WSL file). If a WSL file was loaded into

the FME and modified, it can safed as an intermediate version. In this case, the filename is extended with a version number e.g. <file>-1.wsl. Intermediate versions should be used throughout the whole transformation process. Every applieed transformation creates a new intermediate version. When the transformation process has finished the final WSL version can be saved wiht File as the original filename whereby all intermediate WSL files are deleted (File→Save code and delete intermediate version). If the intermediate versions should be kept it is also possible to save a WSL file to a different file with (File→Save code as WSL file).

3.3 Other functionalities

Apart from the main file handling functions the FME has many other functionalities:

- UNDO/REDO functions which can undo or redo any edit on a WSL file.
- The calls of Actions or Procedures within WSL files can be visualised in Diagrams.
- Find / Replace which can search or replace text patterns in WSL file.
- The editor has the standard copy, paste and cut abilities.
- Selected items in the Abstract Syntax Tree can be highlighted in the code and visa versa.

4 Getting Started

4.1 First experience

The followings are two examples written in WSL. First Example: Hello world in WSL

```
PRINT("Hello World!")
```

The output should look something like this when it is directly executed with the "wsl.pl" script of the engine:

```
Writing: C:\DOKUME~1\TheUser\LOKALE~1\Temp\t11436.scm
Starting Execution...

Hello World!

Execution time: 0
```

To get this result in the FME we create a new project and add an empty file. We then type the statement from above into the editor and save it by selecting "File"→"Save code as intermediate version". Note that the Abstract Syntax Tree in the FME is only updated after the file has been saved. When we now select "File"→"Run WSL File" the following window should appear:

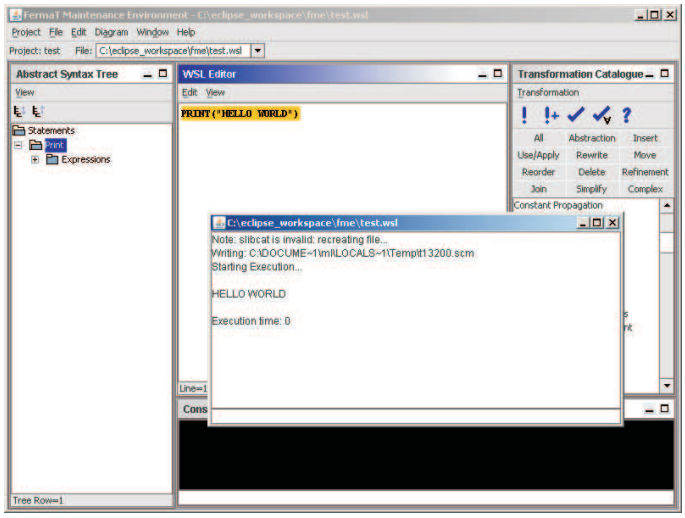

Figure 3: Execution of Hello World

The first line tells the interested reader that a temporary file "t11436.scm" has been written to the temporary directory of the current user (here under

the windows operating system). This is because the FermaT transformation engine converts a WSL program before execution into Scheme. The Scheme interpreter is than used to execute the program. This technique has a serious drawback: The runtime errors detected be Scheme will refer to lines in the Scheme programm. So the user isn't able to trace the error according to line numbers unless he knows exactly which WSL statement of his program is mapped to particular Scheme statement(s) in the executed program. For development language this would be a serious problem but as mentioned before the WSL language is intended to be an intermediate language for migration and analysing tasks rather than for software development tasks.

The second Example should demonstrate an interactive program - A simple guessing game in WSL:

```
VAR <num:=0,guess:=0>:
  num := @Random(100);
  PRINT("I have thought of a number between 1 and 100");
  DO PRINFLUSH("What is your guess? ");
    guess := @String_To_Num(@Read_Line(Standard_Input_Port));
    IF guess = num THEN PRINT("Correct!"); EXIT(1) FI;
    IF guess < num
      THEN PRINT("Too low.")
      ELSE PRINT("Too high.") FI OD;
  PRINT("Goodbye.")
ENDVAR
```

The output should look something like this:

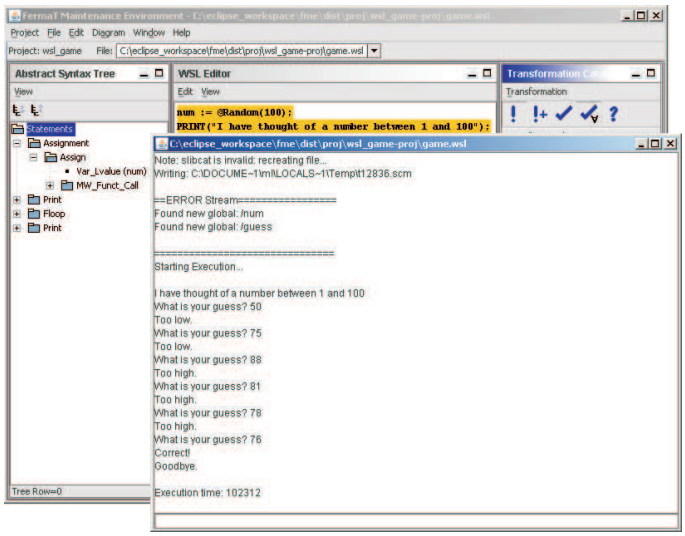

Figure 4: Execution of the guessing game

A more in-depth reference of all possible WSL statements can be found in [5].

4.2 Transformation Example

To demonstrate the transformation facility of FermaT we introduce a small program. The execution will just output "Hello World". The reader is encouraged to save the following chunk of code in a file within a project of his choice.

```
VAR < x := 0, y := 0 >:
DO DO IF x = 0 THEN PRINT("Hello World")
      ELSIF x > (2 + x) - 1
        THEN PRINT("Goodby cruel world")
        ELSE EXIT(2) FI;
      x := x + 1 OD OD ENDVAR
```

A click onto the first *DO* statement should highlight the whole loop:

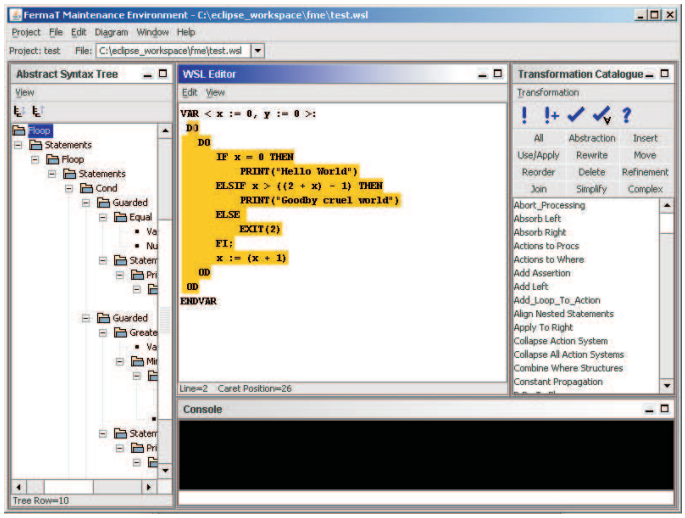

Figure 5: Transformation Example

We can test now for the available transformations on this node with a click on the "Test All Transformations" symbol in the transformation catalogue.

Figure 6: Transformation Catalogue Toolbar

Now some transformations should be highlighted in green. These transformations can be applied on the current selected program item (Of course some of them may have no effect).

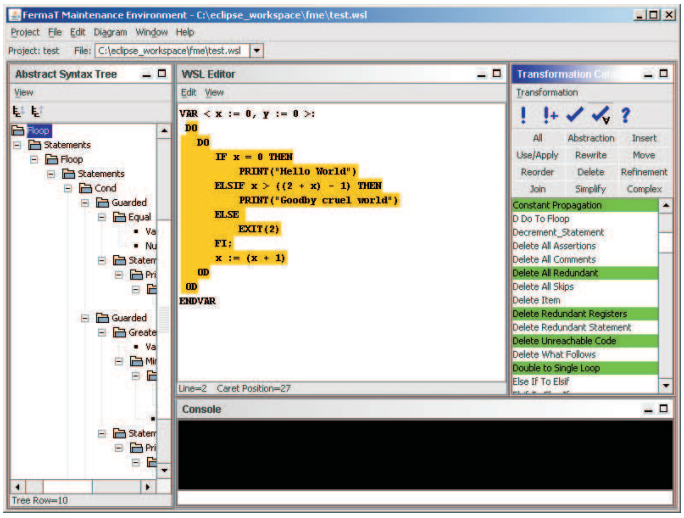

Figure 7: Transformation Example

Now we select the transformation "Double to Single Loop". A click on "Apply Transformation" should result in the following code:

```
VAR < x := 0, y := 0 >:
DO IF x = 0
     THEN PRINT("Hello World")
   ELSIF x > (2 + x) - 1
     THEN PRINT("Goodby cruel world")
     ELSE EXIT(1) FI;
   x := x + 1 OD ENDVAR
```

The double loop has been eliminated and the exit statement inside the loop have been decreased by one. After the transformation has been applied the attentive reader may have recognised that the filename contains now a "-0". Everytime the source code is modiefied and saved the FME will generate an new file (called "intermediate version"). This includes the case when a transformation is applied. If all modifications of the file have been finished the user may select the "Save (final) WSL File and all intermediate versions of the file will be deleted and the last version will replace now the original file. The benefit of this technique is that a maintainer can access in the process of migration all past intermediate versions of the processed program. If he did something wrong or applied the transformations in a wrong order he can easily go back to an older version.

If the file is not saved than all modifications can be undone/redone with the "undo" and "redo" options of the "Edit" menu entry.

Through the "Simplify If" Transformation the program can be simplified to[1]:

[1]The interested readers may do this on their own.

```
VAR < x := 0, y := 0 >:
DO IF x <> 0 THEN EXIT(1) FI;
   PRINT("Hello World");
   x := x + 1 OD ENDVAR
```

4.3 Working with the console

As mentioned before the FME is directly tied to the transformation engine. The engine itself can be accessed via the console. The current program and the current item[2] is changed when a node in the FME's Abstract Syntax Tree Window has been selected. To demonstrate this the reader may open a WSL program of his choice and select a node in the tree. This node is now the current item in the transformation engine. If now the command

```
(@Print_WSL (@I) "")
```

is entered in the Console the engine should output a subtree of the AST with the current item as the root.

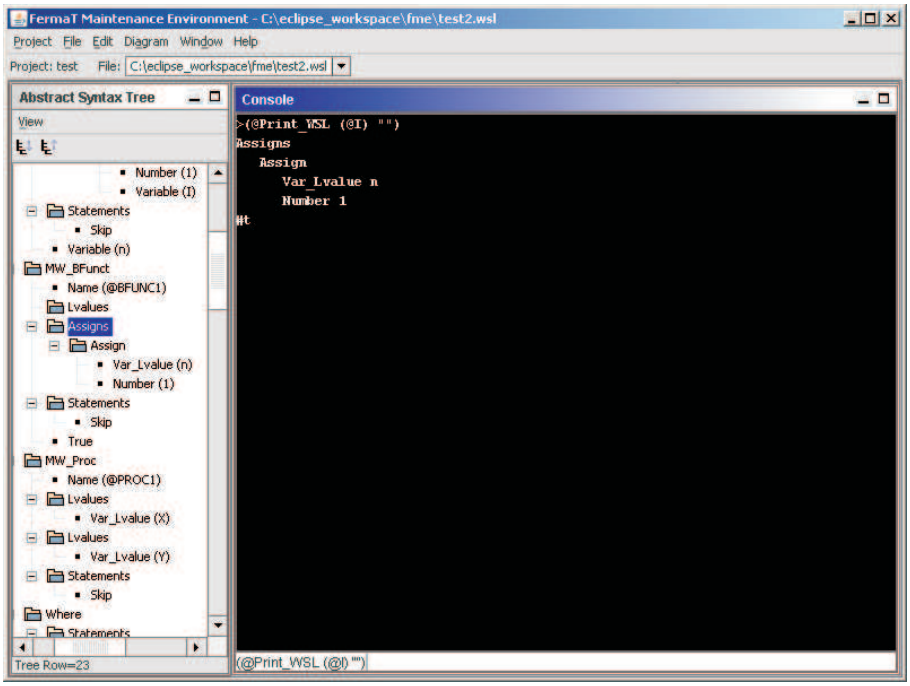

Figure 8: Console to the transformation engine

For all possible commands please see the WSL manual [5].

[2]See [5] for details on these concepts

4.4 Other functionalities of the FME

The FME includes some more functionalities for comfortable usage:

The graphical user interface of the FME consists of internal windows which may be minimised, maximised or closed. The "Window" menu entry gives some options to automatically align these windows.

The "ActionSystem CallGraph" is a very useful feature when analysing action systems.

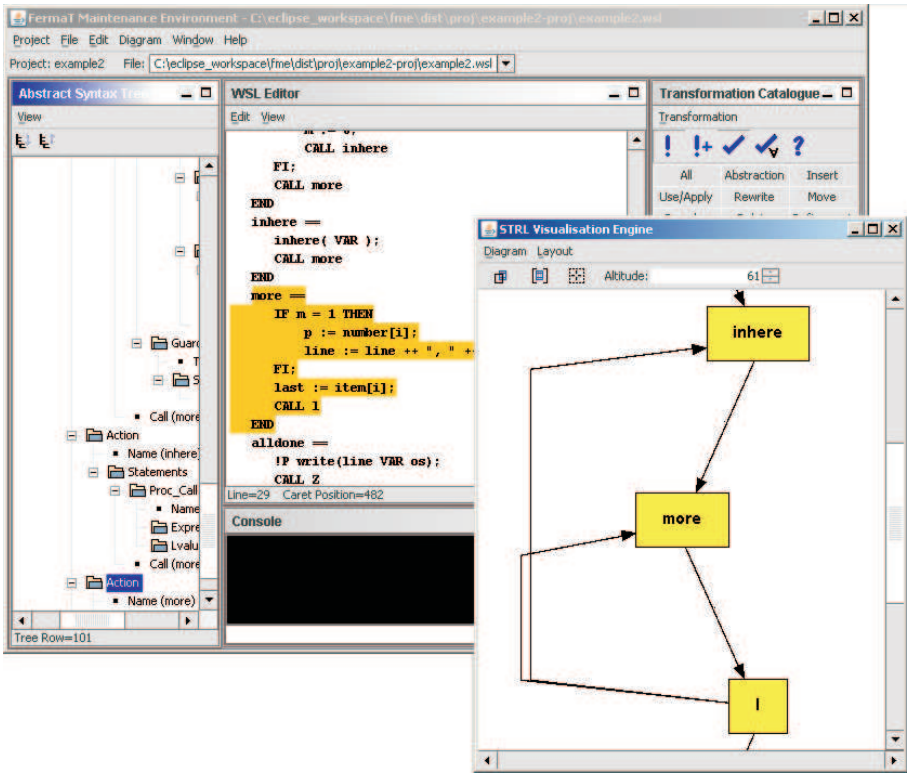

Figure 9: ActionSystem CallGraph

The dialog can be activated within the "Analyse" menu entry if an WSL Action System is present in the current program. It shows the calls in the action system in a call graph. The user can automatically zoom-in ,hide/collapse several nodes, print the graphic and export the graphic to a vector/bitmap file format.

References

[1] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[2] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988.

[3] Martin Ward. Pigs from sausages? reengineering from assembler to c via fermat transformations. *Science of Computer Programming, Special Issue on Program Transformation 52*, pages 213–255, 2004.

[4] Martin Ward and K. H. Bennett. Formal methods for legacy systems. *Journal of Software Maintenance: Research and Practice*, 7(3):203–220, - 1995.

[5] Martin Ward and Tim Hardcastle. *WSL Programmer's Reference Manual*, Nov. 2003.

[6] Martin Ward and Hussein Zedan. Analysing and abstracting legacy assembler code via conditioned semantic slicing. 2006.

Web Links

[web1] The FermaT Engine
http://www.cse.dmu.ac.uk/~mward/fermat.html

[web2] The Software Technology Research Laboratory (DeMontfort University, Leicester)
http://www.cse.dmu.ac.uk/STRL/index.html

[web3] Software Migration Ltd.
http://www.smltd.com

[web4] Teach Yourself Scheme in Fixnum Days
http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html