# The Command-Line Architecture of the TL System 2.00

Victor Winter

February 28, 2010

# Contents

# Chapter 1

# Overview

The TL System is a collection of functions providing general-purpose support for rewrite-based transformation over elements belonging to a (user-defined) *domain language*. In the TL System, a *domain language* is described by a tuple consisting of (1) an EBNF grammar and, (2) a lexical specification of tokens.

The command-line architecture of the TL system has been designed to support the use of TL System functions in specific ways. The command-line architecture consists of three folders: (1) a folder called *Bat Commands* containing dot-bat files can be used at the system-level to create artifacts central to rewrite-based transformation (e.g, a domain-language parser, a pretty-printed version of a target program, etc.), (2) a folder containing domain-specific data (in the download, this folder is generically called *Domain_Data*), and (3) a folder called *TL_System*. Though they are bundled together in the download, these folders need not be grouped together and can be placed anywhere on the system subject to the following constraints:

1. The user must have read, write, and execute privileges over the contents of the *TL_System* folder. The name of the TL_System folder may not be changed.

2. The user must have read, write, and execute privileges over the contents of the *Domain_Data* folder. The name of the Domain_Data can be changed (e.g., the folder could be renamed to *MyDomain*).

3. In the *Bat Commands* folder, the (absolute) paths in the *99_setAbsolutPaths.bat* file must be set to correctly reflect the locations of the *TL_System* folder as well as the (possibly renamed) *Domain_Data* folder. In addition, in the *99_setAbsolutPaths.bat* file all domain specific file names must be set appropriately (e.g., the actual name of the EBNF grammar file to be used, etc.).

## 1.1 Target Audience

The command-line architecture of the TL System has been designed to provide functionality suitable to both *domain developers* as well as *domain clients*.

### 1.1.1 Domain Developer

A *domain developer* is a person whose activities include:

- Development of the domain language. This includes both the specification of lexical tokens as well as the construction of the EBNF description of the language. Validation is an important activity in the development of a domain language. In particular, it is useful to automatically test wether the parser generated from the

lexical tokens and EBNF description is able to successfully parse a (perhaps large) set of programs belonging to the domain language. For example, to test a parser developed for the Java language, it is useful to validate that it can successfully parse all the Java files in a Java library or set of Java libraries (e.g., lang, util, nio, etc.).

- Development of transformations appropriate to a domain. This includes:
  - Implementation of transformations expressed in the language TL.
  - Implementation of general-purpose functionality expressed in the language SML.

  It should be noted that transformation validation efforts can also benefit from the ability to automatically apply a transformation to a set of domain language programs.
- Development of style rules for pretty-printing parse trees belonging to a domain. In this context, it can again be argued that the ability to perform batch-processing is also beneficial.

### 1.1.2 Domain Client

In contrast, a *domain client* is a person whose activities include:

- writing programs in the domain language,
- applying transformations to these programs.

## 1.2 GUI Front-End

The command-line architecture of the TL System has been designed to facilitate its integration with a GUI front-end. Of particular importance is the development of a GUI supporting the batch-processing requirements associated with domain development tasks. The two primary batch-processing activities are: (1) parsing all files (with specified extensions) within a domain-program file hierarchy (e.g., the lang library in Java) and (2) transforming all files either parsed or unparsed (with specified extensions) within a folder hierarchy of domain programs.

# Chapter 2

# Domain Developer

From the perspective of the domain developer, a domain is a fairly complex structure consisting of:

- A set of files related to the domain language. These files include:
  - A file containing a lexical description of the tokens of the language expressed in a form (dialect) suitable to ML-Lex.
  - A file containing an EBNF description of the domain language.
  - A pretty-print style-file, written in SML, specifying how parse trees in the domain language should be formatted when printed as strings.
- A set of files containing transformations expressed in the language TL. Each file corresponds to a transformation module. A module is the TL language scoping construct used to encapsule a set of transformations.
- A set of files comprising a library of general-purpose functions expressed in the language SML.

A domain developer is responsible for designing, implementing, debugging, and validating the functionality of all of the files mentioned above. To do this effectively, the domain developer needs access to the appropriate functions necessary to develop a particular file or set of files.

General purpose TL System functions are:

- **lexer generator:** When given the lexical description of the tokens of the domain language $\mathcal{L}$, this function will create a lexical analyzer for $\mathcal{L}$. This function is useful for developing (and debugging) the lexical specification of the language.
- **parser generator:** When given an EBNF describing the syntax of the domain language $\mathcal{L}$ and a lexical analyzer for $\mathcal{L}$, this function will create parser for $\mathcal{L}$. The parser function can be validated by attempting to parse the file hierarchy.
- **transformation generator:** When given a *transformation*, a parser and a pretty-print specification, this function will create an executable transform function that can be applied to any target program. A transformation can be specified within a heterogeneous language framework consisting of TL and SML.
- **pretty-printer:** This function takes a style-file and a parse tree as input and converts the parse tree into a formatted string. More specifically, formatted string consists (primarily) of the leaves of the parse tree.

A file hierarchy consisting of domain-language files can be useful during validation of any of the domain-specific artifacts produced by the domain developer. In particular, these files can be used to validate (1) the specification of the domain syntax, (2) a particular set of transformations, and (3) the pretty-print style-file for the domain. For example, if Java is given as the domain language, then a Java library such as the util library may be suitable for use as a domain-program file hierarchy.

## 2.1 Domain Creation

A *domain* is the specialization of the TL System targeted to a specific problem. Domain specific data consists of:

- **lexer specification:** A formal description of the tokens of the target language.
- **context-free grammar:** An EBNF grammar describing the syntax of the target language.
- **target program set:** A set of target programs that one wishes to transform.
- **transformation:** The specification of a desired transformation. This specification will typically include a set of TL modules as well as a set of SML structures.
- **prettyprint style file:** A formal description of how parse trees belonging to the target language should be formatted.

Figures 2.1, 2.2, 2.4, and 2.3 are data-flow like diagrams showing the functional behavior of various portions of the TL System. In these figures ovals represent files containing domain-specific data, hollowed rectangles represent domain-independent meta-functions, and filled rectangles represent domain-dependent functions.

## 2.2 Transformation Output

There are two primary domain-independent output formats: (1) the xml representation of the transformed parse-tree, and (2) the pretty-printed form of a parse-tree (transformed or otherwise). The TL System has been designed in such a way that all output can be specified within a transformation. This also supports the development of other output formats such as custom xml representations, or even a variety of distinct pretty-printed output formats. It is because of this design, that the functions produced by 2.4, and 2.3 show no outputs (for this is under the control of the transformation program that is input to the function).

## 2.3 Output from Source-to-Source Transformation

Transformation can be used for a variety of purposes. One purpose is to perform a source-code level manipulation of a target program. The result is (another) target program. In this case, an issue arises regarding how the transformed program should be named and where it should be placed. For example, in the Java Library Migration project the goal is to perform source-code level transformations on a set of Java libraries and produce a transformed set of libraries as a result. This goal requires that the input target program and its transformed counterpart have the same name. To accomplish this requires control over the directory to which transformed programs are written. For reasons such as these, it is important for transformations to have a clear paremeterizable knowledge regarding where the input program resides as well as where the output program should be written (as opposed to placing such knowledge/control within a GUI).

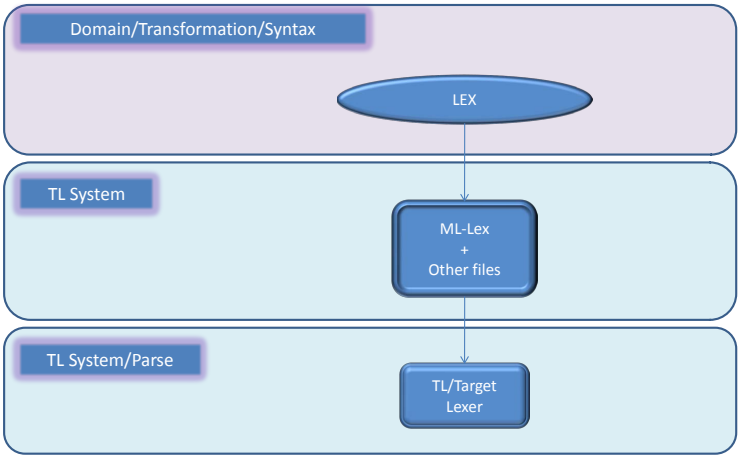


Figure 2.1: The creation of a lexer.

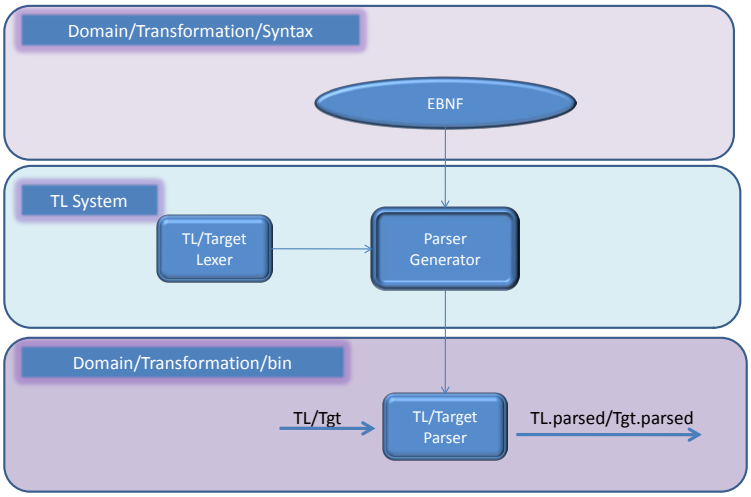


Figure 2.2: The creation of a parser executable.

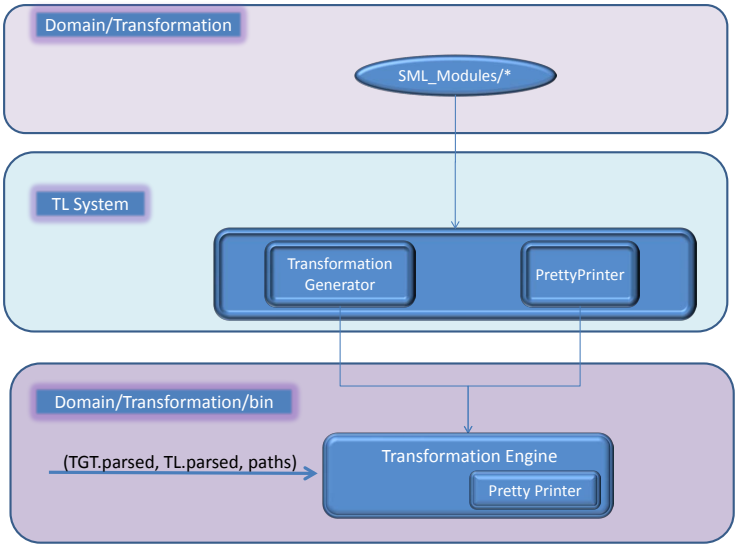


Figure 2.3: The creation of a transform executable with an embedded pretty-printer.

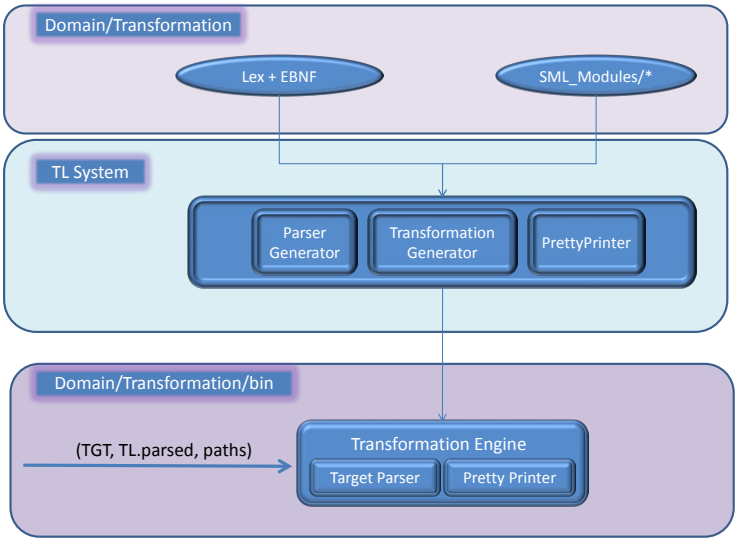


Figure 2.4: The creation of a transform executable with an embedded target parser and pretty-printer.

# Chapter 3

# Domain Client

A domain client is someone interested in using a transformation much in the same way they would use a commercial compiler (e.g., a Java compiler).

C:\> transform filename

For a domain client, the stand-alone specialization of the TL System to the domain is appropriate. An overview of the stand-alone is shown in Figure 3.1. It should be noted that from the perspective of IO destinations, the stand-alone version is treated as a transformation whose source and destination folders are equal (see Section 2.3). Note that when a domain developer creates a stand-alone, the stand-alone is initially placed in the *Domain/Transformation/bin* folder.

## 3.1 Paradigm Compilation: An Example of a Stand-alone

Paradigm is a high-level architecture independent microprogramming language whose inception originated from Sandia National Laboratories. This language is being developed as part of a funded Sandia-UNO effort. A Paradigm compiler takes a Paradigm program as input and produces a microcode program as output and has been implemented in a transformation-based setting. In this example, suppose that a stand-alone transformation, called *paradigm.x86-win32* has been developed and it has been specified that transformed Paradigms programs (i.e., microcode programs) should have the extension "ucode". Let C:/Source denote the directory in which a Paradigm program resides and let C:/Paradigm denote the directory where the *paradigm.x86-win32* compiler is located.

In the C:/Paradigm directory, create a file called *paradigm.bat* whose contents is:

C:/sml/bin/.run/run.x86-win32.exe SMLload=C:/Paradigm/paradigm.X86-win32 %1

From the C:/Source directory, the following command-line call can be used to compile the Paradigm program *model.pdm*:

C:\Paradigm\paradigm model.pdm

The result of the compilation will be the file *model.ucode* which will be written (by the compiler) to the folder C:/Source (i.e., the folder from which the compile command was issued).

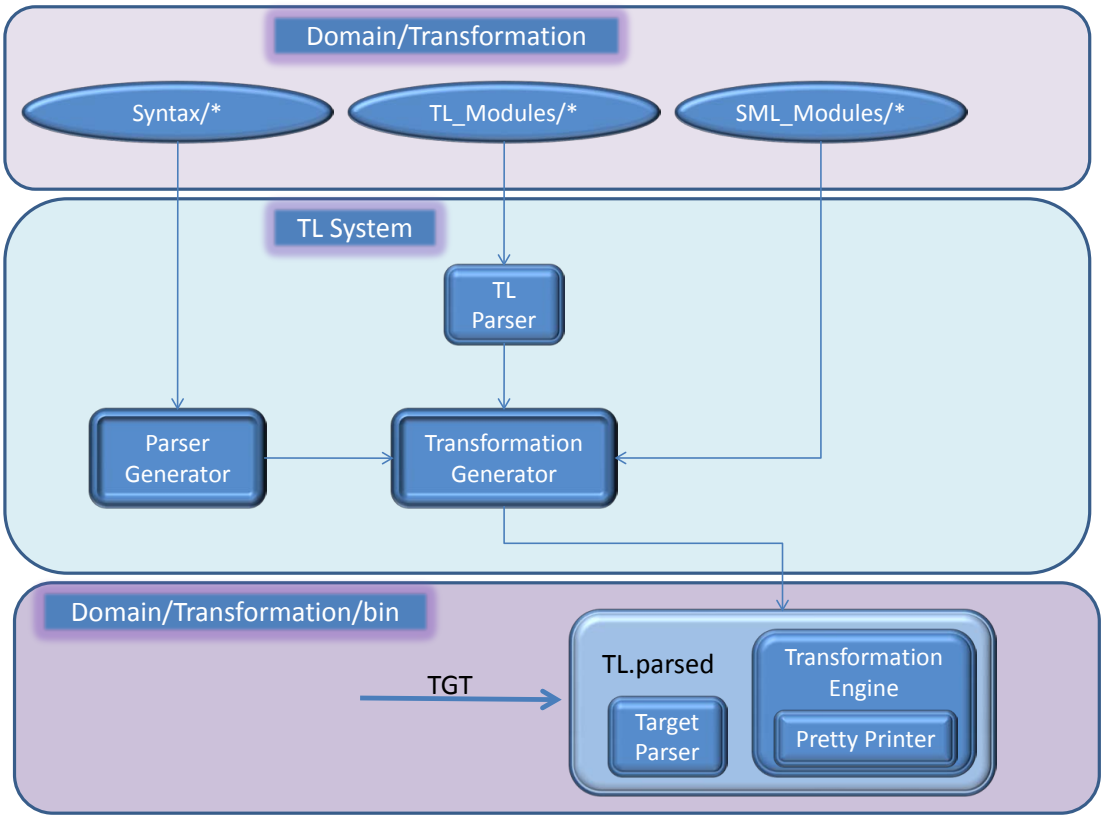


Figure 3.1: The Standalone Specialization of the TL System to a domain.

# Appendix A

# Glossary of Environment Variables

- BNF - the unqualified name of the file containing the EBNF description of the target language (e.g., Paradigm.bnf).
- CREATE_TRANSFORM_EXE_FLAG - A boolean flag that when set to the string "true" directs the transformation engine to generate an executable program (via exportFn) to the desired domain-specific transformation.
- DOMAIN_FOLDER - A string whose value is the fully-qualified path of the domain. In this string, subfolder relationships must be expressed using a forward-slash (e.g., "C:/MyDomain"). This string (or an extension of it) is passed as a parameter to a number of transformation-centric functions (e.g., parse target, parse TL, transform, and prettyprint).
- FORMAT - A string whose value is the unqualified name of the style-file for the domain.
- INPUT_FOLDER - A string whose value is the fully-qualified path of the domain's input folder. In this string, subfolder relationships must be expressed using a forward-slash (e.g., "C:/MyDomain"). In this folder is where the file hierarchy of domain programs should be placed.
- OUTPUT_FOLDER - A string whose value is the fully-qualified path of the domain. In this string, subfolder relationships must be expressed using a forward-slash (e.g., "C:/MyDomain"). In this folder is where the output of batch-processing is placed (e.g., parsed files, transformed files, pretty-printed files).
- STANDALONE_NAME - A string whose value is the unqualified and unextended name of the standalone executable. E.g., for the Paradigm domain the standalone domain is "paradigm").
- START_SYMBOL - A string whose value is the start symbol of the domain grammar.
- SML_RUN - A string whose value is the fully qualified name of the sml "run" executable. This executable is needed to run the compiled functions of the TL system (e.g., the parser, the transformer, the standalone).
- TARGET_TOKENS - A string whose value is the unqualified name of the style-file that is to be used by the pretty-printer.
- TGT - A string whose value is the unqualified and "unextended" name of the target program (i.e., the domain program) to which the pretty-print function is to be applied By "unextended" we mean that the TGT string does not contain the dot-extension of the target program. For example, if the target program is "myProg.java", then TGT will be "myProg".
- TGT_W_EXTENSION - A string whose value is the target program including its "dot-extension". For example, if the target program is "myProg.java", then TGT_W_EXTENSION will be "myProg.java".

- TL - A string whose value is the name of the unextended transformation program to be used to transform target programs.
- TL_SYSTEM_BACKSLASH - A string whose value is the fully-qualified system-appropriate path of the TL system folder. In this string, subfolder relationships must be expressed using a backward-slash (e.g., "C:\MyDomain").
- TRANSFORMATION_FOLDER - A string whose value is the fully-qualified system-appropriate path of the transformation folder corresponding to the domain. In this string, subfolder relationships must be expressed using a forward-slash (e.g., "C:/MyDomain").
- TRANSFORMATION_FOLDER_BACKSLASH - A string whose value is the fully-qualified system-appropriate path of the transformation folder corresponding to the domain. In this string, subfolder relationships must be expressed using a backward-slash (e.g., "C:\MyDomain").

# Appendix B

# Domain Development Tasks and Corresponding Bat-Commands

The following table shows the relationship between domain development tasks and dot-bat files. It should be mentioned that all of the dot-bat files in the table below make use of environment variables that are set in the file *99_setAbsolutePaths.bat*. The file *99_verbose_setAbsolutePaths.bat* is a wrapper file that prints the values of all the environment variables set by the file *99_setAbsolutePaths.bat*.

| **Development Task** | **Related Bat Files** |
|---|---|
| Create Lexer | 00 wrapper_createLexer.bat<br>00_createLexer.bat |
| Create Parser | 01 wrapper_createParser.bat<br>01_createParser.bat |
| Parse Target | 02 wrapper_parseTarget.bat<br>02_parseTarget.bat |
| Parse TL | 03 wrapper_parseTL.bat<br>03_parseTL.bat |
| Create Transform | 04 wrapper_build_transform_without_parser.bat<br>04_build_transform_without_parser.bat<br>05 wrapper_build_transform_with_parser.bat<br>05_build_transform_with_parser.bat |
| Apply Transform | 04b wrapper_apply_transform_without_parser.bat<br>04b_apply_transform_without_parser.bat<br>05b wrapper_apply_transform_with_parser.bat<br>05b_apply_transform_with_parser.bat |
| Pretty-print | 06 wrapper_prettyprint.bat<br>06_prettyprint.bat |
| Create Standalone | 07 wrapper_create_standalone.bat<br>07_create_standalone.bat |

# Appendix C

# A Domain Developer's View of the Paradigm Domain

In this Appendix, we use the Paradigm domain to briefly discuss the structure and contents of a domain. Figure C.1 show the structure of the (generically named) *Domain_Data* folder. As was previously mentioned, the name of this folder can be changed as desired. However, the names of the subfolders *Input*, *Ouput*, and *Transformation* may not be changed. In Figures C.1 and Figure C.2, verbatim names of folders or files (i.e., names that cannot be changed) are enclosed in double-quotes. A description of the contents of each subfolder is as follows:

- *Input* - This folder contains the source files to be transformed. In general, it is a good design principle to treat this folder as "read only". In other words, one should not write the results of transformation or pretty-printing to the *Input* folder.
- *Output* - In general, all the output produced during transformation (e.g., pretty-printed output) should be written to this folder.
- *Transformation* -
  - *bin* - The TL System writes its output to this folder. Output includes: parsers, transformers, and the parsed version of the transformation under development.
  - *Syntax* - The files in this folder specify the domain language as well as a style-file for pretty-printing parse trees corresponding to domain language programs.
  - *TL_Modules* - Transformations are written here in the language TL as a collection of modules. A module may import and then reference transformations defined in another module.
  - *SML_Modules* - During transformation, computations arise whose realizations are well-suited for a general-purpose programming language. In TL, such computations can be implemented in SML subject to the following restrictions:
    - *UserDefined.sml* - The interface between a TL module and SML must occur through a file named *UserDefined.sml*.
    - *UserLibrary.cm* - Dependencies among SML structures must be captured with the make file *UserLibrary.cm*.

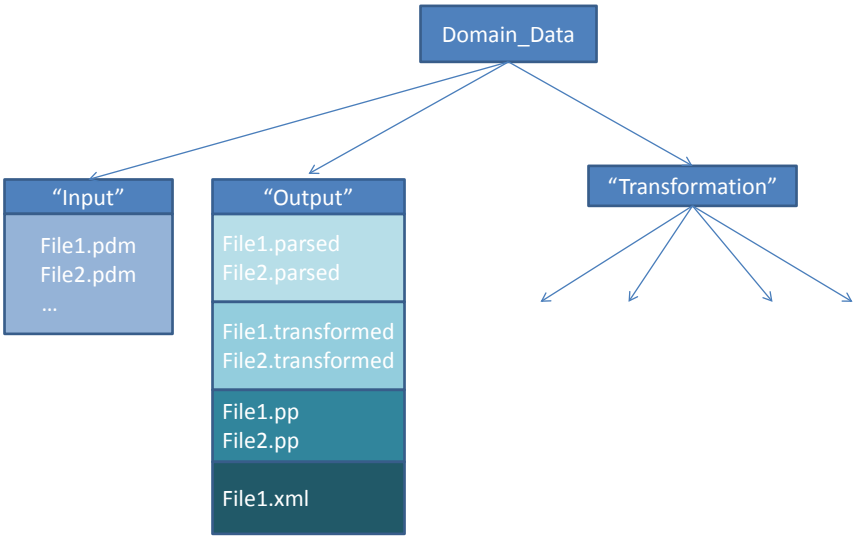


Figure C.1: Domain Data

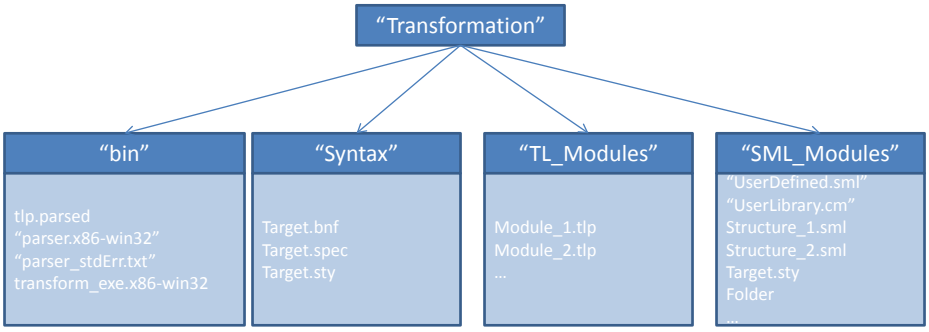




Figure C.2: Transformation Folder