TSF: A Tool Set for Fortran Programs

F. Bodin, S. Chauveau, Y. Mével, R. Quiniou
(bodin,schauvea,ymevel,quiniou)@irisa.fr

contact: tsf@irisa.fr
http://www.irisa.fr/caps/PROJECTS/TSF/
IRISA - CAPS

April 30, 1999

Contents

1 Introduction

This document presents the transformation language for the TSF system.

The transformation language is used under an interpreted form and it aims at providing a way to add simple program transformations to Foresys. This project is partially supported by the FITS Esprit project:

http://www.vcpc.univie.ac.at/activities/projects/FITS/

2 Transformation program structure

Foresys is a product distributed by Simulog for the manipulation of Fortran codes. This document does not claim to describe Foresys but TSF (Tool Set for Fortran), one of its extensions (see figure 1). Foresys makes it possible to carry out many operations on Fortrans codes (translation to the new F90 standard, dependences analysis, ...). These tasks, although very interesting, do not always meet the user's needs. The TSF window is connected to one of the Foresys windows displaying the code of a Fortran unit (a function, a subroutine, a module, ...). This graphical interface provides two basical actions for final users.

- The application of simple scripts to various target ; the current selection, the full unit or the full Forlib[1]. This last functionnality is still experimental and is not fully implemented. The scripting language is quite simple and is dedicated to the manipulation of fortran codes. It provides some simple ways to transform and to get informations in fortran codes. To be executed, the simple scripting language must translated into the lisp dialect used internaly by Foresys. The user does not have to be concerned with this complex phase.

- The matching of a piece of Fortran code in the currently connected unit window according to a pattern (i.e. a model). The pattern can be generated automatically from a piece of fortran program or can be written in the pattern language (see section 11 for more details on the pattern language). The pattern is also used to check the validity of the selection before applying the associated script.

The scripts and the patterns are stored into external files with some additionnal informations (user's documentation, author, ...). The TSF interface provides some facilities to edit those informations but the file format is sufficiently simple so that the manual edition is possible. Several categories, each one being defined by a file, can be used simultaneously.

3 Installation and upgrade

TSF comes with the standard Foresys distribution but you have to ask Simulog[2] to upgrade your version or for a licence that enable TSF. This product is still in development and is subject to frequent changes. You can obtain the last update in our FTP site:

ftp://ftp.irisa.fr/local/caps/F90/TSF.patch.current.tar.gz

The only action of the update consists in installing this file in the Foresys directory. TSF is configured automatically by the standard Foresys distribution but some problems may occur if the following variables are not set properly:

- TSFHOME contains the directory where TSF is installed (i.e. probably identical to FORESYSHOME).

- TSF_PATH contains the path to find the TSF binaries. (i.e. probably $TSFHOME/bin)

- HOST contains the hostname of your computer. It is needed by a few scripts.

[1] A forlib is composed of a full fortran program with all its units.

[2] http://www.simulog.fr/

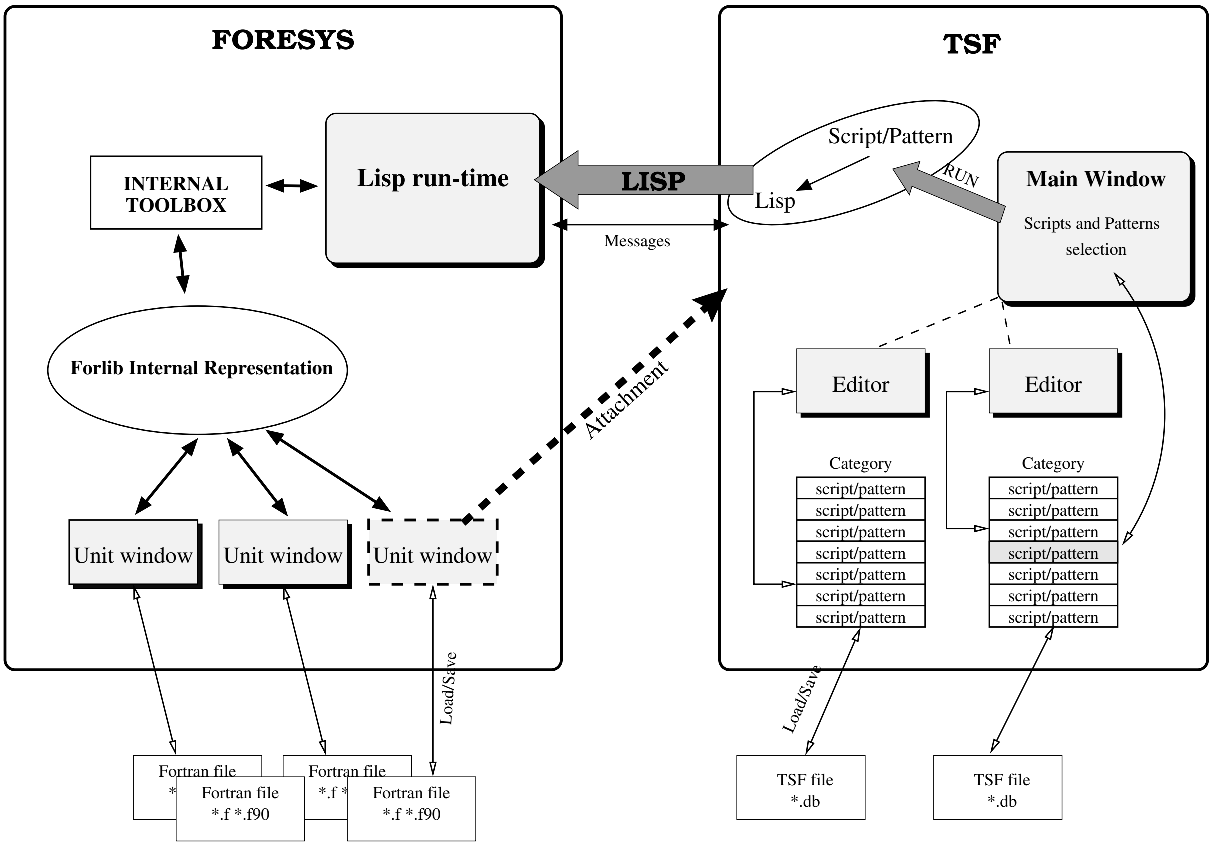

Figure 1: The internal structure of FORESYS/TSF

4 A simple tour in TSF

This section describes the steps of a simple use of FORESYS/TSF. We do not focus on the use of FORESYS but on that of TSF. The graphical interface is briefly described but more informations are given in section 5.2. The goal of this exemple is to normalize a Fortran loop.

▶ **Step 1 - Starting Foresys**

Before starting Foresys, we must create a Forlib that contains all the Fortran files used in your project. The simplest method is to execute a *forlib_create* command in the directory that contains your fortran codes:

forlib_create -name "ProjectName" *.f

A graphical interface is also provided in Foresys for this task (see the menu *Files*). Then we can execute the *foresys* command. We should obtain a window similar to the one given in figure 2.

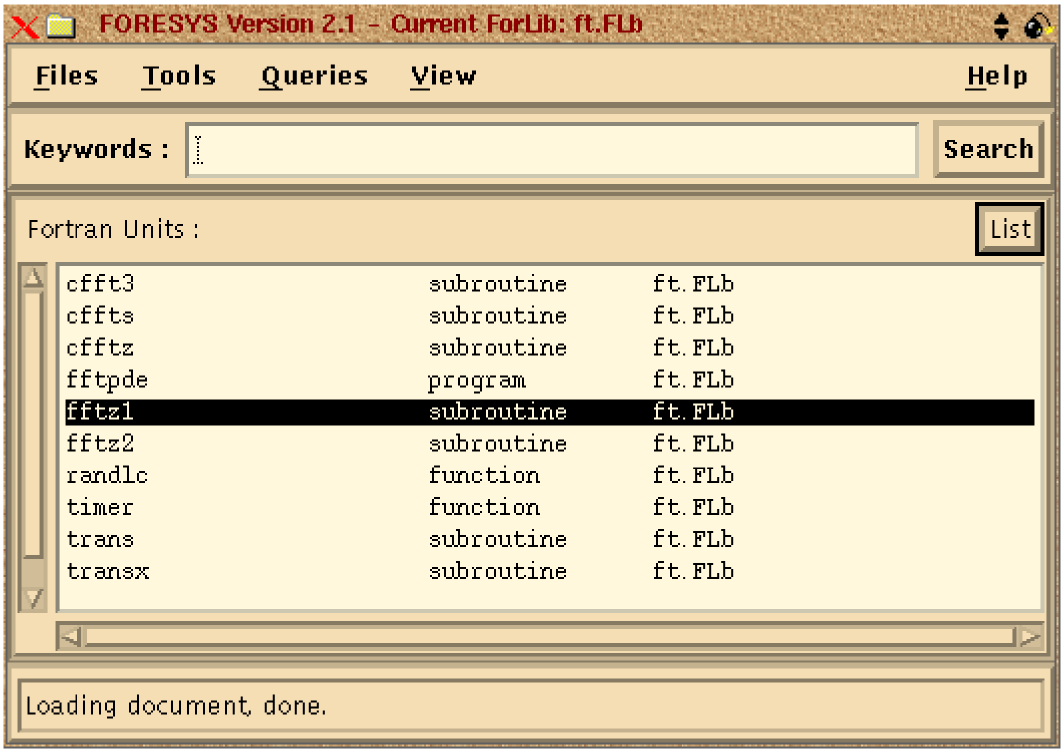

Figure 2: The main window of Foresys

▶ **Step 2 - The fortran window** Each fortran unit (i.e program, subroutine, function, module, ...) is associated to a window displaying the source code. The TSF window is created by using the *Connect* entry in the TSF menu. The *FFTPDE* window (see the Figure 3) is obtained by selecting this unit name in the main Foresys window. The TSF menu appears only if your version of Foresys supports TSF. In the contrary case see section 3. It is possible to open only one TSF window per Foresys session. By default all TSF actions are applied to the window where the menu was used for the last time.

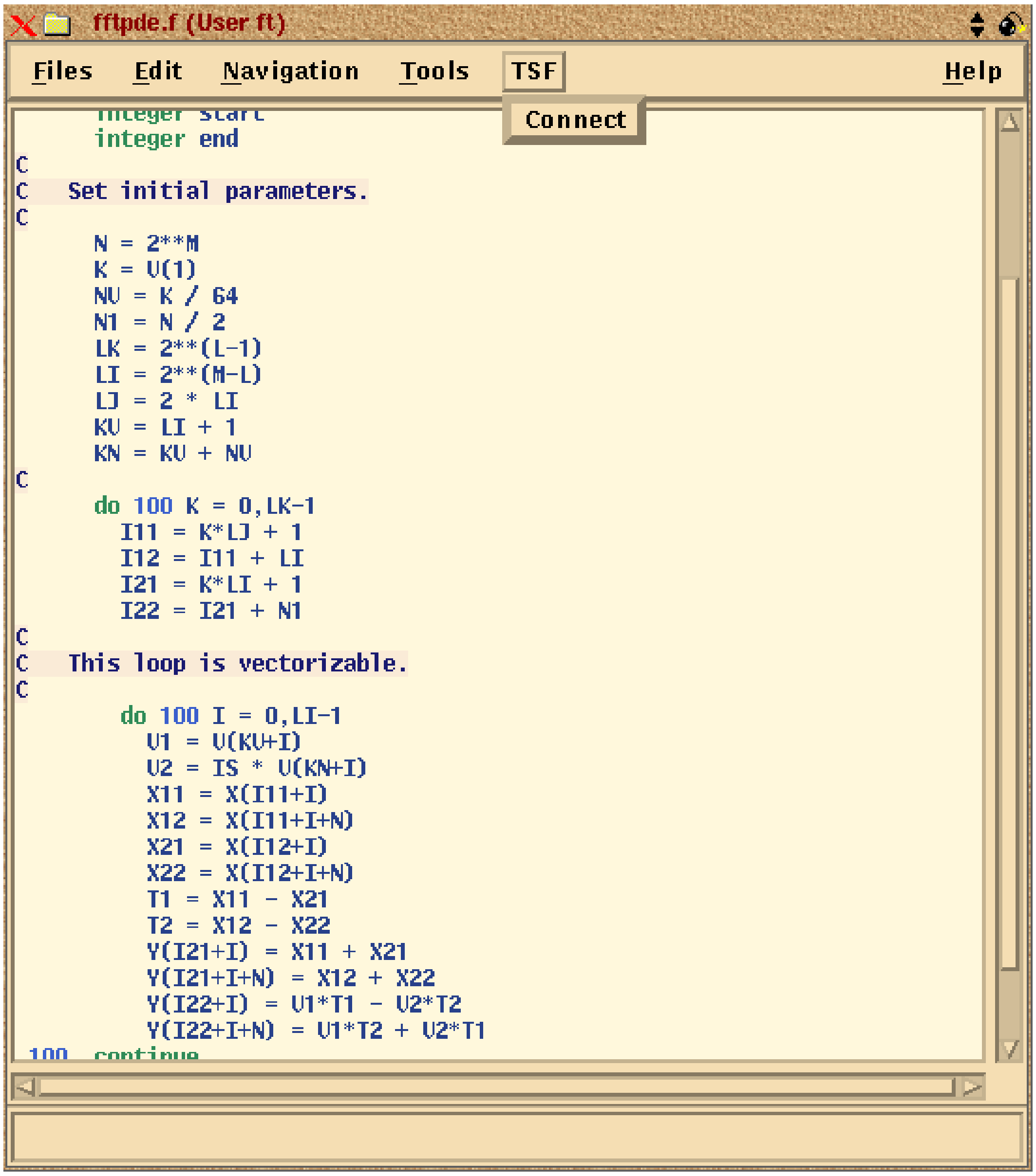

Figure 3: A Foresys fortran window with the TSF/Connect menu

- **Step 3 - The TSF window** An exemple of TSF window is given in the Figure 4. More detailled informations are given in section 5.2. The 'Loop Normalization' transformation is selected.
- **Step 4 - The main actions** The normalization transformation must be applied to a specific loop so the first step is to define the target. The simplest and most used method is to select a statement by hand in the fortran window. Another way is to use the pattern (see section 11) provided for the normalization ; the *Search* button in the TSF window allows to select one

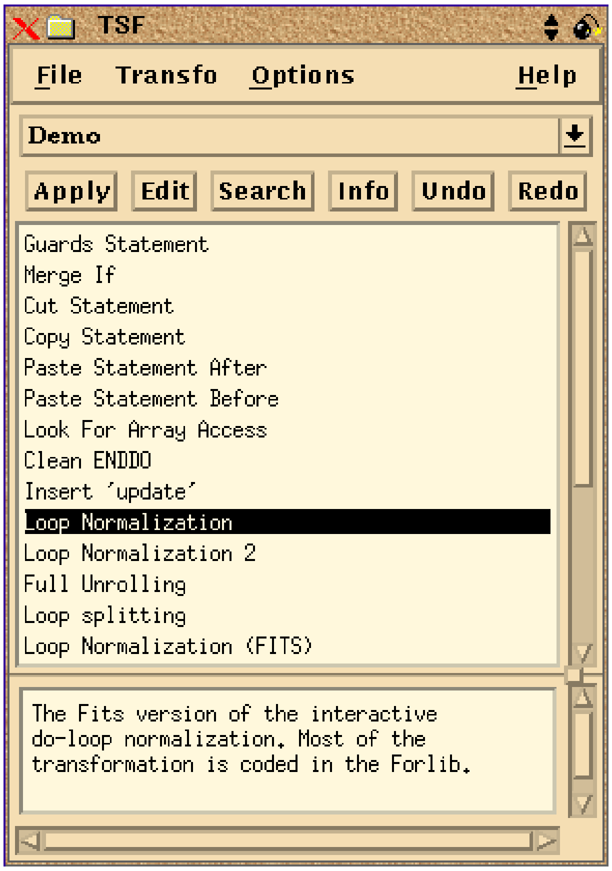

Figure 4: The main window of TSF

by one the loops in the fortran window (see the selection in the left window of the figure 5). More complex patterns can be used. For instance to ignore the loops with a step equal to 1. The right part of the figure 5 shows the resulting code after that the *Apply* button is pressed. The changes can be cancelled by the *Undo* button and then restored by *Redo* button.

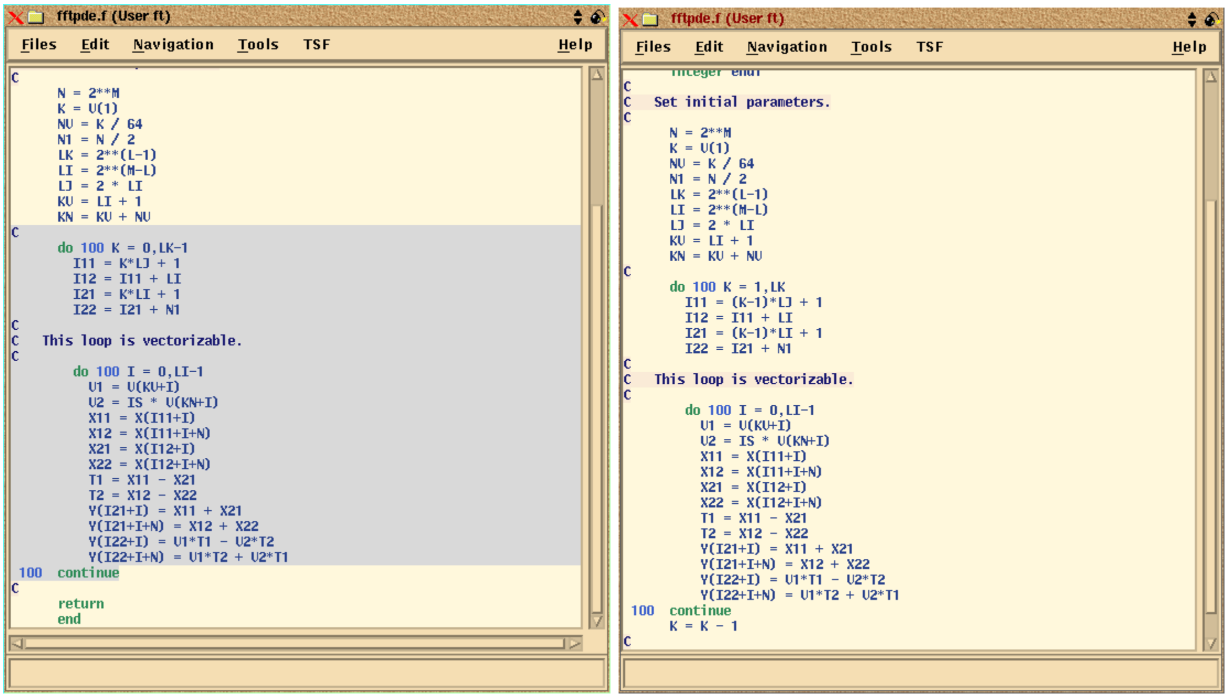

Figure 5: The Fortran window before and after the normalization

5 Interface with the Foresys tool

5.1 The toolkits

The main part of TSF is a graphical interface dedicated to the manipulation of a set of toolkits. Each toolkit is stored in text file that can be dynamically loaded. Several toolkits can be used simultaneously. Each toolkit entry consists in:

- a name used in the main TSF window to reference the entry ;
- an optional architecture name which describes the kind of the system for which the entry is dedicated ;
- an optional application name which describes the context in which the entry was created ;
- a pattern which describes the code fragments to match before applying the transformation script;
- a script which describes the actions applied to the fortran code ;
- a documentation that describes the usage of the entry.

The script is the probably the most important part so we often use the term *script* instead of *toolkit entry*. The documentation is also interesting since it allows the user to be aware of the usage and restriction of the proposed transformation. This is especially important as most techniques are under the control of the user. See example 1. The toolkit is an essential part of the system as it enables users to share program transformation scripts.

▷ **Example 1:** a TSF Toolkit Entry

```
StartCase 14 :
  Name : "insert mpi send for real";
  Application : "send data";
  Author : "xxxx";
  PatternIn : "";
  Transformation : "
     SCRIPT SendMPIReal ()
               ...
     ENDSCRIPT";
  Documentation : "
     insert a call to MPI send ...";
EndCase\\
```

Once loaded the transformations stored in the toolkit become available from a menu (see figure 6). The example 2 shows a script to insert a call to a MPI routine.

▷ **Example 2:** Insertion of a MPI call

```
SCRIPT Send_MPI ()
  IF (NOTUSED ("mpi_ierror")) THEN
       DECLARE("include 'mpif.h'")
       DECLARE("integer mpi_ierror")
  ENDIF
  newStat := PARSESTAT("call MPI_SEND(name,size,MPI_REAL,dest,
      type,MPI_COMM_WORLD,mpi_ierror)")
  verif := PARSESTAT("if (mpi_ierror .ne. MPI_SUCCESS)
      print *, 'ERROR WITH SEND MESSAGE'")
  PASTE("AFTER",verif, $csel)
  PASTE("AFTER",newStat, $csel)
ENDSCRIPT
```

TSF uses the extension '.db' for toolkit files and '.dbm' multiple toolkit files. A '.dbm' file must contain a list of toolkits files (see example 3).

▷ **Example 3:** The content of a simple '.dbm' file

```
/home/foo/TSF/trans/Demo.db
/home/foo/TSF/trans/Utils.dbm
/home/foo/TSF/trans/Unrolling.db
/home/foo/Work/Fortran/Cray/MPI.db
```

For convenience, the file '.TSF.dbm' is automatically loaded in the user home directory when TSF is started.

5.2 Graphical interface

The figure 6 shows the main part of the Motif interface developed on the Foresys tool.

1. A menu bar with the following entries:

   | **File**

      | **New** - Create a new unnamed toolkit.

      | **Open** - Load a toolkit.

      | **Delete** - Delete the current toolkit from the list.

      | **Save** - Save the current toolkit .

      | **Save As** - Save the current toolkit with a new file name.

      | **Save All** - Save the modified toolkits.

      | **Configure** - In the current version, this entry is used to set the name to the current toolkit.

      | **Quit** - stop the execution of TSF.

   | **Entry**

      | **Add** - Add an new empty entry in the current toolkit.

      | **Clone** - Add a copy of the selected entry in the current toolkit.

      | **Delete** - Remove the selected entry from the current toolkit.

   | **Options**

      | **Icons** - Toggle between text/icons buttons

   | **Help**

      | **Help** - Show the TSF help (not yet available)

2. The list of loaded toolkits.

3. A set of buttons.

   Apply - Run the script of the selected entry.

   Edit - Edit the script of the selected entry (see after).

   Search - Apply the pattern to search a matching Fortran code.

   Info - Open a dialog window to show the documentation of the selected transformation.

   Undo - Cancel the last transformation (if possible).

   Redo - Restore the last transformation cancelled by UNDO.

   Ast - Display the structure of the current selection. (see the attribute AST)

4. The list of entries available in the current toolkit.

5. The documentation of the selected entry.

The second kind of TSF window is the editor (see figure 7). An editor can be opened for each entry with the 'EDIT' button in the main window or by a double mouse click of its name.

1. A menu bar with the following entries:

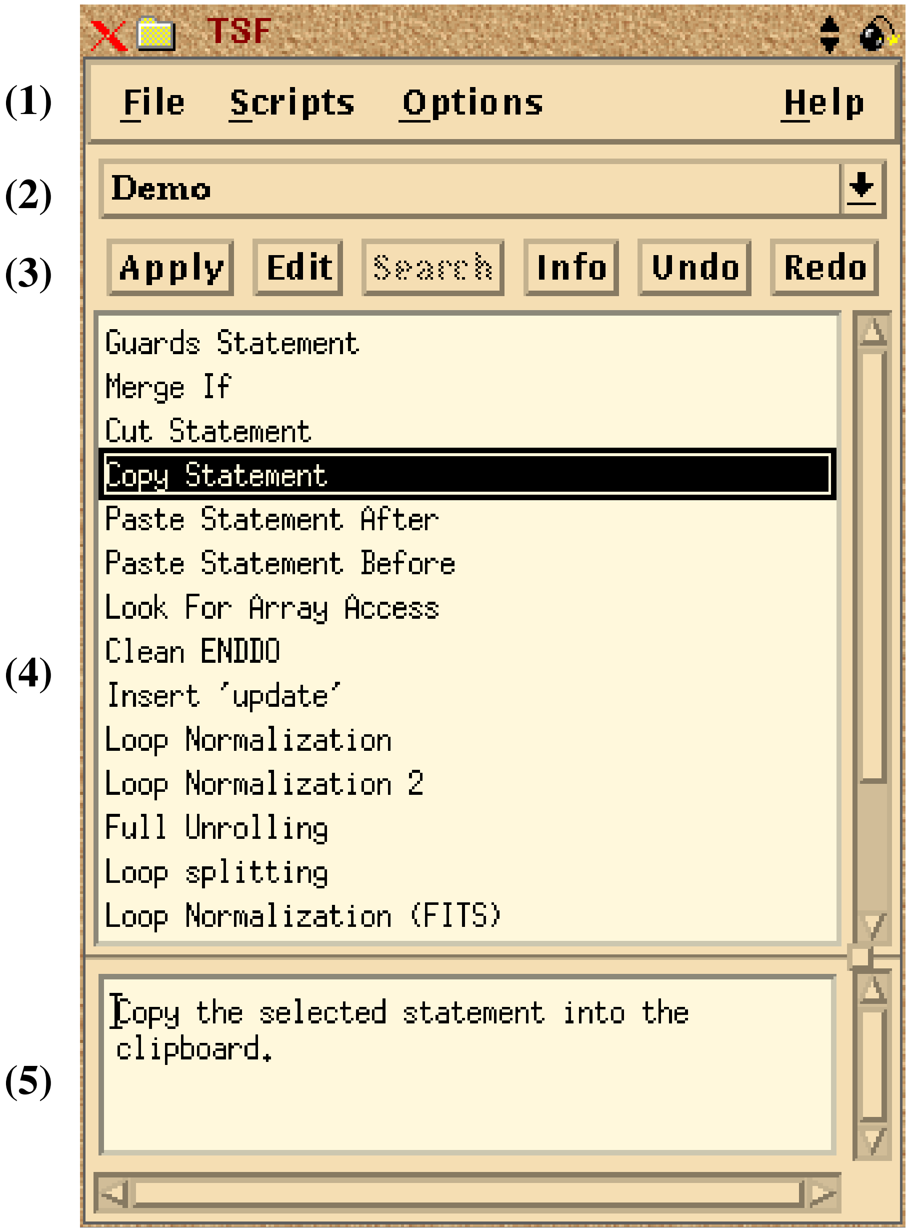

Figure 6: The main window of TSF

| **Case** | **New** - Create a new entry and clear all the fields.
| **Clone** - Create a new entry with the same fields.
| **Commit** - Report all changes in the toolkit.
| **Commit & Save** - Report all changes in the toolkit and save the toolkit
| **Commit As** - Equivalent to 'Clone'+'Commit'
| **Cancel changes** - Restore the fields as before the last 'Commit & Save'
| **Delete** - Remove the entry from the toolkit.
| **Close** - Close the editor.

| **Abstraction** - This menu provides some simple transformations on the pattern.

| **Option**

| **Auto Commit & Save** - Toggle the automatic saving of the fields before running the script.

2. A set of buttons (see the menu 'Case')
3. The **Field** page allows to set the name, the application, the architecture and the author of the entry.
4. The **Pattern** page contains the edition zone for the pattern and the following buttons:

   [Search] - Apply the pattern to search a matching Fortran code.

   [Get] - Use the current Fortran selection to create a pattern.

   [Check Syntax] - Verify the syntax of the pattern.

   The **Abstraction** submenu may be used at this level in order to remove some useless details.
5. The **Script** page contains the edition zone for the script and the following buttons:

   [Apply] - Apply the script to the current Fortran selection.

   [Check Syntax] - Verify the syntax of the script.

   [Goto] - Go to the specified line.
6. The **Documentation** page contains the edition zone for the documentation.

6 Expressions

6.1 Variables

The language used 4 kinds of variables:

- **predefined variables** - These variables concern the current context of the user. In order to distinguish themselves from other variables, they are prefixed by the *$* character. The list is given by the table 1.

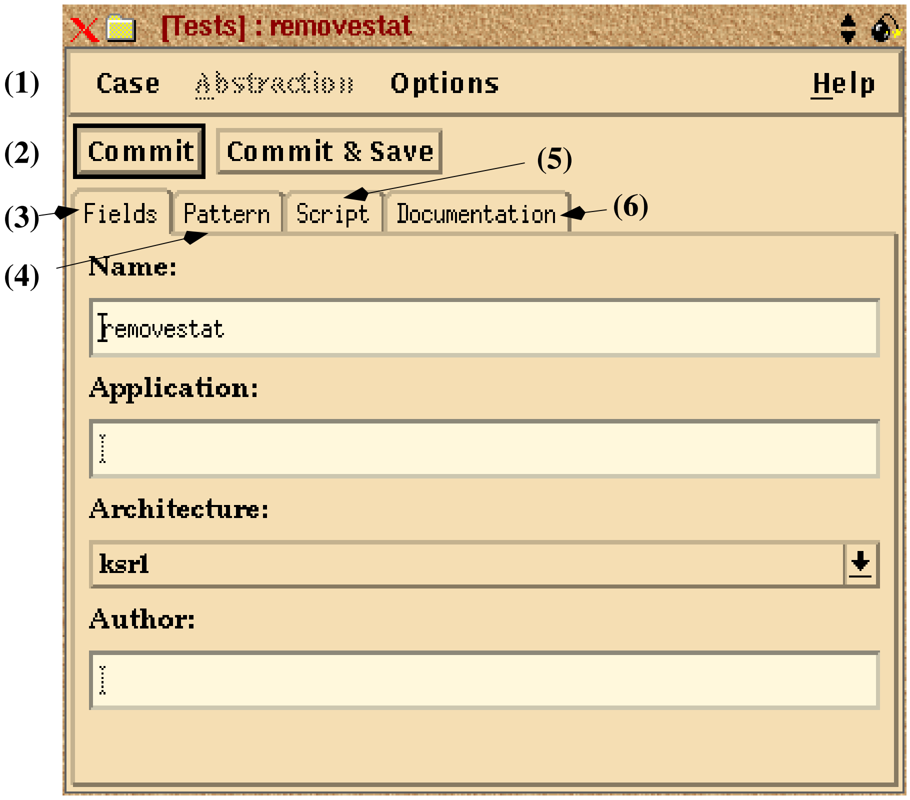

Figure 7: The TSF editor

Name	Type	Description
\$csel	AST	this variable is set when a pattern-matching is successful (mark the beginning of the fortran code matched). It may also be set through the displacement statements (see chapter 7.4) or directly by a selection made by the user in the current fortran window.
\$cstat	AST	this variable is set to the statement containing \$csel or to \$csel (if it is a statement).
\$cfile	Undocumented	This variable describes the file which contains the current unit.
\$cunit	AST	This variable is the root of the AST which defines the fortran code of the current unit. It allows to acces to several informations (statements, symbol table, forlib, ...)
\$cforlib	Undocumented	This variable allows to access to the current forlib.
\$csymtab	Undocumented	This variable allows to access to the current fortran symbol table. In fact, it is never used since the symbol table can be accessed via the unit (i.e. \$cunit)

Table 1: Predefined variables

- **pattern variables** - Those variables are used in pattern definitions. They can be read in the action field (transformation script) only if they are presented in the pattern definition. In a such case, they are used like pointers on the matched fortran program. According to their specific usage, these variables are prefixed by the *?* character.
- **local variables** - Those variables are local to the script which creates them. They are not declared.
- **global variables** - Those variables are declared by a **GLOBAL** statement at the begining of each script which want to use them.

  ▷ **Example 4:** A counter of calls of script

```
SCRIPT hello ()
  GLOBAL call_cpt
  call_cpt := call_cpt+1
  PRINT "hello"
  CALLSCRIPT world()
ENDSCRIPT

SCRIPT world ()
  GLOBAL call_cpt
  call_cpt := call_cpt+1
  PRINT "world"
ENDSCRIPT

SCRIPT main ()
  GLOBAL call_cpt
  call_cpt := 0
  CALLSCRIPT hello()
  PRINT call_cpt
ENDSCRIPT
```

6.2 TSF types

The following documented types are used in the TSF language:

- ▶ **integer** - The integer values have an infinite precision.
- ▶ **real** - The real values are coded by a single precision variable.
- ▶ **string** - The constant strings are bounded by the character ".
- ▶ **boolean** - The predefined constants **TRUE** and **FALSE** are used to create boolean values.
- ▶ **tree** - Fortran expressions and statements are coded in a AST structure.
- ▶ **vector** - A vector is a sequence of values of any types. Each element is referenced within [] by an integer index.

- **hash** - A hash table is a sequence of values of any types. Each element is referenced within by an unique key. We recommand to use simple types for keys (integer, real, string or boolean).

Most of the predefined variables as well as the pattern variables (except $cforlib, $csymtab) are of type *tree*. TSF uses many other undocumented types (lists, tables, symbols, ...). They can be exploited by internal functions. For instance, the type of the predefined variable $symtab is undocumented.

6.3 Operators

The language has the usual logical operators such as: ==, ! =, <, >, <=, >=, |, & and ! as well as the arithmetic operators +, −, ∗ and /. The addition operator + is also used for the string concatenation. The operator **OR** (resp. **AND** and **NOT**) is a synonym for | (resp & and !).

▷ **Example 5:**

```
step := 10
PRINT "The step is "+TOSTRING(step)
```

▷ **Remark**: Most operators can be applied to expressions of different types. For example, the multiplication of an integer by a AST of variant "int_cst" (i.e. a fortran constant) will produce an integer value ; this AST is automatically casted into an integer value. When an appropriate cast can not be found, most operators return FALSE.

7 Statements

7.1 Control Statements

7.1.1 Scripts

Several script constructions can appear in a TSF program. Each one defines a sequence of statements that can be called from another script. The main script (i.e. that is executed to start the program) is the last one and it does not accept parameters.

▷ **Syntax**:

SCRIPT *name* (*list of parameters*)
    list of statements
ENDSCRIPT

When a program contains several script, the execution starts by the last one which can not accept parameters. The statement **CALLSCRIPT** (i.e. or the obsolete **COMMAND**) is used to call a script.

▷ **Example 6:** A 'Hello World' program with two SCRIPT.

```
SCRIPT my_print(str)
  PRINT str
ENDSCRIPT

SCRIPT main ()
  PRINT "Hello "
  CALLSCRIPT my_print("World")
ENDSCRIPT
```

By default, the argument passing protocol is by value (i.e. the script works on a copy). It is possible to use references by adding the prefix "&" before the variable used as an argument. For instance, the example 7 implements a script that increments its argument (if a reference is used of course).

▷ **Example 7:**

```
SCRIPT inc(X)
  X := X+1
ENDSCRIPT

SCRIPT main()
  a:=1
  CALLSCRIPT inc(&a)
  PRINT a   // print 2
  b:=1
  CALLSCRIPT inc(b)
  PRINT b   // print 1 (b is unchanged)
ENDSCRIPT
```

▷ **Remark**: Arguments passed by reference are not restricted to the scripts. Some predefined functions or attributes may use them.

7.1.2 statements vs expressions

Contrary to certain languages, " FITS " does not make it possible to use an expression like a statement. A common way to evaluate an expression without using its result is to store it into a variable.

7.1.3 Conditional statements

The language supports the traditional **IF** statement with some optional **ELSE** and **ELSEIF** parts.

▷ **Syntax**:

```
IF (condition) THEN
  list of statements
ELSEIF (condition) THEN
  list of statements
ELSEIF (...) THEN
```

```
  ...
ELSE
  list of statements
ENDIF
```

The condition is not necessary a boolean expression. A condition is considered as true when it is not equal to the constant FALSE. The ELSEIF statement does not exist in some previous versions of TSF.

7.1.4 Iterative statements

The language supports the FOR and the WHILE iterative statements.

▷ **Syntax**:

```
FOR index := start,end  DO
  list of statements
ENDFOR
```

▷ **Syntax**:

```
WHILE (condition)
  list of statements
ENDWHILE
```

7.1.5 Termination statements

▶ **BREAK**

The statement allows to stop the execution of the current loop. The BREAK statement does not exist in some previous versions of TSF.

▷ **Example 8:**

```
line := $cstat.LINENUMBER
cpt :=0
WHILE (cpt!=10)
  PRINT cpt
  IF (cpt==line) THEN
    BREAK
  ENDIF
  cpt:=cpt+1
ENDWHILE
```

▶ **EXIT**

The statement stops the execution of the program.

▶ **QUIT**

The statement stops the execution of the current script.

▶ **RETURN**

The statement stops the execution of the program. This is a alias for **QUIT**.

▶ **ERROR**

The statement stops the execution of the program and generate an error message. It takes two argument. The first is an error number given by the user and the second is the error message. The ERROR statement does not exist in some previous versions of TSF.

▷ **Example 9:**

```
IF ($cstat.VARIANT!="do") THEN
   ERROR(1,"Please select a DO statement")
ENDIF
```

7.1.6 Assignment statements

The language supports two different types of assignments. The := operator is used to assign a variable.

▷ **Example 10:**

```
x := 1
str := "hello"
```

The second kind of assignment `<-` is used to replace a fortran code (i.e. a AST) by another code. Its left hand side is not necessary a variable

▷ **Example 11:** Replacement of the step of a DO loop statement by the constant 2

```
stat.STEP <- PARSEEXPR("2")
```

▷ **Example 12:** Replacement of the selected statement by 'print *,100'

```
$cstat <- PARSESTAT("~~~~~~~~~print *,100")
```

▷ **Remark**: When the left hand side of the AST assignment is a variable name, this variable is assigned with the AST. So, there is an important difference between the expression ($cstat) and the variable $cstat. For example, the previous transformation is equivalent to the following:

```
tmp := PARSESTAT("   ~~~~~~~~~print *,1")
($cstat) <- tmp
 $cstat := tmp
```

7.2 External tools

TSF provides some simple functions and statements to open a socket connection with an external program. This is an experimental feature that is subject to major changes and improvements. The protocol of communication is described in a C library distributed with TSF (see section 7.2.2).

7.2.1 The TSF part

- ▶ **STARTTOOL**
  The statement starts the program and open the connection. Its single argument is the name of the program.

```
STARTTOOL("/udd/foo/bin/server")
```

- ▶ **CLOSETOOL**
  The statement closes the previously opened connection. Its single argument is the name of the program (it is not used).

```
CLOSETOOL("/udd/foo/bin/server")
```

- ▶ **SEND**
  The statement sends its single argument to the program. This argument can be a string or a string or an integer.

```
SEND("Hello World")
SEND(4)
```

- ▶ **RECV**
  The function waits for a message from the program and returns a string. An obsolete statement **RECV** is also available. Its single argument, a variable, is assigned with the received message.

```
mess:=RECV()
RECV(mess)   // Obsolete
```

The following example use an external tool to translate the French comments to English comments. The script sends a command name, the string to be translated and waits for the answer.

▷ **Example 13:**

```
SCRIPT TranslateComment(stat)
  com :=  stat.COMMENT
  IF (com!=FALSE) THEN
    STARTTOOL("/udd/foo/bin/translator")
    SEND("FrenchToEnglish")
    SEND(com)
    res := RECV()
    CLOSETOOL("/udd/foo/bin/translator")
    INSERTCOMMENT(stat,res)
  ENDIF
ENDSCRIPT
```

7.2.2 The tsfcom library

This section describes the C library **tsfcom**. It defines the protocol of communication used to implement the external tools.

- int TsfCom_init(int *argc,char **argv)
Initialize the library. Must be called with the arguments given to the program. The arguments used by TsfCom_init are removed from the list. The function returns a non zero value when the initialization fails.

- int TsfCom_connect(int s,int ms)
Open the connection and returns 0 if something goes wrong. When nothing appends after *s* seconds and *ms* microseconds then the connection fails and returns a non zero value.

- int TsfCom_pending(void)
Return 1 if an event is waiting else 0.

- int TsfCom_next_event(TsfComEvent *event)

The function does not return before an event occurs. The type of the event is given by `event->type` among the following values. The function returns a non zero value when an error occurs.

- TSF_RECV_INT : an integer (32 bits) has been received. Its value is stored in `event->intval`
- TSF_RECV_STRING : a string has been received. Its value is stored in `event->stringval`.
- TSF_CLOSE : The connection has been closed.

The function returns a non zero value when an error occurs.

- int TsfCom_close(void)
Close the connection. The function returns a non zero value when an error occurs.

- int TsfCom_send_int(int mesg)
Send an integer value (signed 32 bits). The function returns a non zero value when an error occurs. Warning: this function is not yet implemented. So for the moment, send the integer value as a string (see the next function) and cast it to an integer with the TOINT TSF function.

- int TsfCom_send_string(const char *mesg)
Send a string. The function returns a non zero value when an error occurs.

```
#include <tsfcom.h>

int main(int argc, char **argv)
{
  TsfComEvent event ;
  char *str ;
```

```
  if (!TsfCom_init(int *argc,char **argv))
     error("The TSF arguments are missing\n") ;

  if (!TsfCom_connect())
     error("The connection with TSF has failed\n");

  while(1) {

    while(!TsfCom_pending())
      do_something_else() ;

    TsfCom_next_event(&event) ;
    switch(event.type)
     {
       case TSF_RECV_INT:
        printf("I have received the value %d\n",event.intval) ;
        break ;
       case TSF_RECV_STRING:
        /* create a copy of the string */
        str = malloc(sizeof(char)*strlen(event.stringval)+1) ;
        strcpy(str,event.stringval) ;
        /* print a message */
        printf("I have received the string '%s'\n",event.stringval) ;
        /* send the answer */
        if (strcmp(event.stringval,"PI")==0)
          TsfCom_send_string("3.1415") ;
        else if (strcmp(event.stringval,"SEVEN")==0)
          TsfCom_send_int(7) ;
        /* Free the memory allocated for the copy*/
        free(str) ;
        break ;
       case TSF_CLOSE:
        printf("The connection is closed\n") ;
        exit(0) ;
        break ;

     }
  }
}
```

7.3 IO statement

- **PRINT**
  The statement displays the value of it single argument.
- **SYSTEM**
  The statement executes the UNIX command given by its single argument.

▷ **Example 14:** Run the 'make' command in a X terminal

```
SYSTEM("xterm -e make")
```

▶ **OPEN**
The function opens a file. Its first argument is the file name. The second argument is a string that describe the opening mode:

- "r" = open for reading
- "w" = truncate to zero length or create for writing
- "a" = append; open for writing at end of file, or create for writing

The obsolete predefined constants **RD**="r", **WR**="w" and **AP**="a" are available. The **OPEN** function returns the file descriptor used by **CLOSE** and **WRITE**.

▶ **CLOSE**
The function closes the file specified by its single argument.

▶ **WRITE**
The function writes the value of its second argument in the file specified by its first argument.

▷ **Example 15:**

```
fid := OPEN("myfile.txt")
year := 1998
WRITE(fid,"IRISA ")
WRITE(fid,year)
CLOSE(fid)
```

7.4 Browsing in the source code

The fortran code is described in a tree called an AST. The first node of a AST is called the root node. In practice, we use the term *node* or *root node* to reference the first node. In order to move in a AST, we introduce the following attributes and functions:

▶ ***t*.VARIANT**
The expression returns a string that describe the nature of the node t (see the attribute **AST** to obtain more informations about variants). For example, the variant "ass" is used for assignment statements.

▷ **Example 16:** Verify that the selected statement is a DO loop.

```
IF ($cstat.VARIANT!="do") THEN
   ERROR(1,"Please, select a DO statement")
ENDIF
```

▷ **Remark**: In some previous versions of TSF, a comparison such as `($cstat.VARIANT!=do)` was possible. The nature of the identifier `do` is ambiguous ; is `do` a variant/string or a TSF variable ? This syntax is not supported in by new versions of TSF.

▶ ***t*.AST**
The expression returns a string that describe the structure of the AST t. The table 2 shows some examples. The following rules can be used to read those descriptions

$var(p_1, p_2, .., p_n)$: The AST of variant var has n children $p_1..p_n$. The number of arguments is constant.

$var[p_1, p_2, .., p_n]$: The AST of variant var has n children $p_1..p_n$. The number of arguments is not constant.

$(var \quad v)$: The AST is a terminal node (i.e. no child) of variant var associated to the value v. The nature of v depends of the variant. (see CHILD(0)).

$node\{...\}$: An attribute attached to the $node$. For example, a comment attached to a statement.

▶ ***t*.NBCHILD**
The expression returns the number of children of the AST t.

▶ ***t*.CHILD(*n*)**
The expression returns the child of rank n of the AST t. The rank of the first child is 1 and the rank of the last one is *t*.**NBCHILD**. The rank 0 is used to reference the value associated to an terminal AST.

▶ ***t*.PARENT**
The expression is the opposite of **CHILD**.

▶ ***t*.RANK**
The expression returns the index of t in list of children of its parent.

$$t \equiv t.\mathbf{PARENT}.\mathbf{CHILD}(t.\mathbf{RANK})$$

▶ ***t*.FIRST**
The expression returns the first statement in the AST t. When applied to $cunit,the **FIRST** attribute returns the first statement of the fortran function or subroutine. When applied to a loop of to a conditional statement, it returns the first statement of the body.

▶ ***t*.LAST**
The expression is the opposite of *t*.**FIRST**. Its returns the last statement in the AST t.

▶ ***t*.SKIP**
The expression returns the next instruction at the same level of the AST t. (see figure 8 for more details).

▶ ***t*.NEXT**
The expression returns to the next instruction whatever its level may be in the code.

Description	**Fortran expression**	**the result given by AST**
A variable	x	name x
A real constant	0.5	real_cst 0.5
A binary operator	i*2	mul(name i,int_cst 2)

An expression with several operators	(i*2)+j/3-k	sub(
		  add(
		    parenth(mul(name i,
		                int_cst 2)),
		    div(name j,int_cst 3)),
		  name k)

An assignment	i=j	ass(name i,name j)

A conditionnal statement, a function call and some sequences of statements	IF (x .GE. 0) THEN
	  y = x
	ELSE
	  PRINT *, 'Warning'
	  y = abs(x)
	ENDIF
		struct_if(
		  none(),
		  ge(name x, int_cst 0),
		  l_stat[ass(name y, name x)],
		  l_stat[
		    print(star(),
		         l_ioexp[
		           fstring_cst[
		            atom_string_cst Warning]
		           ]),
		    ass(name y,
		        func_call(name abs,
		                  l_exp[name x]))
		    ])

A DO statement with bounds, a labeled statement and a comment	C A COMMENT
	   DO 10 i=1,n,2
	     PRINT *,i
	10 CONTINUE
		do (
		  none(),
		  int_cst 10,
		  bounds(name i,
		         int_cst 1,
		         name n,
		         int_cst 2),
		  l_stat[
		      print(star(), l_ioexp[name i]),
		      labstat(int_cst 10, continue())
		      ]
		  ){^prefix
		      comment_s[comment A COMMENT]}

Table 2: a few examples of AST structures

- ▶ ***t*.PREVIOUS**
  The expression returns the previous instruction whatever its level may be in the code. It is the opposite of *t*.**NEXT**.

The figures 8 illustrates the use of these functions. Note the presence of the keyword header which enables us any manipulations (cut or paste for instance) at any level in the fortran code without being worried by losing the current level.

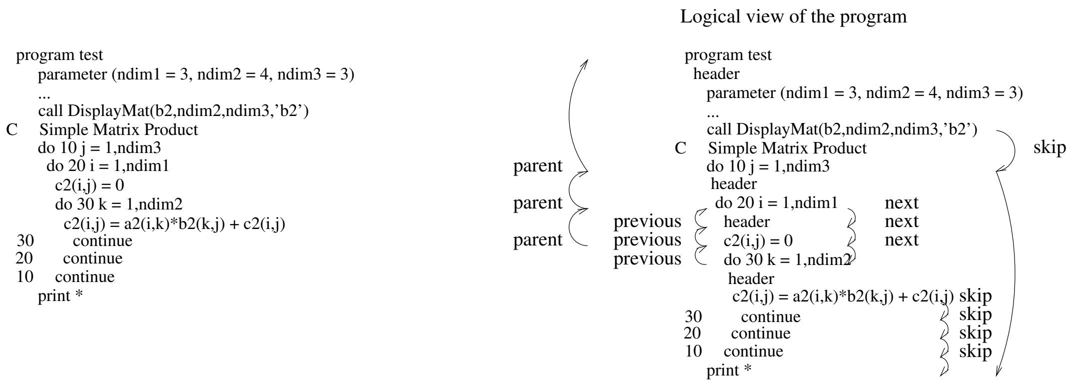

Figure 8: Browse operators in a program

The figure 9 illustrates the same functions on a fortran AST view.

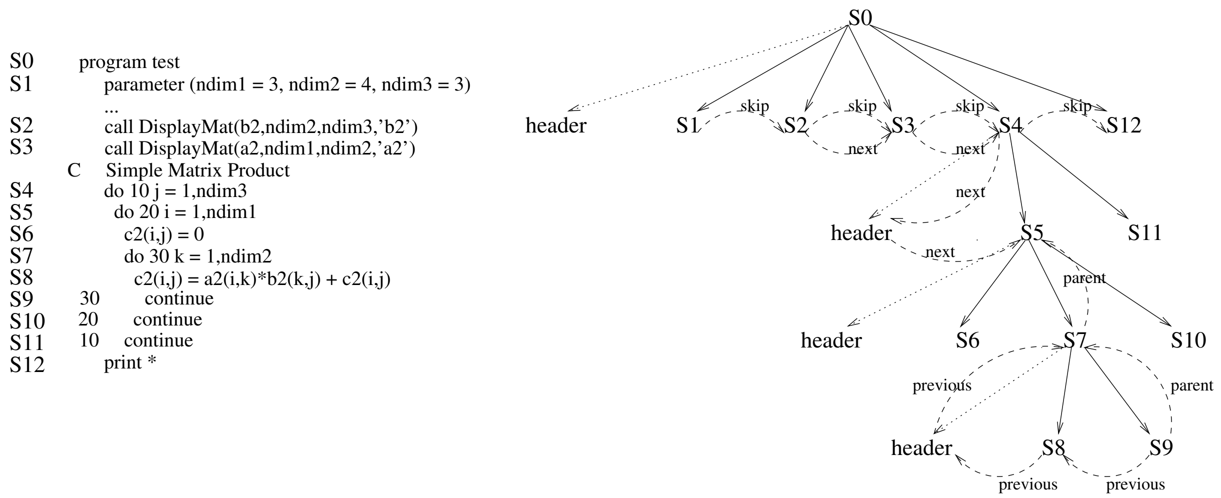

Figure 9: Browse operators on a fortran AST

▷ **Remark**: The attributes **FIRST**, **LAST**, **SKIP**, **NEXT** and **PREVIOUS** are also available as obsolete statements:

- `FIRST` is equivalent to `$cstat:=$cunit.FIRST`
- `LAST` is equivalent to `$cstat:=$cstat.LAST`

- `SKIP` is equivalent to `$cstat:=$cstat.SKIP`
- `NEXT` is equivalent to `$cstat:=$cstat.NEXT`
- `PREVIOUS` is equivalent to `$cstat:=$cstat.PREVIOUS`

7.5 Marking and finding nodes

The language provides some attributes to match nodes in equivalent ASTs.

▶ ***src*.PATH(*dest*)**
The expression returns a path from the node *src* to the node *dest*. If *dest* is not in the AST *src* then the expression returns **FALSE**. The type of the path is undocumented but it can be used by the attribute **APPLYPATH**.

▶ ***src*.APPLYPATH(*path*)**
The expression returns the node in the AST *src* at the position *path* created by **PATH**.

▷ **Example 17:** Insert a copy of the selected statement in which the selected expression replaced by the integer constant 999.

```
SCRIPT test ()
  // Create the path from the statement
  //  to the selected expression
  path :=  $cstat.PATH($csel)
  // Create a copy of the statement
  stat := COPY($cstat)
  // Find the corresponding expression
  expr := stat.APPLYPATH(path)
  // Insert the modified statement
  expr <- CREATEVALUED("int_cst",999)
  PASTE("BEFORE",stat,$cstat)
ENDSCRIPT
```

The language provides a mechanism to mark each node with an unique integer value (i.e. an *id*) .

▶ ***n*.ID**
The expression returns the integer *id* associated to the node *n*. If necessary, a new *id* is created for this node.

▶ ***u*.ALLID**
The expression initialize the *id* of each node in the fortran unit *u* (i.e. see the variable `$cunit`). The first *id* is 0 and the expression returns the number of created *ids*.

▶ ***u*. CLEANID**
The expression cleans all the *ids* in the fortran unit *u*.

▶ ***u*.FINDID(n)**
The expression returns the node of *id* *n* in the fortran unit *u*. in some previous versions of TSF, the obsolete **VARWITHID** was used instead of **FINDID**.

▶ **TREECMP(t1,t2)**

The expression compares the equality of the ASTs t1 and t2. The result is a boolean value. There is an important difference with the == operator. **TREECMP** compares the structure of the ASTs while the operator == compare the AST themselves. The table 3 compares the results given by both expressions.

Operands	==	TREECMP
PARSEEXPR("x") , PARSEEXPR("y")	FALSE	FALSE
PARSEEXPR("x") , PARSEEXPR("x")	FALSE	TRUE
var , var (the same AST)	TRUE	TRUE

Table 3: Comparison of == and TREECMP.

▷ **Example 18:** Print the variant of each node in the current fortran unit

```
SCRIPT printAllVariant ()
  nbid := $cunit.ALLID
  cpt := 0
  WHILE(cpt < nbid)
    node := $cunit.FINDID(cpt)
    PRINT node.VARIANT
    cpt := cpt + 1
  ENDWHILE
  foo := $cunit.CLEANID
ENDSCRIPT
```

▷ **Warning**: The **COPY** function does not preserve *ids*.

7.6 Use the pattern-matching facilities in scripts

In addition to use the pattern-matching mechanism for finding some specifics FORTRAN constructions (see chapter 11), some script statements have been add in order to ease the comparison of AST at the script level.

▶ **GETPAT(***astOrstring***)**

This function allows to create a pattern from a string or from a FORTRAN AST. In this preliminary version, if you need to create a pattern from a string, it is recommend to use the pattern editor (see chapter 5.2) in order to ease this step.

▶ **MATCH(***pattern, ast***)**

This function checks the pattern against an AST. It returns a boolean value according to the result of the comparison. In the case where the comparison is true, the matched variables are automatically add to the script's symbol table (status global, see chapter 6.1 for more details).

▶ **NEXTMATCH(**_pattern, ast_**)**

This function launch the research of the pattern from the specified position by the *ast* argument. In the case where the pattern-matching is successful, the new ast found is return. () otherwise. Warning, the patterns variables are not updated by this function.

▷ **Example 19:** Create a pattern from a string.

```
SCRIPT UsePattern ()
  pat := GETPAT ("(do ?v144 ?v145 (bounds ?index ?start ?end ?step)
                (?? ?body))")

  IF (MATCH (pat,$cstat)) THEN
      PRINT "the selection matches the pattern"
      PRINT ?index
      PRINT ?start
      PRINT ?end
      PRINT ?step
      PRINT ?body
   ELSE
        PRINT "the selection is different from the pattern"
   ENDIF
ENDSCRIPT
```

<table>
<tr><th>Fortran code selection</th><th colspan="2">Results</th></tr>
<tr><td><pre>
      do 100 i = 1,ndim1
        a(i) = var1
        b(i) = var2
        c(i) = var2
  100   continue
</pre></td><td><pre>
Display message
?index
?start
?end
?step
?body
</pre></td><td><pre>
the selection matches the pattern
i
1
ndim1

a(i) = var1
b(i) = var2
c(i) = var2
100 continue
</pre></td></tr>
</table>

▷ **Example 20:** Create a pattern from a FORTRAN AST.

```
SCRIPT UsePattern ()
   pattern := GETPAT(PARSESTAT("               c2(i,j) = 0"))
   PRINT pattern

   IF (MATCH (pat2,$csel)) THEN
        PRINT "the selection is equal to the pattern"
   ELSE
        PRINT "the selection is different from the pattern"
   ENDIF
ENDSCRIPT
```

Fortran code selection	Results	
`c2(i,j) = 0`	Display pattern Display message	(ass (vardim c2 (l_exp i j)) 0) the selection matches the pattern

▷ **Example 21:** Use the NEXTMATCH function to go over the current unit.

```
SCRIPT SearchAllMatch ()
  FIRST // set $cstat

   // pattern created from the pattern editor
   // we are looking for something like:
   //      ?array1(?v117,?index) ?op ?array2(?index, ?v121)
   // we are interested by finding some expressions which involved two arrays
   // with the same indice between their rows and their columns
  pat := GETPAT ("(?op (vardim ?array1 (l_exp ?v117 ?index))
                       (vardim ?array2 (l_exp ?index ?v121)))")

  newMatch := NEXTMATCH(pat,$cstat)
  WHILE (newMatch)
    PRINT ""
    PRINT "A new match has been found"
    PRINT newMatch
    PRINT newMatch.ENCLOSINGSTAT // Get the enclosing statement

    newMatch := NEXTMATCH(pat,newMatch)
  ENDWHILE

  PRINT "No more match"
 // Note: can also use the Motif Builtin Dialog functions
 // to display message
ENDSCRIPT
```

<table>
<tr><th>Fortran code</th><th colspan="2">Results</th></tr>
<tr><td><pre>
  ...

  do k = 1,ndim2
    c2(i,j) = a2(i,k)*b2(k,j)
            + c2(i,j)
  end do
  ...

  do 10 i = 1,ndim1,2
   b(i) = c2(var2,i) + b2(i,1)
   c(i) = var2
0   continue
  ...
</pre></td><td>First match

Second match

End of script</td><td>A new match has been found
a2(i,k)*b2(k,j)
c2(i,j) = a2(i,k)*b2(k,j) + c2(i,j)

A new match has been found
c2(var2,i)+b2(i,1)
b(i) = c2(var2,i) + b2(i,1)

No more match</td></tr>
</table>

7.7 Fortran declarations

Each fortran code contains several objects ; variables of course but also labels, functions and subroutines for example. A simple name is not sufficient to describe one of those objects since some homonyms can exist in several units. We use the undocumented type *fsym* to reference those objects. An *fsym* is composed of two parts; a name and an unit. The unit is taken in the current forlib. The name can be a string or an integer (i.e. for labels). An *fsym* can be used as an entry in the symbol table associated to the fortran unit. A simple string can be used instead of a fsym when the unit is $cunit.

- ***u*.FSYM(*n*)**
  The expression returns the fsym that associates the name n to the unit u. The attribute **SYMBOL** can be used instead of **FSYM** (this usage of **SYMBOL** is obsolete).

- ***u*.SYMTABSIZE**
  The expression returns the number of entries in the symbol table associated to the unit u.

- ***u*.SYMTABENTRY(*k*)**
  The expression returns the fsym that describes the entry of rank k in the symbol table associated to the unit u. The rank of the first entry is 0 and the last one is u.**SYMTABSIZE**-1. The example 22.

▷ **Example 22:** Displays the name of all the variables used in the current unit.

```
SCRIPT ShowSymbolTable()
  nb := $cunit.SYMTABSIZE
  cpt := 0
  WHILE(cpt < nb)
    fsym := $cunit.SYMTABENTRY(cpt)
    IF (fsym.TYPEOFSYMBOL == "variable") THEN
      PRINT fsym.LOWERNAME
    ENDIF
    cpt := cpt + 1
  ENDWHILE
ENDSCRIPT
```

- ***f*.LOWERNAME**
  The expression returns a string that contains the lowercase name associated to the fsym f (see example 22). The value **FALSE** is returned when f does not reference a valid declaration in the unit.

- ***f*.NATUREOFSYMBOL**
  The expression returns a string that contains the nature of the symbol f (see example 22). The value **FALSE** is returned when f does not reference a valid declaration. The most common natures are "variable", "parameter", "label", "function" and "subroutine".

▶ ***f*.ISPOINTER**
The expression returns the value **TRUE** if the fsym s references a pointer else returns **FALSE**.

▶ ***f*.FUNKIND**
The expression returns the nature of the function defined by the fsym f. The string "intrinsic" is returned for the intrinsic functions.

▶ ***f*.SCALARVALUE**
The expression returns the declared scalar value for the symbol table entry corresponding to the fsym f. The value **FALSE** if f does not reference a parameter or a scalar variable with a predefined value.

▶ ***s*.INSTATUS**
The expression returns a string that indicates if the symbol s that references a parameter of the unit, is read. The strings "y", "n" and "?" means respectively *yes*, *no* and *maybe*. The value **FALSE** is returned if s does not reference a parameter of the unit.

▶ ***s*.OUTSTATUS**
The expression returns a string that indicates if the symbol s that references a parameter of the unit, is written. The strings "y", "n" and "?" means respectively *yes*, *no* and *maybe*. The value **FALSE** is returned if s does not reference a parameter of the unit.

▶ **DECLARE(*decl*)**
The statement inserts the fortran declaration specified by the string *decl* into the current unit.

▷ **Example 23:** A set of declarations

```
DECLARE("include 'mpif.h'")
DECLARE("integer ")
DECLARE("include 'mpif.h'")
```

▶ **REBUILDSYMTAB(*unit*)**
The statement rebuild the internal symbol table associated to the specified *unit*. This symbol table contains all informations about the fortran declarations. It must be regenerated to validate all changes in the declarations.

▷ **Example 24:** increase the first dimension of the variable X.

```
d1 := "X".DIMENSION(1)
newd1 := PARSEEXPR("%e+1",d1)
d1 <- newd1
REBUILDSYMTAB($cunit)
PRINT "X".DIMENSION(1)
```

▶ ***var*.TYPE**
The expression returns a string that describes the elementary type of the fortran variable of

name *var*.

▷ **Example 25:** With the Fortran declaration `integer X(m,n)`, the expression `"X".TYPE` returns the string "integer".

▶ ***var*.TYPESIZE**
The expression returns the size in bytes of the type given by *var*.**TYPE**.

▷ **Example 26:** With the Fortran declaration `integer X(m,n)`, the expression `"X".TYPESIZE` returns 4 (i.e. if the size of default integer is 4).

▶ ***var*.DIMENSION**
The expression returns the number of declared dimensions for the variable of name *var*. The value 0 is returned for scalar variables.

▷ **Example 27:** With the Fortran declaration `integer X(m,n)`, the expression `"X".DIMENSION` returns 2.

▶ ***var*.DIMENSION(n)**
The expression returns the dimension of rank n of the variable of name *var*. The AST returned by this attribute is not a copy ; it is the original.

▷ **Example 28:** Exchange the 2 dimensions of the selected variable name.

```
SCRIPT exchangeDims ()
  var := $csel

  IF (var.VARIANT == "array_decl") THEN
    // Select the name of the declaration
    var := var.CHILD(1)
  ENDIF

  IF (var.VARIANT == "name") THEN
    // Get the variable name because we do not
    // to apply the transformation to an array
    // access such as A(i,j)
    varname := var.SYMBOL
    dim1 := COPY(varname.DIMENSION(1))
    dim2 := COPY(varname.DIMENSION(2))
    varname.DIMENSION(1) <- dim2
    varname.DIMENSION(2) <- dim1
  ENDIF
ENDSCRIPT
```

▶ ***var*.LOWERB**

The expression returns the lower bound of an array dimension (i.e. a column expression such as `m:n`). When the dimension is not a column expression then *var*.**LOWERB** returns the integer constant 1.

▷ **Example 29:** Some examples of results given by the attribute **LOWERB**. The target is underlined in each fortran expression:

integer A(x+1)	→	1 (the constant)
integer B(m:n)	→	m (a AST)
print *,C(1:n)	→	1 (a AST)
print *,n	→	1 (the constant)

▶ ***var*.UPPERB**

The expression returns the upper bound of an array dimension (i.e. a column expression such as `m:n`). When the dimension of an expression *var* is not a column expression then *var*.**UPPERB** returns *var*.

▷ **Example 30:** Creates the fortran expression that gives the number of elements in the first dimension of the variable `X`. It returns b-a+1 when the variable `X` is declared by `X(a:b)`

```
varname := "X"
lb :=  varname.DIMENSION(1).LOWERB
ub :=  varname.DIMENSION(1).UPPERB
expr := PARSEEXPR("%e-%e+1",ub,lb)
```

▷ **Warning**: In the left part of a AST assignment (see the previous example), the **DIMENSION** attribute has an effect on the symbol table. A common error is to insert a temporary variable.

▷ **Example 31:** Both forms are not equivalent. The second does not update the symbol table and so is incorrect.

```
varname.DIMENSION(1) <- dim
```

```
tmp := varname.DIMENSION(1)
tmp <- dim
```

▷ **Remark**: The attribute **DIMENSION** can be applied to an array acces (i.e. in a statement such as `A=B(i,j)`) to get or to change the indexes `i` or `j`. Then, it has no effect on the symbol table.

7.8 Type casts

The following function are available to cast a TSF expression into another type :

▶ **TOVALUE(n)**
The function casts the content of n into an integer, a real or a string:

- if n is an integer, a real or a string then returns n.
- if n is an AST of variant "int_cst" (i.e. an integer constant) then returns the integer value of n.
- if n is an AST of variant "real_cst" (i.e. a simple precision constant) then returns a string that contains the value of n. (See **TOREAL** to obtain a real value)
- if n is an AST of variant "double_cst" (i.e. a double precision constant) then returns a string that contains the value of n. (See **TOREAL** to obtain a real value)
- if n is an AST of variant "fstring_cst" then returns a string that contains the value of n.
- if n is an AST of variant "name" then returns a string that contains the Fortran symbol associated to n.
- else returns **FALSE**

▶ **TOINT(n)**
The function casts the content of n into an integer value. (see TOVALUE for more details about AST cases)

▶ **TOREAL(n)**
The function casts the content of n into an real value. (see TOVALUE for more details about AST cases)

▶ **TOSTRING(n)**
The function casts the content of n into a string. (see **TOVALUE** for more details about AST cases)

▶ **TOTREE(n)**
The function casts the content of n into a AST.

- if n is an integer (resp. a real or a string) then return a node of variant "int_cst" (resp. "real_cst" or "fstring_cst").
- if n is an AST then return n.
- else return **FALSE**.

7.9 Creating Fortran statements and expressions

Most of Fortran codes are described by ASTs. Some functions are provided to help to the creation of such ASTs :

▶ **COPY**
The function creates a copy of its first argument (a AST).

▶ **CREATEVALUED(v,n)**
The function is used to create a node without child but with a value. Nodes with the following variants v can be created:

- *int_cst* - an integer constant. The function TOINT is implicitly applied to n.
- *real_cst* - a simple precision real constant. The function TOREAL is implicitly applied to n.
- *double_cst* - a double precision real constant. The TSF language does dot supports the double precision type but the parameter n can be a string (i.e. for example "314.15D87").
- *fstring_cst* - a constant string. The function TOSTRING is implicitly applied to n.
- *name* - a variable name. The parameter n must be a string.

▶ **CREATE(*v*,*c1*,...,*cn*)**
The function creates a node with a specified variant. Its first argument v is the node variant. The remaining arguments *c1*,...,*cn* are the children of the node. The number of children depends on the variant. Those parameters are implicitly transformed into ASTs by the function **TOTREE**.

▷ **Example 32:** Creation of an assignment a:=2

```
varname := "a"
stat := CREATE("ass",CREATEVALUED("name",varname),2)
```

Those functions are sufficient to create all ASTs. However, some convenience functions are provided to create ASTs from fortran strings.

▶ **CREATEEXPR(*expr*)**
The obsolete function call creates a AST from the fortran expression in the string *expr*. This function is obsolete and is emulated by the new function **PARSE**.

▷ **Example 33:** Create a simple expression

```
expr := CREATEEXPR("4*n+3")
```

The **CREATEEXPR** function does not use the symbol table associated to the current fortran unit. Especially, some wrong ASTs can be generated for functions call. For instance, the expression "send(i,j)" will be interpreted as a matrix access even if "send" is known as a function. When the string is constant (i.e. not a variable), it is possible to reference some TSF variables:

- Each occurrence of `?var` is the pattern variable `?var`.
- Each occurrence of `#var` is the TSF variable `var`.

▷ **Example 34:** Creation of the expression `(n+1)*(n-3)`

```
lhs := CREATEEXPR("n+1")
rhs := CREATEEXPR("n-3")
expr := CREATEEXPR("#lhs * #rhs")
```

▷ **Example 35:** A wrong version without a constant string (see PARSEXPR)

```
lhs := CREATEEXPR("n+1")
rhs := CREATEEXPR("n-3")
str := "#lhs * #rhs"
expr := CREATEEXPR(str)
```

▶ **CREATESTAT(*label*,*stat*)**
The obsolete function call and **CREATESTAT**(*stat*) creates a AST from the fortran statement in the string *stat*. **CREATESTAT** works like **CREATEEXPR** but it is not possible to create more than one statement or a complex statement such as an IF or a DO.

▶ **PARSE(*type*,*format*,*p1*,*...pn*)**
The new function call is an improved version of **CREATEEXPR** and **CREATESTAT**. Its first argument *type* must be the string "expr" to create an expression or "stat" to create a statement. The argument *format* is a string with a fortran code. The arguments *p1*,...,*pn* are inserted where the following tokens are found (in lexical order).

- %e or %E - The corresponding argument describes an expression. It can be a AST, a string interpreted a variable name or a numerical value.
- %s or %S - The corresponding argument must be a AST that describes a statement.
- %l or % - The corresponding argument must be an integer that describes a label.
- %% - The % character is inserted. No argument is used.

When an argument is not a root AST (i.e. the attribute PARENT returns an AST) then a copy is performed before the insertion. In doubt, it is recommended to create explicitaly a copy of the argument with the COPY function. The function **PARSE** can create structured statements such as DO loops but can not create sequences of statements. In fact, it is always possible to create a structured statement and to extract its body. For statements, the *format* respects the fixed Fortran format.

▷ **Example 36:** Creation of a DO statement

```
start:=PARSE("expr","1")
end:=PARSE("expr","n")
stat:=PARSE("stat",
"C This is a comment
               DO i:=%e,%e
                 PRINT *,i
               ENDDO",
   start,end)
```

▷ **Example 37:** Creation of a sequence of statement. an IF statement is used to encapsulate the sequence.

```
// Create the IF statement
ifstat:=PARSE("stat",
"          IF (.TRUE.) THEN
             PRINT *,i
             PRINT *,j
           ENDIF")
// Get and detach the sequence
stat := ifstat.THENCLAUSE
stat <- EMPTY
```

▷ **Question**: What is the statement produced by this command ?

```
stat:=PARSE("stat","call execute(10,20)")\\
```

▷ **Answer**: Nothing. This string will be interpreted as a fortran comment. The correct command is

```
stat:=PARSE("stat","          call execute(10,20)")\\
```

▷ **Remark**: The function call **PARSEEXPR**(*format*,*p1*,...,*pn*) is stricly equivalent to **PARSE**("expr",*format*,*p1*,...,*pn*).

▷ **Remark**: The function call **PARSESTAT**(*format*,*p1*,...,*pn*) is stricly equivalent to **PARSE**("stat",*format*,*p1*,...,*pn*).

7.10 Modifying the fortran code

This set of functions allows the user to modify the fortran program. We emphasize the *cut* & *paste* concept in order to facilitate the user's operation:

▶ **CUT(*t*)**
The statement cuts the AST *t* and put it in the default clipboard.

▶ **PASTE(*pos*,*src*,*dest*)**
The statement inserts the AST *src* in the AST *dest*:

- if *pos* is an integer then *src* is inserted as the child of rank *pos* of *dest*. The rank 0 is used to specify the last child so **PASTE**(0,*src*,*dest*) is equivalent to **PASTE**(*dest*).**NBCHILD**+1,*src*,*dest*)
- if *pos* is the string `"BEFORE"` or the obsolete constant **BEFORE** then *src* is inserted as the child before *dest*.
- if *pos* is the string `"AFTER"` or the obsolete constant **AFTER** then *src* is inserted as the child after *dest*.

▷ **Example 38:** The following program applies the transformation described in the example 39

```
SCRIPT test()
  if_stat := $cstat
  IF (if_stat.VARIANT!="struct_if") THEN
    ERROR(1,"Please, select a IF-THEN statement")
  ENDIF
  PASTE("BEFORE",PARSESTAT("        print *,'before'"),if_stat)
  PASTE(1,PARSESTAT("        print *,'start'"),if_stat.THENCLAUSE)
  PASTE(0,PARSESTAT("        print *,'end'"),if_stat.THENCLAUSE)
  PASTE("AFTER",PARSESTAT("        print *,'after'"),if_stat)
ENDSCRIPT
```

▷ **Example 39:** a simple Fortran transformation

```
IF (X.EQ.10) THEN
  Y=X*2
  CALL FOO(X,4)
ENDIF
```

$\longrightarrow$

```
PRINT *,'before'
IF (X.EQ.10) THEN
  PRINT *,'start'
  Y=X*2
  CALL FOO(X,4)
  PRINT *,'end'
ENDIF
PRINT *,'after'
```

- **SWAP(*t1*,*t2*)**
  The statement exchanges the ASTs *t1* and *t2*.
- **REMOVESYMB(*name*)**
  The statement removes the symbol of the current unit.
- **REPLACEIN(*dest*,*new*,*old*)**
  The statement replaces each occurence of the AST *old* in the AST *dest* by a copy of the AST *new*. The example 40 shows how to rename the index of a DO statement.

  ▷ **Example 40:** Rename the index of a the DO statement given in argument.

```
SCRIPT main (dostat)
  // Get the old index
  oldindex := COPY(dostat.INDEX)
  // The new index is "ind"
  newindex := PARSEEXPR("ind")
```

```
    REPLACEIN(dostat,newindex,oldindex)
 ENDSCRIPT
```

7.11 Comments

Unlike fortran statements and expressions, there exist no specific node variant to describe comments. A comment is attached to a statement; not as a child but as an attribute. Let us consider the comment in following fortran statement:

```
    print *,1
C I am fortran comment
    i=1
    print *,2
```

The **AST** attribute applied to the assignment generates the following AST structure:

ass(name i, int_cst 1){^prefix comment_s[comment I am fortran comment]}

So, comments need some specific functionalities in TSF.

- ▶ ***s*.NBCOMMENT**
  The expression returns the number of comments associated to the AST statement *s*.
- ▶ ***s*.COMMENT**
  The expression returns the string concatenation of all comments associated to the AST statement *s*.
- ▶ ***s*.COMMENT(*n*)**
  The expression returns the string of the comment of rank *n* associated to the AST statement *s*. The rank of the first comment is 1. The strings "first" and "last" can be used to access to the first and the last comment.
- ▶ ***s*.PUREINST**
  The expression returns a copy of the AST statement *s* buy without the comments associated to its root node. This attribute does not remove comments in the body of an iterative or conditional statement (see **EXPURGATECOMMENT** for an example)
- ▶ ***s*.EXPURGATECOMMENT**
  The expression returns a copy of the AST *s* without any comment in any subpart of *s*.

▷ **Example 41:** The following script removes the comments associated to a IF statement and to the THEN part of the IF statement. The comments in the ELSE clause are preserved.

```
SCRIPT RemoveIfComment(stat)
  IF (stat.VARIANT=="struct_if") THEN
```

```
    // Remove the comment of the IF statement
    newstat := stat.PUREINST
    //Remove the comments i n the THEN part
    newstat.THENCLAUSE <- newstat.THENCLAUSE.EXPURGATECOMMENT
    stat <- newstat
  ENDIF
ENDSCRIPT
```

▶ **INSERTCOMMENT(*s*,*c*,*op*,*p*)**
The statement adds the comment specified by the string c to the statement s at the position p with the operation op. The position p can be an integer index (i.e. the first index is 1) or one of the string "first" or "last". The operation op must be one of the following keywords (i.e. string):

- **"insert"** - The comment is inserted at the specified position.
- **"replace"** - The comment at the specified position is replaced by the new comment. The position "all" can be used to replace all the comments by the new one.
- **"delete"** - The comment at the specified position is deleted. The position "all" can be used to delete all the comments. The argument c is not used here.

▷ **Example 42:** The following script replaces the comments of the selected statement by the result of the **AST** attribute applied to the statement. A comment "$$ AST" is added before this comment. If it is found, then the new comment does not replace the last comment but it is inserted after.

```
SCRIPT InsertASTComment ()
  stat := $cstat
  IF (stat) THEN
    IF (stat.COMMENT(1)=="$$ AST") THEN
      INSERTCOMMENT(stat,"  "+stat.AST,"insert","last")
    ELSE
      INSERTCOMMENT(stat,"  "+stat.AST,"replace","all")
      INSERTCOMMENT(stat,"$$ AST","insert",1)
    ENDIF
  ENDIF
ENDSCRIPT
```

▶ **INSERTCOMMENT(*s*,*c*)**
The statement adds the comment specified by the string c to the statement s. This is a shortcut to **INSERTCOMMENT**(s,c,"insert","last").

8 Various attributes and functions

8.1 Attributes for labels

The labels are coded as integer constants (i.e nodes of variant "int_cst"). For example, the **AST** attribute applied to the fortran statement `10 print *,i` gives the following string

labstat(int_cst 10, print(star(), l_ioexp[name i]))

We can see that labeled statements are coded by an node of variant "labstat".

▶ ***stat*.STATEMENT**
The expression returns the statement associated to the node *stat* of variant "labstat". (see section 8.2 for more informations).

▶ ***stat*.LABEL**
The expression returns the label associated to a statement *stat*. *stat* can be a "labstat"

8.2 Attributes for conditional statements

Conditionnal statements are coded by several variants:

- The variant "struct_if" refers to a IF-THEN-ELSE statement.

```
IF (X.EQ.Y) THEN
  PRINT *,X
ENDIF
```

- The variant "struct_elseif" refers to a ELSEIF statement in a IF-THEN-ELSE statement.

```
IF (X.EQ.Y) THEN
  PRINT *,X
ELSEIF (X.EQ.Z) THEN
  PRINT *,Z
ENDIF
```

- The variant "log_if" refers to a IF statement applied to only one simple statement.

```
IF (X.EQ.Y) PRINT *,X
```

- The variant "arith_if" refers to an labeled IF statement.

```
IF (X.EQ.Y) 10,20,30
```

- The variant "do" refers to an iterative statement. It can be considered as a conditionnal statement when the variant of its bounds part is "while" (see **BOUND**).

```
DO WHILE (X.EQ.Y)
  PRINT *,X
ENDDO
```

The following attributes gives a quick access to the most useful parts of conditionnal statements.

▶ ***stat*.CONDITION**
The expression returns the condition associated to a conditionnal statement.

▶ ***stat*.THENCLAUSE**
The expression returns the THEN branch of the statement *stat* of variant "struct_if" or "struct_elseif".

▶ ***stat*.ELSECLAUSE**
The expression returns the ELSE branch of the statement *stat* of variant "struct_if" or "struct_elseif".

▶ ***stat*.STATEMENT**
The expression returns the statement associated to the statement *stat* of variant "log_if". (see section 8.1 for more informations).

▷ **Example 43:** The following script inverts the condition and exchanges the branches of an IF-THEN-ELSE statement

```
SCRIPT InvertIf()
  stat := $cstat
  IF (stat.VARIANT=="struct_if") THEN
    stat.CONDITION <- PARSEEXPR(".NOT.(%e)",stat.CONDITION)
    thenpart := stat.THENCLAUSE
    stat.THENCLAUSE <- stat.ELSECLAUSE
    stat.ELSECLAUSE <- thenpart
  ENDIF
ENDSCRIPT
```

8.3 Attributes for DO statements

The following attributes give a quick access to the most useful parts of DO statements. They must be applied to a AST of variant "do".

▶ ***stat*.BOUND**
The expression returns the whole control structure of the DO statement *stat*. Its variant is "bounds" for an indexed DO statement (i.e. something like 'DO I=1,n ...'), "while" for a DO WHILE statement and "none" for an infinite DO.

▷ **Example 44:** How to differentiate DO statements

```
IF (stat.VARIANT=="do") THEN
  IF (stat.BOUND.VARIANT=="bounds") THEN
    PRINT "I am an indexed DO loop"
  ELSEIF (stat.BOUND.VARIANT=="while") THEN
    PRINT "I am a DO WHILE loop"
  ELSEIF (stat.BOUND.VARIANT=="none") THEN
    PRINT "I am an infinite DO loop"
  ELSE
    PRINT "I don't know who I am."
  ENDIF
ENDIF
```

▷ **Remark**: *stat*.**BOUND** is equivalent to *stat*.CHILD(3)

▶ ***stat*.INDEX**
The expression returns the variable index of the indexed DO statement *stat*. This is a shortcut to *stat*.CHILD(3).CHILD(1)

▶ ***stat*.START**
The expression returns the start value of the indexed DO statement *stat*. This is a shortcut to *stat*.CHILD(3).CHILD(2)

▶ ***stat*.END**
The expression returns the start value of the indexed DO statement *stat*. This is a shortcut to *stat*.CHILD(3).CHILD(3)

▶ ***stat*.STEP**
The expression returns the start value of the indexed DO statement *stat*. The step variant is "none" when the step is implicit. This is a shortcut to *stat*.CHILD(3).CHILD(4)

▶ ***stat*.BODY**
The expression returns the body of the DO statement *stat*. This is a shortcut to *stat*.CHILD(4)

▶ ***stat*.LABEL**
The expression returns the label associated to the DO statement *stat* (i.e. the label before the index variable). If there is no label (the statement is ended by ENDDO) then the attribute returns a node of variant "none".

▷ **Example 45:** A simple inversion of the step sign

```
SCRIPT SignInvert ()
  loop := $cstat
  IF (loop.VARIANT == "do") THEN
    IF (loop.BOUND.VARIANT == "bounds") THEN
       // Exchange the start and the step
       start := loop.START
       loop.START <-  loop.END
       loop.END <-  start
       // Invert the condition
       step := loop.STEP
       IF (step.VARIANT=="none" )THEN
          step := PARSEEXPR("-1")
       ELSE
          step := PARSEEXPR("-(%e)",step)
       ENDIF
       loop.STEP <- step
    ENDIF
  ENDIF
ENDSCRIPT
```

8.4 Attributes for unit

the manipulation of the units needs some specifics attributes. We develop in this section some of the most frequently used.

- ▶ ***unitVar*.FILENAME**
  This attribute returns the name of the file (path included) where the unit is defined.

- ▶ ***unitVar*.FILE**
  This attribute returns the file data structure where the unit is defined. In this case, this variable nay be used in order to update in the file the unit while preserving other data

▷ **Example 46:** PRINT $cunit.SAVE(”/tmp/newFile”) // will store the unit in a new File
fileVar := $cunit.FILE PRINT fileVar.SAVE($cunit.FILENAME) // will store the unit in the same file // that where it is defined while keeping the other units which are also // defined in it.

8.5 Objects names and constant strings

Many Fortran constructs such as variables, functions, subroutines have a name. It is coded by a node of variant ”name”.

- ▶ ***t*.IDENT**
  The expression returns the node of variant ”name” associated to the AST *t*. If this node does not exist, then the expression returns FALSE.

- ▶ ***t*.SYMBOL**
  The expression returns a string that contains the symbol associated to the result of *t*.**IDENT**. This is a shortcut to *t*.**IDENT**.**TOSTRING**.

▶ ***t*.SNAME**
The expression returns the result of *t*.**IDENT** as a node of variant ”fstring_cst” i.e. a fortran constant string). This is a shortcut to

CREATEVALUED(”fstring_cst”,*t*.**IDENT**.**TOSTRING**)

▷ **Example 47:** Insert a PRINT statement before a scalar assignment.

```
SCRIPT addprint ()
  stat := $cstat
  // Test an assignment
  ok := FALSE
  IF (stat.VARIANT == "ass") THEN
    dest:=stat.LHS
    // Test if there is a "name" node
    // associated to the destination
    // expression (allways ?) .
    IF (dest.IDENT) THEN
      name_as_symbol := dest.SYMBOL
      name_as_fstring := dest.SNAME
      name_as_var := COPY(dest.IDENT)
      // Check that the variable is declared
      // as a scalar
      IF (name_as_symbol.DIMENSION==0) THEN
        cmd1 := "         print *,'The variable',%e,'is assigned'"
        newstat1 := PARSESTAT(cmd1,name_as_fstring)
        cmd2 := "         print *,'It old value was ',%e"
        newstat2 := PARSESTAT(cmd2,name_as_var)
        PASTE("BEFORE",newstat1,stat)
        PASTE("BEFORE",newstat2,stat)
        ok:=TRUE
      ENDIF
    ENDIF
  ENDIF
  IF (! ok) THEN
    ERROR(1,"Please, select an assignment 'scalar:=expression'")
  ENDIF
ENDSCRIPT
```

8.6 Attributes for expressions

▶ ***expr*.LHS**
The expression (i.e. Left Hand Side) returns the first operand of a Fortran binary operator or of an assignment. In most cases this is a shortcut of *expr*.CHILD(1).

▶ ***expr*.RHS**
The expression (i.e. Left Hand Side) returns the second operand of a Fortran binary operator or of an assignment. In most cases this is a shortcut of *expr*.CHILD(2).

▷ **Example 48:** Transforms an expression `X1-X2` to `-(X2-X1)`

```
SCRIPT trans()
  expr := $csel
  IF (expr.VARIANT=="sub") THEN
    rhs := expr.RHS
    lhs := expr.LHS
    expr <- PARSEEXPR("-(%e-%e)",rhs,lhs)
  ENDIF
ENDSCRIPT
```

8.7 Various attributes

▶ ***n*.LINENUMBER**

The expression returns the line number of the node n in the Fortran code.

- n (input AST) is a statement.
- The attribute returns an line number (INT)

▶ ***t*.SAVE(*f*)**

This expression saves the AST t in the file of name *f*.

- t (input AST) is the AST to be saved.
- f (input STRING) is the filename.
- if a problem occurs then the attribute returns FALSE else TRUE.

8.8 Attributes for data dependencies

A set of attributes have also been implemented in order to provide an access to data dependencies. These one rely on the usual notation to access to the various kind of dependencies which are the following:

- direct dependencies (also named read after write);
- anti dependencies (write after read);
- output dependencies (write after write);
- control dependencies, point out the statements whose the execution depends on the evaluation of an expression;
- value dependencies, point out the reuse of the variables.

In TSF, access to data dependencies requires two steps:

- first you need to create a *DepObjectList* object. This object which belong to the set of objects whose type is not documented, gather all the dependencies computed from a statement.
- second, you access to a particular dependency (that we will call a *DepObject*) by going over the *DepObjectList* object and by displaying useful information.

The following attributes are provided:

▶ ***stat*.DEP()**

This attribute returns the dependencies list computed from the current statement. This list is returned under the form of an undocumented type that we call a *DepObjectList*. A *DepObjectList* is a list of objects of the type *DepObject* which designate a data dependency.

By default, the direct, anti and output dependencies are taken into account if no argument is provided. Otherwise, the following possibilities are also provided:

▷ **Example 49:**

```
depList        := currentStat.DEP() // by default get all the direct, anti
and output dependencies
directDepList  := currentStat.DEP(DIRECT)
antiDepList    := currentStat.DEP(ANTI)
outputDepList  := currentStat.DEP(OUT)
valueDepList   := currentStat.DEP(VALUE)
controlDEpList :=  currentStat.DEP(CONTROL)
directAndAntiDepList   := currentStat.DEP(DIRECT,ANTI)
directAndOutputDepList := currentStat.DEP(DIRECT,OUT)
```

as well as all the other combinations.

The predefined keywords *DIRECT, ANTI, OUT, VALUE, CONTROL* may be replaced directly by their corresponding values (*"direct", "anti", "out", "value", "control"*).

▶ ***depObjectList*.NBDEP()**

This attribute returns the number of dependencies contained in the current *depObjectList* object.

▶ ***depObjectList*.NBDEP(n)**

Return the n^{th} *depObject* contained in the *depObjectList* object.

▶ ***depObject*.SOURCE**

Return the *source* statement of the considered dependency.

▶ ***depObject*.SOURCEEXPR**

Return the *source* expression involved in the considered dependency.

▶ ***depObject*.DESTINATION**

Return the *destination* statement of the considered dependency.

▶ ***depObject*.DESTINATIONEXPR**

Return the *destination* expression involved in the considered dependency.

▶ ***depObject*.KIND**

Return the *kind* of the considered dependency. Kinds returned are: direct, anti, out, value, control and are of type string.

▶ *depObject*.**DISTANCEVECTOR**
Return the whole distance vector of the considered dependency. This distance vector is given from the most internal loop to the most external loop.

▶ *depObject*.**DISTVECTORSIZE()**
Return the size of the current distance vector.

▶ *depObject*.**DISTVECTORSIZE(n)**
Return the n^{th} component of the current distance vector.

▷ **Example 50:** Example on the manipulation of the data dependencies.

Script

```
SCRIPT DisplayOutputDependence ()
 outputDep := $cstat.DEP(OUT)               // Get just the output data dependencies
 nbOutputDep := outputDep.NBDEP()           // Number of output data dependencies
 it := 1                                    // Iterator
 WHILE (it <= nbOutputDep)                  // Go over all the output data dependencies
    PRINT "Kind of the dataDep:  " + outputDep.NBDEP(it).KIND // (P1) Kind of the data dependency
    PRINT "SOURCE"
    PRINT outputDep.NBDEP(it).SOURCEEXPR    // (P2) Source expression of the current data dependency
    PRINT outputDep.NBDEP(it).SOURCE        // (P3) Source statement of the current data dependency
    PRINT "DESTINATION"
    PRINT outputDep.NBDEP(it).DESTINATIONEXPR  // (P4) Destination expression of the current data dep.
    PRINT outputDep.NBDEP(it).DESTINATION   // (P5) Destination statement of the current data dep.
    PRINT outputDep.NBDEP(it).DISTANCEVECTOR// (P6) Distance vector
                                            // (from the most internal loop to the most external loop)
       // Now deal with each component of the distance vector
    vectorSize := outputDep.NBDEP(it).DISTVECTORSIZE() // Get the size of the distance vector
    vectorIt := 1                           // Iterator
    WHILE (vectorIt <= vectorSize)          // Go over all the components of the distance vector
       value := outputDep.NBDEP(it).DISTVECTORSIZE(vectorIt)      // Get the component
       PRINT "component " + TOSTRING(vectorIt) + ":  " + TOSTRING(value) // (P7) Print its value
       // PRINT outputDep.NBDEP(it).DISTVECTORSIZE(vectorIt)
       // // value of the component (ex.  >=)
       vectorIt := vectorIt +1              // Go to the next component of the distance vector
    ENDWHILE
    it := it + 1                            // Go to the next data dependency
    ENDWHILE
ENDSCRIPT
```

Fortran User Selection	Results	

`do 430 i = 1,nsize1`
`  do 440 j = 1,nsize2`
`    s = s + a2(j,i)`
`440 continue`
`430 continue`
	P1 : Kind of the dataDep: out
	SOURCE
	P2 : s
	P3 : s = s + a2(j,i)
	DESTINATION
	P4 : s
	P5 : s = s + a2(j,i)
	P6 : (1 0)
	P7 : component 1: 1
	P7 : component 2: 0
		P1 : Kind of the dataDep: out
		SOURCE
		P2 : s
		P3 : s = s + a2(j,i)
		DESTINATION
		P4 : s
		P5 : s = s + a2(j,i)
		P6 : (0 >)
		P7 : component 1: 0
		P7 : component 2: >

8.9 Controlling the target

By default the script is executed with some initialized variables. The variable *$cunit* defines the current unit corresponding to the connected window and the variable *$csel* specifies the current selection in this window. So, the script can be applied to the selection or to the current selection.

Some features are provided to reach the other units. These ones allows to quickly generalized a script wrote for one unit to the whole forlib[3]. However, they are under development and are thus likely to change.

▶ ***forlibVar*.INITALLUNITS(*u*)**
The expression initializes the list of units in the forlib *forlibVar* (i.e. given by *$forlib*). The argument u must be the current unit (i.e. given by *$cunit*).

▶ ***forlibVar*.NBUNIT**
The expression returns the number of units in the forlib *forlibVar* (i.e. given by *$forlib*).

▶ ***forlibVar*.FINDUNIT(*r*)**
The expression returns one of the units of the forlib *forlibVar*. When r is an integer between 0 and varforlibVar.**NBUNIT**-1, the unit of this rank is returned. A string can also be specified for r to find the unit according to its name. The result is FALSE when search fails.

▶ **SETENVTO(*unit*)**
This statement allows to take into account the specified unit like the default one. Using this statement in conjunction with the previous attributs allows to quickly generalize a script wrote for one unit to the whole forlib. In this mode, the interactive context variable (*$cstat*, *$csel*) are without effects.

▶ **RESTOREENV**
This statement is the dual of the previous one. It allows the user to restore the current TSF environment according to the unit from which TSF has been called.

▶ ***forlibVar*.UPDATE(*logFile*)**
This attributes allows to regenerate the main analysis did at the time of the forlib creation. It allows to take into account the modification did by the user. The *logFile* parameter targets a file which contains the results of the analysis done.

▷ **Example 51:** Declare a new variable and add a new statement after the first executable statement of each unit of the forlib.

```
SCRIPT ApplyToUnit(unit)
   PRINT unit.SNAME
   DECLARE ("integer modifDone")
   newStat := PARSESTAT("       modifDone = 3")
   PRINT newStat
    stat := unit.FIRST
    PASTE(AFTER, newStat, stat)
ENDSCRIPT


SCRIPT main ()
 foo := $cforlib.INITALLUNITS($cunit)
 PRINT $cforlib.NBUNIT
```

[3]Remind, the forlib targets the data structure which contains all the units (subroutines, functions, block data, etc.) defined in the Fortran application.

```
PRINT ""

nbunit := $cforlib.NBUNIT
cpt:=0
WHILE(cpt<nbunit)
  unit := $cforlib.FINDUNIT(cpt)
  IF (unit) THEN
       SETENVTO(unit)
       unit_name:=unit.SYMBOL
       unit_fileName := unit.FILENAME
       unit_file := unit.FILE  // get the file where the unit is defined
       CALLSCRIPT ApplyToUnit(unit)
       keepTrace := unit_fileName
       res := "Unit " + unit_name + " is saved in " + unit_file.SAVE(keepTrace)
       PRINT res
       RESTOREENV
ENDIF
  cpt:=cpt+1 // go to the next unit
ENDWHILE
 log := "/tmp/"+ $cforlib.SYMBOL + ".msg"
 PRINT $cforlib.UPDATE(log)

ENDSCRIPT
```

8.10 Using vectors and hash tables

TSF provides two types for the manipulation of set of values ; the vectors and the hash tables. A vector is an sequence of values referenced by an integer index. The lower index is 1.

▶ ***V*[*i*]**
The expression references the element of index i in the vector V.

- V (input VECTOR).
- i (input INT).

▶ **VECTOR(*v1*,*v2*,...,*vn*)**
The expression returns a vector of length n initialized with the values *v1*,*v2*,...,*vn* of any types.

- *v1*,...,*vn* (input ANY TYPE)

▶ **NEWVECTOR(*n*,*v*)**
The expression returns a vector of length n. All elements are initialized with the same value v.

- n (input INT)
- v (input ANY TYPE)

The following expressions can be used to manipulate hash tables.

▶ $\boldsymbol{H\{k\}}$
The expression returns the element of key k in the hash table H. The value FALSE is returned when the key is not found in the hash table.

- H (input HASH).
- k (input ALL TYPE).

▶ **NEWHASH()**
The expression returns an empty hash table.

▶ **CLEARHASH($\boldsymbol{H}$)**
The statement remove all elements in the hash table H.

- H (input HASH)

▶ **HASHKEYS($\boldsymbol{H}$)**
The expression returns a vector initialized with the keys in the hash table H.

- H (input HASH)

▶ **HASHVALUES($\boldsymbol{H}$)**
The expression returns a vector initialized with the values in the hash table H. The values are given in the same order of the keys returned by the function **HASHKEYS**.

- H (input HASH)

The following are commons to vectors and hash tables.

▶ **LENGTH($\boldsymbol{M}$)**
The expression returns the number of elements in the vector or the hash table M.

- M (input VECTOR or HASH)

▷ **Example 52:** Print the content of a hash table.

```
SCRIPT DumpHashTable(H)
  // Version with the keys
  keys := HASHKEYS(H)
  FOR i := 1,LENGTH(keys) DO
    k := keys[i]
    PRINT TOSTRING(k)+" => "+TOSTRING(H{k})
  ENDFOR
ENDSCRIPT
SCRIPT ShortDumpHashTable(H)
  // Version without the keys
  val := HASHVALUES(H)
  FOR i := 1,LENGTH(val) DO
    PRINT val[i]
  ENDFOR
ENDSCRIPT
```

▷ **Example 53:** A simple use of vectors and hash tables. Associate a string (in english) to its numerical value.

```
SCRIPT main()
  // Create a vector of number (strings)
  v1 := VECTOR("one","two","three","four")
  // Create the equivalent vector of number (integers)
  v2 := NEWVECTOR(LENGTH(v1),FALSE)
  FOR i:=1,LENGTH(v2) DO
   v2[i] := i
  ENDFOR
  // Associate v1 and v2 in a hash table
  h := NEWHASH
  FOR i:=1,LENGTH(v1) DO
    h{v1[i]} := v2[i]
  ENDFOR
  // Print the value associated to the string "two"
  PRINT "The value of 'two' is "
  PRINT h{"two"}
ENDSCRIPT
```

In the current version of TSF, there are some syntaxical restrictions in the usage of vectors and hash tables. A element can not be used as a reference in a function call or in a script call. For example, it is not possible to write the following statement:

`CALLSCRIPT my_script(&V[3])`

When accessing an element (with [] or {}), the left part must be a variable name. So, `V[3]` is allowed but not `V[3][4]` (i.e. the left part V[3] is not a variable name).

8.11 Various functions

The following functions and statements do not fit in any category.

▶ **SYSTEM(*c*)**

This statement executes the UNIX command given by its single argument *c*. See example 54.

▶ **GETENV(*s*)**

This function returns the environment variable of name *s*. The result and the argument are strings but the value FALSE is returned when the environment variable does not exist.

▷ **Example 54:** Start a compilation process in the directory specified by the environment variable WORK_DIR.

```
IF (GETENV("TSFWORK")) THEN
  SYSTEM("cd "+GETENV("WORK_DIR")+" ; make")
ENDIF
```

8.12 Builtin functions

The TSF language provides some functions to access directly to the lisp language provided by Foresys.

- ▶ ***v*.BUILTIN(*n*,*p1*,...,*pn*)**
  The expression call the lisp function of name *n* with the arguments *v*,*p1*,...,*pn*. If the first argument *v* is a variable, then its is passed by reference.

- ▶ ***v*.BUILTIN_(*n*,*p1*,...,*pn*)**
  The **BUILTIN_** attribute is a variant of **BUILTIN** in which the left hand side is not passed as an argument to the lisp function.

▷ **Example 55:** The function Dialog:Util:AskUser (see section 8.15.1) can be implemented by the builtin #:xdialog:exec-ask.

```
foo := foo.BUILTIN("\#:xdialog:exec-ask",
                   "The window title",
                   "I am the message",
                   "Ok|Cancel",
                   3)
```

▷ **Example 56:** Builtins are usefull when the TSF not provided a functionnality such as displaying the content of the undocumented value generated by the **PATH** attribute. The Lisp function `print` is used.

```
foo := foo.BUILTIN_("print",$cstat.PATH($csel))
```

▷ **Warning**: Builtin functions is a dangerous feature. Each error can be fatal to Foresys.

8.13 Clipboard functionalities

The following functions provide a mechanism to preserve informations betweens two executions of scripts.

- ▶ **Clipboard:Set(NAME,INFO)**
  This statement puts an information in a clipboard
  - NAME (input STRING or INT) is the name of the clipboard.
  - INFO (input ALL TYPES) is the information.

- ▶ **Clipboard:Test(NAME)**
  This function tests if there is an information stored in a clipboard.
  - NAME (input STRING or INT) is the name of the clipboard.
  - The function returns TRUE if a value is associated to the clipboard else FALSE.

- ▶ **Clipboard:Get(NAME)**
  This function gets the information stored in a clipboard.

- FOO (input) is not used
- NAME (input STRING or INT) is the name of the clipboard.
- The function returns the value associated to the clipboard.

▷ **Example 57:** A simple script to copy the selected statement in the clipboard "cut0".

```
SCRIPT paste()
  foo := Clipboard:Set("cut0",COPY($cstat))
ENDSCRIPT
```

▷ **Example 58:** A simple script to insert a copy of the statement saved by the example 57.

```
SCRIPT paste()
  IF (Clipboard:Test("cut0")) THEN
    newStat := Clipboard:Get("cut0")
    TT := COPY(newStat)
    PASTE("AFTER",TT,$cstat)
  ENDIF
ENDSCRIPT
```

▷ **Remark**: We recommend to use the clipboard "cut0" to implement the CUT & PASTE functionalities.

8.14 Undo functionalities

The following builtin functions provides a mechanism to implement the UNDO & REDO functionalities.

▶ **Undo:Save(AST,SAVESYMTAB,SAVEDECL)**
This function saves a AST in the UNDO list

- AST (input AST) is the AST to saved. It must be in the current unit (i.e. see $cunit) and must contains the modified AST.
- SAVESYMTAB (input BOOLEAN) indicates that the symbol table must be save (not yet implemented)
- SAVEDECL (input BOOLEAN) indicates that the declaration AST must be saved (not yet implemented)

▷ **Example 59:** Save the AST before a transformation

```
SCRIPT transfWithUndo()
  stat:=$cstat
  foo:=Undo:Save(stat.PARENT,TRUE,TRUE)
  stat <- PARSESTAT("        print *,100")
ENDSCRIPT
```

8.15 Dialog boxes

A set of builtin functions are provided to open dialog boxes. The naming protocol is to prefix each function by *Dialog:*

▷ **Warning**: In the current version, only one window can be open simultaneously. The execution stops until the window is close. There is an exception ; the windows created by Dialog:Util:ErrorMessage and Dialog:Util:ErrorMessage are not managed by TSF but by FORESYS. On the other hand, it is not possible to get a return value from those windows.

8.15.1 Simple dialogue boxes

Some predefined dialogue boxes are available to execute simple tasks such as reading a string or displaying a text. The type STRINGLIST used in the following is a string where the characters '—' and *newline* are used as separators to create a list of strings. For example, "Ok—Cancel" means the strings "Ok" and "Cancel".

▶ **Dialog:Util:ReadInteger(TITLE,MESG,LABEL,DEFAULT)**
The dialog box created by this function asks the user for an integer value.

- TITLE (input STRING) is the dialog window's title.
- MESG (input STRINGLIST) is a message for the user.
- LABEL (input STRING) is a label associated to the input field.
- DEFAULT (input INT) is the default value of the input field.
- The function returns the integer value given by the user (return INT).

▶ **Dialog:Util:ReadText(TITLE,MESG,LABEL,DEFAULT)**
The dialog box created by this function asks the user for a short text (one line).

- TITLE (input STRING) is the dialog window's title.
- MESG (input STRINGLIST) is a message for the user.
- LABEL (input STRING) is a label associated to the input field.
- DEFAULT (input STRING) is the default value of the input field.
- The function returns the short text given by the user (return STRING).

▶ **Dialog:Util:ReadTextWithChoices(TITLE,MESG,LABEL,DEFAULT,CHOICES)**
The dialog box created by this function asks the user for a short text (one line). A list of predefined texts is proposed to the user.

- TITLE (input STRING) is the dialog window's title.
- MESG (input STRINGLIST) is a message for the user.
- LABEL (input STRING) is a label associated to the input field.
- DEFAULT (input STRING) is the default value of the input field.
- CHOICES (input STRINGLIST) is the list of predefined texts. The function returns the short text given by the user (return STRING).

▶ **Dialog:Util:AskUser(TITLE,MESG,BUTTONS,TYPE)**
The dialog box created by this function displays a message and waits to press an exit button.

- TITLE (input STRING) is the dialog window's title.
- MESG (input STRINGLIST) is a message for the user.
- BUTTONS (input STRINGLIST) is a list of button's labels.
- TYPE (input INT) specify the message type. 0:none 1:information 2:warning 3:working 4:question 5:error
- The function returns the index of the exit button selected by the user (return INT).

▶ **Dialog:Util:InfoMessage(MESG)**
The dialog box created by this statement displays a message. No result is returned

- MESG (input STRING) is the message for the user.

▶ **Dialog:Util:ErrorMessage(MESG)**
The dialog box created by this statement displays an error message. No result is returned.

- MESG (input STRING) is the message for the user.

8.15.2 Custom dialogue boxes

There are 5 steps in the life of a custom dialog box:

1. The initialization by a call of the Dialog:Create function.
2. The creation of the input/output fields.
3. The management by one or more calls of Dialog:Run function.
4. The reading of the output fields
5. The closing by a call of the Dialog:Close function.

The steps 1, 3 and 5 are performed by the following functions:

▶ **Dialog:Create(NAME,TITLE,BUTTONS)**
This function creates a new dialogue box.

- FOO (input) is not used
- NAME (input STRING) is the dialog's name.
- TITLE (input STRING) is the dialog window's title.
- BUTTONS (input STRINGLIST) is the list of the exit button's labels.
- The function return a dialog box identifier (return INT).

▶ **Dialog:Run(WIN)**
This function manages a dialog box until one of the exit buttons is pressed. The dialog box is not removed from the view.

- FOO (input) is not used

- WIN (input INT) is the dialog box identifier.

▶ **Dialog:Close(WIN)**
This function remove a dialog box from the view.

- FOO (input) is not used
- WIN (input INT) is the dialog box identifier.

8.15.3 Creation of the fields

Each of the functions described in this section inserts an input or an output field in a dialog box.

▶ **Dialog:InsertText(WIN MESG ALIGN)**
The output field inserted by this function displays a message.

- WIN (input INT) is the dialog box identifier.
- MESG (input STRINGLIST) is the message to be displayed.
- ALIGN (input INT) is the alignment ($<$ 0:left, 0:centered, $>$ 0:right)
- The function returns nothing

▶ **Dialog:InsertText(WIN MESG ALIGN ICON)**
The output field inserted by this function displays a message associated to an icon.

- WIN (input INT) is the dialog box identifier.
- MESG (input STRINGLIST) is the message to be displayed.
- ALIGN (input INT) is the alignment ($<$ 0:left, 0:centered, $>$ 0:right)
- ICON (input INT) is the icon to be displayed:
  0:none 1:information 2:warning 3:working 4:question 5:error
- The function returns nothing

▶ **Dialog:InsertMultiText(WIN MESG LINES)**
The output field inserted by this function displays a message in a scrolling zone.

- WIN (input INT) is the dialog box identifier.
- MESG (input STRINGLIST) is the message to be displayed.
- LINES (input INT) is the number of visible lines in the scrolling zone.
- The function returns nothing

▶ **Dialog:InsertTextField(WIN MESG DEFAULT CHOICES)**
The input field inserted by this function reads a short text (one line). A list of predefined texts is proposed to the user. The value associated to the field is a text.

- WIN (input INT) is the dialog box identifier.
- MESG (input STRING) is the label associated to the input field (one line).
- DEFAULT (input STRING) is the default value of the input field.
- CHOICES (input STRINGLIST) is the list of predefined texts.
- The function returns the field identifier (return INT).

▶ **Dialog:InsertMultiTextField(WIN START LINES)**
The input field inserted by this function reads a long text (many lines). The field can be scrolled.

- WIN (input INT) is the dialog box identifier.
- DEFAULT (input STRING) is the default value of the input field.
- LINES (input INT) is the number of visible lines in the scrolling zone.
- The function returns the field identifier (return INT).

▶ **Dialog:InsertIntegerField(WIN MESG DEFAULT)**
The input field inserted by this function reads an integer value. The value associated to the field is an integer.

- WIN (input INT) is the dialog box identifier.
- MESG (input STRING) is the label associated to the input field (one line).
- DEFAULT (input INT) is the default value of the input field.
- The function returns the field identifier (return INT).

▶ **Dialog:InsertRangeField(WIN MESG DEFAULT MIN MAX)**
The input field inserted by this function reads an integer value in a specified range. The value associated to the field is an integer.

- WIN (input INT) is the dialog box identifier.
- MESG (input STRING) is the label associated to the input field (one line).
- DEFAULT (input INT) is the default value of the input field.
- MIN (input INT) is the lower possible value of the input field.
- MAX (input INT) is the greater possible value of the input field.
- The function returns the field identifier (return INT).

▶ **Dialog:InsertMenuField(WIN MESG ITEMS DEFAULT)**
The input field inserted by this function allows to select an item into a menu. The value associated to the field can be an integer (the index of the selected item) or a text (the selected item).

- WIN (input INT) is the dialog box identifier.
- MESG (input STRING) is the label associated to the menu (one line).
- ITEMS (input STRINGLIST) is the list of possibles items
- DEFAULT (input INT) is the index (in ITEMS) of the default item.
- The function returns the field identifier (return INT).

▶ **Dialog:InsertBooleanField(WIN MESG START)**
The input field inserted by this function reads a boolean value. The value associated to the field is an integer (0 for false and 1 for true).

- WIN (input INT) is the dialog box identifier.
- MESG (input STRING) is the label associated to the input field (one line).
- DEFAULT (input INT) is the default value of the input field.
- The function returns the field identifier (return INT).

8.15.4 Reading the fields

The fields can be read after the first call of #:xdialog:run but before the call of #:xdialog:close.

▶ **Dialog:GetFieldAsInteger(WIN FIELD)**
This function reads the integer value associated to a field.

- WIN (input INT) is the dialog box identifier.
- FIELD(input INT) is the field identifier.
- The function returns the integer value (return INT).

▶ **Dialog:GetFieldAsText(WIN FIELD)**
This function reads the text value associated to a field.

- WIN (input INT) is the dialog box identifier.
- FIELD(input INT) is the field identifier.
- The function returns the text value (return STRING).

8.15.5 Example of dialog box

```
SCRIPT SelectGraph()
  // Create the dialog window
  win := Dialog:Start("TSFMATLAB","1D Graphic","Ok|Cancel")
  // The type of available graphics
  names := "Function Plot|Stairs|Bars"
  // Add the instructions for the user.
  mesg := "Please, select a graphic type"
  foo:= Dialog:InsertText(win,mesg,0)
  // Add the menu
  f_sel := Dialog:InsertMenuField(win,
             "types : ",names,1)
  // Add an empty line
  foo:= Dialog:InsertText(win," ",0)
  // Open the window and wait for the answer
  res := Dialog:Run(win)

  IF (res==0) THEN
    num := Dialog:GetFieldAsInteger(win,f_sel)
    IF (num==0) THEN
      PRINT "You have selected the Function Plot graph"
    ELSEIF (num==1) THEN
      PRINT "You have selected the Stairs graph"
    ELSEIF (num==2) THEN
      PRINT "You have selected the Bars graph"
    ENDIF
  ENDIF
  // And close the window
  foo := Dialog:Close(win)
ENDSCRIPT
```

9 Fits functions

The functions defined for the Fits project are prefixed by the module name *Fits*. An additional module name can be added.

9.1 Constants

The following predefined constants are used by Fits functions:

- Fits:No=0 expresses a false state.
- Fits:Maybe=1 expresses an undefined state.
- Fits:Yes=2 expresses a true state.

9.2 Loop descriptors

Most of the transformations applicable to loops require contextual informations. Rather than to recompute them each time, they are stored in descriptors. Several descriptors stored in a AST structure can be used to describe the structure of a nest of loops. Each transformation updates the AST as well as the associated descriptors. A loop descriptor is stored in an undocumented type of name LOOPDESC. The additional module name is *LoopDesc* (i.e. the full module name is *Fits:LoopDesc*).

The loop descriptors are created by the following functions:

▶ **Fits:LoopDesc:Create(LOOP)**
This function create a loop descriptor associated to the loop LOOP.

- LOOP (input AST) must be a do-loop statement
- The function returns the loop descriptor (LOOPDESC)

▶ **Fits:LoopDesc:RecursiveInit(DESC,DEPTH)**
This statement analyses the nest of loops defined by the loop descriptor DESC (i.e. the outer loop). For each inner loop, a descriptor is created and added in the AST structure. DESC is the root of the AST.

- DESC (input/output LOOPDESC) must be a loop descriptor.
- DEPTH (input INT) is the depth of the recursive analysis.

Some functions are provided to find nodes in a AST of descriptors.

▶ **Fits:LoopDesc:Parent(DESC)**
This function returns the element on top of DESC in the AST of descriptors. If the parent does not exist, a new descriptor is created with a loop equal to FALSE.

- DESC (input/output LOOPDESC) must be a loop descriptor.
- The attribute returns a loop descriptor (LOOPDESC). If there is no parent then a loop descriptor with a statement equal to FALSE is created.

▶ **Fits:LoopDesc:NbChildren(DESC)**
This attribute returns the number of children of DESC.

- DESC (input/output LOOPDESC) must be a loop descriptor.
- The attribute returns a positive integer value (INTEGER).

▶ **Fits:LoopDesc:Rank(DESC)**
This attribute returns the rank of the loop descriptor DESC in the children list of its parent.

- DESC (input/output LOOPDESC) must be a loop descriptor.
- The attribute returns a positive integer value (INTEGER).

▶ **Fits:LoopDesc:Child(DESC,NUM)**
This attribute returns the child of rank NUM of the loop descriptor DESC.

- DESC (input/output LOOPDESC) must be a loop descriptor.
- NUM (input) must be an index .
- The attribute returns a loop descriptor (LOOPDESC).

Several informations are stored in a loop descriptor. They can be accessed by the following functions and statements:

- The statement associated to the loop. This is probably a DO loop.

  ▶ **Fits:LoopDesc:GetStatement(DESC)**
  This function returns the statement (i.e. the loop) associated to the loop descriptor DESC.

  - DESC (input LOOPDESC) must be a loop descriptor.
  - The function returns the loop (AST).

  ▶ **Fits:LoopDesc:SetStatement(DESC,LOOP)**
  This statement puts the statement (i.e. the loop) associated to the loop descriptor DESC to LOOP.

  - DESC(input/output LOOPDESC) must be a loop descriptor.
  - LOOP (input AST) must be a do-loop statement.
  - The function returns the loop (AST).

- The execution status indicates if the loop is executed at least once. The allowed values are Fits:No, Fits:Maybe or Fits:Yes.

  ▶ **Fits:LoopDesc:GetExecutionStatus(DESC)**
  This function gets the execution status.

  - DESC (input LOOPDESC) must be a loop descriptor.
  - The attribute returns one the values Fits:No, Fits:Maybe or Fits:Yes.

  ▶ **Fits:LoopDesc:SetExecutionStatus(DESC,VALUE)**
  This function puts the execution status.

  - DESC (input/output LOOPDESC) must be a loop descriptor.
  - VALUE (input INT) is one the values Fits:No, Fits:Maybe or Fits:Yes.

- The global scope status of the loop index indicates if the index can be used in the loop body without being referenced. When it is set, some transformations are not allowed to modify the index (i.e. its name and its value) in the body of the loop. The allowed values are Fits:No, Fits:Maybe or Fits:Yes.

  ▶ **Fits:LoopDesc:GetIndexGlobalScope(DESC)**
  This attribute gets the global scope status.
  - DESC (input LOOPDESC) must be a loop descriptor.
  - The attribute returns one the values Fits:No, Fits:Maybe or Fits:Yes.

  ▶ **Fits:LoopDesc:SetIndexGlobalScope(DESC,VALUE)**
  This statement puts in the loop descriptor DESC if the loop index has a global scope.
  - DESC (input/output LOOPDESC) must be a loop descriptor.
  - VALUE (input INT) is one the values Fits:No, Fits:Maybe or Fits:Yes.

- The local scope status of the loop index indicates if the index is used after the loop. For several transformations, it means that the original value of the index must be restored after the loop. The allowed values are Fits:No, Fits:Maybe or Fits:Yes.

  ▶ **Fits:LoopDesc:GetIndexExitUsage(DESC)**
  This attribute gets the local scope status.
  - DESC (input LOOPDESC) must be a loop descriptor.
  - The attribute returns one the values Fits:No, Fits:Maybe or Fits:Yes.

  ▶ **Fits:LoopDesc:SetIndexExitUsage(DESC,VALUE)**
  This statement puts in the loop descriptor DESC if the loop index is local to the loop (see before).
  - DESC (input/output LOOPDESC) must be a loop descriptor.
  - VALUE (input INT) is one the values Fits:No, Fits:Maybe or Fits:Yes.

- The positive step status indicate if the step of the loop is positive. The allowed values are Fits:No, Fits:Maybe or Fits:Yes.

  ▶ **Fits:LoopDesc:GetPositiveStep(DESC)**
  This attribute looks in the loop descriptor DESC if the loop index is positive.
  - DESC (input LOOPDESC) must be a loop descriptor.
  - The attribute returns one the values Fits:No, Fits:Maybe or Fits:Yes.

  ▶ **Fits:LoopDesc:SetPositiveStep(DESC,VALUE)**
  This statement puts in the loop descriptor DESC if the loop step is positive.
  - DESC (input/output LOOPDESC) must be a loop descriptor.
  - VALUE (input INT) is one the values Fits:No, Fits:Maybe or Fits:Yes.

- This ending status indicates if the end of the bounds is allways reached by the loop. This information may improve the calculation of the exit value of the index.

  ▶ **Fits:LoopDesc:GetEndIsReached(DESC)**
  This function looks in the loop descriptor DESC if the loop end bound is reached.
  - DESC (input LOOPDESC) must be a loop descriptor.

- The function returns one the values Fits:No, Fits:Maybe or Fits:Yes.

▶ **Fits:LoopDesc:SetEndIsReached(DESC,VALUE)**
This statement puts in the loop descriptor DESC if the loop end bound is reached.

- DESC (input/output LOOPDESC) must be a loop descriptor.
- VALUE (input INT) is one the values Fits:No, Fits:Maybe or Fits:Yes.

10 Simple functions

The following functions are used internally by the FITS transformations.

▶ **Fits:Util:LastStatOfDoLoop(LOOP)**
This function returns the last statement of the do-loop statement LOOP. This is often an node of variant *enddo* or *continue*.

- LOOP (input AST) must be a do-loop statement.
- The function returns a statement (AST).

▶ **Fits:Util:OuterDoLoop(NODE)**
This function returns the do-loop statement that contains the statement or the expression NODE. The falue FALSE is returned if a such do-loop is not found.

- AST (input AST) must be a statement or an expression.
- The function returns a do-loop statement (AST) or FALSE.

▶ **Fits:Util:StatementName(STAT)**
This function returns the name of a DO, IF, FORALL or WHERE statement.

- AST (input AST) must be a DO, IF, FORALL or WHERE statement
- The function returns a node (AST) of variant *name* or FALSE.

▶ **Fits:Util:RemoveStatementName(STAT)**
This statement replaces the statement STAT of variant *labstat* by the statement which it names (i.e. its child of rank 1 is a DO, IF, FORALL or WHERE statement).

- STAT (input AST) must be a *labstat* statement.

▷ **Example 60:** Removing of the name of the loop 'l1'

```
n=4                             n=4
l1: DO i=1,n                    DO i=1,n
  PRINT *,i           -->         PRINT *,i
ENDDO                           ENDDO
```

▶ **Fits:Util:GetTarget(STAT)**
This function returns the target of a GOTO, EXIT or CYCLE statement.

- AST (input AST) must be a GOTO, EXIT or CYCLE statement
- The function returns a statement (AST) .

▶ **Fits:Util:MakeLabelContinue(LABEL)**
This function creates a CONTINUE statement labeled with the specified value. The label must be declared in the fortran symbol table.

- LABEL (input INT) must be a positive integer label.
- The function returns a statement (AST) .

▶ **Fits:Util:MakeGotoLabel(LABEL)**
This function creates a GOTO statement to the specified label.

- LABEL (input INT) must be a positive integer label.
- The function returns a statement (AST) .

▶ **Fits:Util:RemoveCycleOfLoop(LOOP,UNIT)**
This statement transforms all CYCLE statements associated to the specified do-loop statement LOOP into GOTO statements. Some labels may be created by this transformation so an unit UNIT must be specified. The unit can be omited by using the value FALSE when the loop is contained in an unit AST.

- LOOP (input AST) must be a do-loop statement.
- UNIT (input AST) must be an unit AST or FALSE.

▶ **Fits:Util:RemoveExitOfLoop(LOOP,UNIT)**
This statement transforms all EXIT statements associated to the specified do-loop statement LOOP into GOTO statements. Some labels may be created by this transformation so an unit UNIT must be specified. The unit can be omited by using the value FALSE when the loop is contained in an unit AST. A labeled CONTINUE statement can be inserted after LOOP.

- LOOP (input AST) must be a do-loop statement.
- UNIT (input AST) must be an unit AST or FALSE.

▶ **Fits:Util:ChangeLabel(AST,OLD,NEW)**
This statement replaces each occurence in AST of the label OLD by NEW.

- AST (input AST) is the AST to be updated.
- OLD (input INT) is the old label value.
- NEW (input INT) is the new label value.

▶ **Fits:Util:NewLabelValue(UNIT)**
This function returns a label value unused in the specified unit UNIT. The created label is then declared in this unit.

- UNIT (input AST) is a unit AST where the label will be used.
- The function returns a label value (INT).

▶ **Fits:Util:LabelNodesUsedInAst(AST)**
This function returns the list label nodes used in the specified AST AST. Each element of the list (see the attribute GET)

10.1 The clean enddo transformation

The *clean enddo* transformation takes a do-loop statement and replaces its last statement by a *enddo* statement.

▶ **Fits:CleanEnddo:Apply(LOOP)**
This function applies the clean enddo transformation to the statement LOOP.

- LOOP (input AST) must be a do-loop statement
- The function returns the new do-loop statement (i.e. LOOP)

10.2 Loop step standardization transformation

The loop step standardisation transformation set the starting index value and the step of a do-loop statement to 1.

▶ **Fits:Normalization:Apply(DESC)**
This statement applies the clean end-do transformation to the loop described by DESC.

- DESC (input/output LOOPDESC) must be a loop descriptor

10.3 Loop removing transformation

The loop removing consists in transforming a loop into its body.
▷ **Example 61:**

```
DO I=1,N
  A=cos(B)
  C=A*2
ENDDO
```

$$\longrightarrow$$

```
A=cos(B)
C=A*2
```

The following functions are availables.

▶ **Fits:LoopRemoving:Check(LOOP,DIAGMES)**
This functions check if the loop removing transformation can be applied to LOOP.

- LOOP (input AST) must be a do-loop statement
- The function returns an error code
  - 1 if all is correct
  - 0 if LOOP is not a do-loop statement
  - -1 if LOOP contains statements that are not handled by the transformation.
  - -2 if the body of LOOP contains statements that use the loop index.

▶ **Fits:LoopRemoving:Advice(LOOP,DIAGMES)**
This function emits a diagnostic about the interest of the loop removing transformation

- LOOP (input AST) must be a do-loop statement
- DIAGMESG (output STRING) is a diagnostic string for the user
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is recommended
  - 0 if the transformation may be interesting
  - -1 if the transformation in not recommended

▶ **Fits:LoopRemoving:Safe(LOOP,DIAGMES)**
This function emits a diagnostic about the code produced by the transformation

- LOOPS (input AST) must be a do-loop statement
- DIAGMESG (output STRING) is a diagnostic string for the user
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is safe
  - 0 if the transformation may be safe. More information is given by the text in DIAGMESG.
  - -1 if the transformation is not safe

▶ **Fits:LoopRemoving:Apply(LOOP,VERBOSE)**
This function applies the loop removing transformation to LOOP

- AST (input AST) must be a do-loop statement.
- VERBOSE (input INT) is the comment level.
- The function returns the first statement created by the transformation (return AST).

10.4 The loop unrolling and jam transformation

The unrolling and jam transformation must be applied to a perfect nest of do-loop statements. It unrolls the body of one of the inner loops according to the outer loop variable index. When the inner loop is the outer loop (the inner loop of level 0), the unrolling and jam transformation is equivalent to a classical loop unrolling. The knowledge of the sign SIGN of the outer loop step is necessary to obtain an efficient unrolling. This information can be computed automatically or can be specified by the user.

▶ **Fits:UnrollJam:Check(AST,DIAGMESG,LEVEL,FACTOR)**
This functions check if the unrolling and jam transformation can be applied to AST.

- LOOPS (input AST) must be a perfect nest of do-loop statements
- DIAGMESG (output STRING) is the diagnostic string
- LEVEL (input INT) is the level of the inner loop
  - 0 for the outer loop
  - 1 for the first inner loop
  - ¿1 for another inner loop

- FACTOR (input INT) is the unrolling factor.
- The function returns an error code (return INT):
  - 1 if all is correct
  - 0 if AST is not a perfect nest of do-loops of level LEVEL
  - -1 if the inner loop contains statements that are not handled by the transformation.
  - -2 if SIGN does not match the sign of the outer loop step.
  - -3 if FACTOR is lower or equal to 1

▶ **Fits:UnrollJam:Advice(AST,DIAGMESG,LEVEL,FACTOR)**
This function emits a diagnostic about the interest of the loop unrolling transformation

- LOOPS (input AST) must be a perfect nest of do-loop statements
- DIAGMESG (output STRING) is a diagnostic string for the user
- LEVEL (input INT) is the level of the inner loop
  - 0 for the outer loop
  - 1 for the first inner loop
  - ¿1 for another inner loop
- FACTOR (input INT) is the unrolling factor
- SIGN (input INT) is the sign of the outer loop step
  - -1 for a negative sign.
  - 0 for an unknown sign.
  - 1 for a positive sign.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is recommended
  - 0 if the transformation may be interesting
  - -1 if the transformation in not recommended

▶ **Fits:UnrollJam:Safe(AST,DIAGMESG,LEVEL,FACTOR)**
This function emits a diagnostic about the code produced by the transformation

- LOOPS (input AST) must be a perfect nest of do-loop statements
- DIAGMESG (output STRING) is a diagnostic string for the user
- LEVEL (input INT) is the level of the inner loop
  - 0 for the outer loop
  - 1 for the first inner loop
  - ¿1 for another inner loop
- FACTOR (input INT) is the unrolling factor
- SIGN (input/output INT) is the sign of the outer loop step
  - -1 for a negative sign.
  - 0 for an unknown sign.
  - 1 for a positive sign.
- The function returns a diagnostic code (return INT) :

- 1 if the transformation is safe
- 0 if the transformation may be safe. More information is given by the text in DIAGMESG.
- -1 if the transformation is not safe

▶ **Fits:UnrollJam:Apply(DESC,LEVEL,FACTOR)**
This function applies the unroll-and-jam transformation to the loop specified by the descriptor DESC.

- DESC (input LOOPDESC) must be a loop descriptor
- LEVEL (input INT) is the level of the inner loop
  - 0 for the outer loop
  - 1 for the first inner loop
  - ¿1 for another inner loop
- FACTOR (input INT) is the unrolling factor

10.5 Loop distribution

The loop distribution transformation cuts the body of a do-loop statement in two parts. The first statement of the second part specifies the cut position. For example, the following describes a loop distribution according to the statement I3:
▷ **Example 62:**

```
DO I=1,N,4
  PRINT *,I
  PRINT *,I+1
  PRINT *,I+2
  PRINT *,I+3
END DO
```

$$\longrightarrow$$

```
DO I=1,N
  PRINT *,I
  PRINT *,I+1
END DO
DO I=1,N
  PRINT *,I+2
  PRINT *,I+3
END DO
```

▶ **Fits:Distribute:Check(LOOP,DIAGMESG,STAT)**
This functions check if the loop distribution transformation can be applied to LOOP.

- LOOP (input AST) must be a do-loop statement.
- DIAGMESG (output STRING) is the diagnostic string
- STAT (input AST) must be a statement in the body of LOOP.

- The function returns an error code (return INT):
  - 1 if all is correct
  - 0 if LOOP is not a do-loop statement
  - -1 if STAT is not a statement in the body of LOOP
  - -2 if LOOP contains statements not handled by the transformation.

▶ **Fits:Distribute:Advice(LOOP,DIAGMESG,STAT)**
This function emits a diagnostic about the interest of the loop distribution transformation

- LOOP (input AST) must be a do-loop statement.
- DIAGMESG (output STRING) is a diagnostic string for the user
- STAT (input AST) must be a statement in a do-loop body.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is recommended
  - 0 if the transformation may be interesting
  - -1 if the transformation in not recommended

▶ **Fits:Distribute:Safe(LOOP,DIAGMESG,STAT)**
This function emits a diagnostic about the code produced by the loop distribution transformation

- LOOP (input AST) must be a do-loop statement.
- DIAGMESG (output STRING) is the diagnostic string
- STAT (input AST) must be a statement in a do-loop body.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is safe
  - 0 if the transformation may be safe. More information are given by the text in DIAGMESG.
  - -1 if the transformation is not safe

▶ **Fits:Distribute:Apply(LOOP,VERBOSE,STAT)**
This function applies the loop distribution transformation.

- LOOP (input AST) must be a do-loop statement.
- VERBOSE (input INT) is the comment level
- STAT (input AST) must be a statement in the body of LOOP
- The function returns the first do-loop statement created by the transformation (return AST).

10.6 Loop fusion

The loop fusion transformation moves the body of a do-loop statement in another do-loop statement.
▷ **Example 63:**

```
DO I=1,N
  PRINT *,I            DO I=1,N
ENDDO                    PRINT *,I
DO I=1,N        --->     A[I] = F(I)+1
  A[I] = F(I)+1        ENDDO
ENDDO
```

▶ **Fits:LoopFusion:Check(LOOP DIAGMESG LOOP2)**
This function check if the fusion of two do-loop statements is allowed.

- LOOP (input AST) must be the first do-loop statement.
- DIAGMESG (output STRING) is a diagnostic string for the user.
- LOOP2 (input AST) must be the second do-loop statement.
- The function returns an error code (return INT):
  - 1 if all is correct
  - 0 if LOOP is not a do-loop statement
  - -1 if LOOP contains statements not handled by the transformation.
  - -2 if LOOP2 is not a do-loop statement
  - -3 if LOOP2 contains statements not handled by the transformation.
  - -4 if the bounds or the steps of LOOP and LOOP2 are not compatibles.
  - -5 if LOOP and LOOP2 are not in the same sequence of statements

▶ **Fits:LoopFusion:Advice(LOOP DIAGMESG LOOP2)**
This function emits a diagnostic about the interest of the fusion of LOOP2 into LOOP.

- LOOP (input AST) must be a do-loop statement.
- DIAGMESG (output STRING) is a diagnostic string for the user.
- LOOP2 (input AST) must be the second do-loop statement.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is recommended.
  - 0 if the transformation may be interesting.
  - -1 if the transformation in not recommended.

▶ **Fits:LoopFusion:Safe(LOOP,DIAGMESG,LOOP2)**
This functions check if the fusion of LOOP2 in LOOP is safe.

- LOOP (input AST) must be a do-loop statement.
- DIAGMESG (output STRING) is a diagnostic string for the user
- LOOP2 (input AST) must be the second do-loop statement.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is safe
  - 0 if the transformation may be safe. More informations are given by the text in DIAGMESG.
  - -1 if the transformation is not safe

▶ **Fits:LoopFusion:Apply(LOOP,VERBOSE,LOOP2)**
This function applies the loop fusion transformation.

- LOOP (input AST) must be a do-loop statement.
- VERBOSE (input INT) is the comment level
- LOOP2 (input AST) must be the second do-loop statement.
- The function returns the do-loop statement created by the fusion (return AST)

10.7 loop interchanging

The loop interchanging transformation exchanges the indexes, the bounds and the steps of two nested do-loops statements ; the outer loop and an inner loop given by its internal level.

▷ **Example 64:** Interchange of the first and the third loops (indexes I and K). The level is 2.

```
DO I=1,N
  DO J=2,M
    DO K=2,M
      CALL F(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

$$\longrightarrow$$

```
DO K=2,M
  DO J=2,M
    DO I=1,N
      CALL F(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

▶ **Fits:LoopExchangeMulti:Check(OUTLOOP,DIAGMESG,LEVEL)**
This function checks if the loop interchanging transformation can be applied to OUTLOOP.

- OUTLOOP (input AST) must be the outer do-loop statement
- DIAGMESG (output STRING) is the diagnostic string
- LEVEL (input INT) is the inner loop level (1 for the first inner loop).
- The function returns an error code (return INT):
  - 1 if all is correct
  - 0 if OUTLOOP is not a do-loop statement.
  - -1 if the inner loop is not found.
  - -2 if the bounds or the step of the inner loop depend of the index of OUTLOOP.
  - -3 if the inner loop contains statements that are not handled by the transformation.

▶ **Fits:LoopExchangeMulti:Advice(OUTLOOP,DIAGMESG,LEVEL)**
This function emits a diagnostic about the interest of the loop interchanging transformation applied to OUTLOOP.

- OUTLOOP (input AST) must be the outer do-loop statement
- DIAGMESG (output STRING) is the diagnostic string
- LEVEL (input INT) is the inner loop level (1 for the first inner loop).
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is recommended
  - 0 if the transformation may be interesting
  - -1 if the transformation in not recommended

▶ **Fits:LoopExchangeMulti:Safe(OUTLOOP,DIAGMESG,LEVEL)**
This function emits a diagnostic about the code produced by the loop interchanging transformation applied to OUTLOOP.

- OUTLOOP (input AST) must be the outer do-loop statement.
- DIAGMESG (output STRING) is the diagnostic string.
- LEVEL (input INT) is the inner loop level (1 for the first inner loop).
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is safe
  - 0 if the transformation may be safe. More information are given by the text in DIAGMESG.
  - -1 if the transformation is not safe

▶ **Fits:LoopExchangeMulti:Safe(DESC,VERBOSE,LEVEL)**
This function applies the loop interchanging transformation to the loop OUTLOOP.

- DESC (input LOOPDESC) must be a the root node of a AST of loop descriptor with a depth greater of equal to LEVEL.
- VERBOSE (input INT) is the comment level
- The function returns the new do-loop statement created by the transformation (return AST).

10.8 The loop blocking transformation

The loop blocking transformation decomposes a do-loop statement into blocks of a constant blocking factor (the step is multiplied by this factor). The knowledge of the sign of the outer loop step is necessary to obtain an efficient blocking. This information can be computed automatically or can be specified by the user. The blocking factor must be greater than 1. When it is applied to a perfect nest of loops, the outer loop can be blocked in the body of the inner loop of the specified level.

▷ **Example 65:** A loop blocking transformation applied to the loop of index i with a level of 2 and a blocking factor of 4.

```
                    DO Ti=1,m,4
DO i = 1,m            DO j = 1,n
  DO j = 1,n            DO i = Ti,MIN(Ti+4-1,m)
    F(i,j,k)    -->       F(i,j,k)
  ENDDO                 ENDDO
ENDDO                 ENDDO
                    ENDDO
```

The loop blocking transformation is generally followed by a loop interchanging or a loop transformation.

▶ **Fits:Blocking:Check(LOOP,DIAGMESG,FACTOR,LEVEL)**
This function checks if the loop blocking transformation can be applied.

- LOOP (input AST) is the outer do-loop
- DIAGMESG (output STRING) is a diagnostic string for the user
- FACTOR (input INT) is the blocking factor.
- LEVEL (input/output INT) the depth of the inner loop where the blocking is applied. Use 0 to block in the outer loop.
- The function returns an error code (return INT):
  - 1 if all is correct
  - 0 if OUTLOOP is not a perfect nest of do-loops of level LEVEL.
  - -1 if FACTOR is lower or equal to 0.
  - -3 if the bounds or the step of one of the inner loops depend on the outer loop index.

▶ **Fits:Blocking:Advice(LOOP,DIAGMESG,FACTOR,LEVEL)**
This function emits a diagnostic about the interest of the loop blocking transformation

- LOOP (input AST) is the outer do-loop
- DIAGMESG (output STRING) is a diagnostic string for the user
- FACTOR (input INT) is the blocking factor.
- LEVEL (input/output INT) the depth of the inner loop where the blocking is applied. Use 0 to block in the outer loop.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is recommended
  - 0 if the transformation may be interesting
  - -1 if the transformation in not recommended

▶ **Fits:Blocking:Safe(LOOP,DIAGMESG,FACTOR,LEVEL)**
This function emits a diagnostic about the code produced by the loop blocking transformation

- LOOP (input AST) is the outer do-loop
- DIAGMESG (output STRING) is a diagnostic string for the user
- FACTOR (input INT) is the blocking factor.
- LEVEL (input/output INT) the depth of the inner loop where the blocking is applied. Use 0 to block in the outer loop.

- The function returns a diagnostic code (return INT):
  - 1 if the transformation is safe
  - 0 if the transformation may be safe. More information is given by the text in DIAGMESG.
  - -1 if the transformation is not safe

▶ **Fits:Blocking:Apply(DESC,FACTOR,LEVEL)**
This function applies the loop blocking transformation.

- DESC (input LOOPDESC) is the descriptor of the outer loop.
- VERBOSE (input INT) is the comment level
- FACTOR (input INT) is the blocking factor.
- LEVEL (input/output INT) the depth of the inner loop where the blocking is applied. Use 0 to block in the outer loop.
- The function returns the new outer loop (return AST).

10.9 Array padding

The array padding transformation sets a dimension in an array to a new size. The goal of this transformation is to reduce the number of memory system conflicts. The transformation is applied to a full function AST. The new size can be specified by the user or can be computed automatically. The array must be a local variable and it's dimension must be constant.

▶ **Fits:Padding:Check(UNIT,DIAGMESG,ARRAzY,DIM,SIZE)**
This function checks if the array padding transformation can be applied.

- UNIT (input AST) is a foresys unit.
- DIAGMESG (output STRING) is a diagnostic string for the user.
- ARRAY (input STRING) is the array name.
- DIM (input STRING) is the dimension.
- SIZE (input/output INT) is the new size of the dimension. At output, its content is computed if its input value is -1.
- The function returns an error code (return INT) :
  - 1 if all is correct.
  - 0 if ARRAY is not defined.
  - -1 if ARRAY is not a local variable.
  - -2 if DIM is not a valid dimension of ARRAY.

▶ **Fits:Padding:Advice(UNIT,DIAGMESG,ARRAzY,DIM,SIZE)**
This function emits a diagnostic about the interest of the array padding transformation

- UNIT (input AST) is a foresys unit.
- DIAGMESG (output STRING) is a diagnostic string for the user.
- ARRAY (input STRING) is the array name.
- DIM (input STRING) is the dimension.

- SIZE (input INT) is the new size of the dimension. A FALSE value means that the size will be computed automatically.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is recommended
  - 0 if the transformation may be interesting
  - -1 if the transformation in not recommended

▶ **Fits:Padding:Safe(UNIT,DIAGMESG,ARRAzY,DIM,SIZE)**
This function emits a diagnostic about the code produced by the array padding transformation

- UNIT (input AST) is a foresys unit.
- DIAGMESG (output STRING) is a diagnostic string for the user.
- ARRAY (input STRING) is the array name.
- DIM (input STRING) is the dimension.
- SIZE (input INT) is the new size of the dimension. A FALSE value means that the size will be computed automatically.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is safe.
  - 0 if the transformation may be safe. More information is given in DIAGMESG.
  - -1 if the transformation is not safe.

▶ **Fits:Padding:Apply(UNIT,DIAGMESG,ARRAzY,DIM,SIZE)**
This statement applies the array padding transformation.

- AST (input AST) is a Foresys AST.
- VERBOSE (input INT) is the comment level.
- ARRAY (input STRING) is the array name.
- DIM (input STRING) is the dimension.
- SIZE (input INT) is the new size of the dimension. A FALSE value means that the size will be computed automatically (i.e. the old size +1)

10.10 The array dimension interchange transformation

The array dimension interchange transformation exchanges two dimensions of an array. The transformation is applied to a Foresys unit AST.

▶ **Fits:SwapDim:Check(UNIT,DIAGMESG,ARRAY,DIM1,DIM2)**
This function checks if the array padding transformation can be applied.

- AST (input AST) is a Foresys unit.
- DIAGMESG (output STRING) is a diagnostic string for the user.
- ARRAY (input STRING) is the name of the array.
- DIM1 (input INT) is the first dimension to be exchanged.
- DIM2 (input INT) is the second dimension to be exchanged.

- The function returns an error code (return INT):
  - 1 if all is correct.
  - 0 if the function is not an unit AST.
  - -1 if no variable of name ARRAY is not declared.
  - -2 if DIM1 is not a valid dimension for the variable ARRAY.
  - -3 if DIM2 is not a valid dimension for the variable ARRAY or is equal to DIM2.

► **Fits:SwapDim:Advice(UNIT,DIAGMESG,ARRAY,DIM1,DIM2)**
This function emits a diagnostic about the interest of the array dimension interchange transformation.

- AST (input AST) is a Foresys function AST.
- DIAGMESG (output STRING) is a diagnostic string for the user.
- ARRAY (input STRING) is the name of the array.
- DIM1 (input INT) is the first dimension.
- DIM2 (input INT) is the second dimension.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is recommended
  - 0 if the transformation may be interesting
  - -1 if the transformation in not recommended

► **Fits:SwapDim:Safe(UNIT,DIAGMESG,ARRAY,DIM1,DIM2)**
This function emits a diagnostic about the validity of the code produced by the array dimension interchange transformation.

- AST (input AST) is a Foresys function AST.
- DIAGMESG (output STRING) is a diagnostic string for the user.
- ARRAY (input STRING) is the array name.
- DIM1 (input INT) is the first dimension.
- DIM2 (input INT) is the second dimension.
- The function returns a diagnostic code (return INT) :
  - 1 if the transformation is safe
  - 0 if the transformation may be safe. More informations are given by the text in DIAGMESG.
  - -1 if the transformation is not safe

► **Fits:SwapDim:Apply(UNIT,VERBOSE,ARRAY,DIM1,DIM2)**
This statement applies the array dimension interchange transformation.

- AST (input AST) is a Foresys function AST.
- VERBOSE (input INT) is the comment level.
- ARRAY (input STRING) is the name of the array.
- DIM1 (input INT) is the first dimension.
- DIM2 (input INT) is the second dimension.

11 Pattern-Matching Facilities

11.1 Pattern-Matching grammar

The TSF system offers the possibility to use pattern-matching to select Fortran statements. This pattern-matching is fairly standard and operates on the abstract syntax tree (AST) of the program. The current implementation has been derived from Norvig's algorithm which operates on non-structured lists. The syntax of the pattern description language is given table 4. The example 66 shows a pattern given in raw form and as a pseudo-Fortran form. The pseudo-Fortran form is the view produced by the pattern unparser that allows to hide pattern's syntax[4].

pat	var	match any one subtree
	const	match just this atom
	se-pat	match something against a tree
	si-pat	match something against one tree
	(pat . pat)	match the first and the rest
si-pat	(?is var p)	check predicate p on one tree
	(?or pat)	match any pattern on one tree
	(?and pat)	match every pattern on one tree
	(?not pat)	succeed if pattern do not match
segment-pattern	((?* var))	match zero or more trees
	((?+ var))	match one or more trees
	((?? var))	match zero or one tree
	((?if exp))	test if exp is true
var	?char	a symbol starting with ?
const	atom	any nonvariable atom

Table 4: Grammar of the pattern language description. The pattern matching description refers to the abstract syntax tree

▷ **Example 66:**Pattern description example (raw and pseudo-Fortran forms

```
{\bf Raw form}
(do ?name ?label (bounds ?index ?start ?end ?step)
(l_stat
  (ass (vardim ?array1 (l_exp (vardim ?indirect (l_exp ?index))))
    (?op
           (vardim ?array1 (l_exp (vardim ?indirect (l_exp ?index))))
           ?remainderExp))
(?? ?enddoVar)))
```

pseudo-Fortran form

[4] At the present time, this raw form is no more available.

```
?name do ?label ?index = ?start, ?end, ?step
  ?array1(?indirect(?index)) = ?array1(?indirect(?index))
           ?op ?remainderExp
(?? ?enddoVar)
```

One originality of TSF lies in a set of functions that help the user to write pattern descriptions. The idea is to start from a Fortran program and to derive a pattern semi-automatically. These functions, called *abstractors*, generalize patterns according to a criterion. A typical usage of the abstractor starts by selecting a set of Fortran statements. From this set of statements an exact pattern is produced automatically by TSF. This pattern matches exactly the originally selected piece of Fortran code. If the user wishes to look for sets of statements with a similar syntactic structure but not exactly identical he can successively use the abstractor functions to achieve a generalization of the pattern. The pattern abstraction functions are described in the next section.

11.2 Pattern Abstractors

A pattern abstractor is a function that takes as input a pattern and produces a new pattern that will match more Fortran statement than the input pattern. More formally an abstractor *absOp* is defined as follow

$$\forall\ F,\ \forall\ P \in \{Pattern\}, P(F) => absOp(P)(F)$$

where F is a set of Fortran statements. A non exhaustive list of the current abstractors in the system are the following:

- **Name and constant generalization:**
  Anonymous variable names are used by the pattern-matcher instead of using their current values;
- **Loop structure:**
  The pattern corresponding to the end of a loop is replaced by a segment-pattern (see table 4) that will match any end loop structure (*enddo, continue, label statement, ...*);
- **Body extension:**
  The pattern corresponding to the body of a statement is generalized to accept other statements;
- **Assignment statement simplification:**
  The pattern corresponding to an assignment statement is replaced by a pattern that matches any expressions on the left and right hand sides of the assignment.

For instance, applying the pattern abstractor on the code at the top of the example 67, generalizes the pattern corresponding strictly to the Fortran code fragment into a pattern that accepts any variable names and constants. Thus we automatically compute a pattern that matches all loops with a similar syntactic code structure. Of course, the pattern, produced automatically, can then be edited by the user when needed.

All the variables defined in the pattern may be used like some pointers on the subtrees matched at the transformation script level.

▷ **Example 67:**Pattern abstractor

- pseudo-Fortran Code Representation:

```
  DO 10 j = 1,ndim3
    DO 20 i = 1,ndim1
      c2(i,j) = 0
20     CONTINUE
10   CONTINUE
```

- pseudo-Fortran Code Representation (Name and Constant Generalization):

```
  DO ?v161 ?v162 = ?v163, ?v164
    DO ?v165 ?v166 = ?v167, ?v168
      ?v169(?v166, ?v162)=?v172
?v173      CONTINUE
?v174    CONTINUE
```

10, j, 1, ... have respectively been replaced by ?v161, ?v162, ?v163, ..., which may be associated to any kind of constants or variables.

12 tdb: the TSF debugger

In order to help users to follow the execution of their scripts and to understand problems, a debugger has been developed. This one provides the usual operations that you can expect from a tool of this kind.

The debugger is launched from the main TSF window (menu *Options/Debugger*). Its invocation starts a new window which corresponds to the console of the debugger. Once started, it reads commands from the terminal until you tell it to exit with the tdb command *quit*. You can get online help from tdb itself by using the command *help*.

Here is a description of the tdb commands:

▶ **help** [NAME]
Show information about tdb command *NAME*.

▶ **cont** [nb]
Restart the execution. The optional argument is the number of breakpoints to ignore.

▷ **Example 68:**

cont	: continue the execution
cont 10	: continue the execution but ignore 10 breakpoints

▶ **print** [VAR]
Display the content of the variable *VAR*.

▶ **quit**
Quit the debugger and close the tdbcons window.

▶ **break** [SCRIPT] [NAME]
Enable the breakpoint *NAME* in the specified *SCRIPT*.

SCRIPT:

- The symbol . is a shortcut for the current script.
- The symbol * is a shortcut for all scripts (only with @enter and @exit)

NAME:

- A name must start by the symbol @
- Some breakpoints of names @enter and @exit are implicitaly created at the beginning and at the end of each scripts.

▷ **Example 69:**

- break : enable all users breakpoints (not @exit and @enter)
- break test @1 : enable the breakpoint @1 in the script test
- break . : enable all breakpoints in the current script
- break * @exit : enable a breakpoint at the end of each script

▶ **delete** [SCRIPT] [NAME]
Disable the breakpoint *NAME* in the specified *SCRIPT*.

SCRIPT:

- The symbol . is a shortcut for the current script.
- The symbol * is a shortcut for all scripts (only with @enter and @exit)

NAME:

- A name must start by the symbol @
- Some breakpoints of names @enter and @exit are implicitaly created at the beginning and at the end of each scripts.

▷ **Example 70:**

delete	: disable all users breakpoints (not @exit and @enter)
delete test @1	: disable the breakpoint @1 in the script test
delete .	: disable all breakpoints in the current script
delete * @exit	: disable a breakpoint at the end of each script

▶ **list** [SCRIPT]
Display the status of the breakpoints.

▷ **Example 71:**

list	: list all breakpoints
list main	: list the breakpoints in the function main

▶ **display** [VAR]
Print the content of the variable *VAR* after each breakpoint.

▷ **Example 72:**

display i

▶ **trace** [on/off]
Activate/desactivate the breakpoint tracing.

▶ **dump**
Print all variables.

▶ **where**
Print the position in the stack.

▶ **up**
Go up in the stack.

▶ **down**
Go down in the stack.

▶ **stack**
Display the stack.

▶ **go[LEVEL]**
Move directly at the specified level in the stack.

▶ **undisplay** [VAR]
Disable a *display* command.

▷ **Example 73:**

```
display i
undisplay i
```

▶ **ABORT**
When a script is running, the key CTRL-C aborts the execution at the next BREAKPOINT even if it is not activated. A loop without a BREAKPOINT or a script call cannot be stopped.

▶ **error**
Terminate the script by an error. Warning, you have to execute a *cont* command after the *error* command to quit the script.

Breakpoints are set directly in TSF scripts by using the keyword **BREAKPOINT**. The syntax of the command is the following.

▶ **BREAKPOINT** [N]
Set a breakpoint in the script. *N* will be the identifier of the breakpoint and will be used by the tdb commands at the console level.

The figure 11 illustrates the use of the tdb commands during the execution of the script presented figure 10.

```
SCRIPT facto(n,res)
    xx:="test"
    BREAKPOINT(2)
    IF (n<=1) THEN
       res:=1
    ELSE
       CALL facto(n-1,&res)
       res := res*n
    ENDIF
ENDSCRIPT

SCRIPT main()
    a:=1
    PRINT a
    BREAKPOINT(1)
    a:=2
    res:=9999
    CALL facto(5,&n)
    PRINT n
    BREAKPOINT(3)
ENDSCRIPT
```

Figure 10: Example of TSF script which use breakpoints

Action	tdb commands (in console)	Results
Launch the execution of the script		(tdb) Script is running
		BREAK in script main at @enter
	list	All breakpoints are ON
	cont	BREAK in script main at @1
	cont	BREAK in script facto at @2
	stack	3: $<TOPLEVEL>$
		2: main
		* 1: facto
	cont 3	BREAK in script facto at @2
	stack	6: $<TOPLEVEL>$
		5: main
		4: facto
		3: facto
		2: facto
		* 1: facto
	dump	# Symtab facto
		`- n = 2`
		`- xx = 'test'`
		`- res -> res`
		`# Symtab <COMMON>`
		`- $cstat = FALSE`
		`- $cforlib :  TYPE forlib`
		`- $verbose = FALSE`
		`- $cfile :  TYPE flb-file`
		`- $version = 1.01`
		`- $csel = FALSE`
		`- $debug = TRUE`
		`- $cunit :  TREE unit`
		`# Symtab <GLOBAL>`
	where	At level 1/6 in facto
	up	At level 2/6 in facto
	display n	$n = 3$
	down	At level 1/6 in facto
	print n	$n = 2$
	delete facto @2	
	list	facto @2 : OFF
		facto + : ON
		Other breakpoints are ON
	cont	BREAK in script facto at @enter
	delete facto *	
	list	facto @2 : OFF
		facto * : OFF
		facto + : ON
		Other breakpoints are ON
	cont	BREAK in script facto at @exit
	delete .	
	list	facto * : OFF
		Other breakpoints are ON
	cont	BREAK in script main at @3
	dump	# Symtab main
		`- n = 120`
		`- a = 2`
		`- res = 9999`
		`# Symtab <COMMON>`
		`- $cstat = FALSE`
		`- $cforlib :  TYPE forlib`
		`- $verbose = FALSE`
		`- $cfile :  TYPE flb-file`
		`- $version = 1.01`
		`- $csel = FALSE`
		`- $debug = TRUE`
		`- $cunit :  TREE unit`
		`# Symtab <GLOBAL>`
	cont	Script is terminated

Figure 11: Example of use of the tdb commands

13 FAQ

13.1 How can I apply a script to a full forlib ?

By default, the script is applied to the current fortran unit. An experimental functionnality is being developped to go through all the tests.

```
  SCRIPT test ()
  // Initialize the list of units
  foo:=$forlib.INITALLUNITS($cunit)
  // Get the number of units
  nbunit := $forlib.NBUNIT
  cpt:=0
  WHILE(cpt<nbunit)
    // Get the unit AST of rank 'cpt'
    unit := $forlib.FINDUNIT(cpt)
    IF (unit) THEN
      // ### Insert your code here.
      // ### Example:
      PRINT "FILE="+unit.FILENAME
    ENDIF
    cpt:=cpt+1
  ENDWHILE)
ENDSCRIPT
```

You must follow the following rules so that your script works:

- The nature of the unit must be verified.

```
// Get the name of the unit (i.e. a string)
unit_name := unit.SYMBOL
// Get the corresponding entry in the table
// of declarations of the unit
unit_entry := unit.SYMBOL(unit_name)
// Get the declared nature of the unit
unit_type := unit_entry.TYPEOFSYMBOL
// Restrict the actions to some kinds of units
IF ( (unit_type=="subroutine") OR (unit_type=="function")) THEN
  ...
ENDIF
```

- Do not use a simple string such as "X" to describe a fortran object X. By default, the declaration is taken in the current unit (i.e. $cunit). The structure that associate the object X to the unit is given by unit.SYMBOL("X").

13.2 Why does the Undo button not works with my script ?

The undo functionnality is not automatic. You have to save explicitly an AST that contains the piece of code you will change. Look in the section 8.14 or in the following example that inserts a guard around the selected statement.

```
SCRIPT Guards_Inst ()
    // Get the selected statement
    stat := $cstat
    // Save a copy of the including AST (i.e. the parent)
    Undo:Save(stat.PARENT,TRUE,TRUE)
    // Create the condition of the guard
    cond := CREATEEXP("MSGSIZE .ne. 0")
    // Create a conditonnal statement.
    // Its body contains the selected statement.
    new_if := CREATE("struct_if", EMPTY, cond, CREATE("l_stat",stat), CREATE("l_stat"))
    stat <- new_if
ENDSCRIPT
```

13.3 How can I identificate an unit parameter ?

You can look in the unit AST but the simplest method is to test if the INSTATUS or OUTSTATUS attributes return a value different from FALSE.

```
SCRIPT testparameter()
  IF ($cunit.SYMBOL("x").INSTATUS) THEN
    PRINT "The variable x is a parameter"
  ELSE
    PRINT "The variable x is not a parameter"
  ENDIF
ENDSCRIPT
```

13.4 How can I create my own dialog boxes?

The following script creates a dialog box to configure a loop unrolling. All arguments must be passed by reference (i.e. with the prefix &).

```
SCRIPT ReadInfo(level,factor,apply)

  level := 0
  factor :=0

  // Create the dialog window with the buttons "Apply" and "Cancel"
  win := Dialog:Start("DIALOGUNROLL","Loop Unrolling Parameters","Apply|Cancel")
  // Insert a message
  foo := Dialog:InsertText(win," ",-1)
  foo := Dialog:InsertText(win," Loop Unrolling Parameters",0)
  foo := Dialog:InsertText(win," ",-1)
  // Insert the input fields
  // The unrolling level in an integer value between 0 and 20. Its default is 10
  f_level := Dialog:InsertRangeField(win," In level ",10,0,20)
  // The unrolling factor in an integer value between 2 and 20. Its default is 4
```

```
  f_factor := Dialog:InsertRangeField(win," Unrolling Factor   ",4,2,20)
  // Insert a blank line
  foo := Dialog:InsertText(win," ",-1)

  // Show and run the dialog window
  // The result is 0 if the "Apply" button is pressed and 1 for "Cancel".
  res := Dialog:Run(win)

  //  Read the value of each output field
  apply := (res==0)
  IF (apply) THEN
    level := Dialog:GetFieldAsInteger(win,f_level)
    factor := Dialog:GetFieldAsInteger(win,f_factor)
  ENDIF

  // Don't forget to close the window.
  foo := Dialog:Close(win)

ENDSCRIPT
```

13.5 How can I create a structured type ?

TSF does not provide structured types. They can be simulated with hash tables or vectors. The following example creates and prints a list {300,200,100}. We recommand to use vectors when it is possible because hash tables are known to be slower and to use more memory.

```
SCRIPT MakeList (lst)
  // Initialize a list
  lst := NEWHASH
  lst{"head"} := FALSE
ENDSCRIPT

SCRIPT AddToList(lst,val)
  head := lst{"head"}
  elem := NEWHASH
  elem{"value"} := val
  elem{"next"} := head
  lst{"head"}  := elem
ENDSCRIPT

SCRIPT PrintList(lst)
  current :=  lst{"head"}
  WHILE (current)
    PRINT current{"value"}
    current:=current{"next"}
  ENDWHILE
ENDSCRIPT
```

```
SCRIPT main()
  CALLSCRIPT MakeList(&lst)
  CALLSCRIPT AddToList(lst,100)
  CALLSCRIPT AddToList(lst,200)
  CALLSCRIPT AddToList(lst,300)
  CALLSCRIPT PrintList(lst)
ENDSCRIPT
```

13.6 How can I change the dimensions in a declaration ?

The following script applies the transformation A(m:n) –¿ A(m+1:n+1) to the declaration of the selected variable.

```
SCRIPT changeArrayDim ()

  var := $csel

  // -1- Get the symbole of the variable
  // the method depends on the selection

  IF (var.VARIANT == "array_decl") THEN
    // FIRST CASE: this is a selection X(...) in declaration.
    // Using CHILD you can access most of the information in the AST

    // Remark : To see the AST children of the selection, use
    //  PRINT $csel.AST
    // This will give you the variant name and the number of children

    symbol := var.CHILD(1).SYMBOL
  ELSE
    IF ($csel.VARIANT == "name") THEN
     // SECOND CASE: the selection is a symbol in an expression
     symbol := var.SYMBOL
    ELSE
      PRINT "ERREUR : SELECT a variable name or a declaration (like A(m:n,p:q)) "
      EXIT
    ENDIF
  ENDIF


  // -2- verify the number of dimensions
  IF ( symbol.DIMENSION < 1 ) THEN
    PRINT  "ERREUR : SELECT a 1D or +  matrix"
    EXIT
  ENDIF

  // -3- Get the bounds of the dimension
```

```
  dim1 := symbol.DIMENSION(1) // Warning : this is the original ; not a copy

  IF (dim1.VARIANT == "dim_colon") THEN
     // This is the case ; dim1 looks like m:n
     //  The AST structure of "dim_colon" is ("dim_colon" lower_bound upper_bound)
     lb := dim1.CHILD(1) // The lower bound
     ub := dim1.CHILD(2) // The upper bound
  ELSE
     // the default case ; dim1 is an expression
     lb := CREATEEXP("1")
     ub := dim1
  ENDIF
  newlb := CREATEEXP("#lb+1") // The new lower bound
  newub := CREATEEXP("#ub+1") // The new upper bound

  // -4- Create the dim_colon AST with its two childs
  newdim := CREATE("dim_colon",newlb,newub)

  // -5- Set the new dimension (replace the old AST by the new one)
  symbol.DIMENSION(1) <- newdim

ENDSCRIPT
```

Index