Références bibliographiques

[Cha93] Michel Charpentier. Assistance au raffinement de programmes répartis. *Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis (Grenoble)*, 1993.

[CM88] K.M. Chandy and J. Misra. *"Parallel Program Design: A Fondation"*. Addison-Wesley, 1988.

[Dij76] E. W. Dijkstra. *"A Discipline of Programming"*. Prentice-Hall, Englewood Cliffs, 1976.

[Dro89] Geoff Dromey. *"Program Derivation: The Development of Programs From Specifications"*. International Computer Science Series. Addison-Wesley, 1989.

[FG90] Henri Farreny and Malik Ghallab. *"Eléments d'Intelligence Artificielle"*. Hermès, 1990.

[FMP92] M. Filali, P. Mauran, and G. Padiou. Raffiner pour répartir. *Quatrièmes rencontres du parallélisme (Villeneuce D'Ascq)*, 1992.

[FMP93] M. Filali, P. Mauran, and G. Padiou. Méthodologie de raffinage sur un exemple. 1993.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580,583, 1969.

[KZ87] Deepak Kapur and Hantao Zhang. *"RRL: Rewriting Rule Laboratory User's Manual"*. Department of Computer Science,State University of New York at Albany / The University of Iowa, June 1987.

[Lam80] L. Lamport. The "hoare logic" of concurrent programs. *Acta Informatica*, 14, 1980.

[McC90] William W. McCune. Otter 2.0 users guide. Technical Report ANL-90/9, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, March 1990.

[McC91] William W. McCune. What's new in otter 2.2. Technical Report ANL/MCS-TM-153, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, July 1991.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM TOPLAS*, 4(3), July 1982.

[Ray92] Michel Raynal. *"Synchronisation et état global dans les systèmes répartis"*. Collection de la Direction des Etudes et Recherches d'Electricité de France. Eyrolles, 1992.

[RT89] Thomas W. Reps and Tim Teitelbaum. *"The Synthesizer Generator Reference Manual"*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

```
echo '(sum(x,neg(x)) = 0).'
echo '(neg(sum(x,y)) = sum(neg(x),neg(y))).'
echo '(neg(neg(x)) = x).'
echo 'end_of_list.'

echo 'formula_list(sos).'

read formule
echo "$formule"

echo 'end_of_list.'
```

```
signal(SIGPIPE,handlerio);


if (pipe(sg_ot)) {fprintf(stderr,"Ouverture du pipe Editeur -> Otter   impossible \n");exit(1);};
if (pipe(ot_sg)) {fprintf(stderr,"Ouverture du pipe Otter   -> Editeur impossible \n");exit(1);};

if ((n=fork())== -1) {fprintf(stderr,"Creation fork impossible \n");exit(2);};

if (n)  {if ((p=fork())== -1) {fprintf(stderr,"Creation fork impossible \n");exit(2);};

         if (p)  {wait(0);
                  kill(n,9);}

         else    {close(sg_ot[0]); close(ot_sg[1]);
                  dup2(ot_sg[0],0); dup2(sg_ot[1],1);
                  execl("/homes/isis-u/charpov/unity/UNITY.SYN",
                        "/homes/isis-u/charpov/unity/UNITY.SYN",arg[1],arg[2],arg[3],arg[4],(char *)0);}}

else    {close(sg_ot[1]); close(ot_sg[0]);
         dup2(sg_ot[0],0); dup2(ot_sg[1],1);
         execl("/bin/sh","/bin/sh","/homes/isis-u/charpov/unity/OTTER/DEMONTRER",(char *)0);}}
```

B.3. Programmes Shell

B.3.1. DEMONTRER

```
while [ a ]
              do (/homes/isis-u/charpov/unity/OTTER/TALON_OTTER
                  | /homes/isis-u/charpov/unity/OTTER/otter
                  | grep -c PROOF) 2> /dev/null
done
```

B.3.2. TALON_OTTER

```
echo 'set(sos_queue).'
echo 'set(binary_res).'
echo 'set(factor).'
echo 'set(para_into).'
echo 'set(process_input).'

echo 'clear(print_kept).'
echo 'clear(print_back_sub).'
echo 'clear(print_given).'

echo 'assign(max_mem, 15000).'
echo 'set(free_all_mem).'
echo 'assign(stats_level, 0).'
echo 'assign(max_seconds, 15).'

echo 'formula_list(usable).'
echo '(all x (x = x)).'
echo '(all x all y (-lei(x,y) -> lei(neg(x),neg(sum(y,1))))).'
echo '(all x all y all i all j ((lei(x,i) & lei(y,j)) -> lei(sum(x,y),$SUM(i,j)))).'
echo '(all x all i all j ((lei(x,i) & $LE(i,j)) -> lei(x,j))).'

read base
echo $base

echo 'end_of_list.'

read Commande_lex
echo $Commande_lex

echo 'special_unary([neg(x)]).'

echo 'list(demodulators).'
echo '$CONDITIONAL($AND($NUMBER(x),$NUMBER(y)),$eq_infix(x,y),$EQ(x,y)).'
echo '$CONDITIONAL($AND($NUMBER(x),$NUMBER(y)),sum(x,y),$SUM(x,y))'.
echo '$CONDITIONAL($NUMBER(x),neg(x),$DIFF(0,x)).'
echo '$CONDITIONAL($NUMBER(x),lei(sum(x,y),z),lei(y,$DIFF(z,x))).'
echo '$CONDITIONAL($NUMBER(x),lei(x,y),$LE(x,y)).'

echo '(sum(x,y) = sum(y,x)).'
echo '(sum(x,sum(y,z)) = sum(y,sum(x,z))).'
echo '(sum(x,sum(neg(x),y)) = sum(0,y)).'
echo '(sum(0,x) = x).'
```

```
            Equiv(a,a)                         :True,
            Equiv(a,Not(a))                    :False,
            Equiv(Not(a),a)                    :False,
            Not(True)                          :False,
            Not(False)                         :True,
            Not(Not(a))                        :Simplify(a),

            Eq(a,False)                        :Simplify(Not(a)),
            Eq(False,a)                        :Simplify(Not(a)),
            Eq(a,True)                         :Simplify(a),
            Eq(True,a)                         :Simplify(a),

            Sum(NumConst(a),b)                 :Simplify(Sum(b,NumConst(a))),

            Sum(Sum(a,b),c)                    :Simplify(Sum(a,Sum(b,c))),
            And(And(a,b),c)                    :Simplify(And(a,And(b,c))),
            Or(Or(a,b),c)                      :Simplify(Or(a,Or(b,c))),

            Sum(a,b)                           :let x = Sum(Simplify(a),Simplify(b)) in ((x != e)? Simplify(x):x),
            Neg(a)                             :let x = Neg(Simplify(a)) in ((x != e)? Simplify(x):x),
            LessEq(a,b)                        :let x = LessEq(Simplify(a),Simplify(b)) in ((x != e)? Simplify(x):x),
            Eq(a,b)                            :let x = Eq(Simplify(a),Simplify(b)) in ((x != e)? Simplify(x):x),
            And(a,b)                           :let x = And(Simplify(a),Simplify(b)) in ((x != e)? Simplify(x):x),
            Or(a,b)                            :let x = Or(Simplify(a),Simplify(b)) in ((x != e)? Simplify(x):x),
            Implies(a,b)                       :let x = Implies(Simplify(a),Simplify(b)) in ((x != e)? Simplify(x):x),
            Equiv(a,b)                         :let x = Equiv(Simplify(a),Simplify(b)) in ((x != e)? Simplify(x):x),
            Not(a)                             :let x = Not(Simplify(a)) in ((x != e)? Simplify(x):x),
            default                            :e)};

/*          Peut s'ecrire de maniere plus efficace en evitant certains appels recursifs inutiles          */
```

B.2. Programmes C

B.2.1. otter.c

```
# include "structures.h"
# include "attr.h"
# include "user_interface_exp.h"

# include <stdio.h>


FOREIGN Otter(b,n,s)

PROD_INSTANCE b,n,s;

{ char *ch="FALSE" , c;

        puts(str0_to_str(StrValue(b)));
        puts(str0_to_str(StrValue(n)));
        puts(str0_to_str(StrValue(s)));
        fflush(stdout);
        c=getchar();

/*      fputs(str0_to_str(StrValue(b)),stderr);fputs("\n\|= ",stderr);
        fputs(str0_to_str(StrValue(s)),stderr);fputc('\n',stderr);
        fputs("### ",stderr);fputc(c,stderr);fputs(" ###\n",stderr);fflush(stderr);
*/
        if (c !='0') ch = "TRUE";
        c=getchar();
        return Str(str_to_str0(ch));};
```

B.2.2. unity.c

```
# include <stdio.h>
# include <signal.h>

int handlerpipe()
{fprintf(stderr,"Probleme SIGPIPE\n");
signal(SIGPIPE,handlerpipe);}

int handlerio()
{fprintf(stderr,"Probleme SIGIO\n");
signal(SIGPIPE,handlerio);}

void main(nb,arg)

int nb;
char * arg[];

{int sg_ot[2] , ot_sg[2] , p , n;


signal(SIGPIPE,handlerpipe);
```

```
/**************************************************************************************************************************
**                                                      APPEL A OTTER                                                    **
**************************************************************************************************************************/

BOOL Theoreme(exp f, INT i)
        {let fs = Simplify(f) in
                (with(fs)
                        (True                   :true,
                         False                  :false,
                         Implies(a,And(b,c))    :Theoreme(Implies(a,b),i) && Theoreme(Implies(a,c),i),
                         And(a,b)               :Theoreme(a,i) && Theoreme(b,i),
                         Implies(a,b)           :with(OtterSSL("",OtterVar(fs),OtterForm(Not(fs))#"."))
                                                ("TRUE" : true,
                                                 default :with(OtterSSL(OtterForm(a)#".",OtterVar(fs),OtterForm(Not(b))#"."))
                                                        ("TRUE" : true, default : false)),
                          *                     :with(OtterSSL("",OtterVar(fs),OtterForm(Not(fs))#"."))
                                                        ("TRUE" : true, default : false)))};

STR foreign Otter(STR b, STR v, STR s);

STR OtterSSL(STR b, STR v, STR s){Otter(b,v,s)};

theorem : Theo  {theorem.faux = (BOOL && (theorem.pbs == "") && Theoreme(exp.etendu,INT))?"":"[ NON PROUVE ]";};

transform theorem
on "Actif"              Theo(v,true,i):Theo(v,false,i),
on "Passif"             Theo(v,false,i):Theo(v,true,i);
```

B.1.11. *simplify.ssl*

```
/**************************************************************************************************************************
**                                              PRE-SIMPLIFICATEUR DE FORMULES                                           **
**************************************************************************************************************************/

exp Simplify(exp e)
        {with(e)(Sum(NumConst(a),NumConst(b))            :NumConst(a+b),
                 LessEq(NumConst(a),NumConst(b))         :BoolConst(a<=b),
                 Eq(NumConst(a),NumConst(b))             :BoolConst(a==b),
                 Neg(NumConst(a))                        :NumConst(-a),

                 Sum(a,NumConst(0))                      :Simplify(a),
                 Sum(NumConst(0),a)                      :Simplify(a),

                 Sum(a,Neg(a))                           :NumConst(0),
                 Sum(Neg(a),a)                           :NumConst(0),
                 Neg(Sum(a,b))                           :Simplify(Sum(Neg(a),Neg(b))),
                 Neg(Neg(a))                             :Simplify(a),

                 Diff(a,b)                               :Simplify(Sum(a,Neg(b))),

                 Eq(a,a)                                 :True,
                 LessEq(a,a)                             :True,

                 GrtEq(a,b)                              :Simplify(LessEq(b,a)),
                 Grt(a,b)                                :Simplify(Less(b,a)),
                 Less(a,b)                               :Simplify(LessEq(Sum(a,NumConst(1)),b)),
                 NotEq(a,b)                              :Simplify(Not(Eq(a,b))),

                 And(a,True)                             :Simplify(a),
                 And(a,False)                            :False,
                 And(True,a)                             :Simplify(a),
                 And(False,a)                            :False,

                 Or(a,False)                             :Simplify(a),
                 Or(a,True)                              :True,
                 Or(False,a)                             :Simplify(a),
                 Or(True,a)                              :True,

                 And(a,a)                                :Simplify(a),
                 And(a,Not(a))                           :False,
                 And(Not(a),a)                           :False,

                 Or(a,a)                                 :Simplify(a),
                 Or(a,Not(a))                            :True,
                 Or(Not(a),a)                            :True,

                 Implies(True,a)                         :Simplify(a),
                 Implies(a,False)                        :Simplify(Not(a)),
                 Implies(False,a)                        :True,
                 Implies(a,True)                         :True,
                 Implies(a,a)                            :True,
                 Implies(And(a,b),Or(a,c))               :True,
                 Implies(And(a,b),Or(c,a))               :True,
                 Implies(And(a,b),Or(b,c))               :True,
                 Implies(And(a,b),Or(c,b))               :True,
                 Implies(And(a,b),a)                     :True,
                 Implies(And(a,b),b)                     :True,
                 Implies(a,Or(a,b))                      :True,
                 Implies(b,Or(a,b))                      :True,
                 Implies(And(a,b),And(a,c))              :Simplify(Implies(And(a,b),c)),
                 Implies(And(a,b),And(c,b))              :Simplify(Implies(And(a,b),c)),
                 Implies(And(g,And(a,b)),And(c,b))       :Simplify(Implies(And(g,And(a,b)),c)),
                 Implies(And(g,And(a,b)),And(a,c))       :Simplify(Implies(And(g,And(a,b)),c)),

                 Implies(Implies(a,b),Implies(a,c))      :Simplify(Implies(a,Implies(b,c))),
                 Implies(Implies(a,b),Implies(c,b))      :Simplify(Implies(Not(b),Implies(c,a))),
                 Implies(And(g,Implies(a,b)),Implies(a,c)):Simplify(Implies(And(g,a),Implies(b,c))),
                 Implies(And(g,Implies(a,b)),Implies(c,b)):Simplify(Implies(And(g,Not(b)),Implies(c,a))),

                 Implies(a,Not(a))                       :Simplify(Not(a)),
                 Implies(Not(a),a)                       :Simplify(a),

                 Equiv(True,a)                           :Simplify(a),
                 Equiv(a,False)                          :Simplify(Not(a)),
                 Equiv(False,a)                          :Simplify(Not(a)),
                 Equiv(a,True)                           :Simplify(a),
```

```
exp Subst(var v, varList vecteur, expList exps)
        {with(vecteur)
                (VarListPair(x,rv)      : with(exps)
                        (ExpListNil             : VarExp(v),
                         ExpListPair(a,re)      : (v == x)?a:Subst(v,rv,re)),
                 default                : VarExp(v))};

/*************************************************************************************************************************
**                      CONSTRUCTION DE LA LISTE DES CONTRAINTES NON VALIDEES PAR LE DEMONSTRATEUR                      **
*************************************************************************************************************************/

STR Surete(requeteList l)
        {with(l)(RequeteListNil                 : "",
                 RequeteListPair(Req(n,i,f),r)  : ((f != NoExp) && Theoreme(f,i))? Surete(r) : n # " "#Surete(r))};

/*************************************************************************************************************************
**                                      ACTIVATION ET DESACTIVATION DES SPECIFICATIONS                                  **
*************************************************************************************************************************/

transform assertion
on "Actif"              AssertionTripl(v,a,true,i):AssertionTripl(v,a,false,i),
on "Passif"             AssertionTripl(v,a,false,i):AssertionTripl(v,a,true,i);

transform racine
on "Tout Actif"         Rac(p,Spec(l1,l2)):Rac(p,Spec(Tout_a_1(l1,true),Tout_a_2(l2,true))),
on "Tout Passif"        Rac(p,Spec(l1,l2)):Rac(p,Spec(Tout_a_1(l1,false),Tout_a_2(l2,false)));

assertionList Tout_a_1(assertionList l, BOOL b)
        {with(l)(AssertionListPair(AssertionTripl(v,a,x,i),r)   :AssertionListPair(AssertionTripl(v,a,b,i),Tout_a_1(r,b)),
                 default                                        :l)};

theoremList Tout_a_2(theoremList l, BOOL b)
        {with(l)(TheoremListPair(Theo(e,x,i),r) :TheoremListPair(Theo(e,b,i),Tout_a_2(r,b)),
                 default                                        :l)};

/*************************************************************************************************************************
**                                              TRANSFORM SUR LES ASSERTIONS                                            **
*************************************************************************************************************************/

transform assertion
on "Invariant"          <assertion>:AssertionTripl(<var>,Invariant(<exp>),false,0),
on "Stable"             <assertion>:AssertionTripl(<var>,Stable(<exp>),false,0),
on "Constant"           <assertion>:AssertionTripl(<var>,Constant(<exp>),false,0),
on "Unless"             <assertion>:AssertionTripl(<var>,Unless(<exp>,<exp>),false,0),

on "Invariant"          AssertionTripl(v,Stable(e),b,i):AssertionTripl(v,Invariant(e),b,i),
on "Invariant"          AssertionTripl(v,Constant(e),b,i):AssertionTripl(v,Invariant(e),b,i),
on "Invariant"          AssertionTripl(v,Unless(e,x),b,i):AssertionTripl(v,Invariant(e),b,i),

on "Stable"             AssertionTripl(v,Invariant(e),b,i):AssertionTripl(v,Stable(e),b,i),
on "Stable"             AssertionTripl(v,Constant(e),b,i):AssertionTripl(v,Stable(e),b,i),
on "Stable"             AssertionTripl(v,Unless(e,x),b,i):AssertionTripl(v,Stable(e),b,i),

on "Constant"           AssertionTripl(v,Invariant(e),b,i):AssertionTripl(v,Constant(e),b,i),
on "Constant"           AssertionTripl(v,Stable(e),b,i):AssertionTripl(v,Constant(e),b,i),
on "Constant"           AssertionTripl(v,Unless(e,x),b,i):AssertionTripl(v,Constant(e),b,i),

on "Unless"             AssertionTripl(v,Invariant(e),b,i):AssertionTripl(v,Unless(e,<exp>),b,i),
on "Unless"             AssertionTripl(v,Stable(e),b,i):AssertionTripl(v,Unless(e,<exp>),b,i),
on "Unless"             AssertionTripl(v,Constant(e),b,i):AssertionTripl(v,Unless(e,<exp>),b,i);
```

B.1.10. *theorem.ssl*

```
/*************************************************************************************************************************
**                                      THEOREMES AUXILLIAIRES ET INTERFACE AVEC OTTER                                  **
*************************************************************************************************************************/

/*************************************************************************************************************************
**                                              DECOMPILATION CONFORME A OTTER                                          **
*************************************************************************************************************************/

STR OtterForm(exp e)
        {with(e)(NumConst(n)    :INTtoSTR(n),
                 BoolConst(b)   :b? "$T" : "$F",
                 VarExp(v)      :with(v)(VarSimple(n):n,*:""),
                 Neg(a)         :"neg(" # OtterForm(a) # ")",
                 Sum(a,b)       :"sum(" # OtterForm(a) # "," # OtterForm(b) # ")",
                 LessEq(a,b)    :"lei(sum(" # OtterForm(a) # ",neg(" # OtterForm(b) # ")),0)",
                 Eq(a,b)        :"(" # OtterForm(a) # " = " # OtterForm(b) # ")",
                 Not(a)         :with(a)(Not(b):OtterForm(b),*:"-" # OtterForm(a)),
                 And(a,b)       :"(" # OtterForm(a) # " & " # OtterForm(b) # ")",
                 Or(a,b)        :"(" # OtterForm(a) # " | " # OtterForm(b) # ")",
                 Implies(a,b)   :"(" # OtterForm(a) # " -> " # OtterForm(b) # ")",
                 Equiv(a,b)     :"(" # OtterForm(a) # " <-> " # OtterForm(b) # ")",
                 default        :"")};

/*************************************************************************************************************************
**                                      CONSTRUCTION DE LA LISTE DES VARIABLES D'UNE EXPRESSION                         **
*************************************************************************************************************************/

STR OtterVar(exp e) {let l = OtterVars(e) in
                        ((l == "")? "" : "lex([-20,-19,-18,-17,-16,-15,-14,-13,-12,-11,-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0," #
                                        "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20," # l # "neg(x),sum(x,x)]).")};

STR OtterVars(exp e)
        {with(e)(VarExp(v)      :with(v)(VarSimple(s)   :s # ",",
                                         default        :""),
                 Neg(a)         :OtterVars(a),
                 Sum(a,b)       :OtterVars(a) # OtterVars(b),
                 LessEq(a,b)    :OtterVars(a) # OtterVars(b),
                 Eq(a,b)        :OtterVars(a) # OtterVars(b),
                 Not(a)         :OtterVars(a),
                 And(a,b)       :OtterVars(a) # OtterVars(b),
                 Or(a,b)        :OtterVars(a) # OtterVars(b),
                 Implies(a,b)   :OtterVars(a) # OtterVars(b),
                 Equiv(a,b)     :OtterVars(a) # OtterVars(b),
                 default        :"")};
```

```
guardexp        {synthesized guardexp etendu;};
guardexp        : NoGexp        {guardexp.etendu = NoGexp();}
                | Gexp          {guardexp.etendu = Gexp(expList.etendu,guard.etendu);};

guardexpList    {synthesized guardexpList etendu;};
guardexpList    : GuardexpListNil       {guardexpList.etendu = GuardexpListNil();}
                | GuardexpListPair      {guardexpList$1.etendu = (guardexp.etendu == NoGexp())?
                                                guardexpList$2.etendu  : GuardexpListPair(guardexp.etendu,guardexpList$2.etendu);};

aff     {synthesized aff etendu;};
aff     : Af    {aff.etendu = Af(varList,guardexpList.etendu);};

affList {inherited aff affatom;
         synthesized affList etendu;};
affList : AffListNil    {affList.etendu = AffListNil;}
        | AffListPair   {affList$1.etendu = AffListPair(aff.etendu,affList$2.etendu);
                         affList$2.surete = ""; affList$2.affatom = Af(VarListNil,GuardexpListNil);};

assignList      : AssignListPair        {local aff a; a = Atomisation(affList.etendu);
                                         affList.affatom = affList.valide? a : Af(VarListNil,GuardexpListNil);
                                         local STR s; s = affList.valide?
                                                 Surete(Consrequete({racine.assertions},a))
                                                :ToutesContraintes({racine.assertions});
                                         affList.surete = (s == "")? "":"[ " # s # "]";};

requeteList     : RequeteListNil()
                | RequeteListPair(requete requeteList);

requete : Req(STR INT exp);

STR ToutesContraintes(assertionList l)
        {with(l)(AssertionListPair(AssertionTripl(VarSimple(n),a,b,i),r):n # " " # ToutesContraintes(r),
                 default                                                  : "")};

requeteList Consrequete(assertionList liste, aff ins)
        {let Af(vecteur,gexp) = ins in (with(liste)
                (AssertionListNil       : RequeteListNil,
                 AssertionListPair(a,r) : with(a)
                        (NoAssertion                    : Consrequete(r,ins),
                         AssertionTripl(v,aSimple,b,i)  : with(v)
                                (VarSimple(n)   :
                                        b? with(aSimple)
                                                (Unless(e1,e2)  : RequeteListPair(Req(n,i,ToutesGardes(And(e1,Not(e2)),
                                                                                                Or(e1,e2),vecteur,gexp)),
                                                                        Consrequete(r,ins)),
                                                 Invariant(e)   : RequeteListPair(Req(n,i,ToutesGardes(e,e,vecteur,gexp)),
                                                                        Consrequete(r,ins)),
                                                 Stable(e)      : RequeteListPair(Req(n,i,ToutesGardes(e,e,vecteur,gexp)),
                                                                        Consrequete(r,ins)),
                                                 Constant(e)    : RequeteListPair(Req(n,i,
                                                                      And(ToutesGardes(e,e,vecteur,gexp),
                                                                          ToutesGardes(Not(e),Not(e),vecteur,gexp))),
                                                                        Consrequete(r,ins)))
                                        : RequeteListPair(Req(n,i,NoExp),Consrequete(r,ins)),
                                 NoVar          : RequeteListNil))))};

exp ToutesGardes(exp pre, exp post, varList vecteur, guardexpList gexp)
        {with(gexp)(GuardexpListPair(Gexp(l,Gu(g)),r)   :And(Implies(And(g,pre),Wp(post,vecteur,l)),
                                                             ToutesGardes(pre,post,vecteur,r)),
                    default                             :True)};

/*******************************************************************************************************************
**                                      CONCATENATION DE PLUSIEURS AFFECTATIONS SYNCHRONES                       **
*******************************************************************************************************************/

aff Atomisation(affList l)
        {with(l)(AffListNil                     :Af(VarListNil,GuardexpListPair(Gexp(ExpListNil,Gu(True)),
                                                                                GuardexpListNil)),
                 AffListPair(Af(vl,gl),r)       :let Af(vr,gr) = Atomisation(r)  in
                        (Af(vl @ vr, Atombis(vl,gl,gr) @ Atomter(Var_exp(vl),Toutes_fausses(gl),gr))))};

guardexpList Atombis(varList vl, guardexpList gl, guardexpList gr)
        {with(gl)(GuardexpListPair(Gexp(e,Gu(g)),r)     :Atomter(e,g,gr) @ Atombis(vl,r,gr),
                        default                         :GuardexpListNil)};

guardexpList Atomter(expList e, exp g, guardexpList gr)
        {with(gr)(GuardexpListPair(Gexp(ee,Gu(gg)),r)   :GuardexpListPair(Gexp(e @ ee, Gu(And(g,gg))),Atomter(e,g,r)),
                        default                         :GuardexpListNil)};

expList Var_exp(varList l)
        {with(l)(VarListNil             :ExpListNil,
                 VarListPair(v,r)       :ExpListPair(VarExp(v),Var_exp(r)))};

exp Toutes_fausses(guardexpList gl)
        {with(gl)(GuardexpListPair(Gexp(e,Gu(g)),r)     :And(Not(g),Toutes_fausses(r)),
                        default                         :True)};

/*******************************************************************************************************************
**                                      FONCTION DE CALCUL DE WP PAR SUBSTITUTION                                **
*******************************************************************************************************************/

exp Wp(exp e, varList v, expList l)
        {with(e)(VarExp(w)      : Subst(w,v,l),
                 Neg(a)         : Neg(Wp(a,v,l)),
                 Sum(a,b)       : Sum(Wp(a,v,l),Wp(b,v,l)),
                 Diff(a,b)      : Diff(Wp(a,v,l),Wp(b,v,l)),
                 Less(a,b)      : Less(Wp(a,v,l),Wp(b,v,l)),
                 LessEq(a,b)    : LessEq(Wp(a,v,l),Wp(b,v,l)),
                 Grt(a,b)       : Grt(Wp(a,v,l),Wp(b,v,l)),
                 GrtEq(a,b)     : GrtEq(Wp(a,v,l),Wp(b,v,l)),
                 Eq(a,b)        : Eq(Wp(a,v,l),Wp(b,v,l)),
                 NotEq(a,b)     : NotEq(Wp(a,v,l),Wp(b,v,l)),
                 And(a,b)       : And(Wp(a,v,l),Wp(b,v,l)),
                 Or(a,b)        : Or(Wp(a,v,l),Wp(b,v,l)),
                 Implies(a,b)   : Implies(Wp(a,v,l),Wp(b,v,l)),
                 Equiv(a,b)     : Equiv(Wp(a,v,l),Wp(b,v,l)),
                 Not(a)         : Not(Wp(a,v,l)),
                 default        : e)};
```

```
guardexpList     : GuardexpListNil       {guardexpList.valide = true;}
                 | GuardexpListPair      {guardexpList$1.valide = (guardexp != NoGexp) && guardexp.valide && guardexpList$2.valide;
                                          guardexp.seule = guardexpList$1.seule;
                                          guardexpList$2.seule = guardexpList$1.seule;
                                          guardexp.varlistvalide = guardexpList$1.varlistvalide;
                                          guardexpList$2.varlistvalide = guardexpList$1.varlistvalide;
                                          guardexp.varlistlt = guardexpList$1.varlistlt;
                                          guardexpList$2.varlistlt = guardexpList$1.varlistlt;};

aff     {synthesized BOOL valide;};
aff     : Af    {guardexpList.varlistvalide = varList.valide && (varList != VarListNil);
                 guardexpList.varlistlt = varList.lt;
                 guardexpList.seule = with(guardexpList)(GuardexpListPair(x,GuardexpListNil):true,*:false);
                 aff.valide = (guardexpList != GuardexpListNil) && guardexpList.valide;
                 aff.pbs = (guardexpList.valide || (varList.pbs != ""))?"":" [Types Incompatibles] ";};

affList {synthesized BOOL valide;};
affList : AffListNil    {affList.valide = true;}
        | AffListPair   {affList$1.valide = aff.valide && affList$2.valide;};


theorem : Theo  {theorem.pbs = (exp.t == ERREUR_)?" [Types Incompatibles] "
                                 :(exp.t == NOTYPE_)?" [Variable non Declaree] "
                                         :(exp.t != BOOL_)?" [Expression Non Booleenne] ":"";};

init    : NoInit        {init.pbs = "";}
        | InitAssign    {init.pbs = ((expList != ExpListPair(NoExp,ExpListNil)) &&
                                 varList.valide && (varList.lt != expList.lt))?" [Types Incompatibles] ":"";};

assertionSimple {synthesized BOOL valide;};
assertionSimple : Unless        {assertionSimple.pbs1 = (exp$1.t == ERREUR_)?" [Types Incompatibles] "
                                                     :(exp$1.t == NOTYPE_)?" [Variable non Declaree] "
                                                             :(exp$1.t != BOOL_)?" [Expression Non Booleenne] ":"";
                                 assertionSimple.pbs2 = (exp$2.t == ERREUR_)?" [Types Incompatibles] "
                                                     :(exp$2.t == NOTYPE_)?" [Variable non Declaree] "
                                                             :(exp$2.t != BOOL_)?" [Expression Non Booleenne] ":"";
                                 assertionSimple.valide = ((exp$1.t == BOOL_) && (exp$2.t == BOOL_));}
                | Invariant     {assertionSimple.pbs1 = (exp.t == ERREUR_)?" [Types Incompatibles] "
                                                     :(exp.t == NOTYPE_)?" [Variable non Declaree] "
                                                             :(exp.t != BOOL_)?" [Expression Non Booleenne] ":"";
                                 assertionSimple.pbs2 = "";
                                 assertionSimple.valide = (exp.t == BOOL_);}
                | Stable        {assertionSimple.pbs1 = (exp.t == ERREUR_)?" [Types Incompatibles] "
                                                     :(exp.t == NOTYPE_)?" [Variable non Declaree] "
                                                             :(exp.t != BOOL_)?" [Expression Non Booleenne] ":"";
                                 assertionSimple.pbs2 = "";
                                 assertionSimple.valide = (exp.t == BOOL_);}
                | Constant      {assertionSimple.pbs1 = (exp.t == ERREUR_)?" [Types Incompatibles] "
                                                     :(exp.t == NOTYPE_)?" [Variable non Declaree] "
                                                             :(exp.t != BOOL_)?" [Expression Non Booleenne] ":"";
                                 assertionSimple.pbs2 = "";
                                 assertionSimple.valide = (exp.t == BOOL_);};

program {synthesized typedeclList tliste;};
program : Prog  {program.tliste = declareList.tliste;};

racine  {synthesized TABLE[typedeclList] tds;};
racine  : Rac   {racine.tds = MakeTable(program.tliste,1);};


typedeclList Liste_types(varList l , type t)
        {with(l)(VarListNil      :TypedeclListNil,
                 VarListPair(s,r):with(s)(VarSimple(n):with(t)(Ty(i):TypedeclListPair(Tdecl(n,i),Liste_types(r,t))),
                                          default      :Liste_types(r,t)))};
```

B.1.9. *surete.ssl*

```
/***********************************************************************************************************************************
**                                                 CONTROLE DES CONTRAINTES DE SURETE                                             **
***********************************************************************************************************************************/

/***********************************************************************************************************************************
**      CONSTRUCTION D'UNE COPIE DES SPECIFICATIONS PLUS DIRECTEMENT UTILISABLE (MACRO-DEFINITIONS, EXPRESSIONS VIDES, ...       **
**                                                                                                                                **
**                                                 ENORME ERREUR DE CONCEPTION !!!                                                **
***********************************************************************************************************************************/

assertionSimple {synthesized assertionSimple assertions;};
assertionSimple : Unless        {assertionSimple.assertions = Unless(exp$1.etendu,exp$2.etendu);}
                | Invariant     {assertionSimple.assertions = Invariant(exp.etendu);}
                | Stable        {assertionSimple.assertions = Stable(exp.etendu);}
                | Constant      {assertionSimple.assertions = Constant(exp.etendu);};

assertion       {synthesized assertion assertions;};
assertion       : NoAssertion           {assertion.assertions = NoAssertion;}
                | AssertionTripl        {assertion.assertions = with(var)
        (VarSimple(v):AssertionTripl(var,assertionSimple.assertions,BOOL && assertionSimple.valide,INT),
         default     :AssertionTripl(VarSimple(var.varvide),assertionSimple.assertions,BOOL && assertionSimple.valide,INT));};

assertionList   {synthesized assertionList assertions;};
assertionList   : AssertionListNil      {assertionList.assertions = AssertionListNil;}
                | AssertionListPair     {assertionList$1.assertions = AssertionListPair(assertion.assertions,
                                                                                     assertionList$2.assertions);};

specif  {synthesized assertionList assertions;};
specif  : Spec  {specif.assertions = assertionList.assertions;};

racine  {synthesized assertionList assertions;};
racine  : Rac   {racine.assertions = specif.assertions;};

/***********************************************************************************************************************************
**                                         CONSTRUCTION DES FORMULES A SOUMETTRE AU DEMONSTRATEUR                                 **
***********************************************************************************************************************************/

guard   {synthesized guard etendu;};
guard   : NoGuard       {guard.etendu = Gu(True);}
        | Gu            {guard.etendu = Gu(exp.etendu);};
```

```
assertionList   {synthesized exp invariants;
                 synthesized INT niveau;
                 synthesized BOOL cle;};
assertionList   : AssertionListNil      {assertionList.invariants = True;
                                         assertionList.niveau = 0; assertionList.cle = true;}
                | AssertionListPair     {assertionList$1.invariants = (assertion.invariants == NoExp)?
                                                assertionList$2.invariants
                                              : And(assertion.invariants,assertionList$2.invariants);
                                         assertionList$1.niveau = (assertion.niveau > assertionList$2.niveau)?
                                                assertion.niveau : assertionList$2.niveau;
                                         assertionList$1.cle = assertion.cle && assertionList$2.cle;};

specif  {synthesized exp invariants;
         synthesized INT niveau;
         synthesized BOOL cle;};
specif  : Spec  {specif.invariants = (specif.assertions == AssertionListNil)?assertionList.invariants   /* Evite type 3 circ.   */
                                                                             :assertionList.invariants;
                 specif.niveau = assertionList.niveau; specif.cle = assertionList.cle;};

init    {synthesized exp condinit;};
init    : NoInit        {init.condinit = True;}
        | InitAssign    {init.condinit = EgalitesVarVal(varList,expList);};

exp EgalitesVarVal(varList vl, expList el)
        {with(vl)(VarListNil            : True,
                  VarListPair(v,r)      : with(el)
                        (ExpListNil             : True,
                         ExpListPair(e,rr)      : And(Eq(VarExp(v),e),EgalitesVarVal(r,rr))))};

program {synthesized exp condinit;};
program : Prog  {program.condinit = init.condinit;};

racine  : Rac   {specif.pbs = (specif.cle && Theoreme(Implies(program.condinit,specif.invariants),specif.niveau))?
                                                                                 "":"%n%T[ INVARIANTS NON INITIALISES ]%n%n";};
```

B.1.8. types.ssl

```
/*************************************************************************************************************************************
**                                                          GESTION DU TYPAGE                                                       **
*************************************************************************************************************************************/

# define Ftype(n) with(Lookup({racine.tds},n))(TypedeclListPair(Tdecl(n,i),r):i,\
                                                                                  default           :NOTYPE_)

# define Decl_multiple(n) with(Lookup({racine.tds},n))(TypedeclListPair(x,TypedeclListPair(y,z)):true,\
                                                                                  default                       :false)

list typedeclList;
typedeclList    : TypedeclListNil()
                | TypedeclListPair(typedecl typedeclList);

typedecl        : Tdecl(STR INT);

INTList : INTListNil()
        | INTListPair(INT INTList);

var     {synthesized BOOL decl_mul;};
var     : Novar         {var.decl_mul = false;                  var.t = NOVAR_;}
        | VarSimple     {var.decl_mul = Decl_multiple(IDENT);   var.t = Ftype(IDENT);};

varList {synthesized BOOL decl_mul;
         synthesized BOOL pas_decl;
         synthesized INTList lt;
         synthesized BOOL valide;};
varList : VarListNil    {varList.decl_mul = false; varList.pas_decl = false;
                         varList.lt = INTListNil; varList.valide = true; varList.pbs = "";}
        | VarListPair   {local BOOL pas_decl; pas_decl = (var.t == NOTYPE_) || varList$2.pas_decl;
                         local BOOL decl_mul; decl_mul = var.decl_mul || varList$2.decl_mul;
                         varList$1.decl_mul = decl_mul; varList$1.pas_decl = pas_decl;
                         varList$1.valide = !decl_mul && !pas_decl && (var.t != NOVAR_);
                         varList$1.pbs = decl_mul?" [Variable Multi-Declaree] "
                                              :pas_decl?" [Variable Non Declaree] ":"";
                         varList$1.lt = INTListPair(var.t,varList$2.lt);};

declare {synthesized typedeclList tliste;};
declare : Decl  {declare.tliste = Liste_types(varList,type);
                 declare.pbs = varList.decl_mul ? " [Declaration Multiple] " : "";};

declareList     {synthesized typedeclList tliste;};
declareList     : DeclareListNil        {declareList.tliste = TypedeclListNil;}
                | DeclareListPair       {declareList$1.tliste = declare.tliste @ declareList$2.tliste;};

always  : Alw   {always.pbs = ((var.t == NOTYPE_) || (exp.t == NOTYPE_))?" [Variable Non Declaree] "
                                        :(var.t != exp.t)?" [Types Incompatibles] ":"";};

guard   : NoGuard       {guard.pbs = "";}
        | Gu            {guard.pbs = (exp.t == NOTYPE_)?" [Variable Non Declaree] "
                                        :(exp.t == BOOL_)?"":" [Type Non Booleen] ";};

expList {synthesized INTList lt;};
expList : ExpListNil    {expList.lt = INTListNil;}
        | ExpListPair   {expList$1.lt = INTListPair(exp.t,expList$2.lt);};

guardexp        {inherited BOOL seule;
                 inherited BOOL varlistvalide;
                 inherited INTList varlistlt;
                 synthesized BOOL valide;};
guardexp        : NoGexp        {guardexp.pbs = ""; guardexp.valide = true;}
                | Gexp          {local BOOL pbs; pbs = !guardexp.seule && (guard == NoGuard());
                                 guardexp.pbs = pbs?" [Garde Obligatoire] ":"";
                                 guardexp.valide = (guard.pbs == "") && !pbs && (guard != Gu(NoExp)) &&
                                                guardexp.varlistvalide && (guardexp.varlistlt == expList.lt);};

guardexpList    {synthesized BOOL valide;
                 inherited BOOL seule;
                 inherited BOOL varlistvalide;
                 inherited INTList varlistlt;};
```

B.1.6. *always.ssl*

```
/****************************************************************************************************************************
**                                                MACRO-DEFINITIONS ALWAYS                                                  **
****************************************************************************************************************************/

/****************************************************************************************************************************
**                                                CONSTRUCTION DE L'ENVIRONNEMENT                                           **
****************************************************************************************************************************/

list varvalList;
racine  {synthesized TABLE[varvalList] env;};

varvalList      : VarvalListNil()
                | VarvalListPair(varval varvalList);

varval  : VarvalPair(STR exp);

alwaysList      {synthesized varvalList liste;};
alwaysList      : AlwaysListNil         {alwaysList.liste = VarvalListNil;}
                | AlwaysListPair        {alwaysList$1.liste = (always.pbs == "")
                                                ?with(always)
                                                        (Alw(VarSimple(n),e):VarvalListPair(VarvalPair(n,e),alwaysList$2.liste),
                                                                default     :alwaysList$2.liste)
                                                :alwaysList$2.liste;};

program {synthesized varvalList liste;};
program : Prog  {program.liste = alwaysList.liste;};

racine  : Rac   {racine.env = MakeTable(program.liste,1);};


/****************************************************************************************************************************
**                                                EXTENSION DES EXPRESSIONS                                                 **
****************************************************************************************************************************/

exp Extension(exp e, TABLE[varvalList] env)
        {with(e)(NoExp()        :e,
                 NumConst(n)    :e,
                 BoolConst(b)   :e,
                 VarExp(v)      :let fexp = with(v)(VarSimple(n):with(Lookup(env,n))(VarvalListPair(VarvalPair(n,f),r):f,
                                                                                         default              :NoExp()),
                                                    default     :NoExp()) in
                                ((fexp == NoExp)? e : Extension(fexp,env)),
                 Neg(x)         :Neg(Extension(x,env)),
                 Sum(x,y)       :Sum(Extension(x,env),Extension(y,env)),
                 Diff(x,y)      :Diff(Extension(x,env),Extension(y,env)),
                 Less(x,y)      :Less(Extension(x,env),Extension(y,env)),
                 LessEq(x,y)    :LessEq(Extension(x,env),Extension(y,env)),
                 Grt(x,y)       :Grt(Extension(x,env),Extension(y,env)),
                 GrtEq(x,y)     :GrtEq(Extension(x,env),Extension(y,env)),
                 Eq(x,y)        :Eq(Extension(x,env),Extension(y,env)),
                 NotEq(x,y)     :NotEq(Extension(x,env),Extension(y,env)),
                 Not(x)         :Not(Extension(x,env)),
                 And(x,y)       :And(Extension(x,env),Extension(y,env)),
                 Or(x,y)        :Or(Extension(x,env),Extension(y,env)),
                 Implies(x,y)   :Implies(Extension(x,env),Extension(y,env)),
                 Equiv(x,y)     :Equiv(Extension(x,env),Extension(y,env)))};

exp     {synthesized exp etendu;};
exp     :NoExp          {exp$1.etendu = Extension(exp$1,{racine.env});}
        |NumConst       {exp$1.etendu = Extension(exp$1,{racine.env});}
        |BoolConst      {exp$1.etendu = Extension(exp$1,{racine.env});}
        |VarExp         {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Neg            {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Sum            {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Diff           {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Less           {exp$1.etendu = Extension(exp$1,{racine.env});}
        |LessEq         {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Grt            {exp$1.etendu = Extension(exp$1,{racine.env});}
        |GrtEq          {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Eq             {exp$1.etendu = Extension(exp$1,{racine.env});}
        |NotEq          {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Not            {exp$1.etendu = Extension(exp$1,{racine.env});}
        |And            {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Or             {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Implies        {exp$1.etendu = Extension(exp$1,{racine.env});}
        |Equiv          {exp$1.etendu = Extension(exp$1,{racine.env});};

expList {synthesized expList etendu;};
expList : ExpListNil    {expList.etendu = ExpListNil;}
        | ExpListPair   {expList$1.etendu = ExpListPair(exp.etendu,expList$2.etendu);};
```

B.1.7. *init.ssl*

```
/****************************************************************************************************************************
**                                                CLAUSE INITIALLY ET INVARIANTS                                            **
****************************************************************************************************************************/

assertionSimple {synthesized exp invariants;};
assertionSimple : Invariant     {assertionSimple.invariants = exp.etendu;}
                | Unless        {assertionSimple.invariants = NoExp;}
                | Constant      {assertionSimple.invariants = NoExp;}
                | Stable        {assertionSimple.invariants = NoExp;};

assertion       {synthesized exp invariants;
                 synthesized INT niveau;
                 synthesized BOOL cle;};
assertion       : NoAssertion           {assertion.invariants = NoExp;assertion.niveau = 0;assertion.cle = false;}
                | AssertionTripl        {assertion.invariants = assertionSimple.invariants;
                                         assertion.niveau = with(assertionSimple)
                                                (Invariant(e):INT,
                                                 default     :0);
                                         assertion.cle = with(assertionSimple)
                                                (Invariant(e):BOOL,
                                                 default     :true);};
```

```
Var     ::= ()          {Var.trans = Novar();}
        |   (NOIDENT)   {Var.trans = Novar();}
        |   (IDENT)     {Var.trans = VarSimple(IDENT);};
var ~ Var.trans;

VarList {synthesized varList trans;};
VarList ::= (Var)               {VarList.trans = VarListPair(Var.trans,VarListNil);}
        |   (Var ’,’ VarList)   {VarList$1.trans = VarListPair(Var.trans,VarList$2.trans);};
varList ~ VarList.trans;

ExpList {synthesized expList trans;};
ExpList ::= (Exp)               {ExpList.trans = ExpListPair(Exp.trans,ExpListNil);}
        |   (Exp ’,’ ExpList)   {ExpList$1.trans = ExpListPair(Exp.trans,ExpList$2.trans);};
expList ~ ExpList.trans;

Guard   {synthesized guard trans;};
Guard   ::= ()          {Guard.trans = NoGuard();}
        |   (IF Warning Exp)    {Guard.trans = Gu(Exp.trans);};
guard ~ Guard.trans;

Guardexp        {synthesized guardexp trans;};
Guardexp        ::= (ExpList Guard Warning)     {Guardexp.trans = Gexp(ExpList.trans,Guard.trans);};
guardexp ~ Guardexp.trans;

GuardexpList    {synthesized guardexpList trans;};
GuardexpList    ::= (Guardexp)
                        {GuardexpList.trans = GuardexpListPair(Guardexp.trans,GuardexpListNil);}
                |   (Guardexp ’~’ GuardexpList)
                        {GuardexpList$1.trans = GuardexpListPair(Guardexp.trans,GuardexpList$2.trans);};
guardexpList ~ GuardexpList.trans;

Always  {synthesized always trans;};
Always  ::= (Var Warning ’=’ Exp)       {Always.trans = Alw(Var.trans,Exp.trans);};
always ~ Always.trans;

AlwaysList      {synthesized alwaysList trans;};
AlwaysList      ::= ()                          {AlwaysList.trans = AlwaysListNil;}
                |   (Always)                    {AlwaysList.trans = AlwaysListPair(Always.trans,AlwaysListNil);}
                |   (Always ’,’ AlwaysList)     {AlwaysList$1.trans = AlwaysListPair(Always.trans,AlwaysList$2.trans);};
alwaysList ~ AlwaysList.trans;

Init    {synthesized init trans;};
Init    ::= ()                                  {Init.trans = NoInit;}
        |   (NOIDENT)                           {Init.trans = NoInit;}
        |   (VarList Warning ’=’ ExpList)       {Init.trans = InitAssign(VarList.trans,ExpList.trans);};
init ~ Init.trans;

Aff     {synthesized aff trans;};
Aff     ::= (VarList Warning GETS Warning GuardexpList) {Aff.trans = Af(VarList.trans,GuardexpList.trans);};
aff ~ Aff.trans;

AffList {synthesized affList trans;};
AffList ::= (Aff)                       {AffList.trans = AffListPair(Aff.trans,AffListNil);}
        |   (Aff PARALL AffList)        {AffList$1.trans = AffListPair(Aff.trans,AffList$2.trans);};
affList ~ AffList.trans;

AssignList      {synthesized assignList trans;};
AssignList      ::= ()                                  {AssignList.trans = AssignListNil;}
                |   (AffList)                           {AssignList.trans = AssignListPair(AffList.trans,AssignListNil);}
                |   (AffList DISJONC AssignList)        {AssignList$1.trans = AssignListPair(AffList.trans,AssignList$2.trans);};
assignList ~ AssignList.trans;

Declare {synthesized declare trans;};
Declare ::= (VarList ’:’ Type Warning)  {Declare.trans = Decl(VarList.trans,Type.trans);};
declare ~ Declare.trans;

DeclareList     {synthesized declareList trans;};
DeclareList     ::= ()                          {DeclareList.trans = DeclareListNil;}
                |   (Declare)                   {DeclareList.trans = DeclareListPair(Declare.trans,DeclareListNil);}
                |   (Declare ’;’ DeclareList) {DeclareList$1.trans = DeclareListPair(Declare.trans,DeclareList$2.trans);};
declareList ~ DeclareList.trans;

AssignListBloc  {synthesized assignList trans;};
AssignListBloc  ::= ()                  {AssignListBloc.trans = AssignListNil;}
                |   (ASSIGN AssignList) {AssignListBloc.trans = AssignList.trans;};

InitBloc        {synthesized init trans;};
InitBloc        ::= ()                          {InitBloc.trans = NoInit;}
                |   (INITIALLY Init)            {InitBloc.trans = Init.trans;};

AlwaysListBloc  {synthesized alwaysList trans;};
AlwaysListBloc  ::= ()                  {AlwaysListBloc.trans = AlwaysListNil;}
                |   (ALWAYS AlwaysList) {AlwaysListBloc.trans = AlwaysList.trans;};

DeclareListBloc {synthesized declareList trans;};
DeclareListBloc ::= ()                          {DeclareListBloc.trans = DeclareListNil;}
                |   (DECLARE DeclareList)       {DeclareListBloc.trans = DeclareList.trans;};

Program {synthesized program trans;};
Program ::= ()          {Program.trans = Prog(DeclareListNil,AlwaysListNil,NoInit,AssignListNil);}
        |   (PROGRAM DeclareListBloc AlwaysListBloc InitBloc AssignListBloc)
                        {Program.trans = Prog(DeclareListBloc.trans,AlwaysListBloc.trans,InitBloc.trans,AssignListBloc.trans);};

Racine  {synthesized racine trans;};
Racine  ::= (Program SEPAR Specif)      {Racine.trans = Rac(Program.trans,Specif.trans);}
        |   (Program Specif)            {Racine.trans = Rac(Program.trans,Specif.trans);};
racine ~ Racine.trans;
```

```
on "Not"        Or(a,b):Not(Or(a,b)),
on "Not"        Implies(a,b):Not(Implies(a,b)),
on "Not"        Equiv(a,b):Not(Equiv(a,b)),

on "And"        Or(a,b):And(a,b),
on "Or"         And(a,b):Or(a,b),

on "=>"         Equiv(a,b):Implies(a,b),
on "<=>"        Implies(a,b):Equiv(a,b),

on "!="         Eq(a,b):NotEq(a,b),
on "=="         NotEq(a,b):Eq(a,b);

transform guardexp
on "<EXP> IF <EXP>"     <guardexp>:Gexp(<expList>,<guard>);
```

B.1.4. *lex.ssl*

```
/*****************************************************************************************************************************
**                                                  DEFINITIONS DES UNITES LEXICALES                                        **
*****************************************************************************************************************************/

WARNING:        < \[[a-zA-Z\ _\<\>0-9]+\] >;
TRUE:           < [tT][rR][uU][eE] >;
FALSE:          < [fF][aA][lL][sS][eE] >;
IF:             < "if"|"If"|"IF"|"iF" >;
UNLESS:         < [uU][nN][lL][eE][sS][sS] >;
INITIALLY:      < [iI][nN][iI][tT][iI][aA][lL][lL][yY] >;
ALWAYS:         < [aA][lL][wW][aA][yY][sS] >;
ASSIGN:         < [aA][sS][sS][iI][gG][nN] >;
STABLE:         < [sS][Tt][aA][bB][lL][eE] >;
CONSTANT:       < [cC][oO][nN][sS][tT][aA][nN][tT] >;
INVARIANT:      < [iI][nN][vV][aA][rR][iI][aA][nN][tT] >;
SPECIF:         < [sS][pP][eE][cC][iI][fF][iI][cC][aA][tT][iI][oO][nN][sS]? >;
PROGRAM:        < [pP][rR][oO][gG][rR][aA][mM] >;
BOOLEEN:        < [bB][oO][oO][lL] >;
ENTIER:         < [iI][nN][tT] >;
DECLARE:        < [dD][eE][cC][lL][aA][rR][eE] >;
WITH:           < [wW][iI][tT][hH] >;
IDENT:          < [a-zA-Z][a-zA-Z0-9\_]* >;
NOIDENT:        < \<[A-Z_0-9]+\> >;
INTEGER:        < [0-9]+ >;
NOTEQ:          < "!=" >;
LESSEQ:         < "<=" >;
GRTEQ:          < ">=" >;
WHITESPACE:     < [\ \t\n]* >;
IMPLIES:        < "=>" >;
EQUIV:          < "<=>" >;
GETS:           < ":=" >;
DISJONC:        < "[]" >;
PARALL:         < "||" >;
EQ:             < "==" >;
SEPAR:          < _+ >;
```

B.1.5. *entry.ssl*

```
/*****************************************************************************************************************************
**                                                       ANALYSE SYNTAXIQUE                                                 **
*****************************************************************************************************************************/

Warning ::= ()
        |   (WARNING);

Theorem {synthesized theorem trans;};
Theorem ::= (Warning Exp Warning)       {Theorem.trans = Theo(Exp.trans,false,0);};
theorem ~ Theorem.trans;

TheoremList     {synthesized theoremList trans;};
TheoremList     ::= ()                          {TheoremList.trans = TheoremListNil;}
                |   (NOIDENT)                   {TheoremList.trans = TheoremListNil;}
                |   (Theorem)                   {TheoremList.trans = TheoremListPair(Theorem.trans,TheoremListNil);}
                |   (Theorem ',' TheoremList)   {TheoremList$1.trans = TheoremListPair(Theorem.trans,TheoremList$2.trans);};
theoremList ~ TheoremList.trans;

AssertionSimple {synthesized assertionSimple trans;};
AssertionSimple ::= (Exp Warning UNLESS Exp Warning )   {AssertionSimple.trans = Unless(Exp$1.trans,Exp$2.trans);}
                |   (INVARIANT Exp Warning )            {AssertionSimple.trans = Invariant(Exp.trans);}
                |   (CONSTANT Exp Warning )             {AssertionSimple.trans = Constant(Exp.trans);}
                |   (STABLE Exp Warning )               {AssertionSimple.trans = Stable(Exp.trans);};
assertionSimple ~ AssertionSimple.trans;

Assertion       {synthesized assertion trans;};
Assertion       ::= ()                          {Assertion.trans = NoAssertion;}
                |   (NOIDENT)                   {Assertion.trans = NoAssertion;}
                |   (Var AssertionSimple)       {Assertion.trans = AssertionTripl(Var.trans,AssertionSimple.trans,false,0);};
assertion ~ Assertion.trans;

AssertionList   {synthesized assertionList trans;};
AssertionList ::=(Assertion)                  {AssertionList.trans = AssertionListPair(Assertion.trans,AssertionListNil);}
               | (Assertion ',' AssertionList){AssertionList$1.trans = AssertionListPair(Assertion.trans,AssertionList$2.trans);};
assertionList ~ AssertionList.trans;

Specif  {synthesized specif trans;};
Specif  ::= ()                                                  {Specif.trans = Spec(AssertionListNil,TheoremListNil);}
        |   (SPECIF Warning AssertionList)                      {Specif.trans = Spec(AssertionList.trans,TheoremListNil);}
        |   (Warning WITH TheoremList)                          {Specif.trans = Spec(AssertionListNil,TheoremList.trans);}
        |   (SPECIF Warning AssertionList WITH TheoremList)     {Specif.trans = Spec(AssertionList.trans,TheoremList.trans);};
specif ~ Specif.trans;

Type    {synthesized type trans;};
Type    ::= (NOIDENT)   {Type.trans = Ty(NOTYPE_);}
        |   (ENTIER)    {Type.trans = Ty(INT_);}
        |   (BOOLEEN)   {Type.trans = Ty(BOOL_);};
type ~ Type.trans;
```

```
        |   (Exp '>' Exp)                       {Exp$1.trans = Grt(Exp$2.trans,Exp$3.trans);}
        |   (Exp LESSEQ Exp prec LESSEQ)        {Exp$1.trans = LessEq(Exp$2.trans,Exp$3.trans);}
        |   (Exp GRTEQ Exp prec GRTEQ)          {Exp$1.trans = GrtEq(Exp$2.trans,Exp$3.trans);}
        |   (Exp EQ Exp prec EQ)                {Exp$1.trans = Eq(Exp$2.trans,Exp$3.trans);}
        |   (Exp NOTEQ Exp prec NOTEQ)          {Exp$1.trans = NotEq(Exp$2.trans,Exp$3.trans);}
        |   ('~' Exp)                           {Exp$1.trans = Not(Exp$2.trans);}
        |   (Exp '&' Exp)                       {Exp$1.trans = And(Exp$2.trans,Exp$3.trans);}
        |   (Exp '|' Exp)                       {Exp$1.trans = Or(Exp$2.trans,Exp$3.trans);}
        |   (Exp IMPLIES Exp prec IMPLIES)      {Exp$1.trans = Implies(Exp$2.trans,Exp$3.trans);}
        |   (Exp EQUIV Exp prec EQUIV)          {Exp$1.trans = Equiv(Exp$2.trans,Exp$3.trans);}
        |   ('(' Exp ')')                       {Exp$1.trans = Exp$2.trans;};
exp ~ Exp.trans;

/*****************************************************************************************************************************
**                                                       CONSTRUCTION DU TYPE                                                  **
*****************************************************************************************************************************/

exp     {synthesized INT t;};
var     {synthesized INT t;};

exp     : NoExp         {exp.t = NOTYPE_;}
        | NumConst      {exp.t = INT_;}
        | BoolConst     {exp.t = BOOL_;}
        | VarExp        {exp.t = (var.t == NOVAR_)? NOTYPE_ : var.t;}
        | Neg           {exp$1.t = (exp$2.t == INT_) ? INT_ : (exp$2.t == NOTYPE_) ? NOTYPE_ : ERREUR_;}
        | Sum           {exp$1.t = ((exp$2.t == INT_) && (exp$3.t == INT_))? INT_ :
                                        (((exp$2.t == INT_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == INT_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | Diff          {exp$1.t = ((exp$2.t == INT_) && (exp$3.t == INT_))? INT_ :
                                        (((exp$2.t == INT_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == INT_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | Less          {exp$1.t = ((exp$2.t == INT_) && (exp$3.t == INT_))? BOOL_ :
                                        (((exp$2.t == INT_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == INT_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | LessEq        {exp$1.t = ((exp$2.t == INT_) && (exp$3.t == INT_))? BOOL_ :
                                        (((exp$2.t == INT_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == INT_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | Grt           {exp$1.t = ((exp$2.t == INT_) && (exp$3.t == INT_))? BOOL_ :
                                        (((exp$2.t == INT_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == INT_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | GrtEq         {exp$1.t = ((exp$2.t == INT_) && (exp$3.t == INT_))? BOOL_ :
                                        (((exp$2.t == INT_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == INT_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | Eq            {exp$1.t = ((exp$2.t == NOTYPE_) || (exp$3.t == NOTYPE_))? NOTYPE_:(exp$2.t == exp$3.t)? BOOL_:ERREUR_;}
        | NotEq         {exp$1.t = ((exp$2.t == NOTYPE_) || (exp$3.t == NOTYPE_))? NOTYPE_:(exp$2.t == exp$3.t)? BOOL_:ERREUR_;}
        | Not           {exp$1.t = (exp$2.t == BOOL_) ? BOOL_ : (exp$2.t == NOTYPE_) ? NOTYPE_ : ERREUR_;}
        | And           {exp$1.t = ((exp$2.t == BOOL_) && (exp$3.t == BOOL_))? BOOL_ :
                                        (((exp$2.t == BOOL_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == BOOL_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | Or            {exp$1.t = ((exp$2.t == BOOL_) && (exp$3.t == BOOL_))? BOOL_ :
                                        (((exp$2.t == BOOL_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == BOOL_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | Implies       {exp$1.t = ((exp$2.t == BOOL_) && (exp$3.t == BOOL_))? BOOL_ :
                                        (((exp$2.t == BOOL_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == BOOL_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;}
        | Equiv         {exp$1.t = ((exp$2.t == BOOL_) && (exp$3.t == BOOL_))? BOOL_ :
                                        (((exp$2.t == BOOL_) || (exp$2.t == NOTYPE_)) &&
                                         ((exp$3.t == BOOL_) || (exp$3.t == NOTYPE_)))? NOTYPE_ : ERREUR_;};

/*****************************************************************************************************************************
**                                                         TRANSFORMS                                                          **
*****************************************************************************************************************************/

transform exp
on "Neg"        <exp>:Neg(<exp>),
on "+"          <exp>:Sum(<exp>,<exp>),
on "-"          <exp>:Diff(<exp>,<exp>),
on "<"          <exp>:Less(<exp>,<exp>),
on ">"          <exp>:Grt(<exp>,<exp>),
on "<="         <exp>:LessEq(<exp>,<exp>),
on ">="         <exp>:GrtEq(<exp>,<exp>),
on "=="         <exp>:Eq(<exp>,<exp>),
on "!="         <exp>:NotEq(<exp>,<exp>),
on "And"        <exp>:And(<exp>,<exp>),
on "Or"         <exp>:Or(<exp>,<exp>),
on "=>"         <exp>:Implies(<exp>,<exp>),
on "<=>"        <exp>:Equiv(<exp>,<exp>),
on "Not"        <exp>:Not(<exp>),

on "+"          Diff(a,b):Sum(a,b),
on "-"          Sum(a,b):Diff(a,b),

on "<"          LessEq(a,b):Less(a,b),
on "<"          GrtEq(a,b):Less(a,b),
on "<"          Grt(a,b):Less(a,b),

on "<="         GrtEq(a,b):LessEq(a,b),
on "<="         Grt(a,b):LessEq(a,b),
on "<="         Less(a,b):LessEq(a,b),

on ">="         LessEq(a,b):GrtEq(a,b),
on ">="         Grt(a,b):GrtEq(a,b),
on ">="         Less(a,b):GrtEq(a,b),

on ">"          LessEq(a,b):Grt(a,b),
on ">"          GrtEq(a,b):Grt(a,b),
on ">"          Less(a,b):Grt(a,b),

on "Neg"        Neg(e):e,
on "Neg"        Sum(a,b):Neg(Sum(a,b)),
on "Neg"        Diff(a,b):Neg(Diff(a,b)),

on "Not"        Not(e):e,
on "Not"        Less(a,b):Not(Less(a,b)),
on "Not"        Grt(a,b):Not(Grt(a,b)),
on "Not"        GrtEq(a,b):Not(GrtEq(a,b)),
on "Not"        LessEq(a,b):Not(LessEq(a,b)),
on "Not"        Eq(a,b):NotEq(a,b),
on "Not"        NotEq(a,b):Eq(a,b),
on "Not"        And(a,b):Not(And(a,b)),
```

B.1.2. *affdyn.ssl*

```
/********************************************************************************************************************************
**                                                    FORMAT D'AFFICHAGE DYNAMIQUE                                              **
********************************************************************************************************************************/

type    : Ty    {type.strtype = with(INT)(BOOL_:"BOOL",
                                          INT_ :"INT",
                                            *  :"<TYPE>");};

varList : VarListPair   {var.varvide = "<IDENT>";};

always  : Alw           {var.varvide = "<IDENT>";};

exp     : VarExp        {var.varvide = "<IDENT>";};

assertion       {inherited INT num_unl; inherited INT num_inv; inherited INT num_cst; inherited INT num_sta;};

assertionList   {inherited INT num_unl; inherited INT num_inv; inherited INT num_cst; inherited INT num_sta;};

specif  : Spec  {assertionList.num_unl = 1; assertionList.num_inv = 1; assertionList.num_cst = 1; assertionList.num_sta = 1;};

assertionList   : AssertionListPair     {assertionList$2.num_unl = with(assertion)
                                                (AssertionTripl(v,Unless(x,y),b,i):assertionList$1.num_unl + 1,
                                                 default                           :assertionList$1.num_unl);
                                         assertionList$2.num_inv = with(assertion)
                                                (AssertionTripl(v,Invariant(x),b,i):assertionList$1.num_inv + 1,
                                                 default                            :assertionList$1.num_inv);
                                         assertionList$2.num_sta = with(assertion)
                                                (AssertionTripl(v,Stable(x),b,i):assertionList$1.num_sta + 1,
                                                 default                        :assertionList$1.num_sta);
                                         assertionList$2.num_cst = with(assertion)
                                                (AssertionTripl(v,Constant(x),b,i):assertionList$1.num_cst + 1,
                                                 default                          :assertionList$1.num_cst);
                                         assertion.num_unl = assertionList$1.num_unl;
                                         assertion.num_sta = assertionList$1.num_sta;
                                         assertion.num_inv = assertionList$1.num_inv;
                                         assertion.num_cst = assertionList$1.num_cst;};

assertion       : AssertionTripl        {var.varvide = with(assertionSimple)
                                                (Unless(x,y)    : "<UNL_" # INTtoSTR(assertion.num_unl) # ">",
                                                 Invariant(x)   : "<INV_" # INTtoSTR(assertion.num_inv) # ">",
                                                 Stable(x)      : "<STA_" # INTtoSTR(assertion.num_sta) # ">",
                                                 Constant(x)    : "<CST_" # INTtoSTR(assertion.num_cst) # ">");};
```

B.1.3. *exp.ssl*

```
/********************************************************************************************************************************
**                                                           EXPRESSIONS                                                        **
********************************************************************************************************************************/

# define NOTYPE_ 0
# define BOOL_ 1
# define INT_  2
# define NOVAR_ 8
# define ERREUR_ 9

# define True   BoolConst(true)
# define False  BoolConst(false)

/********************************************************************************************************************************
**                                                SYNTAXES ABSTRAITE ET FORMAT D'AFFICHAGE                                      **
********************************************************************************************************************************/

exp     : NoExp()               [^ ::= "<EXP>"]
        | NumConst(INT)         [^ ::= ^]
        | BoolConst(BOOL)       [^ ::= ^]
        | VarExp(var)           [^ ::= ^]
        | Neg(exp)              [^ ::= " -" @]
        | Sum(exp exp)          [^ ::= "(" @ " + " @ ")"]
        | Diff(exp exp)         [^ ::= "(" @ " - " @ ")"]
        | Less(exp exp)         [^ ::= "(" @ " < " @ ")"]
        | LessEq(exp exp)       [^ ::= "(" @ " <= " @ ")"]
        | Grt(exp exp)          [^ ::= "(" @ " > " @ ")"]
        | GrtEq(exp exp)        [^ ::= "(" @ " >= " @ ")"]
        | Eq(exp exp)           [^ ::= "(" @ " == " @ ")"]
        | NotEq(exp exp)        [^ ::= "(" @ " != " @ ")"]
        | Not(exp)              [^ ::= " ~" @]
        | And(exp exp)          [^ ::= "(" @ " & " @ ")"]
        | Or(exp exp)           [^ ::= "(" @ " | " @ ")"]
        | Implies(exp exp)      [^ ::= "(" @ " => " @ ")"]
        | Equiv(exp exp)        [^ ::= "(" @ " <=> " @ ")"];

/********************************************************************************************************************************
**                                                SYNTAXE CONCRETE ET ANALYSE SYNTAXIQUE                                        **
********************************************************************************************************************************/

right IMPLIES EQUIV;
left '|';
left '&';
nonassoc EQ NOTEQ;
left '~';
nonassoc '<' LESSEQ '>' GRTEQ;
left '+' '-';

Exp     {synthesized exp trans;};
Var     {synthesized var trans;};
Exp     ::= ()                          {Exp.trans = NoExp;}
        |   (NOIDENT)                   {Exp.trans = NoExp();}
        |   (INTEGER)                   {Exp.trans = NumConst(STRtoINT(INTEGER));}
        |   (TRUE)                      {Exp.trans = True;}
        |   (FALSE)                     {Exp.trans = False;}
        |   (Var)                       {Exp.trans = VarExp(Var.trans);}
        |   ('-' Exp)                   {Exp$1.trans = Neg(Exp$2.trans);}
        |   (Exp '+' Exp)               {Exp$1.trans = Sum(Exp$2.trans,Exp$3.trans);}
        |   (Exp '-' Exp)               {Exp$1.trans = Diff(Exp$2.trans,Exp$3.trans);}
        |   (Exp '<' Exp)               {Exp$1.trans = Less(Exp$2.trans,Exp$3.trans);}
```

B. Fichiers sources

B.1. Programmes SSL

B.1.1. root.ssl

```
/******************************************************************************************************************************
**                                  STRUCTURE ABSTRAITE ET FORMAT D'AFFICHAGE DU TEXTE EDITE                                 **
******************************************************************************************************************************/

root racine;
racine           : Rac(program specif)                                        [@ ::= "PROGRAM%n" ^ "%n%n%R(_)%n%nSPECIFICATION%n" ^ ];

program          : Prog(declareList alwaysList init assignList)              [^ ::= "DECLARE%n" @ "%nALWAYS%n" @ "%nINITIALLY%n" @
                                                                                     "%nASSIGN%n     " @ ];

specif  {inherited STR pbs;};
specif           : Spec(assertionList theoremList)                            [^ ::= "%F" specif.pbs @ "%nWITH%n" @ ];

optional list declareList;
declareList      : DeclareListNil()
                 | DeclareListPair(declare declareList)                       [^ ::= ^ [";%n"] @ ];

optional list alwaysList;
alwaysList       : AlwaysListNil()
                 | AlwaysListPair(always alwaysList)                          [^ ::= ^ [",%n"] @ ];

optional list assignList;
assignList       : AssignListNil()
                 | AssignListPair(affList assignList)                         [^ ::= "%t%l  " @ "%b" ["%n[]  "] @ ];

optional list assertionList;
assertionList    : AssertionListNil()
                 | AssertionListPair(assertion assertionList)                 [^ ::= ^ [",%n"] @ ];

optional list theoremList;
theoremList      : TheoremListNil()
                 | TheoremListPair(theorem theoremList)                       [^ ::= ^ [",%n"] @ ];

list varList;
varList {synthesized STR pbs;};
varList          : VarListNil()
                 | VarListPair(var varList)                                   [@ ::= ^ [","] @];

optional init;
init    {synthesized STR pbs;};
init             : NoInit()
                 | InitAssign(varList expList)                                [@ ::= @ varList.pbs init.pbs " = " @];

declare {synthesized STR pbs;};
declare          : Decl(varList type)                                         [^ ::= @ " : " @ declare.pbs];

type    {synthesized STR strtype;};
type             : Ty(INT)                                                    [@ ::= type.strtype ..];

always  {synthesized STR pbs;};
always           : Alw(var exp)                                               [^ ::= @ always.pbs " = " @];

list affList;
affList {inherited STR surete;};
affList          : AffListNil()
                 | AffListPair(aff affList)                                   [^ ::= ^ ["%n||  "] @ "%T" affList$1.surete ];

aff     {synthesized STR pbs;};
aff              : Af(varList guardexpList)                                   [^ ::= @ varList.pbs " := " aff.pbs "%L" @ "%b"];

list guardexpList;
guardexpList     : GuardexpListNil()
                 | GuardexpListPair(guardexp guardexpList)                    [^ ::= ^ [" ~%n"] @];

guardexp         {synthesized STR pbs;};
guardexp         : NoGexp()                                                   [^ ::= " <EXP IF EXP> "]
                 | Gexp(expList guard)                                        [^ ::= @ @ guardexp.pbs ];

optional guard;
guard   {synthesized STR pbs;};
guard            : NoGuard()
                 | Gu(exp)                                                    [^ ::= "%TIF " guard.pbs @];

assertion        : NoAssertion()                                              [^ ::= "<ASSERTION>"]
                 | AssertionTripl(var assertionSimple BOOL INT)               [^ ::= @ "%T " @ .. ..];

assertionSimple {synthesized STR pbs1;
                 synthesized STR pbs2;};
assertionSimple : Unless(exp exp)                                             [@ ::= @ assertionSimple.pbs1 " UNLESS "
                                                                                   @ assertionSimple.pbs2]
                 | Invariant(exp)                                             [@ ::= "INVARIANT " @ assertionSimple.pbs1]
                 | Stable(exp)                                                [@ ::= "STABLE    " @ assertionSimple.pbs1]
                 | Constant(exp)                                              [@ ::= "CONSTANT  " @ assertionSimple.pbs1];

theorem {synthesized STR pbs;
         synthesized STR faux;};
theorem          : Theo(exp BOOL INT)                                         [^ ::= theorem.pbs @ "%T" theorem.faux .. ..];

var     {inherited STR varvide;};
var              : Novar()                                                    [^ ::= var.varvide]
                 | VarSimple(IDENT)                                           [^ ::= ^];

list expList;
expList          : ExpListNil()
                 | ExpListPair(exp expList)                                   [@ ::= ^ [","] @];
```

A.2. Algorithme de Peterson - Etat actif

Cette figure représente le même programme que la figure précédente après activation des contraintes et du théorème. Tous les messages d'avertissement ont disparu.

```
UNITY.SYN:mutex
File   Edit   View   Options   Structure   Text   Help
Read prgms/mutex

PROGRAM
DECLARE
UE,UB,UT,UH,VE,VB,VT,VH,TURN  :  BOOL;
Inv1,Inv2,Exclusion  :  BOOL
ALWAYS
Inv1  =  (UE  =>  (UB  &  (  ~VB  |   ~TURN))),
Inv2  =  (VE  =>  (VB  &  (  ~UB  |  TURN))),
Exclusion  =   ~(UE  &  VE)
INITIALLY
UE,UH,VE,VH,UT,UB,VT,VB  =  false,false,false,false,true,false,true,false
ASSIGN
      UE,UT,UB  :=  false,true,false       IF  UE
[]    UH,UT  :=  true,false        IF  UT
[]    VT,VE,VB  :=  true,false,false       IF  VE
[]    VH,VT  :=  true,false        IF  VT
[]    UB,TURN  :=  true,true       IF  (UH  &   ~UB)
[]    UE  :=  true         IF  (UB  &   ~(VB  &  TURN))
[]    VB,TURN  :=  true,false      IF  (VH  &   ~VB)
[]    VE  :=  true         IF  (VB  &   ~(UB  &   ~TURN))

SPECIFICATION
u_inv    INVARIANT Inv1,
v_inv    INVARIANT Inv2
WITH
((Inv1  &  Inv2)  =>  Exclusion)

Context: assertionList   Actif   Stable   Constant   Unless
```

A. Exemples de fenêtres éditeur

A.1. Algorithme de Peterson - Etat passif

La figure suivante visualise un exemple de fenêtre engendrée par l'éditeur de l'environnement. Le programme édité représente l'algorithme d'exclusion mutuelle de **Peterson** pour deux processus et sa propriété fondamentale de sûreté (Cf § III.8). Les deux spécifications et le théorème auxiliaire sont dans l'état *passif*. Aucun contrôle n'est donc effectué. Les spécifications sont non vérifiées et le théorème n'est pas prouvé.

```
UNITY.SYN:mutex
File  Edit  View  Options  Structure  Text  Help
Read prgms/mutex

PROGRAM
DECLARE
UE,UB,UT,UH,VE,VB,VT,VH,TURN : BOOL;
Inv1,Inv2,Exclusion : BOOL
ALWAYS
Inv1 = (UE => (UB & ( ~VB |  ~TURN))),
Inv2 = (VE => (VB & ( ~UB | TURN))),
Exclusion =  ~(UE & VE)
INITIALLY
UE,UH,VE,VH,UT,UB,VT,VB = false,false,false,false,true,false,true,false
ASSIGN
      UE,UT,UB := false,true,false      IF UE   [ u_inv v_inv ]
[]    UH,UT := true,false       IF UT   [ u_inv v_inv ]
[]    VT,VE,VB := true,false,false      IF VE   [ u_inv v_inv ]
[]    VH,VT := true,false       IF VT   [ u_inv v_inv ]
[]    UB,TURN := true,true      IF (UH &  ~UB)  [ u_inv v_inv ]
[]    UE := true        IF (UB &  ~(VB & TURN)) [ u_inv v_inv ]
[]    VB,TURN := true,false     IF (VH &  ~VB)  [ u_inv v_inv ]
[]    VE := true        IF (VB &  ~(UB &  ~TURN))       [ u_inv v_inv ]

SPECIFICATION

        [ INVARIANTS NON INITIALISES ]

u_inv    INVARIANT Inv1,
v_inv    INVARIANT Inv2
WITH
((Inv1 & Inv2) => Exclusion)    [ NON PROUVE ]

Context: assertionList  Passif  Stable  Constant  Unless
```

- prouver automatiquement et *localement* que les transformations introduites par le raffinement maintiennent la cohérence entre le programme et sa spécification. Pour cela, il faut se restreindre à un format particulier de raffinements pour lesquels on connait d'avance les conditions locales à vérifier pour que le raffinement soit valide.

L'avenir du projet peut donc se développer dans deux directions :

- permettre au sein de l'environnement une application (conditionnelle ou inconditionnelle) d'un jeu de raffinements donnés.
- déterminer des raffinements génériques, utiles à la répartition, et applicables sous des conditions facilement vérifiables.

en compte les relations *entre* les instructions du programme, qui sont à l'origine de toute sa puissance.

Il est donc nécessaire que le *principe* de l'algorithme étudié soit présent, d'une manière ou d'une autre, au niveau des spécifications. C'est la raison pour laquelle des propriétés, *intuitivement* vraies pour qui a une vue globale du programme, ne sont *formellement* valides qu'accompagnées d'invariants complexes, le rôle de ces invariants étant de résumer en eux le principe général de l'algorithme.

Par exemple, dans le cas de l'algorithme de **Mattern**, la propriété $VR \leq r_1 + r_2$ est vraie dans tous les états du programme. Par contre, cette propriété n'est pas un invariant puisque l'instruction

$$V_1, VE, VR \;:=\; true, (VE + e_1), (VR + r_1) \quad if\ \neg V_1$$

déplace le programme de l'état

$$[e_1 = 1, r_1 = 1, e_2 = 0, r_2 = 0, VE = 1, VR = 1, V_1 = false, \cdots]$$

à l'état

$$[e_1 = 1, r_1 = 1, e_2 = 0, r_2 = 0, VE = 2, VR = 2, V_1 = true, \cdots]$$

qui ne vérifie pas la propriété. Le fait que l'état de départ ne sera effectivement jamais atteint n'est perceptible qu'à l'aide d'une connaissance globale du programme.

En fait, la confrontation entre un véritable invariant et une instruction correspond formellement à un prédicat qui doit être valide dans tous les états du programme *y compris ceux qui sont inaccessibles*. C'est ce qui permet le type de preuve envisagée, le démonstrateur n'ayant aucune connaissance a priori des états accessibles du programme.

La difficulté est donc d'associer aux propriétés qu'on souhaite vérifier un invariant représentant le fonctionnement global du programme, cet invariant étant d'autant plus difficile à déterminer que le programme est complexe.

VII.3. Perspectives

La version présentée précédemment de l'algorithme de **Mattern** provient d'une suite de raffinements appliqués à un algorithme simple et centralisé pour lequel les spécifications sont simples à démontrer [FMP93]. Ce n'est que lorsque l'algorithme devient réparti que ses propriétés deviennent délicates à établir.

Une méthode envisageable pour contourner les difficultés précédentes peut donc être de procéder aux vérifications du programme en *deux* étapes :

- preuve directe d'un programme simple et de ses spécifications
- application simultanément au programme et aux spécifications de raffinements vers une version plus intéressante de l'algorithme

La preuve du programme simple peut être effectuée à l'aide des méthodes présentées par l'environnement réalisé. En ce qui concerne les raffinements, deux approches sont possibles :

- prouver leur validité par ailleurs, de manière générique et indépendamment du programme auquel ils sont appliqués. Cette preuve peut être réalisée de manière automatique ou non.

aux valeurs réelles (**e1+e2** et **r1+r2**), à savoir que les deux inégalités suivantes sont vraies à tout instant de l'exécution du programme :

$$VR \leq r1 + r2 \qquad \text{et} \qquad VE \leq e1 + e2$$

Bien qu'elles soient constamment vérifiées par le programme, ces deux propriétés *ne sont pas des invariants*, mais sont impliquées par les invariants **Mat_Vision_r** et **Mat_Vision_e**.

Une grande difficulté dans la preuve de ces deux propriétés consiste donc à déterminer les invariants qui les impliquent, tâche qui reste entièrement à la charge de l'utilisateur et pour laquelle on ne dispose pas de stratégie générale.

Par exemple, les deux propriétés les plus intéressantes de l'algorithme précédent sont :

- si **TERMINE** devient vrai, il le restera.
- **TERMINE** détecte bien la terminaison (aspect sûreté).

Exprimées dans le formalisme `UNITY`, ces propriétés deviennent :

- il existe un prédicat I_1 tel que :
  - `Invariant` I_1
  - `Stable` ($I_1 \wedge$ **TERMINE**)
- il existe un prédicat I_2 tel que :
  - `Invariant` I_2
  - $I_2 \wedge$ **TERMINE** $\Rightarrow$ **e1+e2 = r1+r2**

La détermination des prédicats I_1 et I_2 est à ce point délicate qu'il ne nous a pas été possible de les découvrir. La preuve des deux spécifications importantes de l'algorithme reste donc hors de portée du programme, quelle que soit la puissance du démonstrateur automatique.

VII.2.3. Explication intuitive

Les difficultés rencontrées lors de la preuve de l'algorithme de **Mattern** sont tout à fait compréhensibles lorsqu'on examine le processus utilisé avec un peu de recul.

Ce processus est extrêmement simple puisqu'il consiste en la confrontation de chaque spécification avec chaque instruction, le résultat de cette confrontation étant une formule logique représentant la condition pour que l'instruction vérifie la spécification. Cette approche est rendue possible par la propriété qu'ont les spécifications `UNITY` de considérer les instructions *indépendamment les unes des autres.*

Or, la puissance d'un programme n'est pas la somme des puissances de ses instructions. Un programme est d'autant plus complexe que ses instructions interragissent. A spécifications fixées, la preuve mise en œuvre est d'une complexité *proportionnelle* au nombre d'instructions du programme. Elle est donc *la même* quelle que soit la complexité d'un programme de n instructions.

D'autre part, tout le travail concernant la preuve est effectué par le démonstrateur automatique qui fonctionne sans contexte et qui donc ***n'a pas connaissance du programme dont il doit prouver la correction.*** A chaque requête, il n'a connaissance que *d'une* instruction (sémantiquement définie par le calcul de wp), et n'a donc aucun moyen de prendre

VII.2. Le cas de l'algorithme de Mattern

VII.2.1. Présentation

A côté des difficultés purement techniques liées à la démonstration automatique des formules, apparaissent également des limitations propres à l'approche choisie. Celles-ci ont été particulièrement mises en évidence lors de l'étude de l'algorithme de **Mattern**.

Cet algorithme est un algorithme de détection de la terminaison basé sur l'utilisation de compteurs permettant de s'assurer que tous les messages émis ont été reçus (algorithme dit *"des quatre compteurs"*) [Ray92].

Il est présenté ci-dessous sous une forme raffinée et optimisée, accompagné de quelques spécifications élémentaires.

VII.2.2. Algorithme et spécifications

```
PROGRAM
DECLARE
VE,VR_PRED,VR    : INT;
V1,V2            : BOOL;
TERMINE          : BOOL;
e1,e2,r1,r2      : INT;
k                : INT

INITIALLY
VE,VR_PRED,VR,V1,V2,e1,e2,r1,r2 = 0,0,0,false,false,1,1,0,0

ASSIGN
      TERMINE,VR_PRED,V1,V2,VR,VE := (VR_PRED == VE),VR,false,false,0,0 IF (V1 & V2)
[]    V1,VE,VR := true,(VE + e1),(VR + r1)      IF  ~V1
[]    V2,VE,VR := true,(VE + e2),(VR + r2)      IF  ~V2

[]    e1 := (e1 + 1)    IF ((e1 + e2) != (r1 + r2))
[]    r1 := (r1 + 1)    IF ((r1 + r2) < (e1 + e2))

[]    e2 := (e2 + 1)    IF ((e1 + e2) != (r1 + r2))
[]    r2 := (r2 + 1)    IF ((r1 + r2) < (e1 + e2))

--------------------------------------------------------------------------------------------

SPECIFICATION

Appl_Temps     INVARIANT ((r1 + r2) <= (e1 + e2)),
Appl_Croit_e   STABLE    (((e1 + e2) >= k) & (k >= 1)),
Appl_Croit_r   STABLE    (((r1 + r2) >= k) & (k >= 0)),
Appl_Arret     STABLE    ((e1 + e2) == (r1 + r2)),

Mat_Vision_r   INVARIANT ((((((r1 >= 0) & (r2 >= 0)) & ((V1 | V2) | (VR == 0))) &
                (V1 | (V2 => (VR <= r2)))) & (V2 | (V1 => (VR <= r1)))) & (VR <= (r1 + r2))),

Mat_Vision_e   INVARIANT ((((((e1 >= 0) & (e2 >= 0)) & ((V1 | V2) | (VE == 0))) &
                (V1 | (V2 => (VE <= e2)))) & (V2 | (V1 => (VE <= e1)))) & (VE <= (e1 + e2)))
```

Les trois premières instructions concernent le comptage des messages par un système de double vague, et les quatre dernières simulent le comportement de deux processus.

Les deux invariants **Mat_Vision_r** et **Mat_Vision_e** spécifient une propriété importante des valeurs des compteurs observées et calculées de manière distribuée (**VE** et **VR**) par rapport

suivante :

- répéter
  - lire une requête
  - formatter la requête
  - soumettre la requête à `Otter`
  - transmettre la réponse de `Otter`
- indéfiniment

La lecture de la requête et son formattage sont effectués par le programme *Shell* **TALON_OTTER**. Celui-ci produit un fichier de commandes `Otter` contenant le problème. Ce fichier est transmis à `Otter` (commande **otter**) et le résultat produit est analysé par la commande **grep** qui détermine le nombre de preuves effectuées (1 pour une formule démontrée, 0 sinon). C'est ce nombre qui est renvoyé vers l'éditeur.

Le programme **TALON_OTTER** produit les options et axiomes nécessaires à `Otter` et y incorpore la formule admise, la formule à démontrer et la commande **lex** qui sont lues sur le *pipe* (Cf § V.2.12).

L'ensemble des optimisations sont effectuées au niveau des fichiers **theorem.ssl** (§ V.2.10) et **simplify.ssl** (§ V.2.11). Les deux programmes *Shell* sont en annexe B.3.

VII. Conclusion et perspectives

VII.1. Inaptitude de `Otter`

Limitées à un langage d'instructions simple tel que `UNITY`, la traduction des contraintes et la gestion de l'éditeur n'ont pas posé de difficultés particulières. La restriction des spécifications aux seules propriétés de sûreté permettait, de plus, de se limiter à l'utilisation de la logique classique pour laquelle les résultats et les outils abondent.

Pourtant, les difficultés les plus importantes survenues dans la réalisation du projet sont dues à l'inaptitude de `Otter` à démontrer les formules construites. Sa faiblesse provient de son incapacité à gérer efficacement l'arithmétique symbolique qui apparaît nécessairement lors de la traduction des contraintes. Cette inaptitude à l'arithmétique est malheureusement commune à la plupart des démonstrateurs automatiques auxquels nous avons eu accès.

Une part importante du travail pendant ce projet a donc consisté à pallier les faiblesses de `Otter` en arithmétique : choix d'axiomes et introduction des opérateurs **sum** et **lei**, étude d'heuristiques de simplification préalable des formules, restriction de la preuve (activation et désactivation de spécifications). L'utilisation d'un démonstrateur plus adapté (s'il en existe) en remplacement de `Otter` permettrait d'augmenter notablement la puissance du système et donc d'envisager des extensions du langage de programmation (quantification et tableaux) et du langage de spécification (propriétés de vivacité).

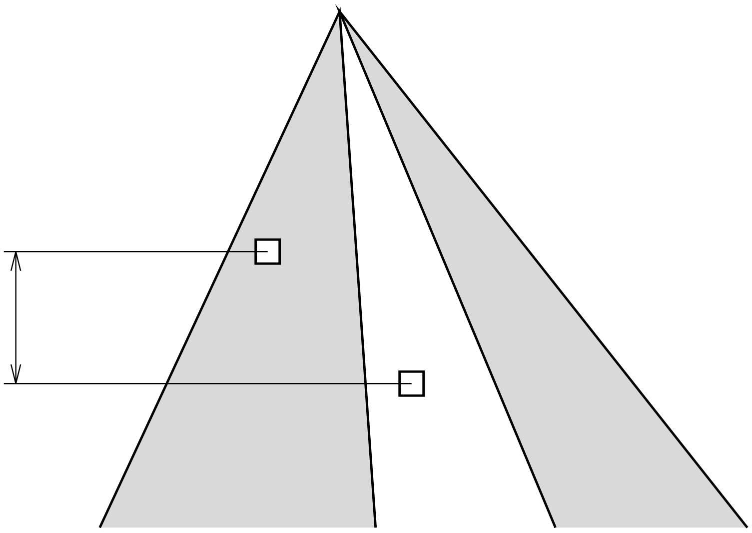

FIG. 1 - . Restriction de l'arbre de recherche

Dans le cas présent, la taille de l'arbre est telle qu'une différence de profondeur de **un** suffit pour passer d'un problème immédiatement résolu en un problème impossible à résoudre en un temps raisonnable.

La difficulté, non encore résolue, est de déterminer au niveau de l'éditeur quelle stratégie sera la meilleure pour la démonstration d'une formule donnée. Actuellement, les deux méthodes sont utilisées successivement (l'une est utilisée en cas d'echec de l'autre). Une utilisation en parallèle serait préférable.

VI.2.4. Résultats

L'utilisation du pré-simplificateur a permis de réduire de manière notable le *nombre* d'appels à `Otter`. Pour la démonstration de l'algorithme de **Peterson** (Cf § III.8), ce nombre est passé de 36 à 7, et pour les premières étapes de la démonstration de l'algorithme de **Mattern** (Cf § VII.2), de 386 à 91.

Le pré-simplificateur influe également sur la *taille* des formules transmises à `Otter`. Ces deux effets combinés ont permis des gains en temps d'un facteur 4 pour l'algotithme de **Peterson** et 2 pour l'algorithme de **Mattern**.

D'autre part, la décomposition en sous-problèmes a permis la démonstration d'une formule quadruple ($A \Rightarrow B \wedge C \wedge D \wedge E$) de manière instantanée. Cette même formule n'était pas démontrée en une heure avant décomposition.

En ce qui concerne la restriction de l'ensemble support, sur les 91 démonstrations nécessaires aux premières spécifications de l'algorithme de **Mattern**, 79 ont été résolues directement et 12 ont nécessité la transformation $\models P \Rightarrow Q \;\; \rightarrow \;\; P \models Q$.

VI.3. Mise en œuvre

Le programme *Shell* **DEMONTRER** est utilisé pour déterminer la validité d'une formule. Il fonctionne comme un serveur de démonstrations dont l'éditeur est le client. Client et serveur sont directement reliés par *pipes* et **DEMONTRER** possède la structure de boucle infinie

est inconsistant (ie non satisfaisable).

Le **principe de résolution** est une règle d'inférence qui génère une nouvelle clause *conséquence logique* de clauses antécédentes. On peut montrer la complétude *pour la réfutation* de ce principe : si un ensemble de clauses est inconsistant, alors la **clause vide** (ie clause toujours fausse) peut être dérivée de cet ensemble à l'aide du principe de résolution. Si toutes les nouvelles clauses sont générées "en largeur d'abord", la stratégie devient *directement* complète, c'est à dire qu'à partir d'un ensemble inconsistant de clauses, la **clause vide** est obtenue *en un nombre fini* de résolutions.

Remarque : La complétude pour la réfutation de principe de résolution ne contredit pas l'indécidabilité de la théorie des prédicats du premier ordre puisque la garantie d'arrêt de l'algorithme nécessite l'hypothèse de l'inconsistance de l'ensemble de clauses considéré. La méthode permet de démontrer toute formule valide, mais ne permet pas de décider de la validité d'une formule.

Stratégie de l'ensemble support : Diverses stratégies de manipulation des clauses ont été étudiées pour l'amélioration de l'algorithme précédent. Leur principe est de réduire le nombre de clauses générées tout en maintenant la complétude du système.

Parmi ces stratégies figure celle de **l'ensemble support** utilisée par `Otter`. Elle consiste à partitionner les clauses initiales en *deux* parties. L'une d'elles constitue l'ensemble support. Le principe de résolution est alors limité à la génération de clauses possédant au moins un ancêtre (non nécessairement un parent) dans cet ensemble.

On montre alors que si l'autre moitié de la partition constitue un ensemble *consistant* de clauses, la stratégie reste complète pour la réfutation. Pilotée en largeur, elle est bien entendu directement complète.

La méthode utilisée ici pour déterminer une partition binaire de l'ensemble de clauses vérifiant la contrainte de consistance d'une des deux parties est de séparer les axiomes (partie supposée consistante) et la négation de la conclusion (devenant l'ensemble support).

Restriction de l'ensemble support : Les formules à démontrer issues du calcul de *wp* sont toutes de forme implicative. Le problème est alors la démonstration de :

$$A \quad \models \quad P \Rightarrow Q$$

où A représente les axiomes utilisés. L'ensemble support est alors le résultat de la mise sous forme clausale de $\neg(P \Rightarrow Q)$. Remplacer le problème précédent par le problème équivalent suivant :

$$A, P \quad \models \quad Q$$

permet de réduire l'ensemble support aux clauses issues de $\neg Q$.

Cette restriction de l'ensemble support permet de réduire la taille de l'arbre de recherche en terme de nombre de clauses générées. Elle a permis de résoudre en quelques secondes certains problèmes qui n'étaient pas résolus par `Otter` en plus d'une heure. Par contre, le phénomène inverse a été constaté pour d'autres problèmes : la restriction de l'arbre de recherche fait perdre la clause vide précédemment utilisée, *mais* la complétude garantit la présence d'une autre clause vide dans la partie restante de l'arbre, malheureusement à un niveau de profondeur plus grand (Fig. 1).

Les formules proposées sont alors plus réduites et sont facilement démontrables par `Otter`. De plus, parmi les n problèmes, $n-1$ sont triviaux et certains seront résolus par le pré-simplificateur seul.

Cas des problèmes conjonctifs : La taille importante des formules à démontrer peut également avoir pour origine la présence de nombreuses expressions gardées au sein d'une même instruction. La formule générée est alors une conjonction d'implications, une pour chaque garde. Démontrer *successivement* chacune des implications permet de réduire la taille des formules manipulées.

Par contre, il convient de se poser la question de l'équivalence entre les deux approches. En effet, si mathématiquement on a bien :

$$\left.\begin{array}{l}\models P\\ \models Q\end{array}\right\} \equiv \models P \wedge Q$$

ce n'est pas nécessairement le cas au niveau des possibilités du démonstrateur automatique. Par exemple, tel qu'il est configuré, `Otter` est incapable de prouver $\exists x, (x + x = 0)$, mais il peut démontrer $\exists x, (x = 0) \wedge (x + x = 0)$.

Dans le cadre du projet, les formules générées ne sont jamais existenciellement quantifiées et le problème précédent ne se présente pas : `Otter` n'utilise jamais une conjonction d'une formule pour la preuve d'une autre conjonction de la même formule.

On peut remarquer que l'utilisation de la technique de décomposition précédente revient à considérer une instruction du type :

$$\begin{array}{lll} v := & e_1 & if \; g_1 \\ & \cdots & if \; \cdots \\ & e_n & if \; g_n \end{array}$$

sous la forme :

$$\begin{array}{lllll} & v & := & e_1 & if \; g_1 \\ [] & v & := & e_2 & if \; g_2 \\ [] & v & := & \cdots & if \; \cdots \\ [] & v & := & e_n & if \; g_n \end{array}$$

L'équivalence entre les deux représentations est liée au déterminisme des instructions `UNITY` (dû à la contrainte d'égalité entre des valeurs dépendant de gardes simultanément vraies). Notamment, la transformation inverse n'est valide que dans le cas où le groupe d'instructions en exclusion mutuelle constitue un programme *déterministe*.

VI.2.3. Réduction de l'arbre de recherche

Principe de résolution et preuves par réfutation : le système `Otter` est basé sur le principe de **résolution** et construit des preuves par **réfutation** [FG90]. Pour ce type de démonstrateurs, un problème

$$Axiomes \stackrel{?}{\models} Conclusion$$

est traduit sous forme clausale. Le démonstrateur cherche alors à établir que l'ensemble de clauses

$$Cl(Axiomes) \;\cup\; Cl(\neg Conclusion)$$

L'objectif des règles suivantes est donc de supprimer les termes apparaissant à la fois à gauche et à droite d'une implication.

$$\begin{array}{lcl} x \wedge y \Rightarrow x & \rightarrow & true \\ x \wedge y \Rightarrow x \vee z & \rightarrow & true \\ x \wedge y \Rightarrow x \wedge z & \rightarrow & x \wedge y \Rightarrow z \end{array}$$

Invariants implicatifs: une autre manière d'établir la propriété Q sur un programme est de montrer que $P \Rightarrow Q$ et P sont des invariants de ce programme. Si l'affectation considérée ne concerne que des variables à droite (resp. à gauche) de l'implication, la formule à démontrer est de la forme: $(P \Rightarrow Q) \Rightarrow (P \Rightarrow R)$ (resp. $(P \Rightarrow Q) \Rightarrow (R \Rightarrow Q)$) et équivaut à $P \Rightarrow (Q \Rightarrow R)$ (resp. $\neg Q \Rightarrow (R \Rightarrow P)$).

Une telle simplification se généralise au cas des affectations gardées par

$$((P \Rightarrow Q) \wedge G) \Rightarrow (P \Rightarrow R) \quad \equiv \quad (P \wedge G) \Rightarrow (Q \Rightarrow R)$$

et

$$((P \Rightarrow Q) \wedge G) \Rightarrow (R \Rightarrow Q) \quad \equiv \quad (\neg Q \wedge G) \Rightarrow (R \Rightarrow P)$$

Les transformations suivantes sont donc utilisées:

$$\begin{array}{lcl} (x \Rightarrow y) \Rightarrow (x \Rightarrow z) & \rightarrow & x \Rightarrow (y \Rightarrow z) \\ (x \Rightarrow y) \Rightarrow (z \Rightarrow y) & \rightarrow & \neg y \Rightarrow (z \Rightarrow x) \\ ((x \Rightarrow y) \wedge g) \Rightarrow (x \Rightarrow z) & \rightarrow & (x \wedge g) \Rightarrow (y \Rightarrow z) \\ ((x \Rightarrow y) \wedge g) \Rightarrow (z \Rightarrow y) & \rightarrow & (\neg y \wedge g) \Rightarrow (z \Rightarrow x) \end{array}$$

Faiblesses de `Otter`: A cause de la gestion qu'il fait de l'égalité, `Otter` est incapable de certaines déductions évidentes, comme l'équivalence entre $\models (a = true)$ et $\models a$. Les deux règles suivantes ont donc été ajoutées pour pallier ces difficultés:

$$\begin{array}{lcl} x = true & \rightarrow & x \\ x = false & \rightarrow & \neg x \end{array}$$

VI.2.2. Décomposition en sous-problèmes

Cas des problèmes implicatifs: l'utilisation d'invariants conjonctifs conduit à démontrer des formules qui restent très imposantes malgré les simplifications précédentes.

Pour une affectation particulière ne modifiant les variables que d'une *partie* de l'invariant, la formule à démontrer est de la forme:

$$P_1 \wedge P_2 \wedge \cdots \wedge P_n \Rightarrow P_1 \wedge P_2 \wedge \cdots \wedge P_{i-1} \wedge Q_i \wedge P_{i+1} \wedge \cdots \wedge P_n$$

Une transformation de ce problème est effectuée dans le but de réduire la taille des formules. Elle consiste à démontrer *successivement*:

$$\begin{array}{ccc} P_1 \wedge P_2 \wedge \cdots \wedge P_n & \Rightarrow & P_1 \\ P_1 \wedge P_2 \wedge \cdots \wedge P_n & \Rightarrow & P_2 \\ \cdots & \Rightarrow & \cdots \\ P_1 \wedge P_2 \wedge \cdots \wedge P_n & \Rightarrow & P_{i-1} \\ P_1 \wedge P_2 \wedge \cdots \wedge P_n & \Rightarrow & Q_i \\ P_1 \wedge P_2 \wedge \cdots \wedge P_n & \Rightarrow & P_{i+1} \\ \cdots & \Rightarrow & \cdots \\ P_1 \wedge P_2 \wedge \cdots \wedge P_n & \Rightarrow & P_n \end{array}$$

Une fonction SSL est donc chargée, préalablement à l'appel de `Otter`, d'un certain nombre de manipulations *élémentaires* inspirées par la façon dont sont générées les formules. Ces simplifications sont de différents types :

Constantes booléennes : les constantes *true* et *false* sont utilisées comme valeurs de fin de récursivité dans les fonctions SSL. Par exemple, la construction d'une conjonction de formules se terminera par $\cdots \wedge true$.

Pour cette raison, et parce que de telles constantes seront également introduites par les autres schémas de simplification, des règles comme celles qui suivent permettent une diminution non négligeable de la taille des formules.

$$\begin{array}{ll} x \wedge false \rightarrow false & x \vee true \rightarrow true \\ x \wedge true \rightarrow x & x \vee false \rightarrow x \\ x \Rightarrow true \rightarrow true & false \Rightarrow x \rightarrow true \\ true \Rightarrow x \rightarrow x & x \Rightarrow false \rightarrow \neg x \\ \neg true \rightarrow false & \neg false \rightarrow true \end{array}$$

Une utilisation récursive de ces règles permet de propager l'apparition de nouvelles constantes.

Constantes arithmétiques : les additions, soustractions et relations d'ordre entre constantes numériques peuvent être réduites par l'arithmétique interne du `Cornell Synthesizer Generator`.

Identité entre termes : un même sous-arbre peut être utilisé plusieurs fois lors de la construction d'une formule. Il est alors intéressant d'éliminer certaines occurrences de ces termes (voire toutes ...) avant qu'ils ne soient décomposés en clauses. De telles simplifications sont possibles au niveau des opérateurs relationnels comme au niveau des opérateurs logiques :

$$\begin{array}{llll} x = x \rightarrow true & x \wedge x \rightarrow x & x \wedge \neg x \rightarrow false & x \Rightarrow x \rightarrow true \\ x \leq x \rightarrow true & x \vee x \rightarrow x & x \vee \neg x \rightarrow true & x \Rightarrow \neg x \rightarrow false \end{array}$$

Invariants conjonctifs : pour montrer qu'une propriété Q est vraie à tout instant de l'exécution d'un programme, on peut être amené à montrer que $P \wedge Q$ est un invariant de ce programme. La formule à démontrer est alors :

$$P \wedge Q \Rightarrow wp\,(P \wedge Q)$$

L'ajoût de Q peut être très important en taille, mais les variables qui y interviennent peuvent n'être modifiées que par un nombre restreint d'affectations. Pour les autres instructions, la formule à démontrer devient alors :

$$P \wedge Q \Rightarrow wp\,(P) \wedge Q$$

éventuellement $P \wedge Q' \wedge R \Rightarrow wp\,(P \wedge R) \wedge Q'$ dans le cas où une partie R de Q est modifiée par l'affectation.

D'une manière générale, ces formules sont alors de la forme :

$$P \wedge Q \Rightarrow P \wedge R$$

ce qui équivaut à

$$P \wedge Q \Rightarrow R$$

Cet axiome implique un type de démonstration **constructif** (la propriété sur un terme est établie par la propriété sur une "partition" de ce terme). Il ne permet pas la preuve d'une relation du type :

$$a \leq 2 \Rightarrow a \leq 5$$

En effet, la connaissance $2 \leq 5$ est dans l'arithmétique interne de `Otter` mais ne figure pas *explicitement* dans la base de connaissances utilisable pour démontrer la relation. Si un problème se réduit à montrer que $2 \leq 5$, il est résolu puisque `Otter` "sait" que $2 \leq 5$. Mais une démonstration ne peut faire intervenir $2 \leq 5$ comme *étape* puisqu'il ne s'agit pas d'un élément de l'ensemble des faits connus (la présence d'un terme symbolique *unique* interdit l'utilisation de l'axiome).

Il est donc nécessaire d'ajouter le second axiome :

$$lei(x, i) \wedge i \leq j \Rightarrow lei(x, j)$$

où x est un terme quelconque et i, j des valeurs numériques.

L'introduction des relations strictes $<$ et $>$ peut se faire de plusieurs manières.

$$x < y \ \equiv \ x + 1 \leq y$$

présente l'avantage de lier la relation d'ordre et la somme. Le problème de l'exemple précédent (Cf § VI.1.1) :

$$a < b \Rightarrow a + 1 \leq b$$

devient alors tautologique, ainsi que la relation très utilisée

$$a + 1 > a$$

En cela, la définition choisie est préférable à une définition du type

$$x < y \ \equiv \ x \leq y \wedge x \neq y$$

ou

$$x < y \ \equiv \ \neg(y \leq x)$$

Par contre, la définition choisie ne permet pas de démontrer $\neg(y \leq x) \Rightarrow x < y$ qui doit être ajouté à la base d'axiomes ($x < y \Rightarrow \neg(y \leq x)$ est démontrable).

La représentation précédente de l'arithmétique permet *théoriquement* (en s'appuyant sur la complétude de la stratégie utilisée par `Otter`) la démonstration par réfutation de toutes les formules générées à partir du calcul de wp dans le cas des relations d'ordre. Dans la pratique, cette complétude n'est assurée que par un parcours en largeur de l'arbre de recherche. La multiplication des clauses rend ce parcours très couteux et limite la profondeur des démonstrations accessibles en un temps raisonnable.

C'est pourquoi diverses optimisations ont dû être étudiées. Elles concernent l'utilisation d'un pré-simplificateur, la décomposition en sous-problèmes et la restriction de l'arbre de recherche.

VI.2. Optimisations

VI.2.1. Pré-simplificateur

Les formules à démontrer, issues du calcul de wp, sont générées à partir d'un processus connu du programmeur. Une manipulation de ces formules au niveau de l'éditeur qui les génère permet une prise en compte de leur format particulier plus efficacement qu'au niveau de `Otter`.

Il faut donc générer, au niveau du `Cornell Synthesizer Generator` et pour chaque appel à `Otter`, une commande **lex** faisant intervenir tous les identificateurs et les nombres de la formule. Actuellement, les variables `UNITY` sont présentées au niveau de `Otter` comme des constantes et leurs noms sont directement inclus dans la commande **lex** générée. Par contre, si on considère ces variables comme universellement quantifiées, il faudra générer une commande **lex** contenant autant de constantes de **Skolem** (de la forme \$c1, \$c2, ...) qu'il y a de variables distinctes dans la formule à démontrer.

En ce qui concerne les valeurs numériques, aucune solution acceptable n'a pu être trouvée. Actuellement, les constantes numériques comprises entre -20 et +20 sont explicitement introduites dans la commande **lex.** L'apparition dans une expression d'une constante hors de cet intervalle risque de provoquer l'échec de la démonstration.

Remarque : Une constante de **Skolem** \$ci ne sera admise par `Otter` au sein d'une commande **lex** que si elle appararaît effectivement au cours d'une skolemisation. Il n'est donc pas possible d'utiliser une commande **lex** générale contenant suffisamment de constantes pour toutes les formules (dont il faudrait alors majorer le nombre d'identificateurs distincts).

Par contre, une constante peut apparaître plusieurs fois dans une même commande **lex.** Seule la dernière occurrence est prise en compte. Ceci facilite la génération automatique de la commande en permettant de considérer *toutes* les variables de la formule sans devoir se restreindre aux variables *distinctes* (Cf § V.2.10).

VI.1.3. Relations d'ordre

La “procédure” précédente de simplification et mise sous forme normale des termes arithmétiques permet une représentation simple des relations d'ordre. On considère ces relations sous l'unique forme :

$$lei(x_1 + x_2 + \cdots + x_n, i)$$

représentant

$$x_1 + x_2 + \cdots + x_n \leq i$$

où $x_1, x_2, \cdots, x_n$ sont des variables ou des négations de variables et i est une *constante* numérique.

La relation $t_1 \leq t_2$ devient $t_1 - i_1 - (t_2 - i_2) \leq i_2 - i_1$ où i_1 (resp. i_2) est la somme des valeurs numériques du terme t_1 (resp. t_2).

$t_1 \geq t_2$ est vu comme $-t_1 \leq -t_2$.

Ainsi, la justification de la relation $a + b \leq b + 3 + a$ est obtenue par simplification et ne nécessite aucune preuve. En effet, elle devient $a + b - b - a \leq 3$ soit $0 \leq 3$ ce qui est connu par l'arithmétique de `Otter`.

Par contre, la preuve de

$$a \geq b \quad \wedge \quad c + d \geq 1 \quad \Rightarrow \quad a - b \geq 1 - c - d$$

nécessite l'introduction de l'axiome :

$$lei(x, i) \wedge lei(y, j) \Rightarrow lei(x + y, i + j)$$

utilisé sous la forme

$$lei(b - a, 0) \wedge lei(-c - d, -1) \Rightarrow lei(b - a - c - d, -1)$$

vérifie la propriété

$$Stable \quad a \leq b$$

La vérification de cette contrainte se fait par la preuve de la formule

$$a < b \quad (\wedge \quad a \leq b) \quad \Rightarrow \quad a + 1 \leq b$$

Ceci suppose donc un minimum de connaissances arithmétiques de la part du démonstrateur utilisé.

Malheureusement, les démonstrateurs logiques généraux du type de `Otter` ne possèdent aucunes notions intrinsèques d'addition ou de relations d'ordre. `Otter` ne manipule ces fonctions qu'au sein de constantes numériques et ne dispose d'aucune arithmétique *symbolique* interne.

Il a donc été nécessaire de lui fournir une base d'axiomes lui permettant d'aborder les problèmes liés à la gestion des compteurs. Celle-ci se limite dans un premier temps aux opérations d'addition et de soustraction, et aux relations d'ordre larges et strictes.

VI.1.2. Addition et soustraction

La soustraction est systématiquement traduite en termes d'addition et de négation. La gestion de l'opération d'addition, que ce soit en preuve par réfutation ou en réécriture, pose un certain nombres de problèmes liés à l'associativité et à la commutativité de la fonction. L'insertion brutale d'une connaissance du type

$$x + y \equiv y + x$$

conduit à une règle délicate à orienter en réécriture et à une explosion combinatoire (en terme de nombre de clauses générées) dans les systèmes de réfutation. Il est donc nécessaire de gérér les multiples formes d'une même expression arithmétique de la manière la plus procédurale possible.

`Otter` offre une possibilité de manipuler par des règles de réécriture les termes d'une loi associative et commutative. Une règle du type

$$x + (y + z) \equiv y + (x + z)$$

associée à un ordre lexical sur les constantes manipulées, permet de gérer à la fois la commutativité et l'associativité. Les clauses ne font alors plus intervenir que des expressions arithmétiques *sous forme normale.* De plus, la commande **special_unary** de `Otter` permet d'ignorer une fonction particulière dans le calcul de l'ordre des termes. Appliquée à la fonction d'opposé neg, elle permet que a et $neg(a)$ soient côte à côte au sein de la forme normale d'un terme où ils apparaissent simultanément. Une règle

$$x + neg(x) \rightarrow 0$$

permet alors la simplification du terme.

L'orientation dynamique des règles en fonction des termes considérés est conditionnée par la connaissance d'un ordre sur les constantes manipulées (constantes de **Skolem** et constantes utilisateur). Un tel ordre est imposé par l'utilisation de la commande **lex.** Malheureusement, bien que la nature précise de cet ordre soit sans importance (l'ordre alphabétique sur les noms des constantes par exemple conviendrait), la présence dans une commande **lex** de *toutes* les constantes rencontrées (y compris les constantes numériques) est absolument nécessaire au bon fonctionnement du mécanisme précédemment décrit.

Theoreme avant tout appel à `Otter`. Ce pré-simplificateur applique récursivement les équivalences décrites au § VI.2.1 jusqu'à ce qu'elles soient sans effet.

V.2.12. otter.c

interface avec `Otter`

Cette fonction est utilisée pour transmettre à l'extérieur de l'éditeur les chaînes de caractères correspondant aux formules et aux commandes destinées à `Otter`. Le langage *SSL* ne disposant d'aucune instruction d'entrée/sortie, la fonction est écrite en langage *C*. Elle se contente d'afficher ses trois chaînes paramètres sur la sortie standard et de lire la réponse de `Otter` sous forme d'un caractère sur l'entrée standard. Ces entrée et sortie standards seront connectées au démonstrateur par des *pipes* au niveau du fichier **unity.c** (Cf § V.2.13).

L'utilisation de la fonction **fflush** est nécessaire au bon fonctionnement des pipes. La réponse de `Otter` est constituée d'un caractère suivi d'un saut de ligne. Le deuxième appel à la fonction **getchar** a pour rôle d'absorber ce saut de ligne.

Les fonctions **Str**, **str_to_str0**, **str0_to_str** et **StrValue** sont issues de la programmation même du `Cornell Synthesizer Generator` et sont utilisées pour la conversion entre la représentation *SSL* des chaînes de caractères et leur représentation *C*.

Remarque : l'affichage des chaînes paramètres sur **stderr** peut permettre, lors de mises au point, de contrôler visuellement les formules soumises au démonstrateur.

V.2.13. unity.c

lien entre l'éditeur et le démonstrateur automatique

Ce programme correspond à la commande **unity** utilisée pour la mise en route de l'environnement. Il se charge d'ouvrir deux *pipes* (l'un de l'éditeur vers le démonstrateur pour les requêtes, et l'autre du démonstrateur vers l'éditeur pour les réponses), et de les associer aux entrées et sorties standard de deux processus fils, l'un étant l'éditeur et l'autre le démonstrateur.

Le programme **DEMONTRER** a pour structure une boucle *infinie* (Cf § VI.3). Pour cette raison, ce processus ne possède pas de fin normale. Le processus père attend donc la terminaison du processus éditeur (*wait*) pour provoquer lui-même celle du processus démonstrateur (*kill*).

VI. Configuration du démonstrateur automatique

VI.1. `Otter` et l'arithmétique

VI.1.1. Introduction

La description d'algorithmes en `UNITY` fait appel fréquemment à la notion de compteurs, représentés par des variables entières. Ces compteurs, mis à jour par les affectations, sont à la base de la plupart des gardes. Typiquement, une instruction est exécutable si la valeur d'un compteur se situe dans un intervalle donné. L'affectation `UNITY`

$$a := a + 1 \quad if \quad a < b$$

La fonction **OtterVar** génère une commande particulière de `Otter` (commande **lex**) sous forme de chaîne de caractères. Cette commande est nécessaire au bon fonctionnement du démonstrateur automatique, mais dépend de la connaissance des noms des variables utilisées par le programme `UNITY`, et ne peut donc être insérée statiquement à la configuration de `Otter` (Cf § VI.1.2).

Doivent apparaître au niveau de cette commande, les opérateurs utilisés par `Otter` et définis par l'utilisateur (*neg* et *sum*), les variables du programme `UNITY` (générées par la fonction **OtterVars**), et les constantes numériques suceptibles d'apparaître dans les formules. Ces constantes numériques peuvent provenir directement du programme `UNITY` ou être issues de la composition ($+$ et $-$) des nombres apparaissant dans le programme édité. Il est donc difficile de déterminer dans quel intervalle elles sont situées. L'intervalle $[-20, +20]$ est utilisé, mais la limite admise par `Otter` (en ce qui concerne le nombre de termes apparaissant dans une commande **lex**) n'a pas été déterminée.

Remarque : une même variable peut apparaître *plusieurs* fois au sein d'une commande **lex** sans nuire au bon fonctionnement de `Otter`. Ceci permet une écriture très simple de la fonction **OtterVars** qui génère les noms des variables *avec* répétition.

La fonction **Theoreme** indique, par sa valeur de retour, si son expression paramètre est démontrée par le démonstrateur automatique (le paramètre entier n'est pas utilisé). Pour cela, elle fait appel au pré-simplificateur **Simplify** et soumet la formule obtenue à `Otter`, si elle n'est pas déjà réduite à **True** ou à **False**. Les problèmes de la forme $P \Rightarrow Q \wedge R$ ou $P \wedge Q$ sont décomposés à ce niveau (Cf § VI.2.2), et ceux de la forme $P \Rightarrow Q$ sont proposés à `Otter` sous *deux* aspects : montrer $P \Rightarrow Q$ en ne sachant rien, puis, en cas d'échec, montrer Q sachant P (Cf § VI.2.3). La formule transmise à `Otter` est la négation ($\neg$) de la formule d'origine (`Otter` fonctionnant par réfutation), et est suivie d'un point (.) pour des raisons de syntaxe.

Theoreme utilise la fonction externe (*foreign*) **Otter** (écrite en langage *C*) dont les trois paramètres sont respectivement la formule admise, la commande **lex**, et la formule à démontrer (sous forme de chaînes de caractères). C'est à ce niveau qu'est compensée l'erreur stratégique liée à la copie des spécifications (Cf § V.2.9). Le `Cornell Synthesizer Generator` permet (par option à la compilation du langage *SSL*) d'associer un *cache* à chaque fonction. Dans ce cache sont mémorisées, automatiquement à chaque appel effectif, les valeurs des paramètres et la valeur de retour de la fonction correspondant à ces paramètres. Si, lors d'un appel, les paramètres coïncident avec ceux d'une des valeurs du cache, cette valeur est renvoyée et l'appel effectif n'a pas lieu. Ce procédé permet donc d'éviter la démonstration effective (par `Otter`) d'une formule déjà démontrée précédemment, à condition qu'elle se trouve encore dans la mémoire cache dont la taille peut être choisie lors de *l'exécution* du programme (option *-c taille*).

Les fonctions externes possédant éventuellement des effets de bord, le `Cornell Synthesizer Generator` ne leur applique pas la technique de cache. C'est pourquoi la fonction **OtterSSL** est utilisée comme intermédiaire entre **Theoreme** et **Otter**. C'est à **OtterSSL** que le cache sera associé.

V.2.11. simplify.ssl

pré-simplificateur de formules

Cette fonction constitue un pré-simplificateur utilisé systématiquement par la fonction

où v et e sont des vecteurs, la formule logique construite est :

$$\bigwedge_{i=1}^{n} [(précondition \wedge g_i) \;\Rightarrow\; wp\,(postcondition,\ v := e_i)]$$

La fonction **Consrequete** utilise des valeurs différentes de précondition et de postcondition selon le type de la spécification, conformément à sa définition en `UNITY` :

- pour P `Unless` Q, précondition : $P \wedge \neg Q$ et postcondition : $P \vee Q$.
- pour `Stable` P et `Invariant` P, précondition : P et postcondition : P.
- pour `Constant` P, **ToutesGardes** est utilisée *deux* fois : précondition-1 : P et postcondition-1 : P, puis précondition-2 : $\neg P$ et postcondition-2 : $\neg P$. La contrainte finale est la conjonction des valeurs de retour de ces deux appels.

La fonction **Surete** propose ensuite au démonstrateur les expressions de la liste des requêtes (via la fonction **Theoreme**), et construit la liste des noms des requêtes non démontrées.

Le calcul de wp lié à une affectation vectorielle élémentaire (c'est à dire $vecteur_1 \; := \; vecteur_2$, inconditionnellement) est réalisé par la fonction **Wp**. Il consiste en la substitution *simultanée* des variables apparaissant dans l'expression logique par la valeur qui leur correspond dans le vecteur d'expressions affectées. Si la variable ne fait par partie du vecteur de variables de l'affectation, elle n'est pas substituée.

Le principe utilisé consiste en une descente récursive dans l'expression logique jusqu'au niveau des variables, puis une recherche de ces variables dans le vecteur affecté. Les vecteurs de variables et d'expressions sont parcourus en parallèle afin d'obtenir, si la variable cherchée est présente, la valeur qui lui correspond.

L'approche inverse, qui constituerait à substituer *successsivement* les variables affectées, n'est pas valable. En effet, certaines substitutions auraient alors lieu récursivement, ce qui n'est pas souhaité. Par exemple, pour l'expression $x - y$ et l'affectation $x, y := y + 1, y - 1$, la méthode utilisée fournit la substitution correcte $y + 1 - (y - 1)\ (= 2)$, alors que la substitution de x puis de y fournit $(y + 1) - y$ puis $(y - 1) + 1 - (y - 1)\ (= 1)$.

Le fichier se termine par les transformations (Cf § III.6.3) sur les spécifications, et la possibilité d'activation et de désactivation globales de toutes les contraintes.

V.2.10. theorem.ssl

théorèmes auxiliaires et interface avec le démonstrateur automatique

Ce fichier gère l'apparition des messages relatifs aux théorèmes auxiliaires. Chaque théorème valide (au sens du contrôle de type) est soumis au démonstrateur (via la fonction **Theoreme**) et le message [*NON PROUVE*] est généré en cas d'echec à la démonstration.

La fonction **OtterForm** traduit la formule à démontrer sous fome de chaîne de caractères, conformément à la syntaxe utilisée par `Otter` (parenthésage complet, utilisation des symboles = (=), & ($\wedge$), | ($\vee$), –> ($\Rightarrow$), <–> ($\Leftrightarrow$), – ($\neg$), \$T (*true*), \$F (*false*), *lei* ($\leq$), *sum* (+), *neg* (– unaire)). Elle suppose la formule issue de la fonction **Simplify**, et ne gère pas la traduction de certains nœuds de *exp* qui ne peuvent plus apparaître (– binaire, >, $\geq$, <).

à une valeur. **surete** est affecté à une chaîne vide et **affatom** à une affectation nulle. Ces valeurs sont sans influence.

La transformation de plusieurs affectations synchrones sous forme d'une unique instruction atomique est rendue délicate par le fait que ces différentes composantes de la même affectation peuvent être gardées différemment (Cf § III.3.4). Le rôle de la fonction **Atomisation** est donc de transformer une affectation de la forme :

$$
\begin{array}{llllll}
 & x, y & := & a_1, b_1 & if & g_1 \\
 & & \sim & a_2, b_2 & if & g_2 \\
\| & z & := & c_1 & if & h_1 \\
 & & \sim & c_2 & if & h_2
\end{array}
$$

en son équivalent :

$$
\begin{array}{lllll}
x, y, z & := & a_1, b_1, c_1 & if & g_1 \wedge h_1 \\
 & \sim & a_1, b_1, c_2 & if & g_1 \wedge h_2 \\
 & \sim & a_1, b_1, z & if & g_1 \wedge \neg h_1 \wedge \neg h_2 \\
 & \sim & a_2, b_2, c_1 & if & g_2 \wedge h_1 \\
 & \sim & a_2, b_2, c_2 & if & g_2 \wedge h_2 \\
 & \sim & a_2, b_2, z & if & g_2 \wedge \neg h_1 \wedge \neg h_2 \\
 & \sim & x, y, c_1 & if & \neg g_1 \wedge \neg g_2 \wedge h_1 \\
 & \sim & x, y, c_2 & if & \neg g_1 \wedge \neg g_2 \wedge h_2 \\
 & \sim & x, y, z & if & \neg g_1 \wedge \neg g_2 \wedge \neg h_1 \wedge \neg h_2
\end{array}
$$

dont le traitement par la suite est classique. D'une manière générale, n affectations synchrones, chacune étant gardée p_i fois, produisent une affectation gardée $\prod_{i=1}^{n}(p_i + 1)$ fois.

A la suite de ces étapes préliminaires commence la vérification des spécifications proprement dite. La fonction **Consrequete** construit, à partir de la seconde forme des spécifications et d'une affectation, une liste de requêtes pour le démonstrateur automatique. Cette fonction n'est utilisée que si l'instruction considérée est valide (au sens du contrôle de type). Dans le cas contraire, aucune vérification n'est entreprise, et les spécifications non vérifiées sont l'ensemble des spécifications, construit à l'aide de la fonction **ToutesContraintes** et mémorisé dans l'attribut **surete**.

Les requête générées ont la forme de triplets (nœud **requete**). Chaque triplet contient la formule logique à démontrer, le nom de la spécification dont elle est issue, et une valeur entière utilisable pour paramétrer le démonstrateur automatique. Le paramètre entier n'est pas utilisé actuellement (Cf § V.2.1).

La fonction **Consrequete** utilise la fonction **ToutesGardes** qui génère une contrainte à partir d'une affectation, d'une précondition et d'une postcondition. **ToutesGardes** produit donc la formule :

$$
\textit{précondition} \Rightarrow wp\,(\textit{postcondition},\ \textit{instruction})
$$

selon le principe exposé au § V.1.1. Plus précisément, pour une instruction de la forme :

$$
\begin{array}{lllll}
v & := & e_1 & if & g_1 \\
 & & \cdots & if & \cdots \\
 & & e_n & if & g_n
\end{array}
$$

Pour cela, une *copie* des spécifications est effectuée dans l'attribut **assertions**. Chaque spécification subit, avant sa copie, les transformations suivantes :

- si l'étiquette associée à la spécification est indéfinie (**Novar**), une étiquette est ajoutée, identique à la représentation à l'affichage de l'étiquette indéfinie. Toutes les spécifications de la copie possèdent donc une étiquette de type **VarSimple**, éventuellement de la forme **Sta_5, Inv_2**, ...
- les expressions booléennes paramètres de la spécification sont étendues. La copie des spécifications ne fait donc plus intervenir de variables macro-définies.
- la clé d'activation de la spécification devient la conjonction d'elle-même et de la validité (au sens du contrôle de type) de la formule. Ainsi, les spécifications incomplètes ou mal typées seront considérées comme passives et ne seront pas démontrées.

La copie des spécifications est mémorisée dans l'attribut **assertions** de **racine** pour pouvoir être accédée à distance sous forme de *remote attribute set* (Cf § V.2.6).

La réalisation de cette copie des spécifications constitue en fait une énorme erreur de conception. En effet, *l'ensemble* des spécifications se retrouvent groupées au sein d'un *unique* attribut. La modification d'une seule spécification entraine la modification de cet attribut. Or, toute la tentative de démonstration repose sur cette copie. L'algorithme d'évaluation incrémentale des attributs redéclanche donc la *totalité* des calculs associés à la preuve, en particulier *tous* les appels au démonstrateur automatique. Du point de vue du `Cornell Synthesizer Generator`, la modification *d'une* spécification influe sur *toutes* les spécifications, et aucune étape de la preuve précédente n'est a priori réutilisable.

Le problème engendré par le regroupement des spécifications sera traîté au niveau des appels au démonstrateur automatique. En effet, le `Cornell Synthesizer Generator` offre une possibilité de généralisation aux fonctions de l'approche incrémentale du calcul des attributs, et utilise pour cela la notion de cache de fonction (Cf § V.2.10).

Une deuxième étape de la traduction s'attache à la préparation des instructions. Le regroupement précédent permet à chaque instruction d'avoir connaissance de l'ensemble des spécifications, et aucun regroupement des instructions n'est nécessaire. Les transformations subies par les instructions sont les suivantes :

- extension des expressions affectées et des expressions des gardes, c'est à dire suppression de toutes les variables macro-définies.
- remplacement d'une garde absente par une garde toujours vraie.
- concaténation de plusieurs affectations synchrones (séparées par $\|$) en une unique affectation. Cette transformation est nécessaire parce que plusieurs affectations synchrones constituent une *unique* instruction atomique qui doit intervenir globalement dans le calcul de *wp*. Elle est réalisée par la fonction **Atomisation** décrite plus loin, et le résultat est mémorisé dans l'attribut **affatom**.

  Remarque : seule la *première* **affList** (c'est à dire la fille d'un nœud **assign**) est concernée par les attributs **surete** et **affatom**. Les autres **affList** ne sont que des parties de cette première liste et n'ont pas à être considérées comme des instructions à part entière. Le `Cornell Synthesizer Generator` impose cependant que leurs attributs soient affectés

par la fonction macro-définie **Decl_multiple** qui détecte les variables apparaissant au moins *deux* fois dans la table.

Les contrôles effectués au niveau d'une liste de variables concernent :

- la définition complète de la liste : toutes les variables de la liste sont définies (absence de <*Ident*>).
- la déclaration des variables : toutes les variables de la liste sont déclarées.
- la déclaration multiple : aucune variable de la liste n'est déclarée plusieurs fois.

Les expressions affectées héritent de l'attribut **seule** qui précise si plusieurs vecteurs d'expressions sont affectés en fonctions de gardes différentes, ce qui n'est connu qu'au niveau de l'affectation. Si cette liste d'expressions n'est pas unique, elle *doit* être gardée (Cf § III.3.4), et le message [*Garde Obligatoire*] est généré si la garde est absente. De plus, un contrôle local à la garde vérifie que l'expression testée est booléenne.

Le contrôle de la cohérence entre les types des variables et les types des expressions lors d'une affectation vectorielle (initialisations et instructions du programme) s'opère par le biais des attributs **lt** et **varlistlt**. Les attributs **lt** sont synthétisés au niveau du vecteur de variables et du vecteur d'expressions, et contiennent la liste ordonnée de leurs types. La valeur de l'attribut **lt** du vecteur de variables est ensuite héritée au niveau du vecteur d'expressions dans **varlistlt**. A ce niveau, il est comparé au vecteur de types de la liste d'expressions et un message d'erreur est généré en cas de non concordance. Cette non concordance peut provenir de types différents en vis-à-vis, ou de la différence de longueur des deux listes (plus d'expressions que de variables ou le contraire).

Lors d'affectations non vectorielles (macro-définitions), le type de la variable est simplement comparé au type de l'expression.

En ce qui concerne les spécifications, les contrôles vérifient que toutes les variables sont définies, déclarées une fois et une seule, et que les expressions manipulées ont un type et que ce type est booléen.

V.2.9. surete.ssl

contrôle des contraintes de sûreté

Les équations de ce fichier réalisent la traduction en formules logiques des contraintes de la spécification vis-à-vis d'une affectation. Les étiquettes correspondant aux formules non établies par le démonstrateur automatique seront ensuite concaténées en un message qui sera affiché à la suite de l'instruction considérée.

La première étape de cette traduction concerne la préparation des spécifications. Il s'agit d'un prétraitement qui s'attache principalement aux étiquettes (adjonction d'une étiquette aux spécifications qui n'en possèdent pas), au contrôle de type (désactivation des spécifications incomplètes ou incorrectes), et aux macro-définitions (substitution de leur valeur aux variables macro-définies).

V.2.7. init.ssl

initialisations et invariants

Ce fichier concerne la clause **initially** d'un programme `UNITY` et l'influence de ces initialisations sur le contrôle des invariants. Les spécifications `Invariant` sont considérées comme des spécifications `Stable` au niveau des affectations, mais sont soumises à la contrainte supplémentaire d'être valides dans l'état initial du programme (Cf § III.7).

Pour une initialisation de la forme $v_1, \cdots, v_n = e_1, \cdots, e_n$, la validité des invariants dans l'état initial correspond à la validité de la formule :

$$\bigwedge_{i=1}^{n} v_i = e_i \quad \Rightarrow \bigwedge_{spécification} invariant$$

Le membre gauche de cette implication est construit dans l'attribut **condinit** par l'intermédiaire de la fonction **EgalitesVarVal**. Le membre droit est synthétisé dans l'attribut **invariants** qui regroupe la conjonction (opérateur **And**) de toutes les spécifications `Invariant`.

L'ensemble de la formule n'est soumis au démonstrateur (fonction **Theoreme** Cf § V.2.10) que si *tous* les invariants sont actifs. Sa clé d'activation est donc la conjonction des clés des invariants (attribut **cle**). De la réponse du démonstrateur dépend l'affichage ou non du message [*INVARIANTS NON INITIALISES*].

Remarque 1 : tous les invariants sont regroupés en une seule formule composée d'opérateurs **And** et soumise au démonstrateur. La fonction d'appel au démonstrateur va ensuite redécomposer cette formule en plusieurs problèmes pour des raisons d'efficacité (Cf § V.2.10 et § VI.2.2).

Remarque 2 : le calcul de l'attribut **invariants** du nœud **specif** fait intervenir un test dont le résultat est sans influence (les deux branches de la conditionnelle impliquent les mêmes calculs). Le rôle de ce test est de forcer, par son évaluation, le calcul de l'attribut **assertions** (et des attributs qui lui sont nécessaires) *avant* le calcul de l'attribut **invariants**. L'objectif est de faciliter la tâche du `Cornell Synthesizer Generator` dans la détermination d'un ordre d'évaluation des attributs. L'absence de ce test multiplie les possibilités au delà des capacités du `Cornell Synthesizer Generator` et provoque une erreur *type 3 circularity.* La technique utilisée ici pour éviter cette erreur est proposée dans [RT89].

V.2.8. types.ssl

contrôle de type

Le contrôle de type est basé sur une table des symboles contenant les noms des variables et les types associés. Cette table est construite d'une manière analogue à celle des macro-définitions (Cf § V.2.6), et est mémorisée dans l'attribut **tds** de **racine**. La création de cette table utilise la fonction **Liste_types** qui génère une *liste* de couples *variable/type* pour chaque déclaration d'une liste de variables. Ces différentes listes sont concaténées (opérateur @) pour former la table.

Le type d'une variable est le type associé à sa *première* occurrence dans la table. Il est obtenu à l'aide de la fonction macro-définie **Ftype**. Les déclarations multiples sont détectées

de 75 conflits *shift/reduce* et de 706 conflits *reduce/reduce*. Néanmoins, l'analyseur imposant engendré fonctionne correctement et a été conservé.

V.2.6. always.ssl

macro-définitions

Cette partie réalise la gestion des macro-définitions de la clause **always** d'un programme `UNITY`. Une table des variables macro-définies et des valeurs qu'elles représentent est construite à partir des déclarations de l'utilisateur. Une fonction permet ensuite de remplacer toutes les variables macro-définies d'une expression par leur valeur.

Le langage **SSL** offre une facilité de construction de tables pour des objets dont une composante au moins est une chaîne de caractères (**STR**). A partir d'une liste de tels objets, **SSL** permet de construire un attribut de type **TABLE** et d'accéder facilement à ses éléments par l'intermédiaire de la fonction **Lookup**. Le `Cornell Synthesizer Generator` se charge de toute la gestion interne de la table, et notamment de la fonction de *H-code* utilisée.

Dans le cas des macro-définitions, une liste de couples est synthétisée dans l'attribut **liste**. Chaque élément de cette liste correspond à une macro-définition et est constitué d'une chaîne de caractères (le nom de la variable macro-définie) et d'une expression (la valeur associée à la variable). Cette liste est transformée en une table par la fonction **MakeTable**, et la table est mémorisée dans l'attribut **env**.

La table construite doit être accessible dans toute partie de l'arbre abstrait où peuvent apparaître des expressions pour pouvoir remplacer les variables macro-définies qu'elles contiennent. Théoriquement, la table devrait être héritée en de multiples endoits de l'arbre abstrait. Le `Cornell Synthesizer Generator` offre une une facilité d'accès à distance de certains attributs (*remote attribute set*). Il faut pour cela garantir que l'attribut accédé sera présent au niveau d'un ancêtre de l'endroit où on l'accède. Dans le cas présent, le seul ancêtre commun à toutes les expressions est la racine de l'arbre (**racine**) qui est donc le nœud auquel l'attribut **env** est associé.

Toute expression contenant des variables macro-définies possède un équivalent pour lequel ces variables ont été remplacées par leur valeur. Cette forme de l'expression est mémorisée dans l'attribut **etendu**. Cet attribut n'est présent que pour des raisons de facilité d'écriture. En effet, il est *impossible* de calculer la valeur étendue d'une expression par simples équations sur des attributs. Ceci provient du fait que la valeur associée à une variable peut elle-même contenir d'autres variables macro-définies. L'expansion des expressions est donc *récursive* et introduirait une circularité au niveau des attributs. Même sans cela, il n'est pas permis en **SSL** d'accéder à un attribut d'un nœud *calculé* (variable locale, valeur de retour d'une fonction, ...).

Le calcul de la valeur étendue d'une expression se fait donc par l'intermédiaire de la fonction **Extension**. Cette fonction est récursive et n'effectue aucun contrôle de circularité sur les macro-définitions. Si un cycle existe, la fonction ne se terminera pas (Cf § III.3.3). Si une variable n'est pas présente dans la table **env**, la fonction laisse cette variable inchangée : c'est le cas des variables non macro-définies.

Les listes d'expressions sont étendues par extension de chacune des expressions qu'elles contiennent.

V.2.4. lex.ssl

définition des unités lexicales

La syntaxe de définition des unités lexicales utilisée par le `Cornell Synthesizer Generator` est très proche de celle de **lex**. Le générateur d'analyseur lexical **flex** est d'ailleurs utilisé lors de la compilation du langage **SSL**.

Les unités définies dans ce fichier concernent les mots-clés (**program, specification, if, declare, initially, always, assign, unless, stable, constant, invariant, with**), les types élémentaires (**int, bool**), les constantes numériques entières, les constantes booléennes (**true, false**), les identificateurs de variables, les messages affichés (entre [et]), les nœuds indéfinis (entre $<$ et $>$), la séparation entre le programme et la spécification (composée de _), et tous les symboles composés de plusieurs caractères (**!=, ==, <=, >=, =>, <=>, :=, [], ||**).

Des unités correspondant aux messages affichés et aux nœuds indéfinis sont nécessaires pour la réanalyse d'un texte partiellement indéfini et/ou comportant des erreurs de types et/ou des spécifications non prouvées.

Les définitions utilisées pour les mots-clés, comme suite de choix entre deux caractères, sont extrèmement coûteuses (en terme de taille de l'analyseur généré). Elles permettent cependant d'éviter toute distinction entre lettres minuscules et lettres majuscules.

La déclaration de l'unité **WHITESPACE** (ici composée d'une suite d'espaces, tabulations et sauts de ligne) est obligatoire et imposée par le `Cornell Synthesizer Generator`.

V.2.5. entry.ssl

analyse syntaxique

La technnique utilisée pour l'analyse syntaxique consiste en la définition d'une grammaire concrète d'une part (dont les identificateurs de nœuds débutent par convention par une lettre *majuscule*), et la mise en correspondance d'éléments des grammaires abstraites et concrètes d'autre part.

Une grammaire concrète est décrite de la même manière qu'une grammaire abstraite. Chaque nœud calcule son équivalent abstrait dans un attribut appelé **trans**. La structure abstraite calculée est ensuite mise en correspondance avec la grammaire abstraite (utilisation du symbole $\sim$). Les conséquences de cette association sont *définitives*, c'est à dire qu'il n'y a aucun moyen pour le programmeur de remplacer un sous-arbre abstrait par un autre dynamiquement en fonction de la valeur des attributs (comme les *transform* le permettent à l'utilisateur). La structure abstraite calculée ne peut donc dépendre du type des variables puisqu'elle ne pourrait être modifiée si ce type changeait. C'est la raison de la grande généralité de la syntaxe abstraite des expressions, identique pour des expressions de types différents.

La grammaire concrète utilisée ici est la copie conforme de la grammaire abstraite. Ce choix ne constitue pas la meilleure approche du problème. En effet, la grammaire abstraite est fortement ambiguë et l'utilisation pour l'analyse syntaxique d'une forme factorisée serait sans doute préférable. A ce défaut vient s'ajouter la présence *éventuelle* de messages (nœuds **Warning**) qui contribue à multiplier les ambiguïtés.

La compilation du langage **SSL** fait appel au générateur d'analyseur syntaxique **bison** (analyse **LR(1)**). Au stade actuel de développement du projet, ce générateur signale la présence

nonassoc. La priorité de $\sim$ ($\neg$) sur $==$ ($=$) empêche la saisie directe de $\sim a == b$ dans le sens $\sim (a == b)$ ($a \neq b$). Si les priorités étaient inversées, $(\sim a) == b$ serait correctement interprété, mais affiché sous la forme $\sim a == b$ par homogénéité avec la représentation des formules de `Otter`, et donc réanalysé comme $\sim (a == b)$. Les priorités choisies sont donc nécessaires pour éviter une incohérence entre le format d'affichage et l'analyse syntaxique, et permettre une réanalyse correcte des textes engendrés. Une solution au problème serait l'abandon du format de `Otter` pour l'affichage des expressions et la définition d'un affichage incomplètement parenthésé compatible avec l'analyse syntaxique.

Le fichier contient également les équations permettant de synthétiser le type d'une expression dans son attribut **t**. Les noms de types sont terminés par un tiret bas (_). Ils sont au nombre de quatre :

- **INT_** : entiers
- **BOOL_** : booléens
- **NOTYPE_** : expressions incomplètement définies, mais correctement typées là où elles sont définies
- **ERREUR_** : expressions partiellement ou complètement définies, mais contenant une incompatibilité de types

Remarque : une distinction importante est faite entre une expression non encore définie correcte là où elle est définie (**NOTYPE_**), et une expression incorrecte (**ERREUR_**).

Par exemple, $5 == true$ est une expression complètement définie et incorrecte, $false + <exp>$ est incomplètement définie et incorrecte, et $2 \leq <exp>$ est incomplètement définie et sans incohérence.

Cette dernière expression, une fois complétée, ne pourra être que booléenne. Son type est donc déjà virtuellement connu. Néanmoins, **NOTYPE_** permet de mémoriser le fait que l'expression est incomplète et donc inutilisable (Cf § V.2.8).

Le principe utilisé pour calculer le type d'une expression est le suivant :

- si la ou les sous-expressions possèdent un type défini (**INT_** ou **BOOL_**) ***et*** si ce type est correct compte tenu de la nature de l'expression, un type défini est affecté à l'expression. Par exemple, si l'expression est $<$ (**Less**) et les types des sous-expressions sont **INT_**, le type **BOOL_** est associé à l'expression.
- si une sous-expression au moins possède un type indéfini (**NOTYPE_**), l'autre type éventuel étant défini correct ou indéfini, le type indéfini **NOTYPE_** est associé à l'expression.
- si une sous-expression au moins possède un type défini incorrect, ou une erreur de type (type **ERREUR_**), le type **ERREUR_** est associé à l'expression.

et l'attribut **faux** constitue le message lié aux théorèmes auxiliaires. Ces chaînes de caractères sont synthétisées au niveau du contrôle de types (Cf § V.2.8) et du contrôle des contraintes (Cf § V.2.9).

Les attributs **varvide** et **strtype** permettent d'insérer des éléments *dynamiques* dans le format d'affichage. L'attribut **varvide** représente l'affichage d'une variable non définie. La forme de cet affichage dépend du contexte d'utilisation de la variable, et sa valeur est héritée de ce contexte. L'attribut **strtype** représente l'affichage d'un nom de type. D'un point de vue interne, les types sont des valeurs entières, mais sont bien entendu représentés à l'affichage par des chaînes de caractères. Les différentes valeurs de ces deux attributs sont définies au § V.2.2.

Les attributs devant, en **SSL**, être définis *avant* d'être utilisés, les attributs mentionnés précédemment sont également définis dans ce fichier.

Deux paramètres (un entier et un booléen) sont associés aux théorèmes et aux spécifications. Le paramètre booléen constitue leur clé d'activation (Cf § III.3.5). Le paramètre entier était utilisé, dans une version précédente du programme, lors de l'appel au démonstrateur automatique. Il permettait, selon sa valeur, un choix dynamique parmi plusieurs configurations de `Otter`. Actuellement, ce paramètre n'est plus utilisé. Ces deux nœuds sont utilisés d'une manière interne et ne sont pas représentés à l'affichage (utilisation du format ..).

V.2.2. affdyn.ssl

formats d'affichage dynamiques

Certains éléments nécessaires au format d'affichage ne peuvent être définis de manière statique. C'est le cas des noms de types (en correspondance avec la valeur entière du type) et des variables indéfinies (dont le format dépend du contexte). Des attributs de type chaîne de caractères sont donc utilisés. Ce fichier contient les définitions des attributs **varvide** et **strtype** (Cf § V.2.1).

strtype fait simplement correspondre une chaîne de caractères à chaque numéro de type (**INT_** → "INT", **BOOL_** → "BOOL", ...).

La valeur de **varvide** est définie en fonction de la nature de la variable à laquelle l'attribut est lié : variable utilisateur, variable macro-définie, expression réduite à une variable ou nom symbolique de spécification. Dans ce dernier cas, la valeur dépend à la fois de la nature de la contrainte (**UNL_**, **INV_**, **STA_**, **CST_**) et de sa position dans la liste des spécifications, ce qui permet de générer facilement des noms distincts. Chaque contrainte porte le numéro de son rang parmi les contraintes de même type (1er `Stable`, 3ème `Invariant`, ...), ce qui nécessite l'utilisation de quatre compteurs auxiliaires (de la forme **num_**). Ces compteurs sont *hérités*, ce qui permet une numérotation croissante du haut de la liste vers le bas.

V.2.3. exp.ssl

définition des expressions

Ce fichier contient toutes les descriptions associées aux expressions. On y trouve en particulier la syntaxe abstraite et le format d'affichage décrits au § III.5.

L'analyse syntaxique se base sur la syntaxe concrète utilisée pour le format d'affichage. Les priorités et associativités des opérateurs sont définies par les commandes **left, right** et

soit l'affectation réellement exécutée. Par contre, la validité d'une garde peut être admise dans la considération d'une affectation particulière et venir renforcer la connaissance liée à la précondition. De plus, à ces n instructions vient s'ajouter l'instruction *nop* (de *wp* identité) pour le cas où aucune garde n'est valide (aucune des n affectations n'est exécutée).

Pour une instruction de la forme :

$$s \equiv \begin{cases} s_0 & \text{si } b_0 \\ \cdots & \cdots \\ s_n & \text{si } b_n \end{cases}$$

on a donc :

$$\{P\}\ s\ \{Q\} \quad \equiv \quad \bigwedge_{i=0}^{n} (P \wedge b_i \Rightarrow wp\,(s_i, Q)) \ \bigwedge\ (P \wedge \neg b_0 \wedge \cdots \wedge \neg b_n \Rightarrow Q)$$

où chaque $wp\,(s_i, Q)$ est calculée par substitution sur Q.

V.1.3. Cas de la logique `UNITY`

Dans le cas de la contrainte `UNITY` : X *Unless* Y, les prédicats P et Q précédents valent respectivement $X \wedge \neg Y$ et $X \vee Y$. Comme $X \wedge \neg Y \Rightarrow X \vee Y$ pour tout X et tout Y, le cas d'une affectation pour laquelle aucune garde n'est vraie n'est pas à considérer. En effet, la contrainte est nécessairement vérifiée par l'instruction *nop*.

Le résultat reste valable pour les contraintes `Stable`, `Constant` et `Invariant` qui ne sont que des cas particuliers de `Unless`.

V.2. Programmation

L'environnement est constitué de deux parties distinctes : l'une concerne la spécification de l'éditeur, l'autre la configuration du démonstrateur automatique.

La première partie est composée de *onze* fichiers décrits ci-après. Ces fichiers utilisent le langage *SSL*, langage de spécification d'éditeurs propre au `Cornell Synthesizer Generator`, dont la description précise se trouve dans [RT89].

La seconde partie utilise les options et paramètres de `Otter` et sera décrite plus loin (Cf § VI.3).

Ces deux parties sont liées par un programme en langage *C* et deux *shell scripts* (*Bourne-Shell*). L'ensemble des programmes *SSL, C* et *Shell* sont ajoutés en annexe B.

V.2.1. root.ssl

structure abstraite et format d'affichage du texte édité

Ce fichier contient la description **BNF** de la grammaire abstraite du programme `UNITY` et de ses spécifications.

Il précise également le format à utiliser pour l'affichage du texte. Ce format fait intervenir des attributs de type chaîne de caractères (**STR**) qui correspondent aux messages entre crochets qui peuvent apparaître dans le texte. Les attributs de la forme **pbs** concernent les problèmes de type, l'attribut **surete** contient les noms des spécifications non vérifiées par une instruction,

V. Un environnement de programmation UNITY Mise en œuvre et programmation

V.1. Principe de traduction des contraintes

V.1.1. Principe général

Le langage de spécifications associé à la logique UNITY utilise le principe d'assertions logiques (*clauses de Hoare*) confrontées aux instructions du programme.

Cette approche est très utilisée et diverses techniques ont été introduites pour la mettre en œuvre. En particulier, la méthode la plus classique consiste à considérer les instructions d'un programme comme des manipulateurs de prédicats (*backward predicate transformers*) [Dij76]. Elle utilise la notion de *weakest precondition* (wp) définie de la manière suivante : pour un prédicat Q et une série d'instructions s, $wp\,(s, Q)$ représente la contrainte *la plus faible* à vérifier *avant* l'exécution de s pour être assuré de la validité de Q *après* l'exécution de s. Un prédicat P ne sera donc une précondition valable à l'établissement de Q que s'il constitue une contrainte *plus forte* que la wp, c'est à dire s'il l'implique. On a donc :

$$\{P\}\ s\ \{Q\} \quad \equiv \quad P \Rightarrow wp\,(s, Q)$$

La fonction $P \rightarrow wp\,(s, P)$ constitue donc une formalisation de la sémantique de la suite d'instructions s.

L'avantage considérable de cette approche est lié à la calculabilité de la wp. En effet, la fonction précédente est parfaitement définie et facilement programmable pour toutes les instructions élémentaires des langages de programmation classiques (affectation, conditionnelle, boucle) et l'enchaînement d'instructions est équivalent à une composition de fonctions :

$$wp\,(s\,;t\ , P) \quad = \quad wp\,(s\ , wp\,(t\ , P))$$

Il est ainsi possible de calculer automatiquement $wp\,(s, Q)$ pour toute suite d'instructions s, et donc de ramener un problème de spécification $\{P\}\ s\ \{Q\}$ à celui de la validité de la formule $P \ \Rightarrow\ wp\,(s, Q)$.

V.1.2. Cas des instructions UNITY

Considéré comme langage de programmation, UNITY n'utilise qu'un unique type d'instruction : l'affectation multiple. De plus, UNITY abandonne toute notion de séquencement et les instructions peuvent être considérées indépendamment les unes des autres.

La seule wp à calculer est donc celle qui définit la sémantique de l'affectation. Une formule Q sera valide après l'exécution d'une affectation si la formule dont elle est issue par l'affectation était valide avant l'exécution de l'instruction. La wp de l'affectation se calcule donc facilement par substitution :

$$wp\,(v_1, \cdots, v_n := e_1, \cdots, e_n\ \ ,\ \ Q) \quad \equiv \quad \sigma(Q)$$

où σ est la substitution $\quad \forall i,\ \ \sigma(v_i) = e_i$.

Ce principe de calcul s'étend au cas où les affectations sont gardées. Une affectation n fois gardée peut être vue comme n affectations différentes selon la valeur des gardes. *Toutes* ces affectations doivent vérifier la contrainte, et la wp doit donc être suffisante quelle que

Spécifié dans la syntaxe de `Otter`, le problème a la forme suivante :

```
set(binary_res).                    % Resolution binaire et factorisation
set(factor).

set(input_sos_first).               % Essai de resolution sur les clauses de l'ensemble
set(sos_queue).                     % support d'abord - l'ensemble support est FIFO

set(process_input).                 % Application des regles orientees sur les
                                    % clauses de depart (hypotheses et conclusion)

assign(max_mem, 15000).             % Memoire maximale
set(free_all_mem).                  % Liberation de la memoire apres usage
assign(max_seconds, 300).           % Temps maximal de recherche : 5 mn

clear(print_kept).                  % Pas d'affichage de clauses
clear(print_back_sub).
clear(print_given).
assign(stats_level, 0).             % Pas de statistiques en sortie

formula_list(usable).
(all x all y all z ((pere(x,y) & pere(y,z)) -> grand_pere(x,z))).          % Hypotheses
fils(Paul,Jacques).
pere(Paul,Pierre).
end_of_list.

list(demodulators).                 % Regle orientee
(fils(x,y) = pere(y,x)).
end_of_list.

formula_list(sos).                  % Negation de la conclusion
-grand_pere(Jacques,Pierre).
end_of_list.
```

et `Otter` produit la preuve :

```
----> UNIT CONFLICT at   0.06 sec ----> 9 [binary,8,4] .

Level of proof is 2, length is 3.

---------------- PROOF ----------------

1 [] (fils(x,y) = pere(y,x)).
2 [] -pere(x,y) | -pere(y,z) | grand_pere(x,z).
3 [demod,1] pere(Jacques,Paul).
4 [] pere(Paul,Pierre).
6 [] -grand_pere(Jacques,Pierre).
7 [binary,6,2] -pere(Jacques,x) | -pere(x,Pierre).
8 [binary,7,3] -pere(Paul,Pierre).
9 [binary,8,4] .

------------ end of proof -------------
```

la liste *n'est pas* sans influence, et les hypothèses apparaissant en premier sont utilisées prioritairement. Un ensemble inconsistant d'hypothèses permet la preuve de n'importe quel prédicat.

- **Règles orientées :** des règles de transformation peuvent être ajoutées afin de faciliter le traitement des égalités. L'orientation de ces règles peut être *statique* (imposée par l'utilisateur) ou *dynamique* (dépendante de poids associés aux formules). Les options précisent sur quels types de clauses et à quels moments ces règles doivent être utilisées.

  Par rapport à des égalités non orientées situées au niveau des axiomes, ces règles permettent de restreindre le nombre de clauses générées, donc à la fois le temps de recherche et la puissance de résolution du système. L'orientation ou non des règles d'égalité constitue donc un compromis.

- **Formule à établir :** le système fonctionnant par réfutation, la formule à établir apparaît sous forme de sa négation.

Remarque 1 : la négation de la formule à établir constitue généralement l'ensemble support initial. Ceci ne constitue pas une obligation. La seule contrainte est que les axiomes constituent un ensemble consistant de clauses, et que la réunion des axiomes et de l'ensemble support initial constitue un ensemble inconsistant de clauses (Cf § VI.2.3).

Remarque 2 : lors de la recherche de preuve, `Otter` ne manipule que des clauses. Néanmoins, il autorise la saisie des axiomes et de la conclusion sous forme de clauses ou de prédicats du premier ordre. Dans ce dernier cas, il gère lui-même le processus de traduction sous forme clausale avant de commencer la recherche.

Remarque 3 : le système `Otter` dispose d'une arithmétique interne sous forme de prédicats prédéfinis sur les valeurs numériques. Ils sont de la forme $SUM, $PROD, $EQ, $LT, $GE, ...

Le fichier de sortie <*fichier.out*> peut contenir :

- la cause de l'arrêt du processus (preuve établie, ensemble support vide, dépassement de la limite en temps, dépassement de la limite en mémoire, ...)
- le détail de la preuve si elle existe (les clauses générées et les résolutions permettant d'atteindre la réfutation)
- des statistiques sur le temps de recherche, l'encombrement mémoire, le nombre de clauses générées, la longueur de la preuve, ...
- d'autres éléments utiles lors d'une mise au point

IV.2.3. Exemple

Mon père est le grand-père de mon fils

Hypothèses :

$$\forall x\ \forall y\ \forall z \quad pere(x,y)\ \wedge\ pere(y,z)\ \Rightarrow\ grand\_pere(x,z)$$

$$fils(Paul, Jacques)$$

$$pere(Paul, Pierre)$$

Règle orientée :

$$fils(x,y)\ \rightarrow\ pere(y,x)$$

Conclusion :

$$grand\_pere(Jacques, Pierre)\ ?$$

Les techniques utilisées en démonstration automatique sont diverses et ont donné naissance à des types variés d'outils. Les systèmes de preuves par réfutation basées sur la résolution et les systèmes de réécriture constituent deux branches parmi les plus importantes. Deux outils ont été étudiés en vue de leur utilisation dans le cadre du projet : `Otter` (*Organized Techniques for Theorem-proving and Effective Research*) [McC90, McC91] qui est un système de preuve par réfutation / résolution, et `RRL` (*Rewriting Rule Laboratory*) [KZ87] qui est un système de réécriture.

Le système `RRL` offre incontestablement une puissance bien plus grande que celle de `Otter` et permet d'envisager des problèmes beaucoup plus complexes. Notamment, il permet d'effectuer des démonstrations inductives qui sont complètement hors de portée de `Otter`. Par contre, plus qu'un simple démonstrateur, il s'agit, comme son nom l'indique, d'un véritable laboratoire *d'aide* à la démonstration, très lourd à manipuler, et son fonctionnement interractif constitue un handicap pour l'utilisation envisagée. De plus, les problèmes délicats qu'il est capable de traiter ne sont pas nécessairement ceux que nous sommes amenés à envisager. En effet, il est plus question, en ce qui nous concerne, de résoudre un problème rapidement que de le résoudre à partir d'une base minimale d'axiomes. De l'aveu même des auteurs, `RRL` peut s'averer particulièrement inefficace sur des problèmes simples ne requérant pas ses multiples possibilités.

A l'opposé, le système `Otter` ne met en jeu aucune méthode originale, mais a été conçu avec le souci constant de l'efficacité. Basé sur les preuves par réfutation / résolution, il utilise la stratégie classique de l'ensemble support (*set of support strategy*), pilotée en largeur, ce qui lui garantit la complétude directe (Cf § VI.2.3). Ecrit directement en langage *C*, il constitue, pour les auteurs de `RRL` (écrit en *Common Lisp*), la référence en matière de rapidité. De plus, `Otter` se présente comme un simple serveur question / réponse au niveau du système d'exploitation, ce qui n'est pas sans avantages par rapport à un système interractif.

Etant donné les spécificités de chacun des deux systèmes et la nature des problèmes à résoudre (sans induction mais avec une forte contrainte sur le temps de résolution), `Otter` a semblé préférable à `RRL` pour une utilisation conjointe à l'éditeur généré.

IV.2.2. Utilisation

`Otter` offre plusieurs possibilités (comme une implémentation de l'algorithme de complétion de **Knuth** et **Bendix**) qui ne sont pas utilisées dans le cadre du projet. Le détail des fonctionnalités de `Otter`, ainsi que leur syntaxe précise se trouve dans [McC90, McC91].

L'utilisation de `Otter` s'effectue par le biais de la commande :

$$otter \quad < \quad fichier.in \quad > \quad fichier.out$$

$<fichier.in>$ contient les données du problème à résoudre et $<fichier.out>$ le détail de la preuve générée par `Otter`. $<fichier.in>$ est composé de *quatre* parties principales :

- **Options et paramètres :** `Otter` admet une multitude d'options et de paramètres permettant de spécifier la stratégie de résolution (hyperrésolution, résolution binaire, factorisation, ...), la gestion de l'ensemble support (FIFO, LIFO, ...), les limites de la recherche (en temps, en mémoire, en taille des clauses, ...), la gestion des règles orientées (sur quelles clauses les appliquer, quand, ...), le format à utiliser pour $<fichier.out>$, ...
- **Formules hypothèses :** tout ce que `Otter` peut admettre pour la résolution du problème doit être spécifié sous forme d'une liste de prédicats (*axiomes*). L'ordre des éléments de

certains éléments de la structure abstraite. L'analyse syntaxique d'une structure n'est possible que si *toutes* les sous-structures disposent déjà d'un équivalent concret. L'analyse globale d'un texte suppose une description concrète à partir du symbole initial (racine) de la grammaire abstraite.

- **Format d'affichage (ou décompilation ou *pretty-printing*) :** ce format décrit la forme d'un texte du langage *en sortie* de l'éditeur. Aucune spécification de ce format n'est obligatoire. Par défaut, les symboles terminaux seront affichés sous leur forme concrète dans un parcours en profondeur de l'arbre. C'est au programmeur, s'il le désire, à insérer dans ce format primitif des chaînes de caractères, sauts de ligne, tabulations, éléments liés à la sémantique, ...

Ces trois descriptions du langage sont *entièrement indépendantes*. L'utilisation la plus courante du `Cornell Synthesizer Generator` consiste à spécifier une syntaxe concrète équivalente à la syntaxe abstraite et permettant l'analyse syntaxique, ainsi qu'un format d'affichage qui n'est que la représentation enjolivée et standardisée de la forme concrète.

Cette approche n'est pas la seule possible, et le `Cornell Synthesizer Generator` offre de grandes libertés dans les relations entre les trois syntaxes. Le format d'affichage peut être sans aucun rapport avec la structure abstraite et refléter, par exemple, des résultats de calculs liés à la sémantique. De la même manière, des éléments concrets peuvent n'avoir aucun équivalent abstrait et constituer des commandes sémantiques.

Sémantique : la forme d'expression de la sémantique utilisée par le `Cornell Synthesizer Generator` est celle des *grammaires attribuées*. Des attributs sont associés à la grammaire abstraite et sont liés par des équations (*context-sensitive relationships*) permettant leur évaluation.

Le `Cornell Synthesizer Generator` restreint le programmeur à l'utilisation de grammaires attribuées ordonnées (*ordered attribute grammars*). Ces grammaires constituent la classe la plus vaste pour laquelle existe un ordre *statique* d'évaluation des attributs, c'est à dire dépendant uniquement de la grammaire et valable pour tout texte édité. Afin de minimiser le nombre d'attributs recalculés après chaque modification, l'algorithme d'évaluation incrémentale de **Kasten** est utilisé.

IV.2. Le démonstrateur automatique de théorèmes `Otter`

IV.2.1. Introduction

La manipulation des attributs réalisée par l'éditeur généré ne permet que de traduire sous une forme classique (la logique des prédicats) les contraintes de cohérence entre le programme et sa spécification. En elle même, elle ne permet pas d'affirmer qu'une contrainte précise est satisfaite ou pas. Elle réduit seulement ce problème à celui de la validité d'une formule logique. La manipulation automatique de formules logiques constitue un domaine abondamment pratiqué de l'informatique, et c'est pourquoi il n'a pas semblé raisonnable d'ignorer sa contribution possible et de chercher à établir la validité des formules construites par nos propres moyens. Au contraire, il a semblé préférable de confier cette tâche à l'un des multiples démonstrateurs automatiques existant, ce type d'outil offrant une puissance impossible à atteindre par nous-mêmes dans le cadre de ce projet.

L'activation de la spécification et du théorème provoque bien entendu la disparition de tous les messages entre crochets.

Remarque : les indications entre accolades ({}) n'apparaissaient pas dans le texte du programme produit par l'environnement et ont été ajoutées pour une plus grande lisibilité des différentes composantes de l'algorithme.

IV. Outils utilisés

IV.1. Le `Cornell Synthesizer Generator`

IV.1.1. Présentation

Le `Cornell Synthesizer Generator` est sans doute le plus répandu des méta-environnements, c'est à dire des générateurs d'environnements de programmation. A partir de descriptions d'un langage de programmation, il produit un logiciel qui peut être vu globalement comme un éditeur syntaxique de ce langage (*language based editor*). Il permet à l'utilisateur de travailler sur un texte du langage et se charge d'une manipulation abstraite associée, spécifiée par le programmeur de l'éditeur.

Pour posséder une représentation abstraite associée, le texte édité doit être syntaxiquement correct. Plus précisément, la contrainte imposée sur le texte est d'être non syntaxiquement incorrect, certaines parties pouvant rester indéfinies.

Les manipulations automatiquement effectuées peuvent être très diverses. Pour un langage de programmation classique, elles peuvent aller d'un simple contrôle de type statique jusqu'à une complète génération de code. Des langages plus particuliers peuvent profiter de manipulations plus spécifiques, α-réduction et β-réduction sur un langage de λ-calcul par exemple.

Tous les éditeurs générés par le `Cornell Synthesizer Generator` sont construits à partir d'un même noyau et disposent d'un nombre important de fonctionnalités communes. Certaines ont été présentées dans le contexte de l'environnement réalisé. Le détail concernant l'utilisation de tels éditeurs se trouve dans [RT89].

IV.1.2. Spécification d'un éditeur

Syntaxe : la description de la syntaxe du langage est triple :

- **Syntaxe abstraite :** elle constitue la représentation interne (arborescente) du langage. La grammaire abstraite du langage est décrite sous une forme propre au `Cornell Synthesizer Generator`, équivalente à une description **BNF** simple (la seule forme de répétition admise est la liste, décrite par une règle récursive à droite).
- **Syntaxe concrète :** elle décrit la forme d'un texte du langage *en entrée* de l'éditeur. Le programmeur n'est tenu de fournir des équivalents concrets que pour les *symboles terminaux* de la grammaire abstraite. La saisie d'un texte s'opère alors par transformations directes de la structure abstraite via le *menu de transformations* (Cf § III.6.3) et par la mise en place des symboles terminaux concrets.

  Le programmeur peut également proposer, pour certaines parties de la grammaire abstraite, une grammaire concrète associée et permettre ainsi une analyse syntaxique de

Remarque : un programme peut être syntaxiquement correct et sémantiquement faux. La présence d'avertissements entre crochets est donc prévue au niveau de l'analyse syntaxique pour la réédition de textes provenant de l'environnement. Ce dernier n'offrant aucune possibilité de *commentaires* dans les programmes édités, des annotations peuvent être ajoutées entre crochets à la suite des instructions. Elle disparaitront dès que le texte sera saisi par l'éditeur, mais ne provoqueront pas d'erreurs de syntaxe. Bien sûr, si le texte initial est remplacé par un texte issu de l'environnement, ces commentaires seront perdus.

III.8. Exemple

On considère l'algorithme d'exclusion mutuelle de **Peterson** pour deux processus [CM88]. La contrainte de sûreté d'un tel algorithme stipule que les deux processus ne sont *jamais simultanément* en zone critique. Elle est représentée par la valeur de vérité de la variable **Exclusion**.

Il est à noter que telle que l'exprime la variable **Exclusion**, la propriété d'exclusion mutuelle n'est pas un invariant du programme. Mais étant impliquée par l'invariant **Spec_1**, elle est bien vérifiée à tout instant par le programme (Cf remarque § **II.3.2**).

Introduit dans l'environnement, le texte de l'algorithme est représenté comme suit :

```
PROGRAM
DECLARE
U_en_ex,U_b,U_hors_ex,U_req_ex,V_en_ex,V_b,V_hors_ex,V_req_ex,TOUR : BOOL;
Inv_Temp,Exclusion : BOOL

ALWAYS
Inv_Temp  = ((U_en_ex => (U_b & (~V_b | ~TOUR))) & (V_en_ex => (V_b & (~U_b | TOUR)))),
Exclusion =  ~(U_en_ex & V_en_ex)

INITIALLY
U_en_ex,U_req_ex,V_en_ex,V_req_ex,U_hors_ex,U_b,V_hors_ex,V_b =
                                              false,false,false,false,true,false,true,false
ASSIGN
                { Comportement du processus U }
      U_en_ex,U_hors_ex,U_b := false,true,false IF U_en_ex                       [ spec_1 ]
[]    U_req_ex,U_hors_ex := true,false          IF U_hors_ex                     [ Spec_1 ]

                { Comportement du processus V }
[]    V_hors_ex,V_en_ex,V_b := true,false,false IF V_en_ex                       [ Spec_1 ]
[]    V_req_ex,V_hors_ex := true,false          IF V_hors_ex                     [ Spec_1 ]

                { Comportement du serveur d'exclusion mutuelle }
[]    U_b,TOUR := true,true                     IF (U_req_ex &  ~U_b)            [ Spec_1 ]
[]    U_en_ex := true                           IF (U_b & ~(V_b & TOUR))         [ Spec_1 ]
[]    V_b,TOUR := true,false                    IF (V_req_ex &  ~V_b)            [ Spec_1 ]
[]    V_en_ex := true                           IF (V_b & ~(U_b &  ~TOUR))       [ Spec_1 ]
------------------------------------------------------------------------------------------

SPECIFICATIONS

        [ INVARIANTS NON INITIALISES ]

Spec_1   INVARIANT Inv_Temp

WITH
(Inv_Temp => Exclusion) [ NON PROUVE ]
```

L'ensemble des modifications sont validées par la touche <*return*>. A ce moment, le texte correspondant au sous-arbre actif est réanalysé dans sa totalité. Si l'analyse échoue, le curseur ^ est placé au voisinage de l'erreur et la modification du texte doit se poursuivre. Si au contraire l'analyse aboutit, le texte est traduit en un arbre abstrait qui vient remplacer le sous-arbre actif. Ce procédé garantit la correction syntaxique du texte du programme entier.

Il existe également une possibilité de détruire le sous-arbre sélectionné. Elle peut s'effectuer par l'intermédiaire du menu *edit* par la commande *delete structure*, ou par l'utilisation simultanée des touches <*shift*>, <*control*> et <*k*>.

III.6.3. Transformations

Un dernier moyen pour modifier le texte édité consiste à *transformer* directement la structure abstraite d'une partie du texte. Chaque type de sous-arbre possède ses transformations propres apparaissant dans le *menu de transformations* situé en bas de la fenêtre. Le fait de cliquer avec la souris sur le nom d'une transformation applique celle-ci au sous-arbre actif.

Ces transformations peuvent s'appliquer à un arbre vide (non défini) et en modifier son squelette. Par exemple, $\boxed{+}$ permet de transformer une expression indéfinie en somme de deux expressions indéfinies.

$$<exp> \quad \rightarrow \quad <exp> + <exp>$$

Elles peuvent également s'appliquer à des arbres partiellement ou complètement définis. Par exemple, $\boxed{\text{Invariant}}$ permet de modifier une spécification :

$$<ident>\ STABLE\ \ n < m \quad \rightarrow \quad <ident>\ INVARIANT\ \ n < m$$

Un type particulier de transformation permet d'ajouter des éléments à une liste (Cf § III.6.1).

III.7. Interprétation des messages affichés

Le rôle de l'environnement est de vérifier la validité de la spécification vis-à-vis des instructions du programme. Le résultat de ce contrôle apparaît au travers de *messages* affichés au sein du texte édité.

Parmi ces messages, un premier type concerne les spécifications proprement dites. De tels messages sont situés à la suite de chaque instruction. Ils contiennent les nom symboliques des spécifications ***NON*** vérifiées par l'instruction, encadrés entre crochets ([ ]). L'absence d'un tel message signifie que l'instruction vérifie *toutes* les spécifications.

Un second type de message concerne les théorèmes de la clause **with**. Ils sont d'une unique forme : [*NON PROUVE*]. Un théorème prouvé valide n'est pas suivi de ce message.

Un dernier message concerne les `Invariants`. De la forme [*INVARIANTS NON INITIALISES*], il apparaît en tête des spécifications dès qu'au moins *un* `Invariant` n'est pas prouvé dans l'état initial. Il disparaît dès que *tous* les `Invariants` sont vérifiés dans l'état initial.

Un `Invariant` non vérifié dans l'état initial, mais maintenu par chaque affectation (ie vérifiant la contrainte `Stable`), n'apparaît pas *dans les messages associés aux instructions. Le non respect de la contrainte n'apparaît alors* que *dans le message* [*INVARIANTS NON INITIALISES*].

De plus, les opérateurs	$+, -, \neg, \wedge, \vee$	sont associatifs à gauche,
	$\Rightarrow, \Leftrightarrow$	sont associatifs à droite,
et	$<, >, \leq, \geq, =, \neq$	sont non associatifs.

Ceci permet la saisie d'expressions du type :

$$\sim a + b < c\&x|y => z => t! = v|w$$

pour :

$$(((\neg((a+b) < c) \wedge x) \vee y) \Rightarrow (z \Rightarrow ((t \neq v) \vee w)))$$

Remarque : avec les priorités précédentes, $\sim a == b$ signifie $(\sim a) == b$, a et b étant des expressions booléennes. Pour saisir $\sim (a == b)$, il faut ajouter explicitement les parenthèses (Cf § V.2.3).

III.6. Modification du texte

III.6.1. Sous-arbre actif

A tout moment de l'édition, seul le sous-arbre actif (apparaissant souligné) peut être modifié. Il existe *trois* moyens principaux de choisir le sous arbre actif :

- **par le menu *structure* :** ce menu offre toutes les possibilités classiques de déplacement dans un arbre (accès au père, aux fils, aux frères, ...). Il est généralement peu utilisé comparé aux autres moyens.
- **par la touche <*return*> :** la touche <*return*> permet d'atteindre le sous-arbre qui suit le sous-arbre actif dans un parcours en profondeur de l'arbre. Ce parcours est le plus naturel et correspond à un déplacement dans le texte de gauche à droite et de haut en bas. Il constitue pour cette raison le moyen le plus utilisé pour choisir le sous-arbre actif.
- **par la souris (bouton gauche) :** le fait de cliquer sur un symbole à l'aide de la souris rend actif le plus petit sous-arbre contenant ce symbole. En particulier, cliquer entre deux termes active un sous-arbre les contenant tous les deux.

  Cliquer de manière répétée permet, à partir d'un point, d'atteindre des sous-arbres de plus en plus grands englobant le précédent. Ceci permet également, au niveau des listes (de variables, d'expressions, de déclarations, de macro-définitions, d'affectations, de spécifications ou de théorèmes), de générer une possibilité supplémentaire du *menu de transformations* (Cf § III.6.3) : l'ajout d'un élément dans la liste.

  La souris permet également de modifier la position d'un curseur de texte $\wedge$ au sein du sous-arbre sélectionné.

Remarque : en fait, seuls certains sous-arbres choisis par le programmeur sont sélectionnables. Ils correspondent aux parties pour lesquelles une analyse syntaxique est possible. Ils sont normalement suffisamment nombreux pour faciliter l'édition.

III.6.2. Insertion et suppression de texte

Le texte correspondant au sous-arbre sélectionné peut être modifié en toute liberté. Les caractères saisis s'insèrent à droite du curseur $\wedge$. La touche d'effacement (*Back Space*) permet la suppression de caractères à gauche du curseur $\wedge$.

- **gardes :** une expression non gardée doit être unique dans une affectation et les expressions constituant les gardes doivent être booléennes (Cf § III.3.4).
- **spécifications :** les paramètres des spécifications doivent être des expressions booléennes (Cf § III.3.5).
- **théorèmes :** les théorèmes auxiliaires doivent être des expressions booléennes (Cf § III.3.6).

Une affectation, une spécification ou un théorème qui ne respecte pas les contraintes de type (présence d'un avertissement) *n'intervient pas dans le calcul de preuve*. Notamment, un théorème non booléen ne sera *jamais* prouvé; une spécification dont un paramètre n'est pas booléen ne sera vérifiée par *aucune* instruction; une instruction affectant des valeurs à des variables de types différents, ou pour laquelle manque une garde ne vérifiera *aucune* spécification.

III.5. Expressions

Dans le contexte de l'environnement décrit, les expressions sont utilisées à la fois pour les spécifications (elles sont alors booléennes) et pour les calculs effectués par le programme (elles sont alors booléennes ou entières).

Elles sont définies par induction structurelle à partir des éléments de base :

- constantes numériques : $n \in Z$
- constantes booléennes : $true$ et $false$
- variables utilisateur

et des constructeurs :

– $+$	: addition,	représenté par : +	– $-$	: soustraction,	represente par : -
– $<$	: inférieur strict,	représenté par : <	– $>$	: supérieur strict,	représenté par : >
– $\leq$	: inférieur large,	représenté par : <=	– $\geq$	: supérieur large,	représenté par : >=
– $=$	: égalité,	représenté par : ==	– $\neq$	: non égalité,	représenté par : !=
– $\wedge$	: *et* logique,	représenté par : &	– $\vee$	: *ou* logique,	représenté par : |
– $\Rightarrow$	: implication,	représenté par : =>	– $\Leftrightarrow$	: équivalence,	représenté par : <=>
– $\neg$	: *non* logique,	représenté par : ~			

Les expressions ne possèdent aucune possibilité de multiplication ni de division.

Le démonstrateur automatique impose que les expressions qu'il manipule soient *complètement* parenthésées. Par simplification, c'est également sous cette forme que les expressions sont affichées.

Par contre, des expressions incomplètement parenthésées peuvent être introduites. La priorité des opérateurs est alors, en ordre décroissant de la gauche vers la droite :

$$
\begin{array}{ccccccc}
+ & < & \neg & = & \wedge & \vee & \Rightarrow \\
- & > & & \neq & & & \Leftrightarrow \\
 & \leq & & & & & \\
 & \geq & & & & &
\end{array}
$$

La syntaxe de la partie **specification** est la suivante :

$$specification\_part \quad ::= \quad 'specification' \quad specification\_unit \quad \{',' \quad specification\_unit\}$$

$$\begin{array}{lcllll} specification\_unit & ::= & & [<ident>] & exp & 'unless' & exp \\ & & | & [<ident>] & & 'stable' & exp \\ & & | & [<ident>] & & 'constant' & exp \\ & & | & [<ident>] & & 'invariant' & exp \end{array}$$

III.3.6. With

interface avec le démonstrateur automatique

Cette partie n'a aucun correspondant ni dans la syntaxe, ni dans la logique de `UNITY`. Elle constitue simplement une interface directe avec le démonstrateur automatique de théorèmes et permet la démonstration de théorèmes utiles à la preuve du programme sous la forme d'expressions booléennes sur les variables du programme (Cf exemple § III.8).

De la même manière que les spécifications, ces théorèmes peuvent ou non être actifs et possèdent les choix $\boxed{\text{Actif}}$ et $\boxed{\text{Passif}}$ dans leur menu de transformations. Leur activité ou passivité est également affectée par les choix $\boxed{\text{Tout Actif}}$ et $\boxed{\text{Tout Passif}}$ de la racine du programme.

La syntaxe des théorèmes est celle d'une liste d'expressions :

$$with\_part \quad ::= \quad 'with' \quad exp \quad \{',' \quad exp\}$$

III.4. Contrôle de type

Pour des raisons d'analyse syntaxique (Cf § V.2.5), la syntaxe abstraite des expressions (*exp*) est très générale. Notamment, les expressions booléennes et les expressions entières ne sont pas distinguées.

Or, certaines parties du programme imposent aux expressions d'être booléennes. C'est le cas des gardes, des paramètres de spécification et des théorèmes.

Il convient donc de contrôler que les expressions soumises au démonstrateur automatique ont bien un sens à son niveau, c'est à dire qu'elles possèdent un type et que ce type est booléen.

Un contrôle de type est donc effectué. Basé sur les déclarations des variables (Cf § III.3.1), il permet de déterminer le type d'une expression s'il existe. Si ce type n'est pas correct compte tenu du rôle de l'expression, un avertissement est affiché. Les différents messages possibles sont :

[*Variable Non Déclarée*] [*Déclaration Multiple*]
[*Types Incompatibles*] [*Type Non Booléen*]

A ces messages s'ajoute [*Garde Obligatoire*] (Cf § III.3.4) qui est considéré comme faisant partie du contrôle de type.

Les contraintes sur les types sont les suivantes :

- **affectation :** une variable doit recevoir une valeur de son type. Les variables d'un même vecteur peuvent être de types différents, mais les expressions correspondantes doivent être du type approprié.

D'autres contrôles sont également manquants. Par exemple, le programme ne détecte pas la présence d'une variable macro-définie *à gauche* d'une affectation.

En parallèle avec l'absence de type tableau, aucune facilité de quantification n'est actuellement implémentée.

La syntaxe complète des affectations du programme est donc :

$$\begin{array}{lcl}
assign_part & ::= & 'assign' \;\; assign_unit \;\; \{'[]' \;\; assign_unit\} \\
assign_unit & ::= & affect \;\; \{' \parallel ' \;\; affect\} \\
affect & ::= & < ident > \{',' < ident >\} \, ':=' \, exp \, \{','exp\} \; [garde] \; \{'\sim' \, exp \, \{','exp\} \; garde\} \\
garde & ::= & 'if' \;\; exp
\end{array}$$

III.3.5. Specification

spécifications du programme

Les quatre contraintes fondamentales de la logique de sûreté de `UNITY` (`Unless`, `Stable`, `Constant` et `Invariant`) sont utilisables. Elles forment une liste de spécifications dont l'ordre des éléments est sans influence. Aucun regroupement entre propriétés de même type n'est imposé.

`Stable`, `Constant` et `Invariant` utilisent un paramètre et `Unless` en nécessite deux. Ces paramètres sont des expressions (*exp*) et leur type doit être *booléen*. Les paramètres sont de part et d'autre du mot-clé `Unless` et *suivent* le mot-clé dans les autres cas.

Chaque contrainte peut être étiquetée par un nom symbolique. Ce nom doit précéder l'ensemble de la déclaration. Dans le cas où aucune étiquette n'est précisée, l'environnement en choisit une. Ces noms symboliques sont utilisés pour désigner les spécifications au niveau de la partie algorithmique du programme (Cf § III.7).

Les contraintes spécifiées peuvent être *actives* ou *passives*. Seules les contraintes actives sont prises en compte par le système, c'est à dire qu'aucune démonstration des contraintes passives n'est mise en œuvre. Cette distinction autorise à ne prendre en compte simultanément qu'un nombre réduit de spécifications, ce qui permet d'alléger la charge du système. En effet, une modification du programme peut remettre en cause *toutes* les spécifications. Cette modification est facilitée si, pendant qu'elle a lieu, peu (ou pas) de clauses sont actives. Un grand nombre d'appels au démonstrateur automatique sont alors évités et il suffit de réactiver les contraintes après la ou les modifications pour s'assurer de la validité de celles-ci.

Lors de l'analyse d'un *texte*, les contraintes sont *toutes* passives dans l'arbre abstrait généré. Le caractère actif ou passif des contraintes peut être modifié globalement (toutes les contraintes) ou localement (seulement une contrainte précise). Cette modification se fait par l'intermédiaire du *menu de transformations* (Cf § III.6.3).

Lorsque la partie sélectionnée de l'arbre abstrait du programme représente une contrainte, le menu de transformations précise l'état d'activation de cette contrainte au travers des indications $\boxed{\text{Actif}}$ ou $\boxed{\text{Passif}}$ qui s'excluent mutuellement. Le choix d'une de ces transformations à l'aide de la souris inverse l'état d'activation de la contrainte et modifie le menu en conséquence.

Lorsque l'arbre entier est sélectionné, le menu offre les possibilités $\boxed{\text{Tout Actif}}$ et $\boxed{\text{Tout Passif}}$ qui respectivement activent et désactivent *toutes* les spécifications du programme ainsi que tous les théorèmes auxiliaires (Cf § III.3.6). $\boxed{\text{Tout Passif}}$ est très utile pour inhiber complètement la vérification des contraintes avant une modification du programme.

Une macro-définition donnée concerne *une* variable et *une* expression. Les différentes macro-définitions sont séparées par des *virgules* (,). La syntaxe générale de cette partie est donc :

$$\begin{array}{lll} define\_part & ::= & 'always' \;\; define\_unit \;\; \{',' \;\; define\_unit\} \\ define\_unit & ::= & <ident> \;\; '=' \;\; exp \end{array}$$

III.3.4. Assign

instructions du programme

Cette partie regroupe les affectations qui constituent le corps du programme. Les affectations sont *vectorielles* et *gardées*. Le symbole d'affectation est $' := '$. Les gardes sont introduites par le mot-clé $'if'$. Les expressions correspondant à différentes gardes sont séparées par le symbole $' \sim '$. Les différentes instructions sont séparées par le symbole '[]', représenté par un crochet ouvrant suivi d'un crochet fermant ([]).

L'absence de garde équivaut à une garde toujours vraie. Elle n'est possible que si elle constitue *l'unique* expression affectée, c'est-à-dire qu'une expression *non gardée* ne peut être suivie par d'autres expressions, gardées ou non. En effet, une affectation comme :

$$\begin{array}{lllll} x & := & y & & \sim \\ & & z & if \; b & \sim \\ & & t & if \; c & \end{array}$$

est dépourvue de signification et n'est pas acceptée.

Le symbole d'affectation synchrone $\|$ est accepté. Les différentes affectations séparées par ce symbole peuvent être gardées différemment (Cf § V.2.9). Ainsi, l'affectation :

$$\begin{array}{lllll} & x, y & := & a, b & if \; u \;\; \sim \\ & & & c, d & if \; v \\ \| & z & := & e & \end{array}$$

équivaut à :

$$\begin{array}{llll} x, y, z & := & a, b, e & if \; u \;\; \sim \\ & & c, d, e & if \; v \;\; \sim \\ & & x, y, e & if \; \neg u \wedge \neg v \end{array}$$

Excepté le contrôle de type, *aucun contrôle sémantique n'est effectué*. Notamment, le langage `UNITY` impose que les affectations soient équivalentes dans le cas de plusieurs gardes simultanément vraies. Par exemple :

$$\begin{array}{llll} x & := & a & if \; u \;\; \sim \\ & & b & if \; v \end{array}$$

n'est une instruction valide que si

$$u \wedge v \;\; \stackrel{prog}{\Rightarrow} \;\; a = b$$

où $\stackrel{prog}{\Rightarrow}$ est une implication *compte tenu du programme*, c'est à dire qu'au sein du programme, si u et v sont simultanément vrais, alors nécessairement $a = b$ ($u \wedge v \Rightarrow a = b$ est valide dans tous les états *accessibles* du programme). De même, l'affectation $x, x \;:=\; a, a + 1$ n'est jamais valide. Ce type de contrôle n'est pas effectué.

Les différentes déclarations sont séparées par des *points-virgules* (;). Le point-virgule est un séparateur; la dernière déclaration n'est donc ***pas*** suivie d'un point-virgule.

Plusieurs variables peuvent être déclarées d'un même type au sein d'une seule déclaration. Elles sont alors séparées par des *virgules* (,). La liste de variables précède le type et en est séparée par *deux points* (:).

Les identificateurs de variables sont des chaînes de caractères constituées de lettres majuscules, de lettres minuscules, de chiffres et du tiret bas (_). Ils débutent nécessairement par une lettre.

La syntaxe générale de cette partie est donc :

$$\begin{array}{rcl}
\mathit{declare_part} & ::= & \mathit{'declare'}\ \ \mathit{declare_unit}\ \ \{';'\ \ \mathit{declare_unit}\} \\
\mathit{declare_unit} & ::= & <\mathit{ident}>\ \ \{','\ \ <\mathit{ident}>\}\ \ ':'\ \ <\mathit{type}> \\
<\mathit{type}> & ::= & \mathit{'INT'}\ \ |\ \ \mathit{'BOOL'} \\
<\mathit{ident}> & ::= & \mathit{letter}\ \ \{\mathit{letter}\ \ |\ \ \mathit{digit}\ \ |\ \ '_'\}
\end{array}$$

III.3.2. Initially

initialisation des variables

Aucune initialisation n'est obligatoire. Les possibilités offertes sont élémentaires : il est possible d'affecter un vecteur d'expressions à un vecteur de variables préalablement au démarrage du programme. Cette initialisation intervient uniquement dans les preuves associées aux invariants : elle permet d'établir quels invariants sont valides dans l'état initial du programme.

Excepté le contrôle de type (Cf § III.4), *aucun contrôle sémantique n'est effectué.* En particulier, la cohérence et la non circularité des initialisations sont à l'entière responsabilité de l'utilisateur. L'initialisation à partir de valeurs non déterminées n'est pas non plus détectée.

Remarque : un ensemble inconsistant d'initialisations, comme $a, b, a = 1, 2, 3$, conduit à un état initial inaccessible pour lequel *tous les* `Invariants` *sonts vérifiés.*

La forme générale d'une initialisation est donc :

$$\mathit{init\_part}\ ::=\ \mathit{'initially'}\ \ <\mathit{ident}>\ \ \{','\ \ <\mathit{ident}>\}\ \ '='\ \ \mathit{exp}\ \ \{','\ \ \mathit{exp}\}$$

La syntaxe des expressions (exp) est donnée § III.5.

III.3.3. Always

macros-définitions

Un nombre quelconque de variables peuvent être *macro-définies.* Ces variables doivent être préalablement déclarées. Toute occurrence ultérieure de la variable sera remplacée par l'expression correspondante.

Comme dans le cas des initialisations, *aucun contrôle n'est effectué.* Une différence importante est qu'une macro-définition circulaire ne se contente pas de produire des résultats erronés, *mais plonge l'exécution du programme dans une boucle infinie.*

$< fichier >$ peut contenir un programme et sa spécification sous une forme arborescente ou sous une forme textuelle. La forme arborescente constitue un format interne au `Cornell Synthesizer Generator` et ne peut provenir que d'une utilisation précédente de l'environnement. Au contraire, la forme textuelle n'est pas liée à l'environnement et peut être générée à partir de n'importe quel éditeur de texte classique. Dans ce cas, le texte proposé est alors soumis à une analyse syntaxique et traduit sous forme abstraite. L'utilisation proprement dite de l'environnement ne pourra commencer que si l'analyse s'est d'abord terminée correctement, c'est à dire si le texte est syntaxiquement valide. Dans le cas contraire, l'éditeur offre une possibilité sommaire d'édition classique de la partie erronée du texte.

Différents menus permettent la gestion du tampon manipulé par l'environnement vis-à-vis du système de gestion de fichiers. En particulier, ils permettent le transfert inverse de celui des appels, c'est à dire la sauvegarde du programme édité dans un fichier spécifié, sous forme arborescente ou textuelle. Les textes générés par l'environnement sont syntaxiquement valides et peuvent être réédités sans difficultés.

III.3. Syntaxe et mots-clés

Un texte édité, représentant à la fois un programme et sa spécification, est introduit par le mot-clé *obligatoire* **Program**. Il est composé de *six* parties, toutes facultatives, introduites successivement par les mots-clés **declare, always, initially, assign, specification** et **with**. Les quatre premières parties concernent l'expression de l'algorithme en `UNITY`, et les deux dernières ses spécifications. Ces deux composantes sont éventuellement séparées par une ligne horizontale de longueur arbitraire, composée de tirets bas (_).

Dans l'ensemble du texte, aucune distinction n'est faite entre majuscules et minuscules, que ce soit en ce qui concerne les mots-clés, les termes de logique `UNITY` (`Unless`, `Stable`, `Constant`, `Invariant`), les noms de types ou les identificateurs d'étiquettes et de variables.

La cinquième partie du texte peut également être introduite par le mot-clé **specifications** au lieu de **specification**.

Dans la suite de ce paragraphe, la syntaxe du langage édité sera présentée sous forme **EBNF** *(Extended-Backus-Naur-Form)* avec les conventions suivantes :

::=	définition d'un terme
[ ]	terme facultatif
{ }	répétition n fois d'un terme, $n \in N$
|	ou logique
<>	unités lexicales
' '	constantes chaînes de caractères

III.3.1. Declare

déclarations des variables et de leur type

Toutes les variables utilisées dans le programme, y compris les variables macro-définies (Cf § III.3.3), doivent être déclarées dans cette partie. Les déclarations multiples (même identiques) sont interdites et détectées.

Les deux types élémentaires utilisés sonts les *entiers* (**INT**) et les *booléens* (**BOOL**). Ils constituent pour le moment les deux seules déclarations possibles, les tableaux, ensembles et suites n'étant pas implémentés.

- Système de preuve : une représentation formelle de l'exécution d'un programme par une suite infinie d'états similaire à celle de [OL82] est utilisée. Elle permet d'établir des théorèmes fondamentaux sur les outils utilisés, à partir desquels les propriétés d'un programme peuvent être formellement démontrées.

  La version courante du système ne permet pas l'intégration de ces théorèmes, et c'est la raison pour laquelle ils n'ont pas été présentés.

- Composition de programmes : `UNITY` possède des opérateurs de composition de programmes et permet la preuve de propriétés sur le programme composé à partir de propriétés des programmes composants. Ces outils de composition, destinés à une conception modulaire, ne sont pas utilisés pour le moment dans le cadre du projet.

- Application à une architecture : `UNITY` propose, de manière trop peu formalisée, des techniques permettant, en phase finale de leur développement, d'implémenter les algorithmes écrits sur divers types d'architectures. Il s'agit d'un aspect important du langage, mais hors de propos au niveau du projet.

III. Un environnement de programmation `UNITY` Spécificités et utilisation

III.1. Présentation

L'environnement développé se présente globalement sous la forme d'un éditeur syntaxique sous `X-Window`. Cet éditeur possède toutes les fonctionnalités des éditeurs générés par le `Cornell Synthesizer Generator` dont le détail peut être trouvé dans [RT89]. Il ne sera fait mention ici que du minimum indispensable à une utilisation des spécificités `UNITY` de l'éditeur.

L'éditeur interprète un texte en fonction de la syntaxe d'un langage donné (*language sensitive edition*). Le langage utilisé au niveau de l'environnement est composé d'une part du sous-ensemble de la syntaxe `UNITY` présenté précédemment, et d'autre part d'une extension de cette syntaxe permettant d'exprimer les spécifications du programme. Un texte édité se compose donc de deux parties distinctes : l'une exprime l'algorithme dans la syntaxe `UNITY` et l'autre exprime les spécifications de cet algorithme dans la syntaxe de la *logique* `UNITY`.

L'environnement ne permet la gestion que de textes *syntaxiquement valides*. Ces textes possèdent alors une expression équivalente sous forme d'arbre dans la grammaire du langage, et l'éditeur ne manipule que cette représentation abstraite. A tout moment, un sous-arbre particulier du texte édité est actif et seule cette partie du texte peut être modifiée (Cf § III.6). Un menu de l'éditeur permet de choisir le sous-arbre actif et la touche <*return*> permet d'activer le sous-arbre suivant dans un parcours en profondeur de l'arbre. Le texte correspondant à la partie active de l'arbre apparaît souligné.

III.2. Appel

En présence d'un environnement `X-Window`, l'éditeur est activé par le biais de la commande

$$unity \quad [<fichier>]$$

où $<fichier>$ est le fichier facultatif à éditer. Si aucun nom de fichier n'est précisé ou si le nom correspond à un fichier vide ou inexistant, un canevas `UNITY` sera édité.

(la valeur de vérité de P ne change pas au cours de l'exécution du programme).

$$Invariant\ P \ \equiv\ (conditions\ initiales \Rightarrow P) \wedge (Stable\ P)$$

(P est vérifié au commencement de l'exécution du programme et à tout instant par la suite).

Remarque : la précision de ces définitions les rend assez restrictives. Elles présentent l'avantage considérable de pouvoir prendre en compte chaque instruction indépendamment des autres. Mais, par exemple, une propriété peut être vérifiée en tout point d'un programme sans être un invariant. Toutefois, il existe dans ce cas une contrainte plus forte possédant la propriété d'invariance, malheureusement difficile à déterminer de manière automatique (Cf exemple § III.8).

II.3.3. Exemple

Le programme `UNITY` suivant calcule la somme de deux entiers a et b.

```
Program Addition

Declare     a, b, k, somme : int;

Initially   k = b  ||  somme = a ;

Assign      somme, k  :=  somme + 1, k - 1    if  k > 0

End Addition
```

Ce programme vérifie les propriétés de sûreté suivantes :

- $k = Cte$ `Unless` $k < Cte$: la valeur de k ne peut que décroître.
- `Stable` $k = 0$: si k devient nulle, elle le restera indéfiniment par la suite.
- `Invariant` $somme + k \ = \ a + b$: l'égalité est vraie à l'initialisation et à tout instant futur.

Pour spécifier complètement le fait que le programme calcule $(a + b)$ dans *somme*, il faut ajouter une propriété de vivacité précisant que k *atteindra* la valeur 0 (Cf § II.3.4).

II.3.4. Aspects non présentés

Cette présentation du formalisme `UNITY` est excessivement sommaire et se limite volontairement aux aspects directement utilisés dans le cadre du projet. Une description complète se trouve dans [CM88], notamment :

- Propriétés de vivacité : la logique associée à `UNITY` formalise également les propriétés de vivacité dans un système équivalent à la *logique temporelle* utilisé par **S. Owicki** et **L. Lamport** [OL82].

  La propriété fondamentale est le `leads-to` noté $\mapsto$. $P \mapsto Q$ spécifie que le programme, s'il atteint un état satisfaisant P, atteindra, après un nombre *fini* de transitions, un état satisfaisant Q. Dans l'exemple d'addition précédent, une propriété pourrait être $(b \geq 0) \mapsto (k = 0)$.

- **quantification des instructions**
  Le principe de quantification précédent est généralisé aux deux opérateurs $\|$ et $[\!]$, de neutre *nop*. Par exemple,
  $$\langle \| \; i : 0 \leq i \leq N :: A[i] := B[i] \rangle$$
  équivaut à
  $$A[0] := B[0] \; \| \; A[1] := B[1] \; \| \cdots \| \; A[N] := B[N]$$

II.3. La logique `UNITY`

II.3.1. Présentation

Au langage d'instructions précédemment décrit sont associés un langage de spécification et un système de preuve de ces spécifications. Ce langage permet de spécifier des propriétés de sûreté et de vivacité sur des expressions logiques faisant intervenir les variables manipulées par le programme. Ces propriétés peuvent ensuite être prouvées formellement au sein d'une théorie et à l'aide d'un ensemble de théorèmes.

La logique `UNITY` est fondée sur des assertions de la forme $\{P\}\ s\ \{Q\}$ (*clauses de Hoare* [Hoa69]) où P et Q sont des prédicats logiques et s une instruction du programme. Une telle écriture signifie que *si l'instruction s débute dans un état du programme satisfaisant P et si elle se termine, l'état résultant vérifiera alors la propriété Q.*

Un programme `UNITY` ne possédant aucun point de contrôle, les propriétés envisagées ne peuvent être attachées à un point précis du programme. Elle sont donc *globales* et s'expriment *indépendamment du texte du programme.* Pour un programme F donné et des prédicats P et Q, elles sont généralement de la forme :

$$\begin{array}{llll} & \forall s \in F & \{P\}\ s\ \{Q\} & \text{(sûreté)} \\ \text{ou} & \exists s \in F & \{P\}\ s\ \{Q\} & \text{(vivacité)} \end{array}$$

Le système de preuve sous-jacent fait l'hypothèse que toute instruction est assurée de se terminer. Ceci permet, au niveau des propriétés de sûreté, de se dispenser de l'utilisation d'une logique particulière (logique des *at, in, after*, logique temporelle) et de se limiter à la logique classique [Lam80].

II.3.2. Syntaxe des propriétés de sûreté

Les propriétés de sûreté sont exprimées à partir de quatre propriétés fondamentales : `Unless`, `Stable`, `Constant` et `Invariant`. Pour un programme F et des prédicats P et Q, elles sont définies comme suit :

$$P\ Unless\ Q \ \equiv\ \langle \forall s : s\ in\ F :: \{P \wedge \neg Q\} s \{P \vee Q\} \rangle$$

(si P est vérifié à un instant donné, P est maintenu au moins jusqu'à ce que Q devienne valide).

$$Stable\ \ P \ \equiv\ P\ \ Unless\ \ false$$

(si P est vérifié à un instant donné, P sera indéfiniment maintenu par la suite).

$$Constant\ \ P \ \equiv\ (Stable\ \ P) \wedge (Stable\ \ \neg P)$$

assign

La partie **assign** constitue le corps proprement dit du programme, c'est à dire l'ensemble (non ordonné) des instructions qui le composent. Ces instructions sont séparées par le symbole d'exclusion $[\!]$.

La sémantique de l'exécution d'un programme est la répétition infinie du processus suivant :

- choix d'une instruction et
- exécution *atomique* de cette instruction

La seule hypothèse sur le choix est celle d'une équité faible : chaque instruction est exécutée infiniment souvent.

Le langage `UNITY` dispose d'une unique instruction élémentaire : l'affectation multiple gardée. Elle est de la forme

$$\begin{array}{lll} v_1, \cdots, v_n & := & e_{11}, \cdots, e_{1n} \quad if\ g_1 \quad \sim \\ & & e_{21}, \cdots, e_{2n} \quad if\ g_2 \quad \sim \\ & & \cdots \qquad\qquad\quad \cdots \\ & & e_{m1}, \cdots, e_{mn} \quad if\ g_m \end{array}$$

A la différence du langage de *commandes gardées* de **Dijkstra**, les valeurs affectées doivent être *identiques* dans le cas de plusieurs gardes simultanément vraies. Chaque affectation est donc *déterministe*. L'instruction est sans effet (*nop*) dans le cas où aucune garde n'est valide.

A ce format d'instructions, le langage ajoute quelques facilités d'écriture :

- **affectations synchrones**
  Le symbole $\|$ permet de séparer plusieurs parties d'une même affectation. Par exemple,

  $$x := a \parallel y, z := b, c \parallel t := d$$

  équivaut à

  $$x, y, z, t := a, b, c, d$$

- **quantification sur les expressions**
  Des opérateurs binaires associatifs et commutatifs peuvent être quantifiés sur un nombre *fini* de valeurs. La syntaxe est la suivante :

  $$\langle op\ x : cond :: exp \rangle$$

  Une telle expression réalise l'application de *op* sur toutes les expressions *exp* pour lesquelles x vérifie la condition *cond*. Si *cond* définit un ensemble vide de variation, la valeur de l'expression quantifiée est l'élément neutre de l'opérateur. Les opérateurs les plus couramment utilisés sont $+, *, min, max, \vee, \wedge$ et $\equiv$, de neutres respectifs $0, 1, +\infty, -\infty, false, true$ et $true$. Par exemple,

  $$\langle +\ i : 0 \leq i \leq N \wedge A[i] \geq 0 :: 1 \rangle$$

  représente le nombre d'éléments positifs du tableau A et

  $$\langle *\ i : 1 \leq i \leq N :: i \rangle$$

  est $N!$.

Dans ce dernier cas, l'environnement se charge de vérifier que les transformations proposées sont effectivement des raffinements, c'est à dire qu'elles maintiennent la cohérence entre le programme et sa spécification.

L'environnement développé se limite dans un premier temps à cette seconde fonctionnalité. Il s'agit donc de prouver de manière automatique qu'un programme donné vérifie bien toutes les contraintes imposées par sa spécification. Nous considérons, pour commencer, les seules contraintes de *sûreté*.

II. Le formalisme `UNITY`

II.1. Introduction

Au cours de leur développement, les algorithmes considérés sont vus à la fois sous un aspect centralisé et sous un aspect distribué, le problème posé étant le passage de l'un à l'autre. Ils doivent donc être exprimés dans un langage général unifiant ces deux aspects. Le formalisme `UNITY` [CM88] offre cette possibilité, puisqu'un algorithme peut être transformé *au sein de ce formalisme* vers une expression favorisant une implémentation répartie. En effet, un aspect important de cette approche est l'indépendance vis-à-vis de toute architecture matérielle, qu'elle soit séquentielle, parallèle synchrone ou parallèle asynchrone.

Par ailleurs, au modèle `UNITY` est associée une logique permettant à la fois d'exprimer une spécification et de valider un programme vis-à-vis de celle-ci. Pour ces deux raisons, ce formalisme a été choisi comme base pour la réalisation d'un environnement d'assistance au raffinement de programmes répartis.

II.2. Les instructions `UNITY`

Un programme `UNITY` est composé de *quatre* sections :

declare

Il s'agit d'un ensemble de déclarations de variables tel qu'on peut le trouver dans les langages de programmation classiques. Les types de base utilisés sont les *entiers* et les *booléens.* Sont également utilisés les *tableaux, ensembles* et *suites* (ou *listes*) de tels types.

initially

Cette partie concerne l'initialisation des variables. Cette initialisation est soumise à une contrainte de non-circularité (*proper set of equations*) spécifiée comme suit :

- Une variable est initialisée au plus une fois avec une expression de son type.
- Il existe un ordre d'exécution de ces initialisations tel qu'une variable n'est initialisée qu'à partir de constantes ou de variables précédemment initialisées.

always

La clause **always** d'un programme `UNITY` est une suite de *macro-définitions* liant les variables du programme. Les variables transparentes (*macros*) ainsi définies sont soumises aux mêmes contraintes que les variables initialisées de **initially**.

Laboratoire d'Informatique et Mathématiques Appliquées, ENSEEIHT/IRIT — 15 juin 1993

Assistance au Raffinement de Programmes Répartis

Michel CHARPENTIER

IRIT/ENSEEIHT,
2, rue Charles Camichel,
31071 TOULOUSE,
FRANCE,
charpov@enseeiht.fr

Résumé :

Dans le contexte d'une conception par raffinements successifs, nous nous intéressons à la réalisation d'un environnement d'assistance au développement de programmes répartis. Basé sur le formalisme **UNITY**, *ce système s'attache au maintien de la cohérence entre un programme et sa spécification. Il réalise pour cela une traduction des contraintes de la spécification en logique des prédicats et fait usage d'un démonstrateur automatique de théorèmes pour la gestion des formules obtenues.*

I. Thème de recherche

Le développement de programmes parallèles met en évidence les difficultés d'une mise au point incrémentale au moyen de tests. Les contraintes étant reportées sur la conception, les méthodes de dérivation et raffinement de spécifications formelles constituent alors un moyen fiable pour garantir la correction d'un algorithme.

Leur principe est de formaliser la spécification d'un programme (généralement à l'aide de la logique mathématique) et de dériver de cette spécification de manière rigoureuse, par raffinages successifs, une suite de programmes satisfaisant tous la spécification dont le dernier est facilement implémentable [Dro89].

Dans le cadre de l'algorithmique répartie, ce processus peut être utilisé pour transformer un algorithme conçu indépendamment de toute considération d'architecture en un équivalent distribué. De telles transformations sont délicates et ne peuvent être menées à bien de manière entièrement automatique.

Deux principes peuvent être utilisés pour surmonter ces difficultés :

- l'étude et la mise en place d'heuristiques générales pour la répartition [FMP92]
- l'utilisation d'un environnement de développement assisté intégrant les heuristiques précédentes, mais offrant en plus la possibilité d'une intervention humaine pour pallier leurs insuffisances

Le travail présenté, intégré au projet ADER (*Aide au Développement Raisonné*) proposé dans le cadre du GRECO de programmation, s'attache plus spécifiquement au deuxième point. Son objectif est la réalisation d'un environnement de programmation qui prenne en charge à la fois un programme et sa spécification [Cha93].

Le rôle d'un tel environnement est double : il doit être capable d'appliquer à un programme donné un jeu d'heuristiques internes, validées par ailleurs, mais permettre également à un utilisateur de tester des transformations nouvelles.