```
   Help
#include <stdlib.h>
#include <stdarg.h>

#include "optype.h"
#include "enums.h"
#include "var.h"
#include "tools.h"
#include "ftools.h"
#include "error_msg.h"
#include "pnl/pnl_vector.h"
#include "config.h"

extern char premiasrcdir[MAX_PATH_LEN];
extern char premiamandir[MAX_PATH_LEN];
extern char *path_sep;

// array to storing VARs passed to FprintVar with pt_user =
    = TOVARARRAY
VAR g_printvararray[255];
// the current size of the array
int g_printvararray_size = 0;

int g_dup_printf = 0;
FILE * g_dup_file = 0;

static int ChkParVar1(const Planning *pt_plan,VAR *x,int ta
    g ) ;


#if defined(_WIN32) && !defined(_CYGWIN)
#else
int Spawnlp( int mode, const char *cmdname, const char *ar
    g0, const char *arg1, const char *arg2 )
{
  char cmd[MAX_PATH_LEN]="";
  int test;

  if ((strlen(cmdname)+strlen( " ")+strlen(arg1) +strlen("
    &"))>=MAX_PATH_LEN)
    {
```

```
      Fprintf(TOSCREEN,"%s{n",error_msg[PATH_TOO_LONG]);
      exit(WRONG);
    }


  strcpy(cmd,cmdname);
  strcat(cmd, " ");
  strcat(cmd,arg1);
  if (mode == 1)
    strcat(cmd," &");
  Fprintf(TOSCREEN,"Opening %s {n",arg1);
  test=system(cmd);
  if((test == -1) || (test == 127))
    Fprintf(TOSCREEN, "WARNING: NO HELP AVAILABLE ON YOUR
    OS{n");
  return OK;
}
#endif

/*_____________________________OUTPUT_FILE_______________
    _____________*/

extern FILE* out_stream;
extern char premiasrcdir[MAX_PATH_LEN];/*defined in premia.
    c*/
extern char* path_sep;/*defined in premia.c*/

/**
 * Fprintf:
 * @param user:
 * @param s:
 * @param ...:
 *
 * Custom printf.
 *
 * @return %OK
 **/
int Fprintf(int user,const char s[],...)
{
  va_list ap;
  int return_value=OK;
```

```
  FILE *out;

  va_start(ap,s);

  switch (user)
    {
    case TOSCREEN:
      out=stdout;
      return_value=vfprintf(out,s,ap);
      if (g_dup_printf)
        return_value=vfprintf(g_dup_file,s,ap);
      va_end(ap);
      break;
    case TOFILE:
      out=out_stream;
      if ( out != NULL) return_value=vfprintf(out,s,ap);
      va_end(ap);
      break;
    case TOSCREENANDFILE:
      out=out_stream;
      if (out != NULL)  return_value=vfprintf(out,s,ap);
      va_end(ap);
      va_start(ap,s);
      out=stdout;
      return_value+=vfprintf(out,s,ap);
      va_end(ap);
      break;
    default:
      break;
    }

  return return_value;
}

/**
 * Valid:
 * @param user:
 * @param status:
 * @param helpfile:
 *
 *
```

```
 *
 * @return
 **/
int Valid (int user,int status,char *helpfile)
{
  char msg,answer;
  char fhelp[MAX_PATH_LEN]="";
  int i;

  for(i=0;i<(int)strlen(helpfile);i++)
    helpfile[i]= (char)tolower(helpfile[i]);

  if ((strlen(premiasrcdir))>=MAX_PATH_LEN)
    {
      Fprintf(TOSCREEN,"%s{n",error_msg[PATH_TOO_LONG]);
      exit(WRONG);
    }

  /*strcpy(fhelp,premiasrcdir);
    for(i=strlen(premiasrcdir);i<(int)(strlen(helpfile)+
    strlen(premiasrcdir));i++)
    fhelp[i] = (char)tolower(helpfile[i]);*/
  strcpy(fhelp,premiamandir);
  strcat(fhelp,path_sep);
  strcat(fhelp,helpfile);

  switch(user)
    {
    case TOSCREEN:

      if (status!=OK)
        {
          Fprintf(TOSCREEN,"{nPlease correct.{n");
        }
      else
        {
          do
            {

              Fprintf(TOSCREEN,"{nAll Right (ok: Return,
    no: n, h for Help) ? {t");
```

```
              msg = (char)tolower (fgetc (stdin));
              answer = msg;
              if (answer==’h’){
                premia_spawnlp(fhelp);
              }

              /* Discard rest of input line. */
              while (msg != ’{n’ && msg != EOF)
                msg = (char)fgetc (stdin);
            }
          while (answer==’h’);

          status=!(answer==’{n’);
        }

      Fprintf(TOSCREEN,"{n");
      break;

    case (TOSCREENANDFILE||TOFILE||NO_PAR):

      break;

    default:
      break;
    }
  return status;
}


/*_____________________________VAR SYSTEM_____________________
    ___________*/

static char **formatV;
int         *true_typeV;
static char **error_msgV;

/**
 * InitVar:
 * @param void:
 *
 * Allocates and Initializes formatV, true_typeV,
```

```
    error_msgV.
 *
 * @return %OK if formatV, true_typeV, error_msgV are well
    allocated else %WRONG.
 **/
int InitVar(void)
{
  formatV= malloc(sizeof(Label)*MAX_TYPE);
  if (formatV==NULL)
    return 1;

  true_typeV= malloc(sizeof(int)*MAX_TYPE);
  if (true_typeV==NULL)
    return 1;

  error_msgV= malloc(sizeof(Label)*MAX_TYPE);
  if (error_msgV==NULL)
    return 1;

  /*For completion*/
  formatV[PREMIA_NULLTYPE]="%d";
  true_typeV[PREMIA_NULLTYPE]=INT;
  error_msgV[PREMIA_NULLTYPE]="Should never be asked !";

  formatV[INT]="%d";
  true_typeV[INT]=INT;
  error_msgV[INT]="Should be an integer !";

  formatV[DOUBLE]="%lf";
  true_typeV[DOUBLE]=DOUBLE;
  error_msgV[DOUBLE]="Should be a double !";

  formatV[LONG]="%lu";
  true_typeV[LONG]=LONG;
  error_msgV[LONG]="Should be a long !";

  formatV[PDOUBLE]="%lf";
  true_typeV[PDOUBLE]=DOUBLE;
  error_msgV[PDOUBLE]="Should be greater than 0!";

  formatV[SNDOUBLE]="%lf";
```

```
true_typeV[SNDOUBLE]=DOUBLE;
error_msgV[SNDOUBLE]="Should be lower than 0!";

formatV[PINT]="%d";
true_typeV[PINT]=INT;
error_msgV[PINT]="Should be greater than 0!";

formatV[DATE]="%lf";
true_typeV[DATE]=DOUBLE;
error_msgV[DATE]="Should be a date!";

formatV[RGDOUBLE]="%lf";
true_typeV[RGDOUBLE]=DOUBLE;
error_msgV[RGDOUBLE]="Should range between 0 and 1 !";

formatV[RGDOUBLE1]="%lf";
true_typeV[RGDOUBLE1]=DOUBLE;
error_msgV[RGDOUBLE1]="Should be greater than  1 !";

formatV[RGDOUBLEM11]="%lf";
true_typeV[RGDOUBLEM11]=DOUBLE;
error_msgV[RGDOUBLEM11]="Should range between -1 and 1 !"
  ;

formatV[RGDOUBLE12]="%lf";
true_typeV[RGDOUBLE12]=DOUBLE;
error_msgV[RGDOUBLE12]="Should range between 1 and 2 !";

formatV[RGDOUBLE02]="%lf";
true_typeV[RGDOUBLE02]=DOUBLE;
error_msgV[RGDOUBLE02]="Should range between 0 and 2 !";

formatV[BOOL]="%d";
true_typeV[BOOL]=INT;
error_msgV[BOOL]="Should be a Bool!";

formatV[PADE]="%d";
true_typeV[PADE]=INT;
error_msgV[PADE]="Should be a Pade!";

formatV[SDOUBLE2]="%f";
```

```
true_typeV[SDOUBLE2]=DOUBLE;
error_msgV[SDOUBLE2]="Should be an integer greater than 2
   !";

formatV[INT2]="%d";
true_typeV[INT2]=INT;
error_msgV[INT2]="Should be an integer greater than 2 !";

formatV[RGINT13]="%d";
true_typeV[RGINT13]=INT;
error_msgV[RGINT13]="Should be an integer between 1 and 3
   !";

formatV[RGINT12]="%d";
true_typeV[RGINT12]=INT;
error_msgV[RGINT12]="Should be an integer between 1 and 2
   !";

formatV[RGINT130]="%d";
true_typeV[RGINT130]=INT;
error_msgV[RGINT130]="Should be an integer between 1 and
  30 !";

formatV[SPDOUBLE]="%lf";
true_typeV[SPDOUBLE]=DOUBLE;
error_msgV[SPDOUBLE]="Should be strictly greater than 0!"
  ;

formatV[RGDOUBLE051]="%lf";
true_typeV[RGDOUBLE051]=DOUBLE;
error_msgV[RGDOUBLE051]="Should range between 0.5 and 1 !
  ";

formatV[PNLVECT]="%lf";
true_typeV[PNLVECT]=PNLVECT;
error_msgV[PNLVECT]="Should be an array of double !";

formatV[PNLVECTCOMPACT]="%lf";
true_typeV[PNLVECTCOMPACT]=PNLVECTCOMPACT;
error_msgV[PNLVECTCOMPACT]="Should be a compact array of
  double !";
```

```
  formatV[RGDOUBLE14]="%lf";
  true_typeV[RGDOUBLE14]=DOUBLE;
  error_msgV[RGDOUBLE14]="Should range between 1 and 4 !";

  formatV[FILENAME]="%s";
  true_typeV[FILENAME]=FILENAME;
  error_msgV[FILENAME]="Should be a valid path !";

  formatV[ENUM] = "%d";
  true_typeV[ENUM] = ENUM;
  error_msgV[ENUM] = "Should be an enumerable type";

  return OK;
}


/**
 * Returns a pointer to the member of the enumeration hold
    by x with id key
 *
 * @param x
 * @param key the value of the choice
 * @param index (out) linear index of the entry with id ke
    y (used in the
 * Nsp interface)
 *
 * @return
 */
PremiaEnumMember * lookup_premia_enum_with_index(const VAR
    * x, int key, int *index)
{
  PremiaEnum * e;
  PremiaEnumMember * em;

  e = x->Val.V_ENUM.members;
  *index = 0;

  for ( em = e->members ; em->label != NULL ; em++ )
    {
      if (em->key == key) return em;
```

```
      (*index) ++;
    }
  return NULL;
}

PremiaEnumMember * lookup_premia_enum(const VAR * x, int ke
    y)
{
  int index;
  return lookup_premia_enum_with_index (x, key, &index);
}

VAR * lookup_premia_enum_par(const VAR * x, int key)
{
  PremiaEnumMember *em;
  em = lookup_premia_enum (x, key);
  if ( em == NULL ) return NULL;
  return em->Par;
}

void display_PremiaEnum(VAR * x)
{
  PremiaEnumMember * em;
  PremiaEnum * e;

  e = x->Val.V_ENUM.members;

  for ( em=e->members ; em->label != NULL ; em++ )
    {
      Fprintf(TOSCREEN, "%d:{t%s{n", em->key, em->label);
    }
  Fprintf(TOSCREEN, "{n");
}

/**
 * ChkVar:
 * @param pt_plan:
 * @param x:
 *
 *
 * Implements the Vtype range tests.
```

```
 * Displays TOSCREEN the error message in error_msgV if ne
    cessary
 * pt_plan is of no use, except as an argument of PrintVar(
    pt_plan, TOSCREEN,x):
 * that is in case x->Viter>=0, ie *x has been selected, th
    e check is performed
 * on the current value of x.
 *
 * @return  %OK if *x is in the range. %WRONG otherwise.
 **/
int ChkVar1(const Planning *pt_plan, VAR *x, int tag )
{
  int status=OK;

  if (x->Viter == IRRELEVANT)
    {
      /* no check is performed, because this variable is
    not used
       * for the current computation
       */
      return OK;
    }

  if (x->Vtype<FIRSTLEVEL)
    {
      switch(x->Vtype)
        {
        case PREMIA_NULLTYPE:
          Fprintf(TOSCREEN,"WARNING: CHKVAR OF PREMIA_NULLT
    YPE TYPE VAR{n");
          break;
        case DATE:
          status=(x->Val.V_DATE<0.); /* DATE>=0.*/
          break;
        case PINT:
          status=(x->Val.V_PINT < 1);
          break;
        case BOOL:
          break;
        case PDOUBLE:
          status=(x->Val.V_PDOUBLE<0.); /* PDOUBLE>=0.*/
```

```
      break;

      case SNDOUBLE:
      status=(x->Val.V_PDOUBLE>=0.); /* SNDOUBLE<0.*/
      break;

    case RGDOUBLE:
      status=((x->Val.V_RGDOUBLE<0.) ||(x->Val.V_RG
DOUBLE>1.)); /*0.<=RGDOUBLE<=1.*/
      break;
    case RGDOUBLE1:
      status=((x->Val.V_RGDOUBLE1<=1.)); /*RGDOUBLE1>1.
*/
      break;
    case RGDOUBLEM11:
      status=((x->Val.V_RGDOUBLE<-1.) ||(x->Val.V_RG
DOUBLE>1.)); /*-1.<=RGDOUBLE_M11<=1.*/
      break;
    case RGDOUBLE12:
      status=((x->Val.V_RGDOUBLE12<1.) ||(x->Val.V_RG
DOUBLE12>2.)); /*1.<=RGDOUBLE12<=2.*/
      break;
    case RGDOUBLE02:
      status=((x->Val.V_RGDOUBLE02<=0.) ||(x->Val.V_RG
DOUBLE02>=2.)); /*0.<RGDOUBLE02<2.*/
      break;

    case SDOUBLE2:
      status=(x->Val.V_SDOUBLE2<=2); /*SDOUBLE2>2*/
      break;

    case INT2:
      status=(x->Val.V_INT2<2); /* 2<=INT2*/
      break;

    case RGINT130:
      status=((x->Val.V_RGINT130<1)||(x->Val.V_RGINT130
>30)); /* 1<=RGINT130<=30*/
      break;
    case RGINT13:
      status=((x->Val.V_RGINT13<1)||(x->Val.V_RGINT13>3
```

```
    )); /* 1<=RGINT13<=3*/
          break;
        case RGINT12:
          status=((x->Val.V_RGINT12<1)||(x->Val.V_RGINT12>2
    )); /* 1<=RGINT12<=2*/
          break;
        case SPDOUBLE:
          status=(x->Val.V_PDOUBLE<=0.); /* SPDOUBLE>0.*/
          break;
        case RGDOUBLE051:
          status=((x->Val.V_RGDOUBLE051<0.5) ||(x->Val.V_RG
    DOUBLE051>1.)); /*0.5<=RGDOUBLE051<=1.*/
          break;
        case RGDOUBLE14:
          status=((x->Val.V_RGDOUBLE14<1.) ||(x->Val.V_RG
    DOUBLE14>4.)); /*1.<=RGDOUBLE12<=4.*/
          break;
          /* the generator type is currently NOT tested */
        case ENUM:
          status = lookup_premia_enum(x, x->Val.V_ENUM.val
    ue) == NULL;
          break;
        case FILENAME: /* test if file exists */
          {
            FILE *fd = fopen(x->Val.V_FILENAME, "r");
            status = (fd==NULL);
            if (fd!=NULL) fclose(fd);
          }
          break;
        default:
          break;
        }
      if (tag == OK && status!=OK)
        {
          Fprintf(TOSCREEN,"{nBad value:{n");
          PrintVar(pt_plan,TOSCREEN,x);
          Fprintf(TOSCREEN,"%s",error_msgV[x->Vtype]);
        }
    }
  else
    {
```

```
      switch(x->Vtype)
        {
        case (NUMFUNC_1):
          status=ChkParVar1(pt_plan,(x->Val.V_NUMFUNC_1)->
    Par,tag);
          break;
        case (NUMFUNC_2):
          status=ChkParVar1(pt_plan,(x->Val.V_NUMFUNC_2)->
    Par,tag);
          break;
        case (PTVAR):
          status=ChkParVar1(pt_plan,(x->Val.V_PTVAR)->Par,
    tag);
          break;
        default:
          break;
        }
    }
  return status;
}

int ChkVar(const Planning *pt_plan, VAR *x)
{
  return ChkVar1(pt_plan,x,OK);
}

/**
 * ChkVarLevel:
 * @param pt_plan:
 * @param x:
 *
 * Returns %OK if @param x is a first level variable.
 *
 * @return  %OK or %WRONG.
 **/
int ChkVarLevel(const Planning *pt_plan, VAR *x)
{
  return  (x->Vtype<FIRSTLEVEL) ? OK : WRONG ;
}

/**
```

```
 * ExitVar:
 * @param void:
 *
 * Desallocates formatV, true_typeV, error_msgV.
 **/
void ExitVar(void)
{
  free(formatV);
  free(true_typeV);
  free(error_msgV);
  return;
}

/**
 * FprintfVar:
 * @param user:
 * @param :
 * @param x:
 *
 *
 *
 * @return
 **/
int FprintfVar(int user,const char s[], const VAR *x)
{
  int vt=true_typeV[x->Vtype],return_value=0;
  switch(vt)
    {
    case DOUBLE:
      return_value=Fprintf(user,s,x->Val.V_DOUBLE);
      break;
    case INT:
      return_value=Fprintf(user,s,x->Val.V_INT);
      break;
    case LONG:
      return_value=Fprintf(user,s,x->Val.V_LONG);
      break;
    case ENUM:
      return_value=Fprintf(user,s,x->Val.V_ENUM.value);
      break;
    case PNLVECT:
```

```
  {
    int i;
    /* compulsary test because at that stage Result
parameters have not
       been mallocated yet! Attempt to dereference a
NULL pointer */
    if (user != NAMEONLYTOFILE)
      {
        Fprintf(user, s);
        for (i=0; i<x->Val.V_PNLVECT->size; i++)
          Fprintf(user, "%f ", x->Val.V_PNLVECT->array[
i]);
        Fprintf(user, "{n");
      }
  }
  break;
case PNLVECTCOMPACT:
  {
    int i;
    Fprintf(user, s);
    if (x->Val.V_PNLVECTCOMPACT->convert == 'd')
      {
        Fprintf(user, "%f ", x->Val.V_PNLVECTCOMPACT->
val);
      }
    else
      {
        for (i=0; i<x->Val.V_PNLVECTCOMPACT->size; i++)
          Fprintf(user, "%f ", x->Val.V_PNLVECTCOMPACT-
>array[i]);
      }
    Fprintf(user, "{n");
  }
  break;
case FILENAME:
  Fprintf(user, s, x->Val.V_FILENAME);
  break;
default:
  Fprintf(TOSCREEN,"WARNING: UNKNOWN TRUETYPE IN THE
VAR SYSTEM{n");
  return_value=0;
```

```
      break;
    }
  return return_value;
}


/**
 * Calls PrintVarRec with arg isrec = 0.
 * This function recursively prints Par arg of enumerations
 *
 * @param pt_plan
 * @param user
 * @param x
 *
 * @return
 */
int PrintVar(const Planning *pt_plan,int user,const VAR *x)
{
  return PrintVarRec (pt_plan, user, x, 1);
}


/**
 * PrintVar:
 * @param pt_plan a Planning describing iterations if any
 * @param user an integer describing the kind of printing.
    Possible values
 * are: TOVARARRAY, TOSCREEN, TOFILE, TOSCREENANDFILE, NA
    MEONLYTOFILE,
 * VALUEONLYTOFILE
 * @param x the address of a VAR
 * @param isrec an integer 0 or 1. If 1 Par arg of enums ar
    e recursively
 * printed
 *
 * Print the name and/or the value of *x depending on x->Vi
    ter and user:
 * .PrintVar(&plan,NAMEONLYTOFILE,x):
 *   Fprint(TOFILE,"#%s{n",x->Vname);
 * .PrintVar(&plan,VALUEONLYTOFILE,x):
 *   Fprint(TOFILE,"formatV[x->Vtype]{t",x->Vname);
```

```
 * .PrintVar(&plan,TOFILE,x):
 * if x->Viter==ALLOW or FORBID, (ie *x has not been selec
    ted for iteration)
 * Fprint(TOFILE,"#%s{tformatV[x->Vtype]{n,x->Vname,x->Val)
    ;
 * else
 * Fprint(TOFILE,"#%s{t:from formatV[x->Vtype] to formatV[x
    ->Vtype] step
 * %d{n",
 * x->Vname,Min.Val,Max.Val,StepNumber);
 * where Min, Max and StepNumber are the fields of plan->
    Par[Viter].
 *
 * PrintVar(&plan,TOSCREENANDFILE,x): the same with
 * Fprintf(TOSCREENANDFILE,..)
 *
 * !!!!!!!!!!!!!!! WARNING!!!!!!!!!!!!!!!
 * (i) Fprintf(user,"formatV[x->Vtype] formatV[y->Vtype]",x
    ->Val,y->Val)
 * DOES NOT WORK,
 * so such a Fprintf is cut into parts.
 * (ii) No test is made to check the temporary string: char
 * string[MAX_CHAR]; is smaller than MAX_CHAR
 * !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 *
 * @return  the return_value of the last Fprintf
 **/
int PrintVarRec(const Planning *pt_plan,int user,const VAR
    *x, int isrec)
{
  char string[MAX_CHAR_X4];
  const Iterator* pt_it;
  int return_value=1;
  PremiaEnumMember *em;

  if (x->Vtype<FIRSTLEVEL)
    {

      if (x->Viter!=IRRELEVANT)
        {
          switch (user)
```

```
          {
          case TOVARARRAY:
            {
              g_printvararray[g_printvararray_size++] = *
x;
              break;
            }
          case TOSCREEN:

            if (x->Viter >= ALREADYITERATED)
              {

                pt_it=&(pt_plan->Par[x->Viter-ALREADYITER
ATED]);

                strcpy( string, x->Vname );
                strcat( string, " from " );
                strcat(string,formatV[x->Vtype]);
                FprintfVar(TOSCREEN,string, &(pt_it->Min)
);
                strcpy( string, " to " );
                strcat(string,formatV[x->Vtype]);
                FprintfVar(TOSCREEN,string, &(pt_it->Max)
);
                strcpy(string," step %d");
                strcat(string,"{n");

                return_value=Fprintf(TOSCREEN,string,pt_
it->StepNumber);
              }
            else if ((x->Viter==ALLOW)||(x->Viter==FORBID
))
              {
                if (x->Vtype == ENUM)
                  {
                    strcpy(string, x->Vname);
                    strcat(string, ":{t" );
                    strcat(string,formatV[x->Vtype]);
                    strcat(string," (");
                    if ((em = lookup_premia_enum(x, x->
Val.V_ENUM.value))==NULL)
```

```
                    strcat(string, "NOT A VALID CHOICE"
);
                  else
                    strcat(string,em->label);
                  strcat(string," )");
                  strcat(string, "{n" );
                  return_value=FprintfVar(TOSCREEN,stri
ng, x);
                  if ( isrec == 1 && em != NULL )
                    {
                      int i;
                      for ( i=0 ; i<em->nvar ; i++ )
                        {
                          PrintVar(pt_plan, user, &(em-
>Par[i]));
                        }
                    }
                }
              else
                {
                  strcpy(string, x->Vname );
                  strcat(string, ":{t" );
                  strcat(string,formatV[x->Vtype]);
                  strcat(string, "{n" );
                  return_value=FprintfVar(TOSCREEN,stri
ng, x);
                }
            }
          else
            {
              pt_it=&(pt_plan->Par[x->Viter]);

              strcpy(string, x->Vname );
              strcat( string, " from " );
              strcat(string,formatV[x->Vtype]);
              FprintfVar(TOSCREEN,string, &(pt_it->Min)
);
              strcpy( string, " to " );
              strcat(string,formatV[x->Vtype]);
              FprintfVar(TOSCREEN,string, &(pt_it->Max)
);
```

```
                strcpy(string," step %d");
                strcat(string,"{n");

                return_value=Fprintf(TOSCREEN,string,pt_
it->StepNumber);
              }
            break;

          case TOFILE:

            if (x->Viter >= ALREADYITERATED)
              {
                pt_it=&(pt_plan->Par[x->Viter-ALREADYITER
ATED]);

                strcpy(string,"#");
                strcat( string, x->Vname );
                strcat( string, " from " );
                strcat(string,formatV[x->Vtype]);
                FprintfVar(TOFILE,string, &(pt_it->Min));
                strcpy( string, " to " );
                strcat(string,formatV[x->Vtype]);
                FprintfVar(TOFILE,string, &(pt_it->Max));
                strcpy(string," step %d");
                strcat(string,"{n");

                return_value=Fprintf(TOFILE,string,pt_it-
>StepNumber);
              }
            else if ((x->Viter==ALLOW)||(x->Viter==FORBID
))
              {
                strcpy(string,"#");
                strcat( string, x->Vname );
                strcat( string, ":{t" );
                strcat(string,formatV[x->Vtype]);
                strcat( string, "{n" );

                return_value=FprintfVar(TOFILE,string, x)
;
              }
```

```
            else
              {
                pt_it=&(pt_plan->Par[x->Viter]);

                strcpy(string,"#");
                strcat(string, x->Vname );
                strcat(string, " from " );
                strcat(string,formatV[x->Vtype]);
                FprintfVar(TOFILE,string, &(pt_it->Min));
                strcpy(string, " to " );
                strcat(string,formatV[x->Vtype]);
                FprintfVar(TOFILE,string, &(pt_it->Max));
                strcpy(string," step %d");
                strcat(string,"{n");

                return_value=Fprintf(TOFILE,string,pt_it-
>StepNumber);
              }
            break;

          case TOSCREENANDFILE:

            if (x->Viter >= ALREADYITERATED)
              {
                pt_it=&(pt_plan->Par[x->Viter-ALREADYITER
ATED]);

                strcpy(string,"#");
                strcat( string, x->Vname );
                strcat( string, " from " );
                strcat(string,formatV[x->Vtype]);
                FprintfVar(TOSCREENANDFILE,string, &(pt_
it->Min));
                strcpy( string, " to " );
                strcat(string,formatV[x->Vtype]);
                FprintfVar(TOSCREENANDFILE,string, &(pt_
it->Max));
                strcpy(string," step %d");
                strcat(string,"{n");

                return_value=Fprintf(TOSCREENANDFILE,stri
```

```
ng,pt_it->StepNumber);
            }
          else if ((x->Viter==ALLOW)||(x->Viter==FORBID
))
            {
              strcpy(string,"#");
              strcat( string, x->Vname );
              strcat( string, ":{t" );
              strcat(string,formatV[x->Vtype]);
              strcat( string, "{n" );

              return_value=FprintfVar(TOSCREENANDFILE,
string, x);
            }
          else
            {
              pt_it=&(pt_plan->Par[x->Viter]);

              strcpy(string,"#");
              strcat( string, x->Vname );
              strcat( string, " from " );
              strcat(string,formatV[x->Vtype]);
              FprintfVar(TOSCREENANDFILE,string, &(pt_
it->Min));
              strcpy( string, " to " );
              strcat(string,formatV[x->Vtype]);
              FprintfVar(TOSCREENANDFILE,string, &(pt_
it->Max));
              strcpy(string," step %d");
              strcat(string,"{n");

              return_value=Fprintf(TOSCREENANDFILE,stri
ng,pt_it->StepNumber);
            }
          break;


        case NAMEONLYTOFILE:

          return_value=Fprintf(TOFILE,"#%s{n",x->Vname)
;
```

```
            break;

          case VALUEONLYTOFILE:

            strcpy(string,formatV[x->Vtype]);
            strcat( string, "{t" );
            return_value=FprintfVar(TOFILE,string, x);
            break;

          default:
            break;
          }
      }
  }
else
  {
    switch(x->Vtype)
      {
      case (NUMFUNC_1):
        return_value=ShowParVar(pt_plan,user,(x->Val.V_
  NUMFUNC_1)->Par);
        break;

      case (NUMFUNC_2):
        return_value=ShowParVar(pt_plan,user,(x->Val.V_
  NUMFUNC_2)->Par);
        break;

      case (PTVAR):
        return_value=ShowParVar(pt_plan,user,(x->Val.V_PT
  VAR)->Par);
        break;

      case (PNLVECT):
        if (x->Viter==IRRELEVANT) break;
        strcpy(string, x->Vname );
        strcat(string, ":{t" );
        return_value=FprintfVar(user, string, x);
        break;

      case (PNLVECTCOMPACT):
```

```
          {
            strcpy(string, x->Vname );
            strcat(string, ":{t" );
            return_value=FprintfVar(user, string, x);
          }
          break;

        default:
          break;
        }
    }

  return return_value;

}

/**
 * initializes a PnlVect pointer from a string
 * containing the different values.
 *
 * @param v : a PnlVect already mallocated
 * @param s : the string containing the values to put in the
 * array.
 */
static int charPtr_to_PnlVect(PnlVect *v, const char *s)
{
  int i, n, count;
  double tmp;
  const char *s_addr = s;
  n = 0;
  while (sscanf(s, "%lf%n", &tmp,&count)>0) { s+=count; n++
    ;}
  pnl_vect_resize (v, n);

  for(i=0; i<n; i++)
    {
      sscanf (s_addr, "%lf%n", &(v->array[i]), &count);
      s_addr += count;
    }
  return(i);
}
```

```
/**
 * initializes a PnlVectCompact pointer from a string
 * containing the different values.
 *
 * @param v : a PnlVectCompact already mallocated
 * @param s : the string containing the values to put in the
 * array. If the string is of length 1, the compact stora
    ge is used
 */
static int charPtr_to_PnlVectCompact(PnlVectCompact *v,
    const char *s)
{
  char *tok;
  char *s_copy = malloc(sizeof(char)*(strlen(s)+1));
  char *init = s_copy; /* only to keep address for delete *
    /
  int i;
  strcpy(s_copy, s);
  if ((v->array=malloc(v->size*sizeof(double)))==NULL)
    {
      PNL_ERROR("Memory allocation error", "charPtr_to_PnlV
    ectCompact");
    }
  v->convert = 'a';
  for(i=0; i<v->size; i++, s_copy=NULL)
    {
      tok = strtok(s_copy, " ");
      if(tok == NULL)
        break;
      v->array[i] = atof(tok);
    }
  if(i==1)
    {
      /* use compact storage */
      v->convert = 'd';
      v->val = v->array[0];
      free (v->array);
    }
  if(i!=1 && i<v->size)
    {Fprintf(TOSCREEN, "size mismatched in charPtr_to_PnlV
```

```
    ect"); exit(1);}
  free(init);
  return(i);
}

/**
 * ScanVar:
 * @param pt_plan:
 * @param user:
 * @param x:
 *
 *
 *
 * @return
 **/
int ScanVar(Planning *pt_plan,int user,VAR *x)
{
  Iterator *pt_iterator;
  int return_value=1;
  int msg,answer;
  char input[MAX_CHAR] = "";

  if (x->Vtype<FIRSTLEVEL)
    {

      if (x->Viter!=IRRELEVANT)
        {
          /*
           * Print the current value, but do NOT go throug
    h Par arg of
           * enums because the Par arg depends on the choic
    e of the enum
           */
          PrintVarRec(pt_plan,user,x,0);

          if ((pt_plan->Action=='p')
              &&(pt_plan->VarNumber<(MAX_ITERATOR-1))
              &&(x->Viter!=FORBID)&&(x->Viter<ALREADYITER
    ATED))

            /*if ((pt_plan->VarNumber<(MAX_ITERATOR-1))&&(x
```

```
->Viter==ALLOW))*/

        {
          Fprintf(TOSCREEN,"{t Ok:Return Modify:m Iter:
i {t?");
          fflush(TOSCREEN);

          answer = tolower(fgetc(stdin));
          msg = answer;
          while( (answer != '{n') && (answer != EOF))
            answer = fgetc(stdin);

          switch(msg)
            {
            case 'i':
              if (x->Viter==ALLOW) /*x has not been se
lected before*/
                {
                  pt_iterator=&(pt_plan->Par[pt_plan->
VarNumber]);
                  pt_iterator->Location=x;
                  (void)CopyVar(x,&(pt_iterator->Defau
lt));

                  x->Viter=pt_plan->VarNumber;

                  pt_iterator->Min.Vtype=x->Vtype;
                  pt_iterator->Max.Vtype=x->Vtype;

                  pt_plan->VarNumber=pt_plan->VarNum
ber+1;
                  (pt_plan->Par[pt_plan->VarNumber]).Mi
n.Vtype=PREMIA_NULLTYPE;

                  pt_iterator->Min.Vname = x->Vname;
                  pt_iterator->Max.Vname = x->Vname;

                  pt_iterator->Min.Viter=FORBID;
                  pt_iterator->Max.Viter=FORBID;
                }
              else if (x->Viter>ALREADYITERATED)
```

```
                {
                  pt_iterator=&(pt_plan->Par[x->Viter-
ALREADYITERATED]);
                }
              else /*x has already been selected befor
e*/
                {
                  pt_iterator=&(pt_plan->Par[x->Viter])
;
                }

              Fprintf(TOSCREEN,"Min value{t?");
              do {
                scanf(formatV[x->Vtype],&((pt_iterator-
>Min).Val.V_INT));
                /*CopyVar(&(pt_iterator->Min),x);*/

              } while (ChkVar(pt_plan,&(pt_iterator->Mi
n))!=OK);

              Fprintf(TOSCREEN,"Max value{t?");
              do {
                scanf(formatV[x->Vtype],&((pt_iterator-
>Max).Val.V_INT));
              } while
                  ((ChkVar(pt_plan,&(pt_iterator->Max))
!=OK)
                   ||(LowerVar(user,&(pt_iterator->Min)
,&(pt_iterator->Max))!=OK));

              Fprintf(TOSCREEN,"Number of steps{t?");
              do
                {
                  return_value=scanf("%d%*c",&(pt_itera
tor->StepNumber));
                } while (ChkStepNumber(user,pt_iterator
,pt_iterator->StepNumber)!=OK);

              break;

            case '{n':
```

```
                return_value=OK;
                break;

              case 'm':
                if (x->Vtype == ENUM)
                  display_PremiaEnum(x);
                if (x->Viter!=ALLOW)
                  ShrinkPlanning(x->Viter,pt_plan);
                x->Viter=ALLOW;
                Fprintf(TOSCREEN,"Value{t?");
                return_value=scanf(formatV[x->Vtype],x->
    Vtype == ENUM ? &x->Val.V_ENUM.value : &(x->Val.V_INT));
                scanf("%*c");
                break;

              default:
                return_value=OK;
                break;
              }
          }
        else /* if (x->Viter==ALLOW)*/
          {

            Fprintf(TOSCREEN,"{t Ok:Return Modify:m{t?");

            fflush(TOSCREEN);
            answer = tolower(fgetc(stdin));
            msg = answer;
            while ((answer != '{n') && (answer != EOF))
              answer = fgetc(stdin);

            switch(msg)
              {
              case '{n':
                return_value=OK;
                break;
              case 'm':
                if (x->Vtype == FILENAME)
                  {
                    Fprintf(TOSCREEN,"Value{t?");
                    return_value=scanf(formatV[x->Vtype],
```

```
  x->Val.V_FILENAME);
                    return  return_value;
                  }
                if (x->Vtype == ENUM)
                  display_PremiaEnum(x);
                Fprintf(TOSCREEN,"Value{t?");
                return_value=scanf(formatV[x->Vtype],x->
  Vtype == ENUM ? &x->Val.V_ENUM.value : &(x->Val.V_INT));
                scanf("%*c");
                break;

              default:
                return_value=OK;
                break;
              }

          }

      }/*Irrelevant*/
  }
else /*Vtype>FirstLevel*/
  {

    switch(x->Vtype)
      {
      case (NUMFUNC_1):
        do
          {
            return_value=GetParVar(pt_plan,user,(x->Val.
  V_NUMFUNC_1)->Par);
          }
        while (ChkParVar(pt_plan,(x->Val.V_NUMFUNC_1)->
  Par)!=OK);
        break;

      case (NUMFUNC_2):
        do
          {
            return_value=GetParVar(pt_plan,user,(x->Val.
  V_NUMFUNC_2)->Par);
          }
```

```
      while (ChkParVar(pt_plan,(x->Val.V_NUMFUNC_2)->
Par)!=OK);
      break;


    case (PTVAR):
      return_value=GetParVar(pt_plan,user,(x->Val.V_PT
VAR)->Par);
      break;

    case PNLVECT:
      {
        if (x->Viter==IRRELEVANT || x->Vsetable==UNSETA
BLE) break;
        PrintVar(pt_plan,user,x);
        Fprintf(TOSCREEN,"{t Ok:Return Modify:m{t?");
        answer = tolower(fgetc(stdin));
        msg = answer;
        while ((answer != '{n') && (answer != EOF))
          answer = fgetc(stdin);

        switch(msg)
          {
          case '{n':
            return_value=OK;
            break;
          case 'm':
            Fprintf(TOSCREEN,"Value{t?");
            /* if fgets returns NULL, nothing to be read
               in stdin */
            if(fgets(input, MAX_CHAR, stdin)!=NULL)
              {
                /* remove newline characters if any */
                while(input[strlen(input)-1] == '{n')
                  input[strlen(input)-1]='{0';
                return_value=charPtr_to_PnlVect(x->Val.
V_PNLVECT, input);
              }
            break;
          default:
            return_value=OK;
```

```
            break;
          }
      }
      break;
    case PNLVECTCOMPACT:
      {
        if (x->Viter==IRRELEVANT || x->Vsetable==UNSETA
BLE) break;
        PrintVar(pt_plan,user,x);
        Fprintf(TOSCREEN,"{t Ok:Return Modify:m{t?");
        answer = tolower(fgetc(stdin));
        msg = answer;
        while ((answer != '{n') && (answer != EOF))
          answer = fgetc(stdin);

        switch(msg)
          {
          case '{n':
            return_value=OK;
            break;
          case 'm':
            Fprintf(TOSCREEN,"Value{t?");
            /* if fgets returns NULL, nothing to be read
               in stdin */
            if(fgets(input, MAX_CHAR, stdin)!=NULL)
              {
                /* remove newline characters if any */
                while(input[strlen(input)-1] == '{n')
                  input[strlen(input)-1]='{0';
                return_value=charPtr_to_PnlVectCompact(
x->Val.V_PNLVECTCOMPACT, input);
              }
            break;
          default:
            return_value=OK;
            break;
          }
      }
      break;
    }
```

```
    }

  /*
   * if *x is an enumeration, recursively call ScanVar on
    the Par arg
   */
  if ( x->Vtype == ENUM )
    {
      int i;
      PremiaEnumMember *em;
      if ((em = lookup_premia_enum(x, x->Val.V_ENUM.value))
    ==NULL) return FAIL;
      for ( i=0 ; i<em->nvar ; i++ )
        {
          return_value += ScanVar(pt_plan, user, &(em->Par[
    i]));
        }
    }
  return return_value;
}

/**
 * CheckIterationValue:
 * @param InputFile:
 * @param pt_plan:
 * @param user:
 * @param x:
 * @param nbline:
 * @param nbchar:
 *
 *
 *
 * @return
 **/
int CheckIterationValue(char **InputFile,Planning *pt_plan,
    int user,VAR *x,int nbline,int nbchar)
{
  int i,j;
  int return_value,stepnumber;
  VAR *xmin;
  VAR *xmax;
```

```
char line[MAX_CHAR_LINE];

return_value=OK;

xmin = malloc(sizeof(VAR));
xmax = malloc(sizeof(VAR));
xmin->Val = x->Val;
xmin->Viter = x->Viter;
xmin->Vname =  x->Vname;
xmin->Vtype=x->Vtype;
xmax->Val = x->Val;
xmax->Viter = x->Viter;
xmax->Vname =  x->Vname;
xmax->Vtype=x->Vtype;
j=nbchar;
for(i=0;i<MAX_CHAR_LINE;i++)
  line[i]='{0';
while((isdigit(InputFile[nbline][j]) !=0 ) ||  (InputFil
  e[nbline][j]=='.')|| (InputFile[nbline][j]=='-')){
  line[j-nbchar]=InputFile[nbline][j];
  j++;
}
nbchar=j+4;
sscanf(line,formatV[xmin->Vtype],&(xmin->Val.V_INT));
if (ChkVar(pt_plan,xmin)!=OK){
  printf("Warning!!!! Error in the iteration value of %s;
  {n{n",x->Vname);
  return_value=WRONG;
}
j=nbchar;
for(i=0;i<MAX_CHAR_LINE;i++)
  line[i]='{0';
while((isdigit(InputFile[nbline][j]) !=0 ) || (InputFile[
  nbline][j]=='.')|| (InputFile[nbline][j]=='-')){
  line[j-nbchar]=InputFile[nbline][j];
  j++;
}
nbchar = j+6;
sscanf(line,formatV[xmax->Vtype],&(xmax->Val.V_INT));
if((ChkVar(pt_plan,xmax)!=OK)||(LowerVar(user,xmin,xmax)!
  =OK)){
```

```
    printf("Warning!!!! Error in the iteration value of %s;
    {n{n",x->Vname);
    return_value=WRONG;
  }
  j=nbchar;
  for(i=0;i<MAX_CHAR_LINE;i++)
    line[i]='{0';
  while((isdigit(InputFile[nbline][j]) !=0 )){
    line[j-nbchar]=InputFile[nbline][j];
    j++;
  }
  sscanf(line,"%d%*c",&stepnumber);
  if ((stepnumber<1) || (stepnumber>1000)){
    printf("Warning!!!! Error in the iteration value of %s;
    {n{n",x->Vname);
    return_value=WRONG;
  }
  return return_value;
}

/**
 * FScanVar:
 * @param InputFile:
 * @param pt_plan:
 * @param user:
 * @param x:
 *
 *
 *
 * @return
 **/
int FScanVar(char **InputFile,Planning *pt_plan,int user,
    VAR *x)
{
  Iterator *pt_iterator;
  int return_value=1;
  int msg,i,j,j0,i0,k;
  char line[MAX_CHAR_LINE];
  VAR xtmp = *x;
  /* avoid warning */
  j0=0;
```

```
/* Recherche de la ligne ou est definie la variable */
i0=-1;
for(i=0;(i<MAX_LINE) && (i0<0);i++){
  j=0;
  while(j<(signed)(strlen(InputFile[i])-strlen(x->Vname))
  )
    {
      for(k=j;k<j+(signed)strlen(x->Vname);k++)
        line[k-j] = InputFile[i][k];
      line[j+(signed)strlen(x->Vname)]='{0';
      if (strcmp(x->Vname,line) == 0){
        i0 = i;
        j0 =j+(signed)strlen(x->Vname)+1;
      }
      j++;
    }
}
if(i0<0){
  if((x->Viter!=IRRELEVANT) && (x->Viter!=FORBID))
    printf("No %s found, default value is: {n",x->Vname);
  PrintVar(pt_plan,user,x);
  printf("{n");
  return_value=OK;
}else{
  if(InputFile[i0][0] == 'i')
    if (CheckIterationValue(InputFile,pt_plan,user,x,i0,
  j0)==WRONG){
      msg='{n';
    }else{
      msg='i';
    }
  else
    if(InputFile[i0][0] == 'm'){
      msg='m';}
    else
      msg='{n';

  if (x->Vtype<FIRSTLEVEL){
    if (x->Viter!=IRRELEVANT){
      /*PrintVar(pt_plan,user,x);*/
```

```
    if ((pt_plan->Action=='p')&&(pt_plan->VarNumber<(
MAX_ITERATOR-1))
        &&(x->Viter!=FORBID)&&(x->Viter<ALREADYITERATE
D))
      {

        /* Recherche d'une valeur ou d'une iteration ou
 autre*/
        switch(msg)
          {
          case 'i':
            if (x->Viter==ALLOW) /*x has not been selec
ted before*/
              {
                pt_iterator=&(pt_plan->Par[pt_plan->
VarNumber]);
                pt_iterator->Location=x;
                (void)CopyVar(x,&(pt_iterator->Default)
);

                x->Viter=pt_plan->VarNumber;

                pt_iterator->Min.Vtype=x->Vtype;
                pt_iterator->Max.Vtype=x->Vtype;

                pt_plan->VarNumber=pt_plan->VarNumber+1
;
                (pt_plan->Par[pt_plan->VarNumber]).Min.
Vtype=PREMIA_NULLTYPE;

                pt_iterator->Min.Vname = x->Vname;
                pt_iterator->Max.Vname = x->Vname;

                pt_iterator->Min.Viter=FORBID;
                pt_iterator->Max.Viter=FORBID;
              }
            else if (x->Viter>ALREADYITERATED)
              {
                pt_iterator=&(pt_plan->Par[x->Viter-ALR
EADYITERATED]);
```

```
              }
            else /*x has already been selected before*/
              {
                pt_iterator=&(pt_plan->Par[x->Viter]);
              }
            /* On cherche la valeur minimal */
            return_value=WRONG;
            j=j0;
            for(i=0;i<MAX_CHAR_LINE;i++)
              line[i]='{0';
            while((isdigit(InputFile[i0][j]) !=0 ) ||
(InputFile[i0][j]=='.')|| (InputFile[i0][j]=='-')){
              line[j-j0]=InputFile[i0][j];
              j++;
            }
            j0=j+4;
            sscanf(line,formatV[x->Vtype],&((pt_itera
tor->Min).Val.V_INT));
            if (ChkVar(pt_plan,&(pt_iterator->Min))!=OK
){
              printf("Error in the value of %s; assumed
 default value{n",x->Vname);
              return_value=OK;
            }
            j=j0;
            for(i=0;i<MAX_CHAR_LINE;i++)
              line[i]='{0';
            while((isdigit(InputFile[i0][j]) !=0 ) || (
InputFile[i0][j]=='.')|| (InputFile[i0][j]=='-')){
              line[j-j0]=InputFile[i0][j];
              j++;
            }
            j0 = j+6;
            sscanf(line,formatV[x->Vtype],&((pt_itera
tor->Max).Val.V_INT));
            if((ChkVar(pt_plan,&(pt_iterator->Max))!=OK
)||(LowerVar(user,&(pt_iterator->Min),&(pt_iterator->Max))
!=OK)){
              printf("Error in the value of %s; assumed
 default value{n",x->Vname);
              return_value=OK;
```

```
            }
            j=j0;
            for(i=0;i<MAX_CHAR_LINE;i++)
              line[i]='{0';
            while((isdigit(InputFile[i0][j]) !=0 )){
              line[j-j0]=InputFile[i0][j];
              j++;
            }
            sscanf(line,"%d%*c",&(pt_iterator->StepNumb
er));
            if (ChkStepNumber(user,pt_iterator,pt_itera
tor->StepNumber)!=OK){
              printf("Error in the value of %s; assumed
 default value:10{n",x->Vname);
              pt_iterator->StepNumber=10;
              return_value=OK;
            }
            if (return_value==OK){
              printf("--> var number : %d{n", pt_plan->
VarNumber);
              pt_plan->VarNumber=pt_plan->VarNumber-1;
            }else{
              return_value=OK;
            }
            break;

          case '{n':
            return_value=OK;
            break;

          case 'm':
            if (x->Viter!=ALLOW)
              ShrinkPlanning(x->Viter,pt_plan);
            x->Viter=ALLOW;
            j=j0;
            for(i=0;i<MAX_CHAR_LINE;i++)
              line[i]='{0';

            while((isdigit(InputFile[i0][j]) !=0 ) || (
InputFile[i0][j]=='-')|| (InputFile[i0][j]=='.')){
              line[j-j0]=InputFile[i0][j];
```

```
              j++;
            }
            return_value=sscanf(line,formatV[xtmp.Vtyp
e],&(xtmp.Val.V_INT));
            if(ChkVar(pt_plan,x)==OK){
              sscanf(line,formatV[x->Vtype],&(x->Val.V_
INT));
              return_value=OK;
            }
            break;

          default:
            return_value=OK;
            break;
          }
      }else{
      switch(msg)
        {
        case 'p':
          printf("No iteration allowed for %s; assuming
 default value{n",x->Vname);
          return_value=OK;
        case '{n':
          return_value=OK;
          break;
        case 'i':
          printf("No iteration allowed for %s; assuming
 default value{n",x->Vname);
          return_value=OK;
          break;
        case 'm':
          j=j0;
          for(i=0;i<MAX_CHAR_LINE;i++)
            line[i]='{0';
          while((isdigit(InputFile[i0][j]) !=0 )|| (Inp
utFile[i0][j]=='-') || (InputFile[i0][j]=='.')){
            line[j-j0]=InputFile[i0][j];
            j++;
          }
          return_value=sscanf(line,formatV[xtmp.Vtype],
&(xtmp.Val.V_INT));
```

```
          if(ChkVar(pt_plan,&xtmp)==OK){

            sscanf(line,formatV[x->Vtype],&(x->Val.V_
INT));
            return_value=OK;
          }
          break;

        default:
          return_value=OK;
          break;
        }
    }
  }
}else{

  switch(x->Vtype)
    {
    case (NUMFUNC_1):
      FGetParVar(InputFile,pt_plan,user,(x->Val.V_
NUMFUNC_1)->Par);
      if(ChkParVar(pt_plan,(x->Val.V_NUMFUNC_1)->Par)==
OK)
        {
          return OK;
        }else{
        printf("Error in parameter %s; exiting....{n",x
->Vname);
        return WRONG;
      }
      break;

    case (NUMFUNC_2):
      FGetParVar(InputFile,pt_plan,user,(x->Val.V_
NUMFUNC_2)->Par);
      if(ChkParVar(pt_plan,(x->Val.V_NUMFUNC_2)->Par)==
OK)
        {
          return OK;
        }else{
```

```
            printf("Error in parameter %s; exiting....{n",x
    ->Vname);
            return WRONG;
          }

        case (PTVAR):
          return_value=FGetParVar(InputFile,pt_plan,user,(x
    ->Val.V_PTVAR)->Par);
          break;

        case PNLVECT:
          {
            charPtr_to_PnlVect(x->Val.V_PNLVECT, &(InputFil
    e[i0][j0]));
          }
          break;

        default:
          break;
        }

    }
  }
  return return_value;
}

/**
 * ChkParVar:
 * @param pt_plan:
 * @param x:
 *
 * The list version of the former.
 *
 * @return
 * -OK if every item has a pertaining value.
 * -The number of bad values otherwise.
 *
 **/
static int ChkParVar1(const Planning *pt_plan,VAR *x,int ta
    g )
{
```

```
  int status=OK;

  while (x->Vtype!=PREMIA_NULLTYPE)
    {
      status+= ChkVar1(pt_plan,x,tag);
      x++;
    }

  return status;
}

int ChkParVar(Planning *pt_plan,VAR *x)
{
  return ChkParVar1(pt_plan,x,OK);
}

/**
 * GetParVar:
 * @param pt_plan:
 * @param user:
 * @param x:
 *
 * The list version of ScanVar.
 *
 * @return
 * -OK if every item has been well ScanVared.
 * -The number of bad scans otherwise.
 **/
int GetParVar(Planning *pt_plan,int user,VAR *x)
{
  int status=OK;

  while (x->Vtype!=PREMIA_NULLTYPE)
    {
      if (x->Viter>ALREADYITERATED){
        x++;
      }
      else{
        status+=(ScanVar(pt_plan,user,x)<=0);
        x++;
      }
```

```
    }
  return status;
}

/**
 * FGetParVar:
 * @param InputFile:
 * @param pt_plan:
 * @param user:
 * @param x:
 *
 * The list version of ScanVar.
 * @return
 * -OK if every item has been well ScanVared.
 * -The number of bad scans otherwise.
 *
 **/
int FGetParVar(char **InputFile,Planning *pt_plan,int user,
    VAR *x)
{
  int status=OK;


  while (x->Vtype!=PREMIA_NULLTYPE)
    {
      if (x->Viter>ALREADYITERATED){
        x++;
      }
      else{
        status+=(FScanVar(InputFile,pt_plan,user,x)<=0);
        x++;
      }
    }
  return status;
}

/**
 * ShowParVar:
 * @param pt_plan:
 * @param user:
 * @param x:
```

```
 *
 * .The list version of PrintVar.
 *
 * @return
 * -OK if every item has been well PrintVared.
 * -NO_PAR if the list is empty.
 * -The number of bad print messages otherwise.
 **/
int ShowParVar(const Planning *pt_plan,int user,const VAR *
    x)
{
  int status=OK;
  const VAR *pt_x = x;
  if (pt_x->Vtype==PREMIA_NULLTYPE)
    return NO_PAR;

  while (pt_x->Vtype!=PREMIA_NULLTYPE)
    {
      if (pt_x->Vsetable == SETABLE)
        status+=(PrintVar(pt_plan,user,pt_x)<0);
      pt_x++;
    }

  return status;
}

/**
 * LowerVar:
 * @param user:
 * @param x:
 * @param y:
 *
 * .Displays to user a message if x->Val>y->Val.
 * !!!!!!!!!!!!!! WARNING!!!!!!!!!!!!!!
 * Assumes that x->Vtype==y->Vtype, no check is performed;
 * the Vtype is read in x->Vtype
 * AND is assumed to be in the range of the beginning ifs;
    otherwise
 * the return value is OK
 * !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 * @return
```

```
 * -OK if x->Val<=y->Val, cf also WARNING
 * -WRONG otherwise.
 *
 **/
int LowerVar(int user,VAR *x, VAR*y)
{
  int status=OK;
  int vt=true_typeV[x->Vtype];

  switch(vt)
    {
    case DOUBLE:

      status+=(x->Val.V_DOUBLE>y->Val.V_DOUBLE);
      break;

    case INT:

      status+=(x->Val.V_INT>y->Val.V_INT);
      break;

    case LONG:

      status+=(x->Val.V_LONG>y->Val.V_LONG);
      break;

    default:
      break;
    }

  if (status!=OK)
    Fprintf(user,"Min %s should be less than Max %s{n",x->
    Vname,y->Vname);

  return status;
}

/**
 * CopyVar:
 * @param srce:
 * @param dest:
```

```
 *
 *
 **/
void CopyVar(VAR *srce,VAR *dest)
{
  memcpy(dest,srce,sizeof(VAR));
}


/*_____________________________________PLANNING____________
    ____________________*/



/**
 * ResetPlanning:
 * @param pt_plan:
 *
 * .Reset *pt_plan:
 * .at the first call, (pt_plan->Par)[0] is set to PREMIA_
    NULLTYPE, pt_plan->VarNumber to 0.
 * .the next calls in addition (and before) the pt_plan->
    Par[i].Viter are set to ALLOW
 *
 **/
void ResetPlanning(Planning *pt_plan)
{
  static int first=1;
  int i;

  if (first)
    {
      first=0;
    }
  else
    {
      for (i=0;i<pt_plan->VarNumber;i++){
        pt_plan->Par[i].Default.Viter=ALLOW;
        (void)CopyVar(&(pt_plan->Par[i].Default),pt_plan->
    Par[i].Location);
        /*  (pt_plan->Par[i].Location)->Viter=ALLOW;  */
```

```
      }
    }

  (pt_plan->Par)[0].Min.Vtype=PREMIA_NULLTYPE;
  pt_plan->VarNumber=0;
  pt_plan->NumberOfMethods=0;

  return;
}

/**
 * ShowPlanning:
 * @param user:
 * @param pt_plan:
 *
 * .Displays the iterated VARs selected in *pt_plan, prece
    ded by {n##{n and
 * followed by ##{n
 * in case user==NAMEONLYTOFILE
 * .If user is not NAMEONLYTOFILE, VALUEONLYTOFILE or TOSCR
    EEN,
 * doesn't do anything.
 *
 **/
void ShowPlanning(int user,const Planning *pt_plan)
{
  int i;


  switch(user)
    {
    case NAMEONLYTOFILE:

      Fprintf(TOFILE,"{n##{n");
      for (i=1;i<=pt_plan->VarNumber;i++)
        PrintVar(pt_plan,NAMEONLYTOFILE,(pt_plan->Par)[i-1]
    .Location);
      Fprintf(TOFILE,"##{n");

      break;
    case VALUEONLYTOFILE:
```

```
      for (i=1;i<=pt_plan->VarNumber;i++)
        PrintVar(pt_plan,VALUEONLYTOFILE,(pt_plan->Par)[i-1
    ].Location);

      break;

    case TOSCREEN:

      for (i=1;i<=pt_plan->VarNumber;i++)
        PrintVar(pt_plan,TOSCREEN,(pt_plan->Par)[i-1].Locat
    ion);

      break;
    default:
      break;
    }

  return;
}

/**
 * ShrinkPlanning:
 * @param index:
 * @param pt_plan:
 *
 *
 * .Removes the item no index in *pt_plan and shrinks *pt_
    plan, resetting the
 * coreesponding values
 * of the Viter fields of the selected VARs.
 *
 **/
void ShrinkPlanning(int index,Planning *pt_plan)
{
  int i;
  Iterator *pt_it, *pt_next_it;

  pt_it=&(pt_plan->Par[index]);
```

```
  for (i=index;i<pt_plan->VarNumber;i++)
    {
      pt_next_it=&(pt_plan->Par[i+1]);

      (void)CopyVar(&(pt_next_it->Min),&(pt_it->Min));

      (void)CopyVar(&(pt_next_it->Max),&(pt_it->Max));
      (void)CopyVar(&(pt_next_it->Default),&(pt_it->Default
    ));
      pt_it->Location=pt_next_it->Location;

      if (pt_it->Min.Vtype!=PREMIA_NULLTYPE)
        (pt_it->Location)->Viter-=1;

      pt_it=pt_next_it;
    }

  pt_plan->VarNumber-=1;

  /*pt_plan->Par[pt_plan->VarNumber].Min.Vtype=PREMIA_NULLT
    YPE;*/

  return;
}

/**
 * ChkStepNumber:
 * @param user:
 * @param pt_iterator:
 * @param step:
 *
 * .Checks for step to be in the range [1,MAX_ITER] which
    is defined in
 * optype.h.
 * .Displays to user an error message if not.
 * Return:
 * -OK if step is in the range.
 * -WRONG otherwise.
 *
 *
 * @return
```

```
 **/
int ChkStepNumber(int user,Iterator *pt_iterator,int step)
{
  static int INT_dummy;
  static long LONG_dummy;

  int vt=true_typeV[pt_iterator->Min.Vtype];

  if ((step<1)|| (step>MAX_ITER))
    {
      Fprintf(user,"should range between 1 and %d{n",MAX_
    ITER);
      return WRONG;
    }

  switch(vt)
    {
    case INT:
      INT_dummy=(pt_iterator->Max.Val.V_INT-pt_iterator->Mi
    n.Val.V_INT)/(int)(pt_iterator->StepNumber);
      if (INT_dummy<1)
        {
          pt_iterator->StepNumber=pt_iterator->Max.Val.V_
    INT-pt_iterator->Min.Val.V_INT;
          Fprintf(user,"WARNING: NUMBER OF STEPS SET TO %d{
    n",pt_iterator->StepNumber);
        }
      break;

    case LONG:
      LONG_dummy=(pt_iterator->Max.Val.V_LONG-pt_iterator->
    Min.Val.V_LONG)/(long)(pt_iterator->StepNumber);
      if (LONG_dummy<1)
        {
          pt_iterator->StepNumber=(int)(pt_iterator->Max.
    Val.V_LONG-pt_iterator->Min.Val.V_LONG);
          Fprintf(user,"WARNING: NUMBER OF STEPS SET TO %d{
    n",pt_iterator->StepNumber);
        }

      break;
```

```
    default:
      break;
    }


  return OK;
}


/**
 * NextValue:
 * @param count:
 * @param pt_iterator:
 *
 * .Compute the next value of pt_iterator.Location->Val ac
    cording to the fields
 * of pt_iterator
 *
 * !!!!!!!!!!!!!! WARNING!!!!!!!!!!!!!!
 * Assumes that pt_iterator->Min.Vtype is in the range of
    the beginning ifs;
 * otherwise
 * dosn't do anything
 * !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 *
 **/
void NextValue(int count,Iterator* pt_iterator)
{
  static double DOUBLE_dummy;
  static int INT_dummy;
  static long LONG_dummy;
  int vt=true_typeV[pt_iterator->Min.Vtype];

  /*FprintfVar(TOSCREEN,formatV[(pt_iterator->Location)->Vt
    ype],pt_iterator->Location);*/

  switch(vt)
    {
    case DOUBLE:
      DOUBLE_dummy=(pt_iterator->Max.Val.V_PDOUBLE-pt_itera
```

```
    tor->Min.Val.V_PDOUBLE)/(double)(pt_iterator->StepNumber);
      (pt_iterator->Location)->Val.V_PDOUBLE+=DOUBLE_dummy;
      if (count==(pt_iterator->StepNumber-1))
        (pt_iterator->Location)->Val.V_PDOUBLE=pt_iterator-
    >Max.Val.V_PDOUBLE;
      break;

    case INT:
      INT_dummy=(pt_iterator->Max.Val.V_INT-pt_iterator->Mi
    n.Val.V_INT)/(int)(pt_iterator->StepNumber);
      (pt_iterator->Location)->Val.V_INT+=INT_dummy;
      if (count==(pt_iterator->StepNumber-1))
        (pt_iterator->Location)->Val.V_INT=pt_iterator->Max
    .Val.V_INT;
      break;

    case LONG:
      LONG_dummy=(pt_iterator->Max.Val.V_LONG-pt_iterator->
    Min.Val.V_LONG)/(long)(pt_iterator->StepNumber);
      (pt_iterator->Location)->Val.V_LONG+=LONG_dummy;
      if (count==(pt_iterator->StepNumber-1))
        (pt_iterator->Location)->Val.V_LONG=pt_iterator->
    Max.Val.V_LONG;
      break;

    default:
      break;
    }

  return;
}

/**
 * ShowParVarTestRes:
 * @param pt_plan:
 * @param user:
 * @param x:
 *
 * This routine displays needed values and needed arrays
    to plot stock
 *    and P&L trajectories IN COLUMNS in the premia.out file
```

```
 *
 *    BE CAREFULL : doublearrays must be the LAST arguments
      of Test->Res[] !!!!
 *
 *    if you want to put a new output argument in Test->Res
     [], you must
 *    right-shift the indices of the arrays
 *
 * @return
 **/
int ShowParVarTestRes(Planning *pt_plan, int user, VAR *x)
{
  int status=OK, k;
  long size;
  VAR *y;
  char string[MAX_CHAR_X3];

  if (x->Vtype==PREMIA_NULLTYPE)
    return NO_PAR;

  while ((x->Vtype!=PREMIA_NULLTYPE)&&(x->Vtype!=PNLVECT))
    {
      status+=(PrintVar(pt_plan,user,x)<0);
      x++;
    }


  if (x->Vtype!=PREMIA_NULLTYPE)
    {
      if (x->Val.V_PNLVECT == NULL) return status;
      size=x->Val.V_PNLVECT->size;
      for (k=0;k<size;k++)
        {
          y=x;
          while (y->Vtype!=PREMIA_NULLTYPE)
            {
              if (y->Val.V_PNLVECT->size==2)
                {
                  strcpy(string,formatV[y->Vtype]);
                  strcat(string," ");
                  strcat(string,formatV[y->Vtype]);
```

```
                strcat(string," ");
                status+=(Fprintf(user,string,y->Val.V_PN
    LVECT->array[0],y->Val.V_PNLVECT->array[1])<0);
                y++;
              }
            else if (y->Val.V_PNLVECT->size>k)
              {
                strcpy(string,formatV[y->Vtype]);
                strcat(string," ");
                status+=(Fprintf(user,string,y->Val.V_PN
    LVECT->array[k])<0);
                y++;
              }
            else
              y++;
          }
        Fprintf(user,"{n");
      }
    }
  return status;
}


/* Utilities to communicate to outside world
 */

static char **formatV;
int         *true_typeV;
static char **error_msgV;

void premia_Vtype_info(VAR *x,char **format,char **
    error_msg,int *type)
{
  *format = formatV[x->Vtype];
  *type = true_typeV[x->Vtype];
  *error_msg= error_msgV[x->Vtype];
}



/*********************************************/
```

```
/** Free functions ***************************/
/*************************************************/
void free_premia_var(VAR *x);


/**
 * free a PtVar struct or a VAR[MAX_PAR]
 *
 * @param x : an array of VAR
 */
void free_premia_par_var(VAR *x)
{
  VAR *it=x;
  while (it->Vtype!=PREMIA_NULLTYPE)
    {
      free_premia_var(it);
      it++;
    }
}


/**
 * free a var when necessary
 *
 * @param x : a pointer to a VAR
 */
void free_premia_var(VAR *x)
{
  switch (x->Vtype)
    {
    case PNLVECT :
      pnl_vect_free (&(x->Val.V_PNLVECT));
      break;

    case PNLVECTCOMPACT :
      pnl_vect_compact_free (&(x->Val.V_PNLVECTCOMPACT));
      break;

    case FILENAME:
      if (x->Val.V_FILENAME!=NULL)
        {
```

```
          free(x->Val.V_FILENAME); x->Val.V_FILENAME=NULL;
        }
      break;

    case PTVAR:
      free_premia_par_var(x->Val.V_PTVAR->Par);
      break;

    case NUMFUNC_1:
      free_premia_par_var(x->Val.V_NUMFUNC_1->Par);
      break;

    case NUMFUNC_2:
      free_premia_par_var(x->Val.V_NUMFUNC_1->Par);
      break;

    case NUMFUNC_ND:
      free_premia_par_var(x->Val.V_NUMFUNC_ND->Par);
      break;
    case ENUM:
        {
          PremiaEnum * e;
          PremiaEnumMember * em;

          e = x->Val.V_ENUM.members;
          if ( e == NULL ) return;

          for ( em = e->members ; em->label != NULL ; em++
    )
            {
              int i;
              for  ( i=0 ; i<em->nvar ; i++ )
                free_premia_var (&(em->Par[i]));
            }
        }
      break;
    default:
      break;
    }
}
```

```
/**
 * free a model instance when needed (i.e. when some
 * variables are mallocated)
 *
 * @param Mod : a pointer to a model
 */
void free_premia_model(Model *Mod)
{
  void* pt=(Mod->TypeModel);
  int nvar = Mod->nvar;
  VAR *var = ((VAR*) pt);
  int i;

  for (i=0; i<nvar; i++)
    free_premia_var(&(var[i]));
  Mod->init=0;
}


/**
 * free an option instance when needed (i.e. when some
 * variables are mallocated)
 *
 * @param Opt : a pointer to an option
 */
void free_premia_option(Option *Opt)
{
  void* pt=(Opt->TypeOpt);
  int nvar = Opt->nvar;
  VAR *var = ((VAR*) pt);
  int i;

  for (i=0; i<nvar; i++)
    free_premia_var(&(var[i]));
  Opt->init =0;
}


/**
 * free a method instance when needed (i.e. when some
```

```
 * variables are mallocated)
 *
 * @param Met : a pointer to a method
 */
void free_premia_method(PricingMethod *Met)
{
  free_premia_par_var(Met->Par);
  free_premia_par_var(Met->Res);
  Met->init=0;
}
```

References