MULTITRANSLATOR

An Integrated Environment for Multilanguage Translation

Version 2.8
Released 20.03.05

User's Guide

Department of Computer Engineering
Taganrog State University of Radio Engineering
Russia

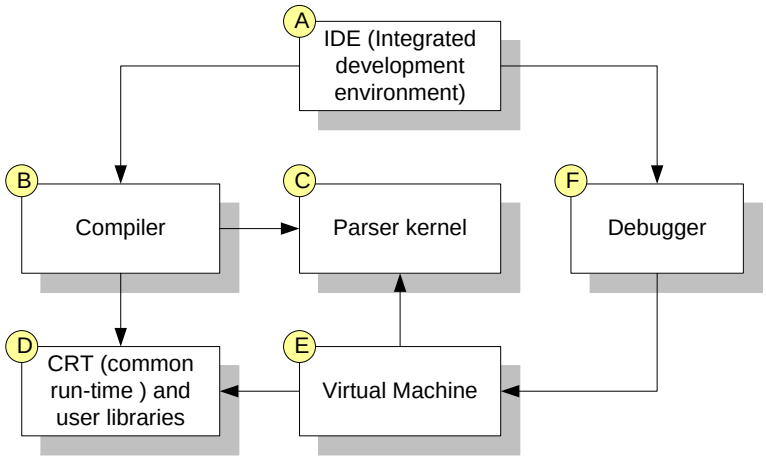
CONTENTS

1. Introducing MultiTranslator

MultiTranslator is a complete set of development tools for developing peculiar translators of various algorithmic languages, based on its own language of describing the process of translation. It has compiler, virtual machine and debugger. All these tools are united by integrated development environment (IDE), which provides all facilities user needs to design, develop, test, debug, and deploy translator applications, allowing rapid prototyping and a shorter development time.

2. MultiTranslator's structure

The principle of functioning of MultiTranslator program can be represented by the next figure:



The principle of functioning of MultiTranslator lies on the next algorithm:

1. Firstly, the user creates a set of grammatical rules and actions to translate texts from one language into another. He can use IDE (A) to type text of program or any another text editor
2. The obtained source is compiled by Compiler (B) and transformed to p-code which then is used by Virtual Machine (E) to run and translate texts.
3. In order to perform translation Virtual Machine (E) uses Parser Kernel (C) which comprises all necessary tools to parse texts without binding to certain language
4. MultiTranslator has a set of common run-time libraries (D) to facilitate the user's work and to widen the possibilities of translators. If the abilities, offered by CRT Library, are not enough, user can develop and distribute his own libraries
5. The MultiTranslator applications are easily debugged using Debugger (F) that allows to user to observe the run-time behavior of the program and determine the location of semantic errors

(A) Integrated development environment

When you start MultiTranslator, you are immediately placed within the integrated development environment, also called the IDE (**mtdev.exe)**. The IDE includes all the tools necessary to start designing applications, such as the:

- Code editor for writing and editing the underlying program logic;

- Visual tools for creating grammatical rules, that provides simplest and obvious way of developing translation modules for specific language;
- Object View for displaying and changing component's' logical relationships. It allows user to examine and navigate to symbols in the solution. The symbols, which are organized by project, display in a hierarchical tree view, indicating the containment relationships between them;
- Solution Explorer View, which provides to user an organized view of loaded projects and their files as well as ready access to the commands that pertain to them.
- Integrated debugger for finding and fixing errors in your code;
- Many other tools such as standard libraries viewer, optimization wizard etc;
- Command-line tools including compilers, linkers, and other utilities.

Some tools may not be included in all editions of the product.

(B) Compiler

Compiler (**grmc.exe**) works from a command line. For compilation it is necessary to specify one or several initial grammar files and name of the target file. The name of a target file should be last in the list. Besides, you can use a make-file that provides simpler way to describe configuration of the project. The using make-file allows user to link CRT and additional libraries to his program, set additional switches of compilation, etc. The linker is combined with compiler in one module, so you must not use the linker separately from the compiler. The linker has a key /SFX for generation of EXE-files. As a rule of thumb, there is no necessity to use compiler from the command line. It is more efficient and reliable to use IDE to achieve the same purposes.

(C) Parser Kernel

This tool (**grmkrnl.dll**) provides wide service for translation and interpretation of input texts. You must not use this tool directly, but it is used by MultiTranslator and all produced MultiTranslator's programs constantly and independently of you. It comprises tools for parsing input texts, determine lexemes, form lexical and syntactical chains, etc. If you intend to distribute program, built by MultiTranslator, you have to include this module into your deployment packet

(D) CRT Libraries

MultiTranslator has a set of standard libraries (**grmcrt.dll**), which allows you to enhance the functional capabilities of your translator modules. The functions included to CRT libraries are described below. These libraries also must be included to your deployment in case you are going to distribute your translator modules separately from MultiTranslator.

(E) Virtual Machine

It is an interpreter of a binary code (**grmexec.exe**). The compiler generates either EXE-files, or binary files. As a rule, users use a generation of exe-files. In that case the body of Virtual Machine (VM) is placed altogether with binary code of user's translator in the same module. VM starts first, prepares data for translator and then starts translator inside of it. But you can use non-exe-generation when the VM is not included to output file. This approach gives opportunity to reduce the size of output modules, but in that case the VM is necessary when you distribute your translator modules. To start the VM it is necessary to specify in a command line a name of the binary file for interpretation.

(F) Debugger

The MultiTranslator debugger is a powerful tool that allows you to observe the run-time behavior of your program and determine the location of semantic errors. The debugger understands features that are built into programming language and associated libraries. With the debugger,

you can break (suspend) execution of your program to examine your code, evaluate and edit variables in your program, see the instructions created from your source code, and view the memory space used by your application.

Other tools and features

- **cab.exe** - program for drawing up archives. With the help of the given program it is possible to make archive of several files and then to carry out its unpacking during translation. It is very convenient when the outing text of the translated program should be included into some set of sample files.
- **Set of examples.** The examples of use of various languages, grammars examples of translations for ACSL, Pascal, SML and Modelica languages into C++ are included in set.
- **Set of examples of libraries**. With the help of this set you can learn how to develop and link up to your translator your own libraries

3. Grammar modules' development

The work of the translator is based on preliminary development of grammatical modules (grammars) of input languages and their subsequent automatic use for the analysis of the initial program in input language. The analysis consists of two parallel processes: recognition and appropriate actions. In this connection the rules of the translator's behavior can conditionally be divided on described by language of analysis (grammar description language) and language of actions. Generally these rules can be submitted in the following kind (see fig.1)

Definition of recognition rules

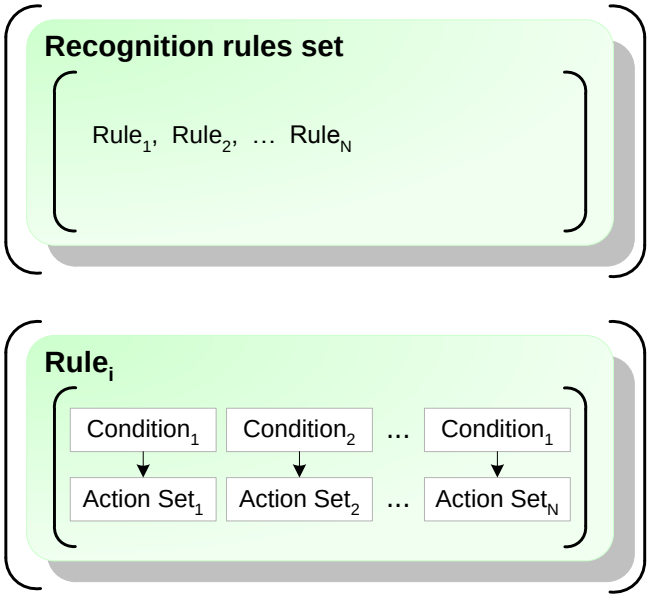


Fig. 1

At the grammar description the following grammatical units of analysis are used:

- **rule** –- represents the detached expression and describes the certain block of translated language. All rules represent the completed grammatical model of language. By a rule it is possible to describe any construction of translated language, watching for correctness and logicality of connections between rules in grammar according to the specification of described language.

- **variant** – grammatical unit which is included in structure of a rule. Any rule can have branching, in this case each similar branch should be described by an appropriate variant. As an example we shall consider the conditional operator in a context of ACSL.

```
rule "Conditional statement"
  variant "logical IF"
  variant "IF with ELSE"
  variant "IF without ELSE"
```

- **term** – base grammatical unit. Two types can represent it: - a terminal and a nonterminal. The terminal describes final units of language: keywords, operators, predefined identifiers etc. Nonterminal represents a call of other rule. As an example we shall consider complete realization of the conditional operator

```
rule "Conditional statement"
  variant "logical IF"
    terminal IF
    nonterminal "Expression"
    nonterminal "Statement"
  variant "with ELSE"
    terminal IF
    nonterminal "Expression"
    terminal THEN
    nonterminal "Statement block"
    terminal ELSE
    nonterminal "Statement block"
    terminal ENDIF
  variant "without ELSE"
    terminal IF
    nonterminal "Expression"
    terminal THEN
    nonterminal "Statement block"
    terminal ENDIF
```

As it is visible from an example the given unit is redundant, as it repeats in itself the same terms in different variants. It is possible to apply the optional terms to avoid similar redundancy. Optional term is a term, which can present or not present in analyzing unit. Using optional terms an above mentioned example could be transformed to the following form for the second and third variants

```
rule "Conditional statement"
  terminal IF
  nonterminal "Expression"
  terminal THEN
  nonterminal "Statement block"
  nonterminal ["ELSE block"]
  terminal ENDIF

rule "ELSE block"
  terminal ELSE
  nonterminal "Statement block"
```

It is necessary to pay attention that the separate rule for the block ELSE was created. It is connected that the declaration of terms

```
  terminal ELSE
  nonterminal "Statement block"
```

as optional inside the conditional statement would be logically incorrect since two optional rules can be executed or not be executed independently from each other, and in our case they have to be executed or to not be executed simultaneously.

To bypass the similar inconvenient situations it is possible to use *optional-dependent terms*. Such terms are ignored, if their parent-term was not involved at analysis. Let's explain it on an example:

```
rule "Conditional statement"
   terminal IF
   nonterminal "Expression"
   terminal THEN
   nonterminal "Statement block"
   terminal [ELSE]
   nonterminal ["Statement block"]* – optional-dependent term
   terminal ENDIF
```

After introduction of optional terms the description of a rule «Conditional statement» has become essentially simpler, but it is necessary to use optional terms with care. Quite often there are situations resulting to direct or indirect indissoluble recursion. As an example we shall consider organization of the list

```
rule "List"
   nonterminal ["List element"]
   nonterminal ["List"]
```

This example demonstrates rules infringement of nonterminal recursive calls.

4. Grammars description language

Grammars description language has the following syntax:

```
rule <"Rule Name">
{
   // Variants description block.
   // Each rule has to have one variant at least
   before:   // This unit is not obligatory
           // Arbitrary statements block
   variant
   {
      // Terminal and nonterminal description block
      // It have to be one terminal or one nonterminal in
      // variant at least
      before:
      // Arbitrary statements block
      term "Terminal name"
      {
         // Actions block for terminal
      }
      before:
         . . .
      term <"Nonterminal name">
      {
      // Actions block for nonterminal
      }
      // Actions block for variant
   }
   // Actions block for rule
}
```

The name of a rule (keyword rule) should to be enclosed in angular brackets and in inverted commas. The amount of rules in the program is not limited. Each rule contains one or several variants (keyword variant). The variant has not the name. Each variant has the set of terminals and nonterminals (keyword term) inside. Their amount and order of the description is not limited. A sole condition - the variant should contain at least one terminal or nonterminal. The compiler distinguishes the terminal from nonterminal on angular brackets framing its name.

Names of terminals, nonterminals and rules can be only string constants. The actions block in a rule and variant can be placed only after the appropriate descriptions block. The block of actions can include any units allowed in actions language for the functions description. A rule, variant and term do not return values, so application of the operator return therefore is allowed only without arguments. The double slash means the comment. At incorrect record of rules, variants or terms the compiler gives out the appropriate messages.

At the rule description the following options of analysis can be specified:

- **error** = `"Line of the message"` - specifies what message to give out at detection of a error in this rule during analysis. By default the message "Syntax Error" is given out.
- **SolitaryWords** = `on/off/auto` - way of comparison of terms. On - at the end of a term should be delimiter, off - the presence delimiter is not checked, auto - accepts value of global variable SolitaryWords. By default auto is used.
- **IgnoreWhiteSpaces** = `on/off/auto` - Specifies whether to ignore odd spaces between words. By default is set in auto.

Example:

```
rule <"Rule Name"> error = "Incorrect declaration",
  SolitaryWords = auto,
  IgnoreWhiteSpaces = auto
```

Any of these parameters can be omitted. At the terminal declaration it is possible to specify an option error, for example

```
rule <"Conditional statement">
{
  variant
  {
    term "IF" : error = "If Absent"
    { }
  }
}
```

If the rule will be not recognized owing to the given terminal, the message on a mistake will be taken just from the given variant, ignoring presence of an option **error** at the rule.

Also there is an opportunity to set the instructions before execution of variants and terms with the help of the operator **before**. Let's explain it by program example:

```
rule <"Preparation">
{
  variant
  {
    before:
      print("before term 1\n");
    term <"Program header">  {
      print ("in term 1");
    }
```

```
   }
}

rule <"Program header">: error = "Wrong program header"
{
  before:
    print("before variant\n");
  variant
  {
    before:
      print("before term\n");
    term "program": error= "'program' word absent"{
      print("in term\n");
    }
    term STD_ID {
    }
    term ";" {
    }
    print("in variant");
  }
}
```

After execution of the given program we'll have:

```
before term 1
before variant
before term
in term
in variant
in term 1
```

It is possible to set in grammar the optional terms, which the compiler can pass. Let's consider it on an example of the list:

```
// The list standart description
rule <"List">
{
  variant
  {
    term <"Element"> {}
    term <"List"> {}
  }
  variant
  {
    term <"Element"> {}
  }
}

// The list suggested description
rule <"List">
{
  variant
  {
    term <"Element"> {}
    term [<"List">] {}
  }
}
```

Optional terms can be either terminals or nonterminals. Except optional terms it is possible to set optional-dependent terms. The optional-dependent term should have square brackets framing its name and behind them the asterisk should be specified. The optional-dependent terms should be right after the optional term. The insert of a usual term in the list of optional-

dependent IS NOT SUPPOSED. In this case the connection of optional-dependent terms with their parent will be broken. If the optional-dependent term has not the parent it is considered as usual term.

Let's consider example of optional-dependent term's description:

```
rule <"Begin-End">
{
   variant
   {
      term ["Begin"]{}
      term ["End"]* {}  // optional-dependent term
   }
}
```

Below is shown the wrong description:

```
rule <"Begin-End">
{
   variant
   {
      term ["Begin"]{}
      term <"List of statements">{} // Wrong usage of ordinary term
      term ["End"]* {}   // The connection with parent is broken
   }
}
```

The compiler supports a set of predefined terminals:

Name	Description
STD_LINE	All string from the current position up to line feed character
STD_ID	The standard identifier, i.e. the lexeme which is limited on the right by delimiter and begins with the letter or symbol of underlining and consists of the letters or digits.
STD_INT_CONST	An integer constant. Does not support any marks for the indication of a notation of a type $ or 0x. Does not support also hexadecimal letters: A, B, C, D, E, F.
STD_DIGIT_CONST	A real constant. The usual form of a kind x.xxxxxx and the form of x.xxxE±xxxx are supported.
STD_SINGLE_QUOTES_STR_CONST	A string in unary inverted commas. The managing symbols inside a string are not supported. The string is considered complete when the second unary inverted comma appears.
STD_DOUBLE_QUOTES_STR_CONST	A string in double inverted commas. As well as the previous variant, does not support the managing symbols.
STD_COMMON_CONST	Any constant, described above (string unary, string double, integer or real)
STD_COMMON_CONST_OR_ID	The combination of all predefined terminals mentioned above.

As it can be seen the descriptions of the predefined terminals are base and universal. But they are not capable to assort such lexemes, as, for example, “xxx \” xxx “, therefore in such cases it is necessary to make a grammar for analysis of similar expressions.

Standard terminal usage example:

```
rule <"Operand">
{
  variant
     {
     term STD_ID {write(GetText()," ");}
     term <"Array indexes list"> {}
  }
}
```

5. Actions description language for grammar

The action language of MultiTranslator has the large possibilities for work with text files and flexible syntax permitting to not block up main algorithm by secondary tasks. The given language is a description tool of an output file, generated by the translator. As the prototype of the actions language the language C, distinguished large flexibility and ease of control was selected. Basic features of the actions language:

- Support of the C language syntax;
- Structures support;
- Arrays support;
- Support of operators Increment, Decrement, etc.;
- The extended type conversion;
- The extended assignment of variables, including exemplar of structures;
- Support of hashing functions;
- Support of built-in functions and global variables;
- The extended syntax for passing parameter in the function;
- Presence of the sizeof operator, operating as on simple types, and structure;
- Presence of the size operator, detecting a size of arrays;
- The extended possibilities of working with arrays, including copying and dynamic memory allocation.
- The action language supports rules of description and calls of the C language functions.

In addition language has the following built-in functions:

The function ***print*** is for output information on the screen. The function has a variable amount of parameters of any type down to structures and returns the void type:

```
void print ([expression], expression [],...)
```

The function ***write*** is for output information in the file. The function has a variable amount of parameters of any type down to structures and returns the void type:

```
void write([expression], expression [],...)
```

The function ***Parse*** is for start of the analysis for grammars. The function has the one parameter of a string type pointing, from what rules to start analysis and returns a **int** value:

```
int Parse(string)
```

The function ***GetText*** return the expression, recognized by a current rule:

```
string GetText()
```

The function ***GetFilePos*** return position in current file

```
int GetFilePos()
```

The functions ***InsertBefore*** and ***InsertAfter*** insert the text into the file before the reference position or after it accordingly. The functions have two parameters: the pointer in the file (or text for searching), where will be produced an insertion and text. Return a new value of the pointer of the file:

```
int InsertBefore(int (or string), string)
```

```
int InsertAfter (int (or string), string)
```

The function ***IgnoreBlock***. It allows the grammar module to skip the text between Start and End positions. It is useful for the comments and other units skipping. It is possible to call this function several times but before the ***Parse*** call.

```
void IgnoreBlock (string Start, string End).
```

The function ***ReplaceBlock*** – replace in input text all strings *src* by string *dst* before analysis. It is possible to call this function several times but before the ***Parse*** call.

```
void ReplaceBlock (string src, string dst)
```

The function ***format***– works similarly to *sprintf()* in C/C++.

```
string format (string strFormat, ....)
```

Besides standard functions there are the set of predefined variables in the actions language:

Name	Description
SolitaryWords	
IgnoreWhiteSpaces	
IgnoreCase	specifies whether to ignore the case by search of terminals. There can be *on/off/auto*;
Delimiters	specifies what symbols could be used as delimiters. For example – "{}<>!@#$%*()_+|-[],.;=:/";
ParamStr	array of strings with command line parameters. String with index 0 saves path to executed program.
ParamCount	number of parameters in command line.
SourceFileName	saves the name of source translated file. For example: SourceFileName =ParamStr[1];
DestFileName	saves the name of output file. For example: DestFileName ="RESULT\\Acsl.h";

Actions language allows making various type conversions string to number and number to string automatically. All simple types are cast to each other automatically, but it is not allowed to cast various structures.

At assigning one variable to another occurs automatic cast of types. Besides it is possible to assign instances of structures, provided that they use the same structure, and also arrays.

6. Common run-time libraries

MultiTranslator offers a set of standard run-time libraries, which can be used by user. The main library module **grmcrt.dll** is linked to program automatically if at least one function from this module is used. You haven't to explicitly declare the functions from standard libraries or include additional modules or header files – all actions are performing by MultiTranslator automatically, in case it finds the use of such function in the program text. But if you developed your own library and intend to use it, you have to set the path to it in the settings of the project. The content of any runtime libraries is accessible to see from the tool "Libraries' contents" which can be called from IDE. By default, this tool shows the content of **grmcrt.dll.**

So, the standard libraries comprise next functions:

Operations with maps

Using of maps is very helpful for translation. The main idea of a map is to bind some value with the key and the retrieve this value with the help of this key. Or you can just add values into the map in order to check later if some value was found in text or not. For example, you can add the names of found variables into the map during parsing their declaration and afterwards to check their presence in the map during further translation.

Name	Description
CreateMap	The function **CreateMap** creates a map for language units
	`bool CreateMap (string Name)`
AddToMap	The function **AddToMap** adds to indicated map a new unit, key and value. Value type is arbitrary and could be used for different interpretation of appropriate unit.
	`AddToMap (string Name, string Key, Value)`
LookupInMap	The function **LookupInMap** searches value in a map guided by a key.
	`LookupInMap (string name, string Key, Value)`

Operations with strings

Name	Description
StrAnsiToOem	Translates a specified string into the OEM-defined character set.
	`string StrAnsiToOem(string strSrc)`
StrOemToAnsi	Translates a string from the OEM-defined character set into an ANSI or a string.
	`string  StrOemToAnsi(string strSrc)`
StrCat	Appends one string to another. Returns a string which holds the combined strings. Parameter nLen specifies the count of characters of appended string that must to be added. If nLen specifies more characters than are available, whole string is added

	`string StrCat(`
	`  string strSrc,`
	`  string strAppendStr,`
	`  int nLen){`

StrCmp	Compares two strings to determine if they are the same. The comparison *is* case-sensitive.
	`bool StrCmp(string strSrc1, string strSrc2)`

StrCmpI	Compares two strings to determine if they are the same. The comparison is *not* case-sensitive.
	`bool StrCmpI(string strSrc1, string strSrc2)`
StrDel	Removes a substring from a string. **StrDel** removes a substring of nLen characters from string strSrc starting with strSrc[nPos]. If index is larger than the length of the string or less than 0, no characters are deleted. If count specifies more characters than remain starting at the index, **StrDel** removes the rest of the string. If count is less than or equal to 0, no characters are deleted.
	`string StrDel(string strSrc, int nPos, int nLen)`
StrDeleteFrames	Deletes specified frames from the beginning and the end of the string. Both of characters must present. In case one character does not present the second is not removed as well
	`string StrDeleteFrames(string strSrc, char chLeftFrame, char chRightFrame)`
StrFillChar	Returns a string with a specified number of repeating characters.
	`string StrFillChar(string strSrc, char ch, int nCount){`
StrGetLast	Returns last character of the string. Parameter bSkipSpaces specifies whether space characters have to be skipped or not.
	`char StrGetLast(string strSrc, bool bSkipSpaces){`
StrInsert	Inserts a substring into a string beginning at a specified point. If the strSubStr parameter is an empty string, Insert does nothing.
	`string StrInsert(string strSubStr, string strSrc, int nPos)`
StrIntersect	Intersects to string. Function **StrIntersect** returns string, containing only characters present in both input strings
	`string StrIntersect(string strString1, string strString2)`
StrIsFramed	Checks if specified string is framed by specified characters
	`bool StrIsFramed(string strSrc, char chLeftFrame, char chRightFrame)`
StrLen	Calculates length of the specified string
	`int StrLeft(string strSrc)`
StrLeft	Returns the substring of a specified length that appears at the start of a string.
	`string StrLeft(string strSrc, int nCount)`

StrMid	Extracts a substring of length *nCount* characters from a strSrc string, starting at position *nFirst* (zero-based). The function returns a copy of the extracted substring.
	`string StrMid(string strSrc, int nFirst,int nCount)`
StrMove	Moves substring of length nLen in specified string from position nPos to position nNewPos
	`string StrMove(string strSrc, int nPos, int nLen, int nNewPos)`
StrPos	Returns the index value of the first character in a specified substring that occurs in a given string. This function *is* case-sensitive.
	`long StrPos(string strSrc, string strSubStr, int nStart=0)`
StrPosI	Returns the index value of the first character in a specified substring that occurs in a given string. This function is *not* case-sensitive.
	`long StrPosI(string strSrc, string strSubStr,int nStart=0){`
StrQuant	Calculates the number of repetition of specified character in the specified string. This function *is* case-sensitive.
	`int StrQuant(string strSrc, char ch)`
StrQuantI	Calculates the number of repetition of specified character in the specified string. This function is *not* case-sensitive.
	`int StrQuantI(string strSrc, char ch)`
StrReplace	Returns a string with occurrences of one substring replaced by another substring. If bReplaceAll = false StrReplace only replaces the first occurrence of strOld. Otherwise, StrReplace replaces all instances of strOld with strNew. The comparison operation *is* case sensitive.
	`int StrReplace(string strSrc, string strOld, string strNew, bool bReplaceAll)`
StrRight	Extracts the last (that is, rightmost) *nCount* characters from a strSrc string and returns a copy of the extracted substring. If *nCount* exceeds the string length, then the entire string is extracted.
	`string StrRight(string strSrc, int nCount)`
StrSubstract	Deletes from strString1 all characters that present in strString2 and returns the result of operation as a new string
	`string StrSubstract(string strString1, string strString2)`
StrToLower	Converts string to lower case
	`string StrToLower(string strSrc)`
StrToUpper	Converts string to upper case
	`string StrToUpper(string strSrc)`

StrTrim	Trims leading and trailing spaces and control characters from a string.
	`string StrTrim(string strSrc)`
StrTrimLeft	Trims leading spaces and control characters from a string.
	`string StrTrimLeft(string strSrc)`
StrTrimRight	Trims trailing spaces and control characters from a string.
	`string StrTrimRight(string strSrc)`
StrUnion	Returns the string, which includes characters from both input strings. Each character may be represented only once in the result string no matter how many times it has been found in input string
	`string StrUnion(string strString1, string strString1)`
StrWrap	Returns a copy of input string wrapped at the beginning and the end with specified characters
	`string StrWrap(string strSrc, char chLeftFrame, char chRightFrame)`

Operations with files

Name	Description
FileOpen	Opens the specified file for reading only, **FileOpen** returns the handle of opened file. If the function fails, the return value is 0.
	`int FileOpen(string strFileName)`
FileRead	Reads data from specified file. Data can be read only as a string. The length of reading string is set by 3rd parameter. The function returns the number of bytes that have been read.
	`string FileRead(int nFileHandle, int nBytesToRead, int & nActuallyRead)`
FileCreate	Creates a new file or opens the existing file for writing. If the file already exists, it is truncated to zero length. **FileCreate** returns the handle of opened file. If the function fails, the return value is 0.
	`int FileCreate(string strFileName)`
FileAppend	Appends the specified data to the end of indicated file. The file must be opened by function FileCreate. The data that will be written may be of any type. The function returns bookmark of added record. You can use this bookmark lately with functions **FileInsertBefore** and **FileInsertAfter**
	`int FileAppend(int nFileHandle, variant Data)`
FileInsertAfter	Inserts the specified data after indicated bookmark of the file. The file must be opened by function FileCreate. The data that will be written may be of any type. The function returns bookmark of added record.
	`int FileInsertAfter(int nFileHandle, variant Data, int nBookmark)`

FileInsertBefore	Inserts the specified data before indicated bookmark of the file. The file must be opened by function FileCreate. The data that will be written may be of any type. The function returns bookmark of added record.
	`int FileInsertBefore(int nFileHandle, variant Data, int nBookmark)`
FileClose	Closes the specified file.
	`bool FileClose(int nFileHandle)`
FileRemove	Removes the specified file. If the function succeeds, the return value is true (1)
	`bool FileRemove(string strFileName)`
FileExists	Ensures whether the specified file exists or not. If this file exists, the return value is true (1)
	`bool FileExists(string strFileName)`
DirExists	Ensures whether the specified directory exists or not. If this directory exists, the return value is true (1)
	`bool FileExists(string strFileName)`
FileChangeExt	Changes an extension of specified filename
	`string FileChangeExt (string strFileName, string strNewExt)`
FileFullPath	Returns the full path of an indicated file name. It attempts to find the file by searching standard directories such as Windows, System32 and the directories specified in the PATH environment variable. If path cannot be found, the function returns the same string to the input one. If the specified file name is already full path the function does nothing
	`string FileFullPath (string strFileName)`

Other operations

The function ***InstallPackage*** installs the specified archive to the specified directory. By default archive has the "*.pkg" extension. Archives may be created by **CAB.exe** program.

```
bool InstallPackage(string strFormatPackageName, string
FolderToInstall).
```

7. Integrated Development Environment

Integrated development environment (IDE) gives user more suitable interface to work with translator and interpreter. It allows user to open program files, edit, translate and run them. IDE supports multidocument mechanism.

The integrated environment is presented by the project IDE. The environment also consists of the project manager, the class manager, debugger, editor and the set of tools such as the syntactic diagrams constructor, library viewer, statistic and optimization of grammar rules, etc.. The view of an environment is shown on fig.2.

View of integrated environment

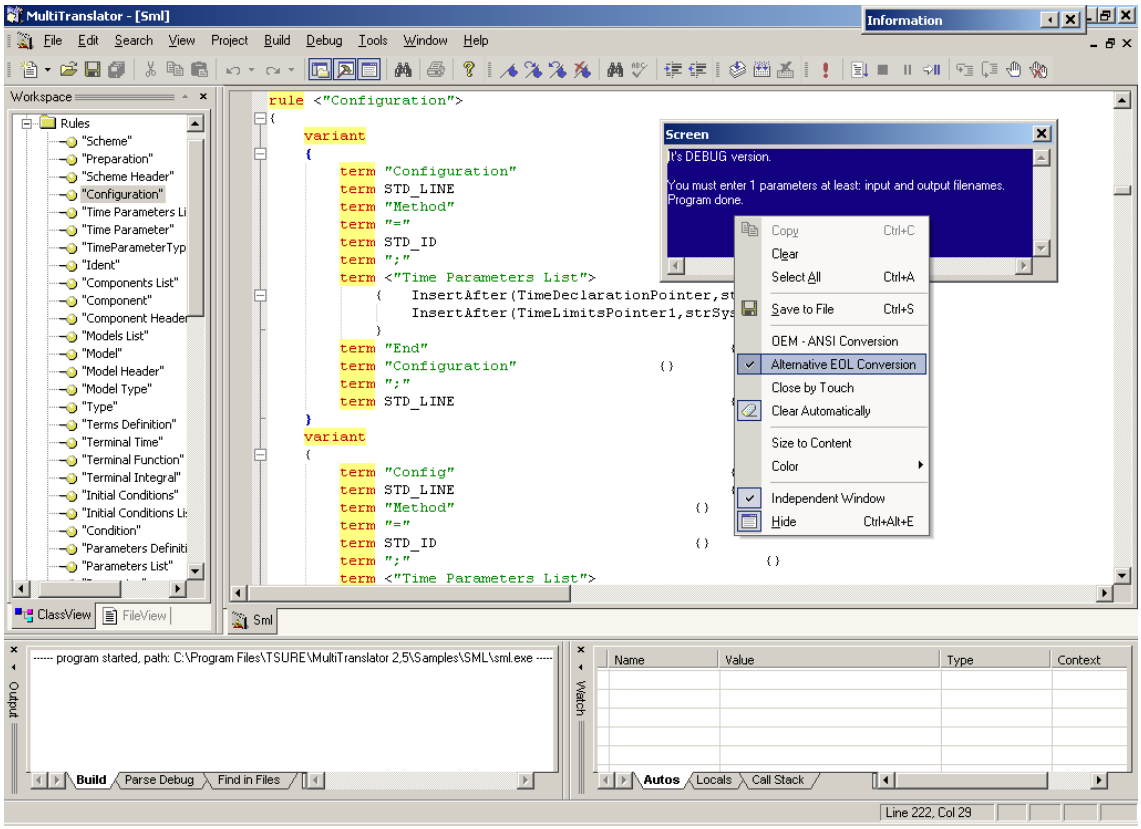


Fig.2.

The IDE environment works with the projects. For grammar description it is necessary to create the new project and to attach to it at least one grammar file (*.grm). The number of grammar files in the project is not limited. Each grammar file is compiled separately, and then all compiled object files are linked into the executable file. Besides of grammar files any other files can be included in a project. The files that have extension different from *.grm are not considered by MultiTranslator as a source files but accessible to view and edit.

In order to manage projects (include and exclude files from the project, open some project simultaneously, choose active project, etc) the project manager is engaged. The project manager comprises the list of the included files with their accessory to the certain group. The given list is represented as a tree.

The file from the tree can be loaded into the editor just by clicking on its name. The editor changes its own interface depending on a type of a file. It has the opportunity of syntax highlight that realized for a lot of types of documents. The syntax highlighting is accessible for the following files:

Kind	Supported extensions
Multitranslator files	*.grm; *.pjt; *.rpt
Acsl files	*.csl
Ada files	*.ads;*.adb
Avennue files	*.ave
Bullant files	*.ant
Apache Configuration files	*.conf
C++, C, Java, IDL, JavaScript files	*.c; *.cc; *.cpp;.*.cxx;; *.cs;.*.h;.*.hh;.*.hpp;.*.hxx; *.sma
Eiffel files	*.e
HTML files	*.html;*.htm;*.asp;*.shtml;*.php3;*.phtml;*.php;*.htt;*.cfm;*.tpl;*.dtd; *.xml;*.xsl;*.svg;*.xul;*.xsd;*.dtd
LaTeX files	*.tex;*.sty
lisp files	*.lsp;*.lisp
Lua files	*.lua
Crontab files	*.tab
Pascal files	*.pas;*.inc
Perl and Bash files	*.pl;*.pm;*.cgi;*.pod; *.sh;*.bsh
Python files	*.py
Ruby files	*.rb
SQL and PL/SQL files	*.sql; *.spec;*.body;*.sps;*.spb;*.sf;*.sp
Tcl and itcl files	*.tcl; *.itcl
VB files	*.vb;*.bas;*.frm;*.cls;*.ctl;*.pag;*.dsr;*.dob
Other files	*.properties;*.ini;*.inf;*.reg;*.url;*.cfg;*.cnf;*.aut; *.txt;*.log;*.lst;*.doc;*.diz;*.nfo; *.bat;*.cmd;*.nt; *.diff;*.patch; *.mak

It is possible to describe your own language for highlighting. The files of the descriptions are placed in the directory **Highlight**. The editor has a feature of an automatic formatting of the text, besides the text can be reformatted completely using command **"Format Selection"**

The project manager includes in the structure the classes' manager. Objectives of the given manager are creation of rules, structures, scanning of the source text and search of errors "passing by", drawing up of a tree of rules, structures and functions. The updating occurs automatically in process of the source text changing.

MultiTranslator has a set of standard toolbars, which appear when you start IDE for the first time. They contain the most popular commands. Their layout and contents may be changed further just by simple dragging or choosing the command **"Customize"**. The standard toolbars are shown below at the Fig 3.

View of standard toolbars.

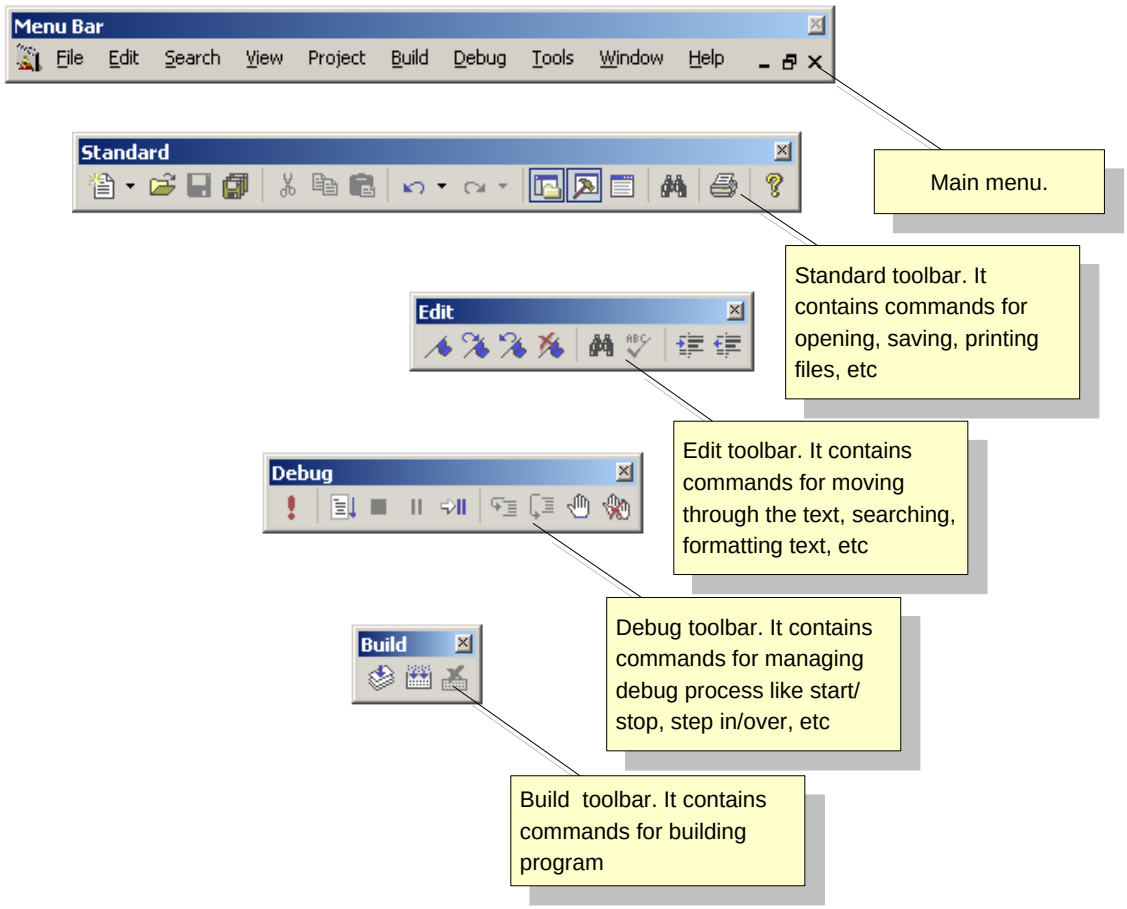


Fig.3.

The integrated environment includes debugger. The debugger can not only track the execution of actions of grammatical module, but also track the process of recognition of the input text. With its help it is possible to see the process of lexeme analysis and of moving on a grammar tree. The process of translation can be reflected either to **Parse Output Window** or to text file depending on specified options. There are to basic modes of logging the process of translation: complete log (all enters and exits from any rule are written to log) and success-point-way log (only success rules and terms are written to log). The second mode is shorter than the first one, but the first is more exploratory. The 2 ways of tracking the process of translation exists: the non-stop way and step-by-step way. Using the second way of tracking you can see the process of translation in interactive mode. To tune the output to reflection and the mode of tracking of translation you have to use set of commands in the menu “Debug\Parsing….”. The example of using the tracking is represented below, at fig 4

An example of using the tracking of translation process

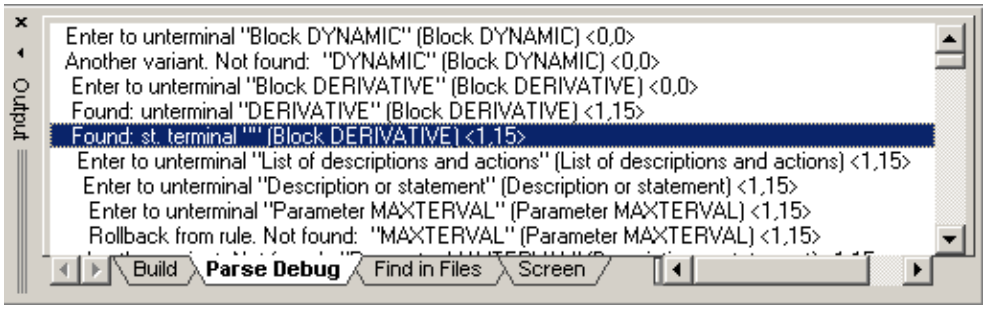


Fig.4.

The contents of any standard common run-time library can be seen by *Library Viewer*.(Fig 5) By default, this viewer shows the contents of **grmcrt.dll.**

View of Library Viewer

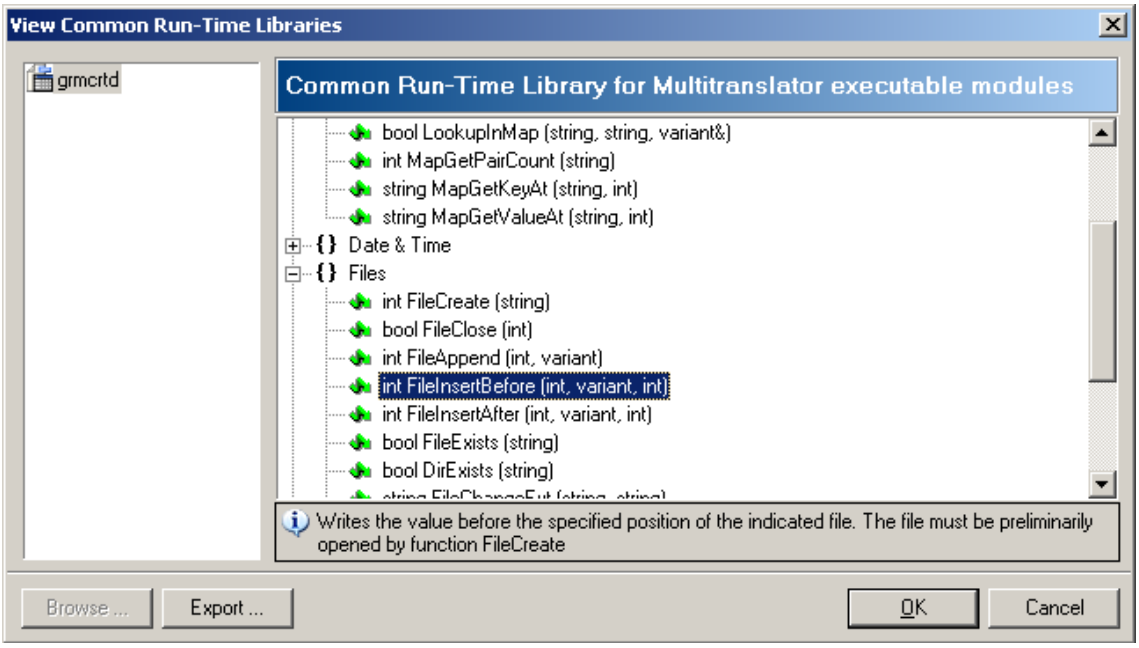


Fig.5.

Structure of the project also includes the syntactic diagram manager. Its task is creation of the syntactical diagram tree for available grammars. The manager displays all connections between rules, their relations, gives an opportunity of moving, scaling of a tree etc. The view of the syntactical manager is shown on the fig.6.

Syntactical diagram manager

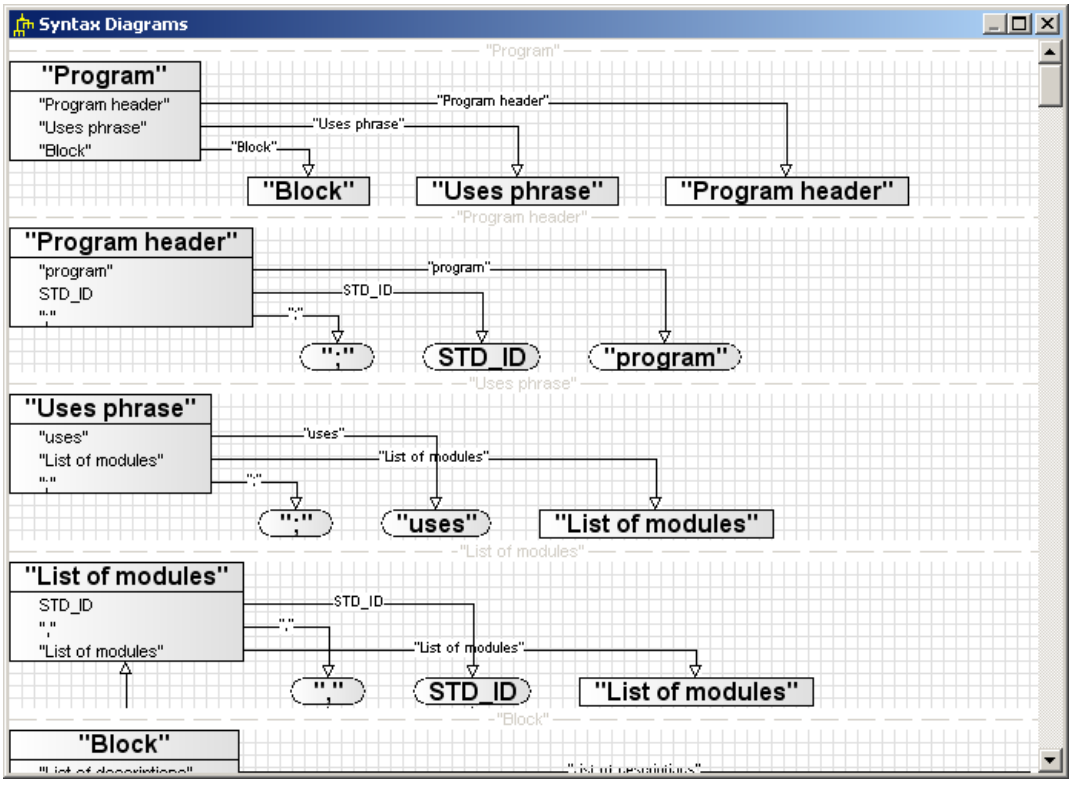


Fig.6.

8. System requirements for Multitranslator environment

For the project's functioning it is necessary to have:

- Minimal size of free disk space to complete project – 45 MB.
- Minimal memory size to complete project – 32 MB.
- Operation system – Windows 98/NT/2000/XP.

9. Using Multitranslator from other application based on COM technology

The Multitranslator is implemented as the automation server, that allows to use it from other program applications. Using OLE Automation you can fulfill the majority of operations accessible from the integrated environment, namely:

- to load and to unload the integrated environment;
- to open and to close the projects;
- to install the active project;
- to sort out all files, which are included in the project to add and to delete files from the project;
- to control customizations of the project, keys of compilation and other;

For Multitranslator usage from your programs it is necessary to use the compiler supporting import of type libraries (Type Library). The type library of the Multitranslator is included directly into the executable Multitranslator program and is accessible on the address:

($InstallPath)\ MTDev.exe,

where ($InstallPath) – the path, where the program was installed. By default program is installed into the directory: Program Files\TSURE\MultiTranslator.

Let's consider the examples of libraries import for several compilers.

For the Visual C ++ compiler it is necessary to write into the heading file (better into *stdafx.h*) the following string:

```
#import "C:\ProgramFiles\TSURE\Multitranslator\MTDev.exe"
```

The compiler will create the file MTDev.tlh, which contains all COM-interfaces translated on C ++. It can be included to any unit with the help of the directive #include:

```
#ifdef _DEBUG
        #include "Debug\\MTDev.tlh"
#else
        #include "Release\\MTDev.tlh"
#endif
```

For the Delphi compiler it is necessary to fulfill the following operations:

1. To create or to open the existing project.
2. To select the menu-item "Project\Import Type Library".
3. In the opened dialogue to find in the list MTDev (MultiTranslator) Type Library.
4. To push the button "Create Unit"

After execution of the enumerated operations Delphi will create the file *MTDevLib_TLB.pas*, which will include all COM-interfaces and MultiTranslator's data types translated into the ObjectPascal language.

At usage C ++ .NET or C# for importing the library it is necessary to fulfill the following steps:

1. To create new or to open the existing project
2. To open in Windows the command line and to run the command: TlbImp.exe "C:\Program Files\TSURE\MultiTranslator 2,7\MTDev.exe" /out: MTDev_TLB.dll. The program TlbImp.exe is included into .NET package. After execution it will create the file MTDev_TLB.dll. Having fulfilled the command ildasm MTDev_TLB.dll, you can view contents MTDev_TLB.dll by browse facilities of .NET.
3. In the project it is necessary to register the following strings:

```
#using <mscorlib.dll>
#using “MTdev_TLB.dll”

using namespace System::Runtime::InteropServices;
using namespace MTdev_TLB
```

The import of COM-library in any other compilers, supporting this technology, is implementing in similar way. To obtain the detailed instructions about installation rules of COM-libraries you should to apply for information to the reference book of your compiler.

All interfaces written on the IDL language (standard language for the interfaces description) and stored in COM-library of the MultiTranslator are listed and described below.

```
IApplication interface
//------------------------------------------------------------------------
// Interface IApplication
//------------------------------------------------------------------------
[
   object,
   uuid("EE3E904C-6AA0-401A-BCB5-A936E3B717F6"),
   dual, oleautomation, helpstring("IApplication Interface"),
   pointer default(unique)
]
__interface IApplication : IDispatch
{
   [id(1)] HRESULT Quit([in] VARIANT_BOOL SaveChanges);
   [id(2)] HRESULT ScreenRefresh(void);
   [id(3)] HRESULT Resize([in] LONG Width, [in] LONG Height);
   [id(4)] HRESULT Activate(void);
   [id(5)] HRESULT OpenProject([in]BSTR Path,[out,retval]IProjectDocument** pDoc);
   [id(9)] HRESULT CreateProject([in] BSTR Path, [out,retval] IProjectDocument **pDoc);
   [propget, id(6)] HRESULT Visible([out, retval] VARIANT_BOOL* pVal);
   [propput, id(6)] HRESULT Visible([in] VARIANT BOOL newVal);
   [propget, id(7)] HRESULT ActiveDocument([out, retval] IProjectDocument** pVal);
   [propget, id(8)] HRESULT MRUDocument([out, retval] BSTR* pVal);
};
```

The given interface is a basic one. With its help you can call the shell of the MultiTranslator (to load MTDev.exe or, if the MultiTranslator is already loaded, to connect with MT). Main commands:

- Quit - to complete operation and to unload the MultiTranslator
- Visible - to show or to hide the main form of the MultiTranslator
- Activate - to make the MultiTranslator active (to transfer its main form on the foreground always on top)
- Resize - to resize the main window
- OpenProject - to open the indicated project. The given method resets the pointer on IProjectDocument, which will be considered below
- ActiveDocument - return the current active project.

Interface of IProjectDocument project

This interface divided into two sub interfaces: the base IDocument and IProjectDocument:

```
//-----------------------------------------------------------------------
// Document
//-----------------------------------------------------------------------
[
   object,
   uuid("C7242A99-D06C-4A9E-818D-5D3180EB8D90"),
   dual, oleautomation, helpstring("IDocument Interface"),
   pointer_default(unique)
]
__interface IDocument : IDispatch
{
   [id(1)] HRESULT Save(void);
   [propget, id(2)] HRESULT Application([out, retval] IApplication** pVal);
   [propget, id(3)] HRESULT Modified([out, retval] VARIANT_BOOL* pVal);
   [propput, id(3)] HRESULT Modified([in] VARIANT_BOOL newVal);
};

//-----------------------------------------------------------------------
// IProjectDocument
//-----------------------------------------------------------------------
[
   object,
   uuid("C7242A99-D06C-4A9E-818D-5D3180EB8D99"),
   dual, oleautomation, helpstring("IProjectDocument Interface"),
   pointer_default(unique)
]
__interface IProjectDocument : IDocument
{
   [id(5)] HRESULT SaveAll([out,retval] VARIANT BOOL* Result);
   [id(6)] HRESULT Build([out,retval] VARIANT_BOOL* Result);
   [id(7)] HRESULT Run([out] LONG* ExitCode, [out,retval] VARIANT_BOOL* Result);
   [id(8)] HRESULT Clean(void);
   [id(9)] HRESULT AddFile([in] BSTR FilePath, [out,retval] VARIANT_BOOL* Result);
   [id(11)]HRESULT RemoveFile([in] BSTR FilePath, [out,retval] VARIANT BOOL* Result);
   [id(12)]HRESULT GetFileFirst([out]BSTR* FileName, [out,retval] VARIANT_BOOL* Result);
   [id(13)] HRESULT GetFileNext([out]BSTR* FileName, [out,retval] VARIANT_BOOL* Result);
   [propget, id(14)] HRESULT Active([out, retval] VARIANT BOOL* pVal);
   [propput, id(14)] HRESULT Active([in] VARIANT BOOL newVal);
   [propget, id(15)] HRESULT TargetPath([out, retval] BSTR* pVal);
   [propget, id(16)] HRESULT TargetDirectory([out, retval] BSTR* pVal);
   [propget, id(17)] HRESULT Notes([out, retval] BSTR* pVal);
   [propput, id(17)] HRESULT Notes([in] BSTR newVal);
   [propget, id(18)] HRESULT Params([out, retval] BSTR* pVal);
   [propput, id(18)] HRESULT Params([in] BSTR newVal);
   [propget, id(19)] HRESULT Switches([out, retval] BSTR* pVal);
   [propput, id(19)] HRESULT Switches([in] BSTR newVal);
   [propget, id(20)] HRESULT LangGens([out, retval] BSTR* pVal);
   [propput, id(20)] HRESULT LangGens([in] BSTR newVal);
};
```

Main methods of the interface:

- Modified - the flag, whether was modified the project;
- Build - to start compilation of the project;
- Run - to start the project on execution;
- AddFile - to add the file into the project;
- RemoveFile - to delete the file from the project;
- Active - to define, whether the project is active and to set it active.

In the directory ($InstallPath) \Type Library are placed the examples on the different programming languages on usage of the reduced MultiTranslator's interfaces.

10. Implementation of model generation wizard for Multitranslator

One of the main goals of Multitranslator is to come close to necessary format of generated models and also maximally reduce manual operations at models' development and finishing before their usage.

One of the problems originating at models' conversion into XML format, using *the Multitranslator,* is representation of models in this format. This representation is structurally rigid, functionally limited and demands exact definition of model as a set of branches connecting input, output and internal nodes. For each branch the equation describing the physical process flowing past in it should be given and the pair variables *across* and *through* (current and voltage for electrical circuits) should be defined. The equation should be solved for a variable *through* as: *through = f (across)*. At the same time, source languages (for example, Modelica), as a rule, implement flexible possibilities of descriptions of models, and for them such branch-nodes representation is only one of many alternatives of representations which is used not always. Completely the automatic process of conversion in this case is rather complicated, because it needs the intelligent analysis of the source text of model and implementation of symbolical conversions of source equations of model to a form *through = f (across)*.

Effective way to solve of this problem is dialogue interaction with the user during translation of the model's source text with the purpose of a solution of originating uncertainty. To automate the translation process *the Model generation wizard* has been created. It gives the user, in uncertain situations, a possibility to set branches and to assign appropriate equations and variables.

This Wizard is organized as a set of the functions obtaining the data on the equations, variables and the terminals of the source model according to the user's assignments. This set of functions is implemented as BranchWizard3 dll-library. It is simple enough to connect this library to grammar module *of the Multitranslator*, since it is invariant concerning source languages and can work together with any grammar modules. Let's consider functions of BranchWizard library.

Function `bool AssignBranches (string ports, string vars, string eqs)` - calls the dialogue of model's branches assigning. Ports parameter contains model's nodes obtained during the source text analysis where each node is presented by sequence of four parameters: "name", "mode", "nature", "type". These sequences are divided by a line feed symbol. For example, the representation of a two-terminal circuit with nodes anode1 and cathode1 will look like: *ports = "anode1 inout electrical terminal \n cathode1 inout electrical terminal"*. Parameter vars contains the list all identified variables of model as string (for example, vars = "i1 v1 v2 iC2 R1 R2 C1"). Parameter eqs contains the list of all identified equations of model as text string in which the equations are divided by line feed symbol (for example, eqs = "i1=u1 / (R1+R2) \n iC2=C*der (v2)"). The function returns *true*, if the user has confirmed the parameter setup and *false* otherwise.

Function `void AssignNodeTypes (string modes, string natures, string types)` - defining a set of possible values of parameters "mode", "nature" and "type" accordingly. Each value is separated by space. For example: modes = “in out inout”.

Function `void AddBrc (string name, string nodefrom, string nodeto, string across, string through, string equation)` - adds a branch with appropriate parameters to the list of branches.

After successful finishing of the branches assigning dialogue there is a possibility to get the information about branches, nodes and equations defined by the user. In this case used the appropriate functions:

`int GetNodeCount ()` - returns number of the model's nodes assigned by the user;

`int GetBrcCount ()` - returns number of the model's branches assigned by the user;

`int GetICCount ()` - returns number of the initial conditions assigned by the user (the common equations, which are not describing interaction between *across and through*).

Other functions return parameters of node, branch or equation by their index:

`void GetNodeData (int index, string name, string mode, string nature, string type)` - returns a name of a node ("name"), an operating mode of a node ("mode"), a physical nature of a node ("nature") and the type ("type").

`void GetBrcData (int index, string name, string nodefrom, string nodeto, string across, string through, string equation)` – returns a name of a branch ("name"), a source node ("nodefrom"), a destination node ("nodeto"), variables ”across”, ”through” and the branch equation ("equation") on the source language.

`void GetICData (int index, string eq)` – returns the common equation as “eq” string.

Also the following functions are accessible:

`bool EditEq (string err, string eq)` – show error message with the text "err" and after pressing "Ok" shows the window dialog with an edit field where it is possible to edit string "eq". This function returns "true" if user pressed “Ok” button and "false" otherwise.

`bool IsVarAcross (string var)` - checks, whether the variable "var" as *across* in any branch is assigned or not.

`bool IsVarThrough (string var)` - checks, whether the variable "var" as *through* in any branch is assigned or not.

All functions should be defined in the beginning of the grammar module:

```
extern void AssignNodeTypes(string modes, string natures, string
types) library "Add-ins\\BranchWizard3.dll","AssignNodeTypes";
extern bool AssignBranches(string ports, string vars, string eqs)
library "Add-ins\\BranchWizard3.dll","AssignBranches";
extern int GetBrcCount() library "Add-ins\\BranchWizard3.dll",
"GetBrcCount";
extern void GetBrcData( int index, string name, string nodefrom,
string nodeto, string across, string through, string equation ) library
"Add-ins\\BranchWizard3.dll","GetBrcData";
extern void AddBrc( string name, string nodefrom, string nodeto, string
across, string through, string equation ) library "Add-
ins\\BranchWizard3.dll","AddBrc";
extern int GetNodeCount() library "Add-ins\\BranchWizard3.dll",
"GetNodeCount";
extern void GetNodeData( int index, string name, string mode, string
nature, string type ) library "Add-ins\\BranchWizard3.dll",
"GetNodeData";
extern int GetICCount() library "Add-ins\\BranchWizard3.dll",
"GetICCount";
extern void GetICData( int index, string equation ) library "Add-
ins\\BranchWizard3.dll","GetICData";
extern bool EditEq(string err, string eq) library "Add-
ins\\BranchWizard3.dll","EditEq";
extern bool IsVarAcross(string var) library "Add-
ins\\BranchWizard3.dll","IsVarAcross";
extern bool IsVarThrough(string var) library "Add-
ins\\BranchWizard3.dll","IsVarThrough";
```

At translation of the source model the grammar module does not form the text on XML, but only recognizes the equations, variables and, probably, nodes, collects their representations in appropriate strings. The collected information is passed to the *AssignBranches* function which activates a dialog box. It displays the collected information and permits to create new nodes, branches, equations and assign their parameters by "drag-and-drop" appropriate elements. Before call *AssignBranches* it is necessary to define a set of possible values of parameters of nodes with help of *AssignNodeTypes* function. Also, if it is necessary, it is possible with the help of *AddBrc* function add elements to the list of branches:

```
if (!Parse("Programm"))
{
      print("Parsing failed\r\n");
      return ;
}

// allocate memory for strings
strBrcName = StrFillChar( ' ',64 );
strBrcNodeFrom = StrFillChar( ' ',64 );
strBrcNodeTo = StrFillChar( ' ',64 );
strBrcAcross = StrFillChar( ' ',64 );
strBrcThrough = StrFillChar( ' ',64 );
strBrcEquation = StrFillChar( ' ',512 );
strNodeName = StrFillChar( ' ',64 );
strNodeMode = StrFillChar( ' ',64 );
strNodeNature = StrFillChar( ' ',64 );
strNodeType = StrFillChar( ' ',64 );
strICEquation = StrFillChar( ' ',512 );

// define possible parameter values
AssignNodeTypes( "inout in out", "electrical mechanical thermal
optical magnetic", "terminal quantity signal logic" );

if ( AssignBranches( PortCount,strVariables,strEquations) )
{
      // user clicks “Ok” in the dialog box
      // obtain assigned by user nodes,
      // branches and equations
      // forms main sections of the XML-description
}
else
{
      // user choices “Cancel”
}
```

Reading of the data assigned by the user is implemented through call of appropriate functions, for example:

```
…
for( int i = 0; i < GetBrcCount() )
{
  GetBrcData( i, strBrcName, strBrcNodeFrom, strBrcNodeTo,
          strBrcAcross, strBrcThrough, strBrcEquation );
  // generate in output file branch section
  MakeBranchDefinition();
} next( i++ );
…
```

It is important, that equations returned by GetBrcData function represented in the source format. Therefore, they must be translated into XML format. For this purpose a file should be created and equation in the source format should be written into it. Then this file needs to be specified as source file for parsing. The parse starts from equation rule of grammar module. In the actions of this rule it is necessary to provide two alternatives of their call: at first translation when collecting equations of model and at translation of the final version of the equation. In the first case it is necessary to receive string with the equation in the initial format, and in second - with its XML representation. Also it is necessary to take into account, that the user can edit the equations, and, therefore can bring to them some errors. Function ParseEq shown below can handle such situation, calling in case of an error the edit box with equation need to be corrected.

```
void ParseEq( string eq )
{
	int res = 0;
	while ( !res )
	{
		// next cycle when error found
		// create source file for parsing
		int nFile = FileCreate("eqparse.tmp");
		FileAppend( nFile, eq+";" );
		string tmpstr = SourceFileName;
		SourceFileName = "eqparse.tmp";
		FileClose( nFile );
		// cleans XML-description
		strEquationXML = "";
		// parse equation
		res = Parse("Equation_Clause1");
		SourceFileName = tmpstr;
		// if errors were found,
		// a dialog window for error correction should be shown
		// othrewise writes translated equation
		// into destination file
		if (!res)
		{
			if ( EditEq( "Error in equation", eq ) )
				res = 1;
		}
		else
			write( strEquationXML );
		FileRemove("eqparse.tmp");
	}
}
```

Wizard's main dialog box is shown in a figure 1. On the right part of box ("Found terms") the information obtained by analysis of the model's source text is displayed. On the left part a state of the current branch ("Branch i of N") is shown. At the left below assigned branches ("Assigned Branches") showed in graph form. Movement on the list of branches is implemented with the help of "Next" and "Prev" keys, adding new elements into the list of branches and nodes with "Add" button, deleting of a current element from these lists – with "Del" button. Parameters of a selected node can be changed, having pressed "Edit button". The changes parameter values and addition to the common equations list realized via "drag-and-drop" operation from appropriate lists. When equation from source equations list is "drag-and-dropped" into common equation list, it deletes from source list. Double clicks on equation from the common equations list deletes this equation from this list and moves it to list of the source equations. The field "Branch equation" of a current branch permits editing of equation.

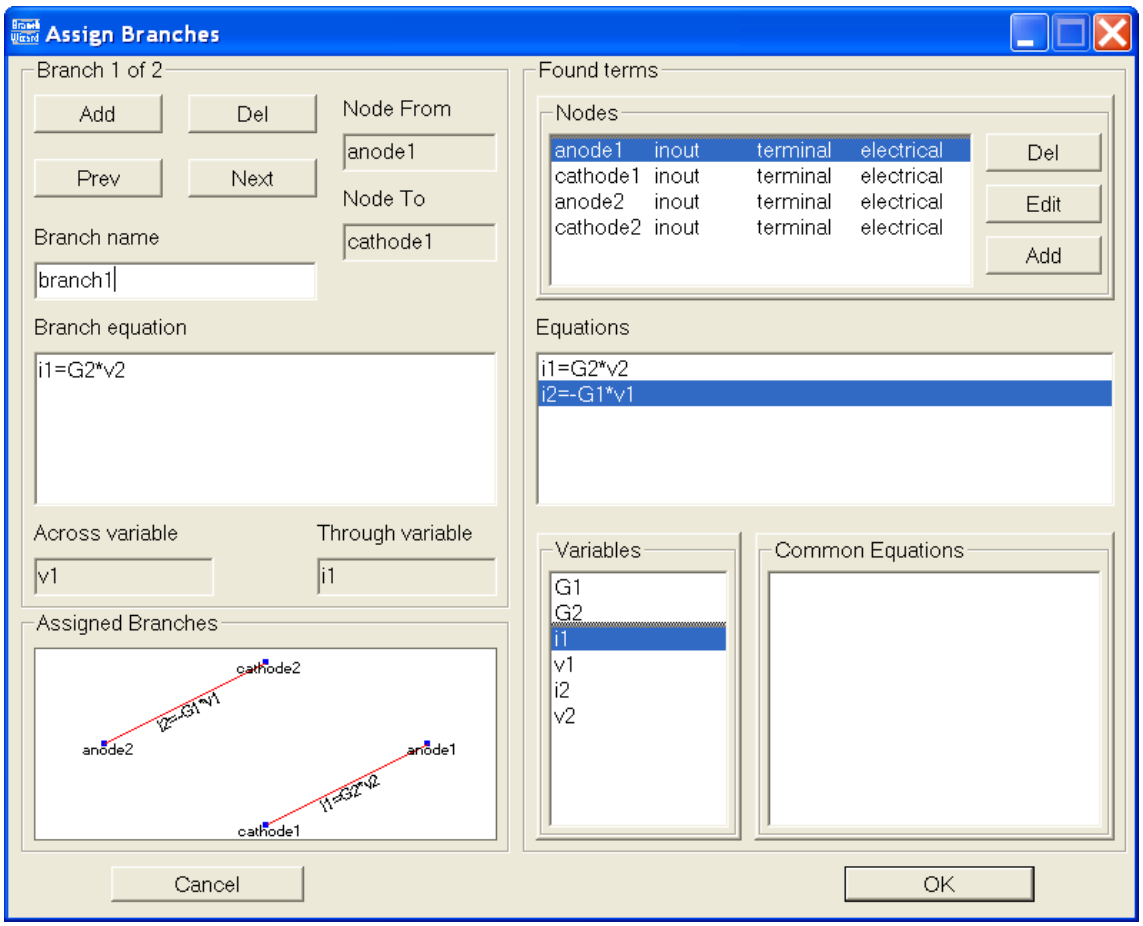


Figure 1 – Main dialog window of the model generation wizard