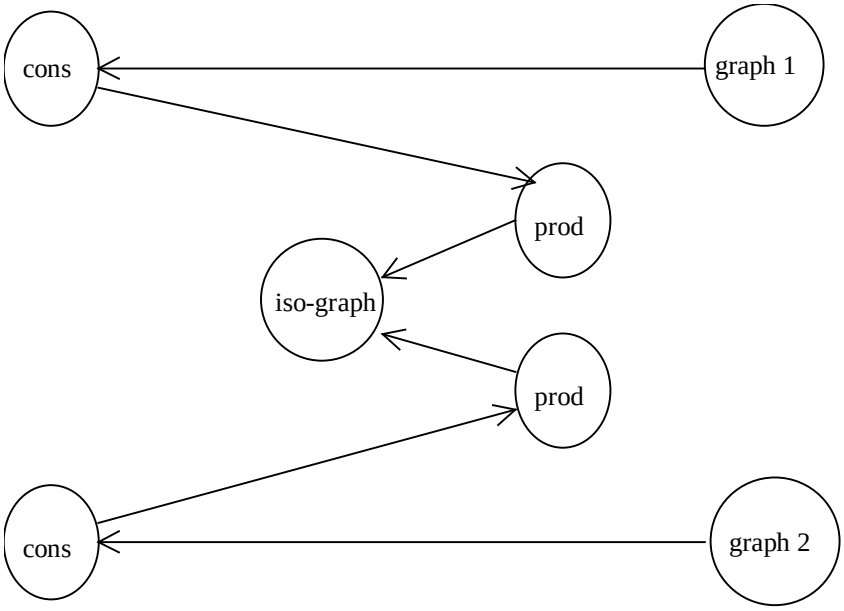
CCP language expressiveness – mechanisms available
(Chapter 6.5.6 of Ph.D.Thesis)

Taking into account the language specification [Mur98] and chapters 6.5.2 to 6.5.5 of the thesis, some observations can be made. First, the language subsumes the Prolog, GHC, Parlog and CP languages. This means that most of the programs written in those languages may be translated into CCP. The feature is similar to those of AKP [Har90] and therefore paradigms present in different domains are available in a single language.

Before presenting the mechanisms available, we'll give a complete program written in CCP, together with the translation in P_f, which emphasise the strength and complexity of the language.

A program that determines graph isomorphism (with default reasoning) is:

```
:-ensure_loaded('trad_ccp').
:-dynamic prod/2,cons/2,goi/2,go/2,decl_ccp/1,p/2,delete/3,perm_eq/5,
graph_iso/4, eq_edge/4, eq_vertex/4, graph1/1,graph2/1.
decl_ccp([prod/2,cons/2,delete/3,perm_eq/5,graph_iso/4,eq_edge/4,eq_v
ertex/4]).

delete(H,[H+|T],T):-true;.
delete(X,[H+|T],[H|Td]):-delete(X,T,Td);.

perm_eq([]+,[]+,_,L,L):-true;.
perm_eq([H1+|T1],L2+,EQ,Li,Lo):-
            delete(H2,L2,T2),P =.. [EQ,H1,H2,Li,L],call(P),
            perm_eq(T1,T2,EQ,L,Lo);.

graph_iso(L1,L2,Li,Lo):-perm_eq(L1,L2,eq_edge,Li,Lo).

eq_edge(v(N1+,L1),v(N2+,L2),Li,Lo):-
            eq_vertex(N1,N2,Li,L),perm_eq(L1,L2,eq_vertex,L,Lo);.

eq_vertex(N1+,N2+,L,L):- p(N1,N2) ! true;.
eq_vertex(N1+,_,_,_):- p(N1,_) ! fail.
eq_vertex(_,N2+,_,_):- p(_,N2) ! fail;.
eq_vertex(N1+,N2+,[(N1,N2)|T],T):- asserta(p(N1,N2));.
eq_vertex(N1+,N2+,_,_):- retract(p(N1,N2)) ! fail.

graph1([v(a,[b,c,d]),v(b,[a,c]),v(c,[a,b,d]),v(d,[a,c])]).
graph2([v(1,[2,4]),v(2,[1,3,4]),v(3,[2,4]),v(4,[1,2,3])]).

prod([],[]+):-true;.
prod([Hi|To],[Hi+|Ti]):-prod(To,Ti);.

cons([]+,[]):-true;.
cons([Hi+|Ti]+,[Hi|To]):-cons(Ti,To);.

go(Li,Lo):- graph1(L1),graph2(L2),
            graph_iso(LL1,LL2,Li,Lo),
            cons(L1,LL1),cons(L2,LL2).
goi(Li,Lo):-cons(L1,LL1),cons(L2,LL2),
            graph_iso(Lg1,Lg2,Li,Lo),
            prod(Lg1,LL1),prod(Lg2,LL2),
            graph1(L1),graph2(L2).
```



The picture represents the graph of goals execution at call. The reactive behaviour may be observed, the goals being chosen from left to right (at call). The arrows represent the direction of arguments passing.
The translated code (program in P_f) is:

```
delete(A, B, C) :-
            set_guard(D),
            freeze(B, user:(B=[E|_]->freeze(E,(E=_->
               freeze(off,(delete(A,B,C,1,D);off=F));off=F));off=F)),
            freeze(B, user:(B=[G|_]->freeze(G,(G=_->
               freeze(F,(delete(A,B,C,2,D);F=on));F=on));F=on)),
            true.

delete(A, [A|B], B, 1, C) :- active_guard(C), true.
delete(A, [B|C], [B|D], 2, E) :- active_guard(E),delete(A, C, D).

perm_eq(A, B, C, D, E) :- set_guard(F),
      freeze(A, user:(A=[]->freeze(B,(B=[] ->
      freeze(off,perm_eq(A,B,C,D,E,1,F);off=G));off=G)); off=G)),
      freeze(A,user:(A=[H|_]->
      freeze(H,(H=_->freeze(B,(B=_->freeze(G,(perm_eq(A,B,C,D,E,2,F);
      G=on)) ; G=on));G=on));G=on)),
      true.

perm_eq([], [], _, A, A, 1, B) :- active_guard(B),true.
perm_eq([A|B], C, D, E, F, 2, G) :- active_guard(G),
                              delete(H, C, I), J=..[D,A,H,E,K],
                              call(user:J), perm_eq(B, I, D, K, F).

graph_iso(A, B, C, D) :-      set_guard(E),
            freeze(off,user:(graph_iso(A,B,C,D,1,E);off=on)),true.

graph_iso(A, B, C, D, 1, E) :-      reset_guard(E), true,
                              perm_eq(A, B, eq_edge, C, D).

eq_edge(A, B, C, D) :-
```

```
      set_guard(E),
      freeze(A, user:(A=v(F,_)->freeze(F,(F=_->
      freeze(B,(B=v(G,_)->freeze(G,(G=_->
      freeze(off,(eq_edge(A,B,C,D,1,E) ;
      off=on));off=on));off=on));off=on));off=on)),
      true.

eq_edge(v(A,B), v(C,D), E, F, 1, G) :-
        active_guard(G),
        eq_vertex(A, C, E, H),
        perm_eq(B, D, eq_vertex, H, F).

eq_vertex(A, B, C, D) :-
        set_guard(E),
      freeze(A, user:(A=_->freeze(B,(B=_->
       freeze(off,(eq_vertex(A,B,C,D,1,E);off=F));off=F));off=F)),
      freeze(A, user:(A=_->freeze(F,
       (eq_vertex(A,B,C,D,2,E);F=G));F=G)),
      freeze(B, user:(B=_->freeze(F,
       (eq_vertex(A,B,C,D,3,E);G=H));G=H)),
      freeze(A, user:(A=_->freeze(B,(B=_->
       freeze(H,(eq_vertex(A,B,C,D,4,E);H=I));H=I));H=I)),
      freeze(A, user:(A=_->freeze(B,(B=_->
       freeze(I,(eq_vertex(A,B,C,D,5,E);I=on));I=on));I=on)),
      true.

eq_vertex(A, B, C, C, 1, D) :-
           (active_guard(D),p(A,B)!reset_guard(D),true,true).
eq_vertex(A, _, _, _, 2, B) :-
           (active_guard(B),p(A,_)!reset_guard(B),true,fail).
eq_vertex(_, A, _, _, 3, B) :-
      (active_guard(B),p(_,A)!reset_guard(B),true,fail).
eq_vertex(A, B, [(A,B)|C], C, 4, D) :-
           (active_guard(D), asserta(user:p(A,B)).
eq_vertex(A, B, _, _, 5, C) :-(active_guard(C),retract(p(A,B))!
           reset_guard(C),true,fail).

graph1([v(a,[b,c,d]),v(b,[a,c]),v(c,[a,b,d]),v(d,[a,c])]).

graph2([v(1,[2,4]),v(2,[1,3,4]),v(3,[2,4]),v(4,[1,2,3])]).

prod(A, B) :-
      set_guard(C),
      freeze(B, user:(B=[]->
            freeze(off,(prod(A,B,1,C);off=D));off=D)),
      freeze(B, user:(B=[E|_]->
      freeze(E,(E=_->freeze(D,(prod(A,B,2,C);D=on));D=on));D=on)),
      true.

prod([], [], 1, A) :- active_guard(A), true.
prod([A|B], [A|C], 2, D) :- active_guard(D),prod(B, C).

cons(A, B) :-
        set_guard(C),
freeze(A, user:(A=[]->freeze(off,(cons(A,B,1,C);off=D));off=D)),
freeze(A, user:(A=[E|_]->
      freeze(E,(E=_->freeze(D,(cons(A,B,2,C);D=on));D=on));D=on)),
true.

cons([], [], 1, A) :- active_guard(A),true.
cons([A|B], [A|C], 2, D) :-active_guard(D),cons(B, C).
```

```
go(A, B) :-graph1(C), graph2(D),
           graph_iso(E, F, A, B),cons(C, E),cons(D, F).
goi(A, B) :-cons(C, D),cons(E, F),
           graph_iso(G, H, A, B),
           prod(G, D),prod(H, F),graph1(C),graph2(E).
```

At execution, the following results are provided:

```
Lgo=[[(d,3),(b,1),(a,2),(c,4)],
      [(b,1),(a,4),(c,2),(d,3)],
      [(b,3),(d,1),(a,2),(c,4)],
      [(b,3),(d,1),(a,4),(c,2)]     ]
Lgoi=[[(d,3),(b,1),(a,2),(c,4)],
      [(b,1),(a,4),(c,2),(d,3)],
      [(b,3),(d,1),(a,2),(c,4)],
      [(b,3),(d,1),(a,4),(c,2)]     ]
```

The complexity of the translated code, which is very difficult to be written by the user, must be noticed.

Next we present the CCP mechanisms available, by comparing them with those in committed choice languages like Parlog (which were presented in chapter 4.5.2 of the thesis, and were executed in our own implementation of Parlog).

1. Prolog *incompleteness*, due to the unfair depth-first search, may be *removed* because of the concurrency mechanisms. By trying concurrently many alternatives (OR parallel clauses), the existence of a solution is found even if there exist some infinite branches (in the left part of the execution tree, which means that in a sequential depth first search, like in pure Prolog, the other ones, if any, are never found). The example given here is for a nondeterministic search in a graph (the same example was run also in Parlog. It's execution in Prolog enters in infinite branch). The sequence that performs the search is:

```
path(N+,N+,[N]).
path(S+,E+,[S|R]):- neighbours(S,Ns), one_way(Ns,E,R).

one_way([N|Ns]+,E+,P):- one_way(Ns,E,P) ! true.
one_way([N|Ns]+,E+,P):-path(N,E,P) ! true.
```

2. The *stream* processing is realised by synchronisation through input variable annotation (+). The process that uses the stream cannot work ahead. It has to wait for the instantiation of the annotated arguments. The example written in Parlog, in CCP is:

```
term_processing(X+):-processing(X).
stream_processing([]+).
stream_processing([Element+|Rest_stream]):-
            term_processing(Element),
            stream_processing(Rest_stream).
```

The `stream_processing` process has all the necessary synchronisation mechanisms inside; therefore it is called concurrently with the other processes which share the variable (stream).

3. Another mechanism introduced by the concurrency is the *logic variable as communication channel*. The processes that share a variable may communicate through it (the mechanism is present in a set of examples that explain other paradigms). We exemplify with the sequence of two processes that try to instantiate the same variable (output in both processes) in order to find out if the profile of two trees is the same. The nodes of the trees are added (in inorder) in the same list. The process that is more eager adds the node, and the lazier one verifies. Thus, the CW (with policy concurrent write at equal) access to shared memory is implemented [AKL89]. During the execution the processes may change their place, depending on their relative speed. The execution ends successfully if both trees were completely traversed. Otherwise, the execution fails in the moment the first different node is found.

```
      same_profile(T1+,T2+,L):-wl(L),profile(0,T1,L),profile(0,T2,L),
c(L).

      profile(I+,nil+,[]).
      profile(I+,t(nil+,N,Dr),[m(N,R)|T]):-profile(R,Dr,T).
      profile(I+,t(t(SS,NS,DS)+,N,D),L):-
                        profile(I,t(SS,NS,t(DS,N,D)),L).
      c([]+).
      c([m(N+,R)|T]):-R=N,c(T).

      a1(t(t(t(nil,1,nil),2,t(nil,3,nil)),4,nil)).
      a2(t(nil,1,t(t(nil,2,t(nil,3,nil)),4,nil))).
      a3(t(nil,1,t(t(nil,2,nil),4,t(nil,5,nil)))).

      wl([m(H,R+)|T]):-write(m(H,R)),wl(T).
      wl([]+).
```

A succesful execution is:

```
t(X,Y,L):-a2(Y),a1(X),same_profile(X,Y,L).
```

the response being:

```
m(1,1)m(2,2)m(3,3)m(4,4)
Lt=[[m(1,1),m(2,2),m(3,3),m(4,4)]]
```

And a failed execution is:

```
f(X,Y,L):-a3(Y),write(Y),nl,a1(X),same_profile(X,Y,L).
```

with the response:

```
t(nil,1,t(t(nil,2,nil),4,t(nil,5,nil)))
m(1,1)m(2,2)
Lf=[]
```

The response lists represent the common profile of the trees. In the second execution, the list is empty, the profile being different.

4. *The fair communication* between processes, with fair resources access is ensured by the balanced merging of data that come at/leave a process and has a single destination/source.

The fair merge mechanism, which takes the data in the first moment it is present is represented by the following sequence (and corresponds to the one in Parlog in chapter 5.4.2).

```
fair_merge([H1+|T1],[H2+|T2],[H1,H2|Rest]):-
            fair_merge(T1,T2,Rest).
fair_merge([H1+|T1],L2,[H1|Rest]):-
            var(L2):fair_merge(T1,L2,Rest).
fair_merge(L1,[H2+|T2],[H2|Rest]):-
            var(L1):fair_merge(L1,T2,Rest).
fair_merge([]+,L,L).
fair_merge(L,[]+,L).
```

The reverse operation is resolved by the fair split:

```
fair_split([]+,[],[]).
fair_split([H+|Rest],[H|T1],L2):-fair_split(Rest,L2,T1).
```

The same mechanism may be used to merge (split) as many streams as necessary.

5. The one way producer consumer communication is the simplest way of communication between processes that share a variable (the communication channel). The producer process sends information to the consumer, in this way influencing its behaviour. The producer acts independently, the communication being one-way.

```
correct(107).
consum([I+|Resti]):-    correct(I) ! true.
consum([I+|RestI]):-    not(correct(I))!
                        consum(Resti).
prod(Info):-integer(0,1000,Info).
integer(I+,I+,[I]).
integer(I1+,I2+,[I1|Resti]):- I1<I2  !
                        In is I1+1 , integer(In,I2,Resti).
```

In a similar manner may be also described the two-way synchronisation.

6. A more complex example, which suggests the language strength, is the *incomplete messages technique.* By using this technique, two (or many) processes interact, the communication being performed by a single communication channel – a single shared variable.

The channel is an output variable for the producer and an input one for the consumer. The messages passing the channel are streams, a unit element of the stream being now a compound term, not a simple variable. In such a term, together with the useful information, the producer also sends a response variable. After sending a first message on the channel, the producer suspends, waiting for the instantiation of the response variable given by the consumer. This argument is input annotated at the consumer. The way the producer continues the execution according to the response given by the consumer (the instantiation of the response variable). Immediately after the consumer has computed the response for the current element of the stream, the value “is seen” at the producer, which decides to stop or the way to continue the execution.

Complex mechanisms of communication processes may be implemented by using this technique. But it is important to understand the mechanism very well, otherwise the deadlock can occur (because the safe guard property is not respected).

```
consum([m(N+,R)|RestI]):- correct(N) ! R=yes, stop(RestI).
consum([m(N+,R)|RestI]):- high(N) ! R=high, consum(RestI).
consum([m(N+,R)|RestI]):- mic(N) ! R=mic, consum(RestI).

stop([]+).
stop([m(X+,R)|RestI]):- R=stop,stop(RestI).

correct(107).
high(N):-N>107.
mic(N):-N<107.

prod1(X+,[m(100,R)|RestI]):- sync_p(i(50,100,150),RestI,R).
prod2(X+,[m(100,R)|RestI]):- sync_p(i(90,100,110),RestI,R).

sync_p(i(Min,G,Max),[],stop+).
sync_p(i(Min,G,Max),[],yes+).
sync_p(i(St,N,Dr),[m(Nu,R)|RestI],high+):-
            Dr1 is N-1, Nu is (St+Dr1)//2,

      sync_p(i(St,Nu,Dr1),RestI,R).
sync_p(i(St,N,Dr),[m(Nu,R)|RestI],mic+):-
            St1 is N+1, Nu is (St1+Dr)//2,
            sync_p(i(St1,Nu,Dr),RestI,R).
```

A goal in the following form will execute such a mechanism.

```
gow1(N):-swritelist(N),(prod1(N);prod2(N)),consum(N).
```

7. All previous communication models are for eager producers, meaning that the communication begins (and ends) because of the producer, and all interaction between processes is accomplished only through the communication channel. If the other process (the consumer) drives the interaction, the progress is achieved according to its action. Thus, the mechanism is one of the lazy producer – consumer type. The mechanism description of such a communication is given in chapter 5.4.2 (example 8 in Parlog).

```
consuml([m(X+,Y+,_)|R]):-
            correctX(X),correctY(Y) ! R=[m(X,Y,yes)].
consuml([m(X+,Y+,_)|RestC]):-
            correctX(X), \+ correctY(Y) !

RestC=[m(X,NY,yes_abs)|R],consuml(RestC).
consuml([m(X+,Y+,_)|RestC]):-
            \+ correctX(X),correctY(Y) !
      RestC=[m(NX,Y,yes_ord)|R],consuml(RestC).
consuml([m(X+,Y+,_)|RestC]):-
            \+ correctX(X), \+ correctY(Y) !
                  RestC=[m(NX,NY,no)|R],consuml(RestC).
correctX(15).
correctY(3).

prodX(C):-lazy_intX(10,20,C).
```

```
prodY(C):-lazy_intY(0,10,C).

lazy_intX(_,_,[]+).
lazy_intX(_,_,[m(_,yes_abs+)|_]).
lazy_intX(_,_,[m(_,yes+)|_]).
lazy_intX(N1,N2,[m(X,yes_ord+)|RestC]):-
            N1<N2,X=N1,N is N1+1, lazy_intX(N,N2,RestC).
lazy_intX(N1,N2,[m(X,no+)|RestC]):-
            N1<N2,X=N1,N is N1+1, lazy_intX(N,N2,RestC).

lazy_intY(_,_,[]+).
lazy_intY(_,_,[m(_,yes+)|_]).
lazy_intY(_,_,[m(_,yes_ord+)|_]).
lazy_intY(N1,N2,[m(Y,yes_abs+)|RestC]):-
             N1<N2,Y=N1,N is N1+1, lazy_intY(N,N2,RestC).
lazy_intY(N1,N2,[m(Y,no+)|RestC]):-
             N1<N2,Y=N1,N is N1+1, lazy_intY(N,N2,RestC).
```

The execution is performed by a goal:

```
go(C):-     C=[m(X,Y,nu)|R],consuml(C),
            split(C,C1,C2),prodX(C1),prodY(C2).
```

With the response:

```
Lgo=[[m(10,0,no),m(11,1,no),m(12,2,no),m(13,3,no),m(14,3,yes_ord),m(1
5,3,yes_ord),m(15,3, yes)]]
```

or:

```
      goi(C):-    prodX(C1),prodY(C2),
                  C=[m(X,Y,no)|R],consuml(C),split(C,C1,C2).
Lgoi=[[m(10,0,no),m(11,1,no),m(12,2,no),m(13,3,no),m(14,3,yes_ord),m(
15,3,yes_ord), m(15,3,yes)]];

      gow(C):-    swritelist(C),C=[m(X,Y,no)|R],
                  consuml(C),split(C,C1,C2),prodX(C1),prodY(C2).
      m(10,0,no)m(11,1,no)m(12,2,no)m(13,3,no)m(14,3,yes_ord)m(15,3,y
es_ord)m(15,3,yes)
Lgow=[[m(10,0,no),m(11,1,no),m(12,2,no),m(13,3,no),m(14,3,yes_ord),m(
15,3,yes_ord),m  (15,3, yes)]]
```

8. The implementation of *data-flow algorithms* is another important feature of the language. As an example we have chosen the data-flow model of a systolic architecture, because this allow us to strength two main aspects: modelling a systolic architecture itself and pointing out the common points and the differences between the two implementations. Such an analysis was implemented as well in our own Parlog and presented in chapter 4.5.2 (example 7, for an architecture that computes the product between a matrix and a vector).

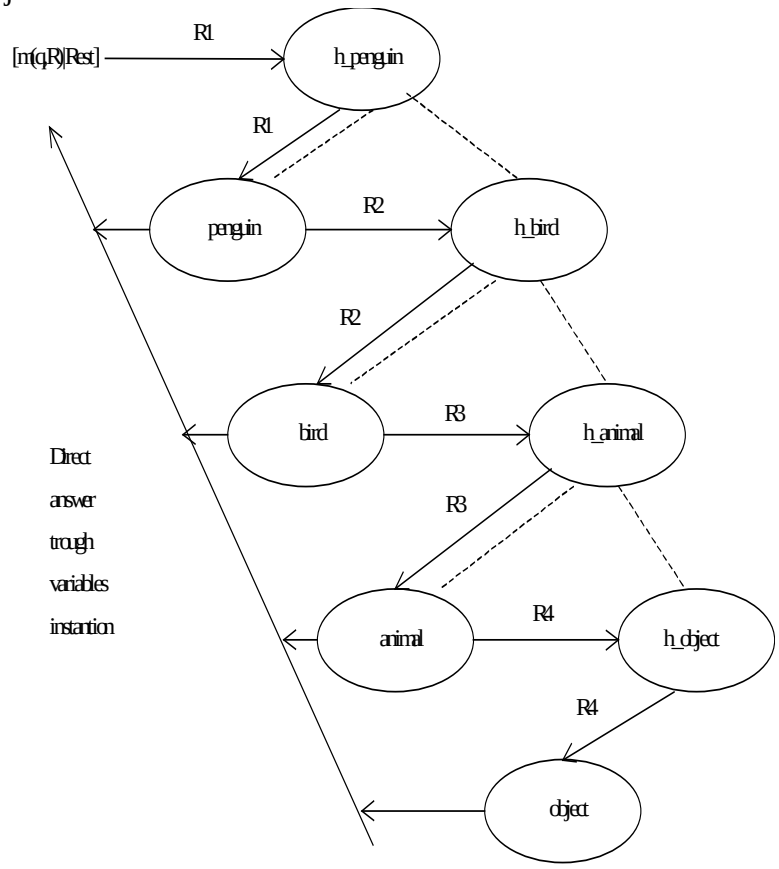
```
mul_mat([]+,_+,[]).
mul_mat([Ai+|RAi+],B+,[Ci|RCi]):-
      vect_mat(Ai,B,Ci),mul_mat(RAi,B,RCi).
vect_mat(_+,[]+,[]).
vect_mat(Ai+,[Bk+|RBk+],[Cik|RCik]):-
      vect_mul(Ai,Bk,Cik),vect_mat(Ai,RBk,RCik).
```

```
vect_mul([]+,[]+,0).
vect_mul([Aij+|RAij+],[Bjk+|RBjk+],Cik):-
      inst(Aij,Bjk,Sum,Cik),vect_mul(RAij,RBjk,Sum).

inst(X+,Y+,Z+,R):-R is X*Y+Z.
```

The execution is realised by a goal of the form:

```
test(R):-mul_mat([[1,2],[3,4]],[[5,6],[7,8]],R).
```

9. *Objects* are implemented as processes that take a stream of demands, which identifies the objects. The associated information is kept as process arguments. The object definition is in fact a clause that makes a query. This way, clauses correspond to methods while individual subgoals correspond to instances. An example that implements objects is:




```
      h_penguin(S):-penguin(S,L),h_bird(L).
      h_bird(S):-bird(S,L),h_animal(L).
      h_animal(S):-animal(S,L),h_object(L).
      h_object(S):-object(S).

object([]+).
object([m(_+,undif)|T]):-object(T).

animal([]+,[]).
animal([m(mod_h+,without_f)|T],L):-
      animal(T,L);.
animal([M|T]+,[M|T1]):-animal(T,T1).

bird([]+,[]).
bird([m(movement+,zboara)|T],L):-bird(T,L).
```

```
      bird([m(cover+,pene)|T],L):-bird(T,L);.
bird([M|T]+,[M|T1]):-bird(T,T1).

penguin([]+,[]).
penguin([m(movement+,paseste)|T],L):-
      penguin(T,L);.
penguin([M|T]+,[M|T1]):-penguin(T,T1).

      test(X,Y,Z,T):-
         h_penguin([m(movement,X),m(cover,Y),m(mod_h,Z),m(a,T)]).
```

6.6 Interpretation of the results

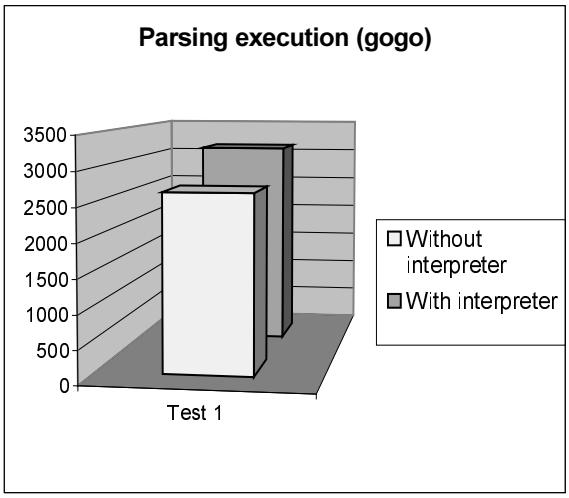
This chapter concludes the implementation by presenting the execution times for the programs that represent the benchmark set for CCP language validation. It presents both the execution times for direct execution (e.g. in SICStus Prolog, by using the built in freeze) and for the execution with our proposed meta-interpreter. This comparison is not only a simple validation of the meta-interpreter but also allows a classification of benchmark tests according to the ratio of execution times using the alternatives. The differences are due to the suspended processes queue manipulation, which is a sequential one for the meta-interpreter and has direct access in Prolog with freeze. Moreover, the differences express the length of suspended processes queue during the execution. For those tests with little execution time differences, a small suspension signifies a superficial processes interaction, so a small queue. The examples with big execution times differences are explained by a large dimension of the suspended processes queue, corresponding to a deep interaction between processes. This aspect may also be noticed in the data-flow representation of processes interaction. Following, we include the tests without/with the meta-interpreter for the presented situations, together with the data-flow diagram in the cases in which it is representative.

1.Data-flow parallel parsing [Con89]

The test is obtained for a call:

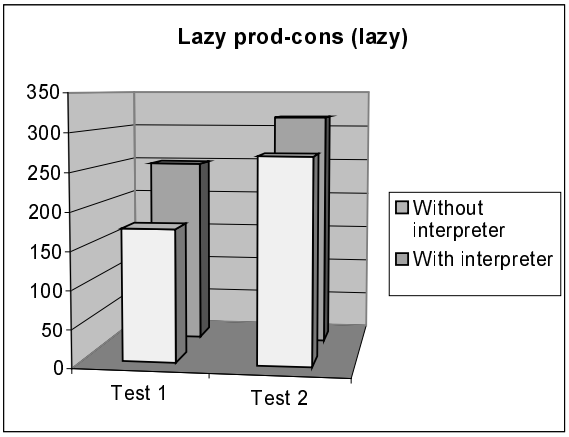
```
test4(R):-p_lhs([this,nice,boy,sees,a,long_hair
,dog],s,R).
```

and the diagram is:



2. Producer-consummer

Lazy producer consumer interaction, with a single communication channel (weak interaction). The consumer request is split between the two producers.



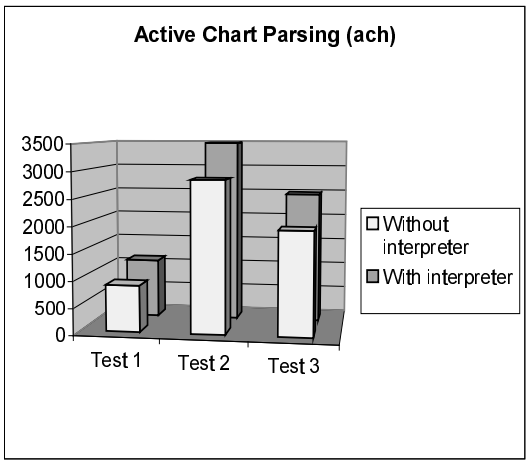
Test1 for the query:

```
go(C):-consuml(C),split(C,C1,C2),prod1(C1),
prod2(C2).
```

Test2 for the query:

```
gow(C):-swritelist(C),consuml(C),split(C,C1,C2),prod1(C2),prod2(C1).
```

3. Active charts parsing [Gaz89]



Test1 for the query:

```
p1:-parse([this,boy,sees]).
```

Test2 for the query:

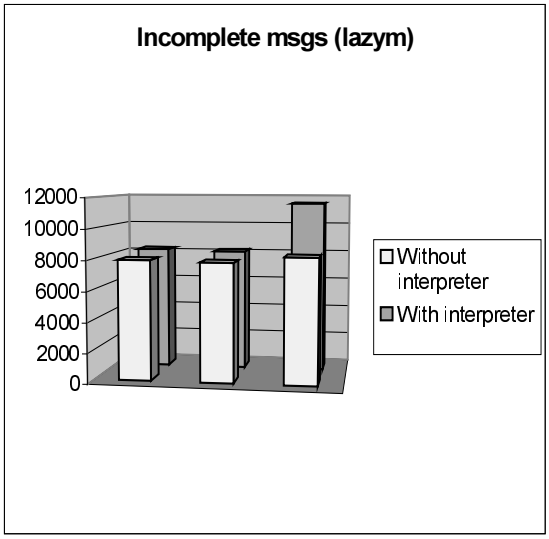
```
p5:-parse([this,nice,boy,sees,a,long_hair,dog])
.
```

Test3 for the query:

```
p4:-parse([this,nice,boy,sees,a,dog ]).
```

4. Incomplete messages technique

The communication between two lazy producers with incomplete message technique is tested below. The tests were performed for the following queries.



Test1:

```
go(C):-C=[m(X,Y,no)|R],consuml(C),
split(C,C1,C2),prodX(C1),prodY(C2).
```

Test2:

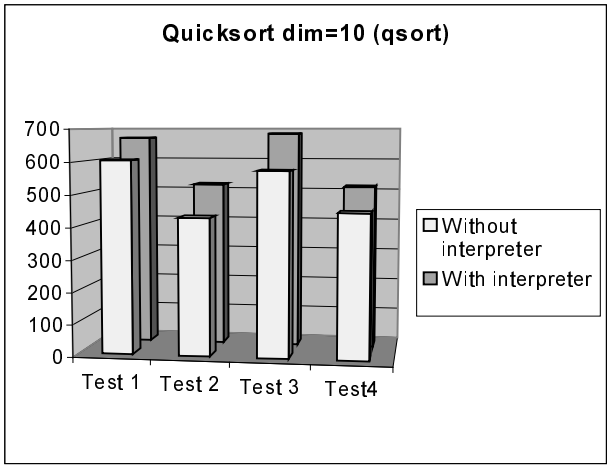
```
goi(C):-prodX(C1),prodY(C2),C=[m(X,Y
,no)|R], consuml(C),split(C,C1,C2).
```

Test3:

```
gow(C):-swritelist(C),C=[m(X,Y,no)|R],c
onsuml(C),split(C,C1,C2),prodX(C1),pro
dY(C2).
```

5. Quicksort

The tests have been performed for two input data dimensions and for programs with append (go tests) and with difference lists (god tests), writing or not the results during their computation. We may observe the significant growing of the suspended processes queue.



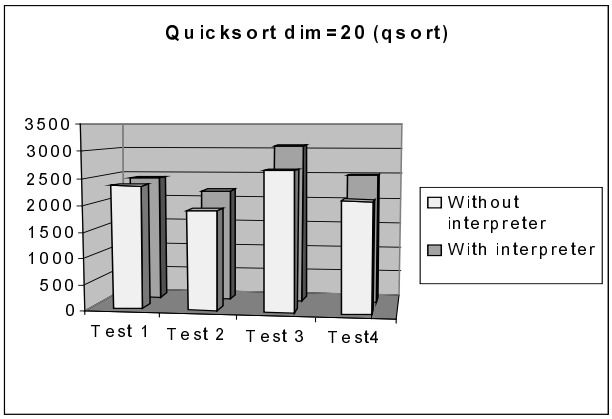
Test1:

```
go(R):-gen(L),qs(L,R).
```

Test2:
god(R):-gen(L),qsd(L,R,[]).

Test3:
gow(R):-swritelist(R),gen(L),qs(L,R).

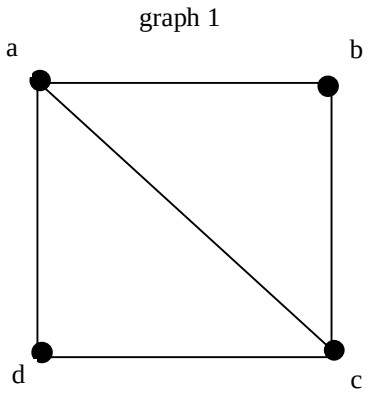
Test4:
godw(R):-swritelist(R),qsd(L),R,[]).

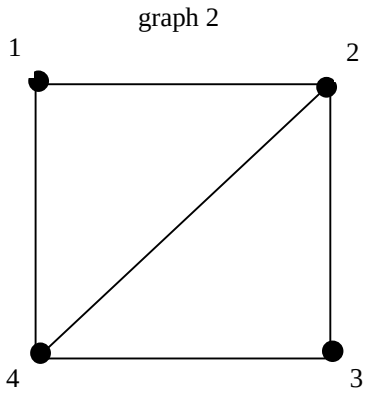


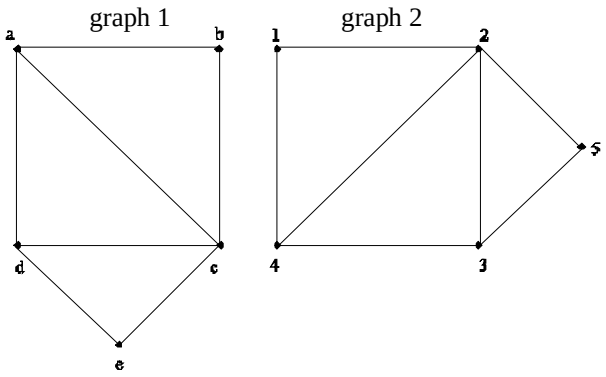
All previous examples have a supplementary execution time with 10% in the case of using the interpreter.

6. Graph isomorphism

The tests are made for the graphs given below and they correspond to the following query:

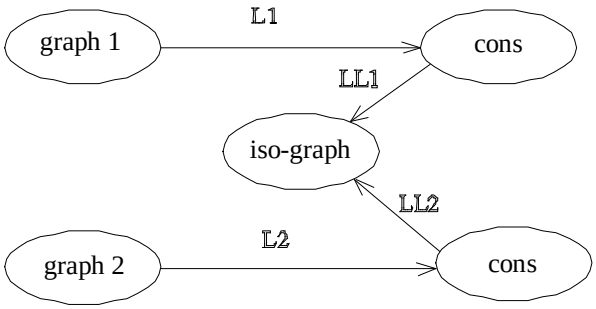






Test1:

```
go(Li,Lo):-retractall(p(_,_)),
        graph1(L1),graph2(L2),
        graph_iso(LL1,LL2,Li,Lo),
        cons(L1,LL1),cons(L2,LL2).
```

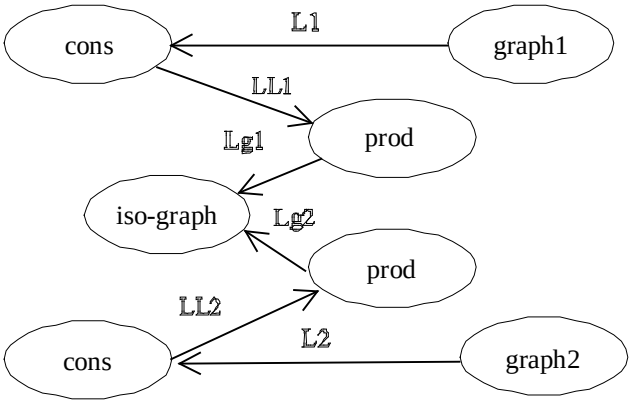


Test2:

```
goi(Li,Lo):-retractall(p(_,_)),
        cons(L1,LL1),cons(L2,LL2),
        graph_iso(Lg1,Lg2,Li,Lo),
        prod(Lg1,LL1),prod(Lg2,LL2),
        graph1(L1),graph2(L2).
```

Test3:

```
g(Li,Lo):-retractall(p(_,_)),
        graph_iso([v(a,[b,c]),v(b,[c,a]),
        v(c,[a,b])],[v(1,[2,3]),v(2,[3,1]),
        v(3,[1,2])],Li,Lo).
```

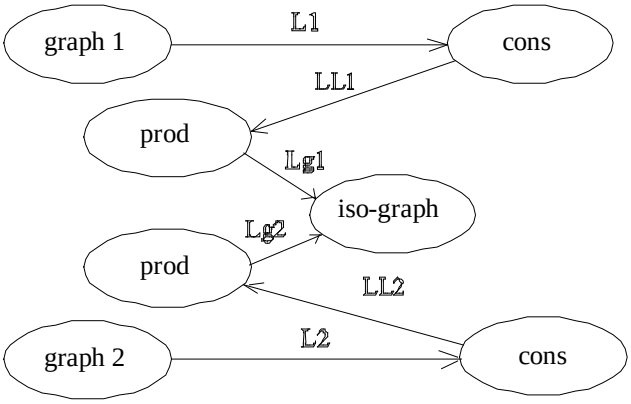


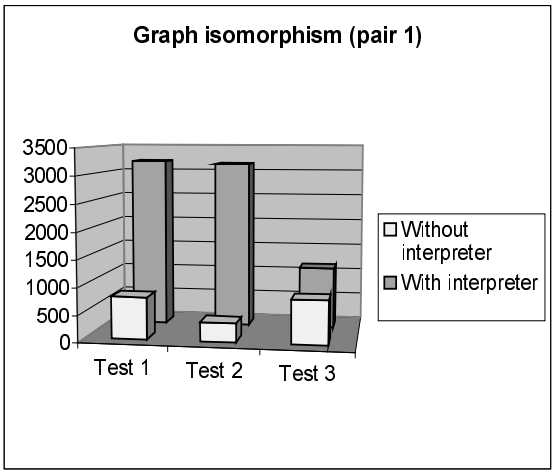
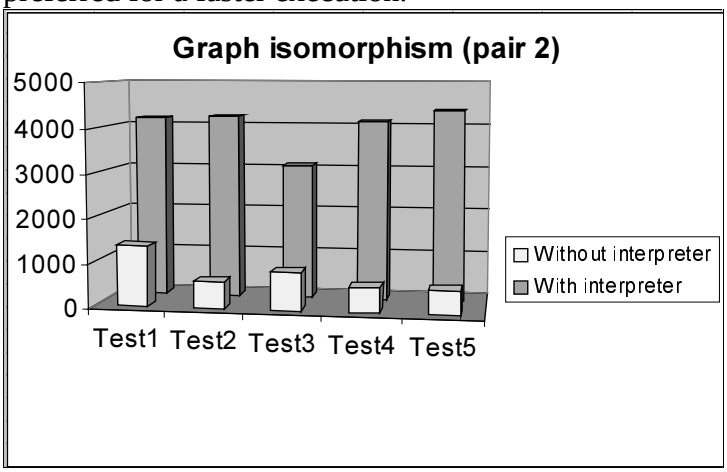
Test4:

```
goo(Li,Lo):-retractall(p(_,_)),
        graph1(L1),graph2(L2),
        prod(Lg1,Ll1),prod(Lg2,Ll2),
        graph_iso(Lg1,Lg2,Li,Lo),
        cons(L1,Ll1),cons(L2,Ll2).
```

Test5:

```
go(Li,Lo):-retractall(p(_,_)),
        graph1(L1),graph2(L2),
        graph_iso(LL1,LL2,Li,Lo),
        prod(LL1,L1),prod(LL2,L2).
```





These tests have greater execution times differences on the execution with and without the interpreter, due to the length of the queue. The third test has the minimum difference, the number of suspended processes being minim (graphs instantiation is direct without any delay). Differences in execution times between test 1 and test 5 are explained as follows: test 1 has a don't know nondeterminism search (backtracking, sequential clauses in cons) while test 5 has a don't care nondeterminism search (parallel clauses in prod). The conclusion is that in the case of disjoint alternative clauses, guarded clauses are preferred for a faster execution.