TR-631

Co-operative Hierarchical Layout Problem Solver on Parallel Inference Machine

by

T. Watanabe & K. Komatsu (Hitachi)

March, 1991

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Co-operative Hierarchical Layout Problem Solver on Parallel Inference Machine

Toshinori Watanabe and Keiko Komatsu

Systems Development Laboratory, Hitachi, Ltd.

1099 OHZENJI, ASAO-KU, KAWASAKI-SHI, KANAGAWA, 215 JAPAN

Tel: (044) 966-9111, Telex: 3842-577, Fax: (044) 966-6862

Co-HLEX is a co-operative hierarchical layout problem solver being developed by ICOT as an application program of parallel inference machines; Multi-PSI and PIM. The kernel of **co-HLEX** is a hierarchical recursive concurrent theorem prover for layout nicknamed **HRCTL**. Due to its recursive nature, **HRCTL** has a size of O(1,000) lines in KL1; the kernel language of ICOT. Due to its stream-parallel and distributed-memory architecture, $O(N^{0.8})$ time complexity could be attained. Moreover, due to the co-operations among concurrent processes, abutment of wires between circuit modules could be attained. Currently, **co-HLEX** spawns O(10,000) concurrent processes on 16-PE Multi-PSI illustrating the parallel processing power of the machine. Program maintainability in faces of rapid LSI technology innovations could be enhanced through modularized program descriptions in clausal forms. In this paper, current status of **co-HLEX** is reported.

1. INTRODUCTION

The principle of recursion; deriving a large complex whole from repeated applications of a few simple rules, had been and will be one of the most useful ideas in problem solving. For us with limited scope of thought, it is a powerful tool to save thoughts. We can understand the idea of infinity by way of recursive inductions only [1]. We can use it as a primitive element of computations [2]. Recently, researchers in fractal theory point out that the nature herself adopts the principle [3]. Despite these attractive features, it had been a minority in programming arts due to its run-time inefficiency and weak representation power. Many real world problems, which are more or less network typed, can not be represented in trees pertinent to the recursion formula. Important links among problem elements would be neglected.

This anomaly can be found in LSI layout problems. Divide-and-conquer for a tree typed or hierarchical circuit is the usual strategy, but important links among sub-circuits become neglected. As a rescue, abutment of sub-circuits; module shapes and wire connection points, should be made [4]. Otherwise, the chip area becomes large due to dead spaces.

Parallel logic programming languages and their processors give a new chance to the recursion principle. Elegant and short recursive layout program can now be processed efficiently. Co-operations among processes in forms of message passing and automatic suspension can compensate the missed links among sub-problems generated by recursive layout programs.

LSI layout programs are, as is well known, one of the most complex and large scaled softwares in our society. It can become as large as $O(10^6)$ steps in FORTRAN. In contrast to the rapid innovation of LSI technologies, that of software are very slow. To cope with technological innovations, layout programs are often rewritten using huge cost, time, and programmers' labour. New programming paradigms to enhance both man-man and man-machine communications are required by which traditional huge volumed documents and program

codes could be perished [5]. Decade ago, Kowalski and Shapiro pointed out that logic programming can be a candidate [6, 7].

Through the realization of **co-HLEX** on Multi-PSI and PIM, we hope to prove that logic can be an attractive journeying vehicle toward the solution of this software crisis in VLSI design automation.

2. BASIC CONCEPTS

2.1. Layout Problem

The original layout problem which **co-HLEX** solves can be specified by a goal:

```
:-mode solve_a_layoutproblem(+,-,+,+).
?-solve_a_layoutproblem(CirNet,LPlan,Proc,Constr).
```

where:

CirNet:=A network data denoting circuit modules and module connection nets.
LPlan:=Geometrical data representing module placements and connection wires among them.
Proc:=LSI fabrication process on which layout rules depends.
Constr:=Pairs of modules in placement. The topmost chip layout plan; the planned size of the chip and the external connector placements or pad placements are included.

2.2. Representations of Layout Plan

The *LayoutPlan* in the *solve_a_layoutproblem(...)* can take various forms depending on the fundamental layout model used by the problem solver. In **co-HLEX**, we adopted the classical quadtree representation proposed by Samet [9,10], Otten [11], and Luk [12]. Figure 2.1 gives their images.

LPlan:=[PQtree,Wires].
PQtree:=Quadtree representation of rectangular slices, each node of the tree carrying a module name placed in it, external connector names placed on the north, west, south, and east edges, etc.
Wires:=[Conns,Lines].
Conns:=Set of a list of terminal points, vias, and dummy connectors of a net. Dummy connector is placed at each wire crossing point on the edge of the slice.
Lines:=Set of lines spanning two connectors.

2.3. Constrained Recursive Layout Problem Solving

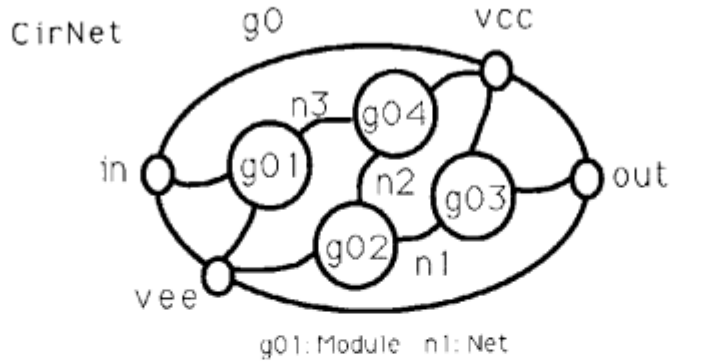
The slicing structure representation of layouts permits us both elegant and simple recursive formulation of the problem solver. An elegant but impractical prolog specification of the solver is:

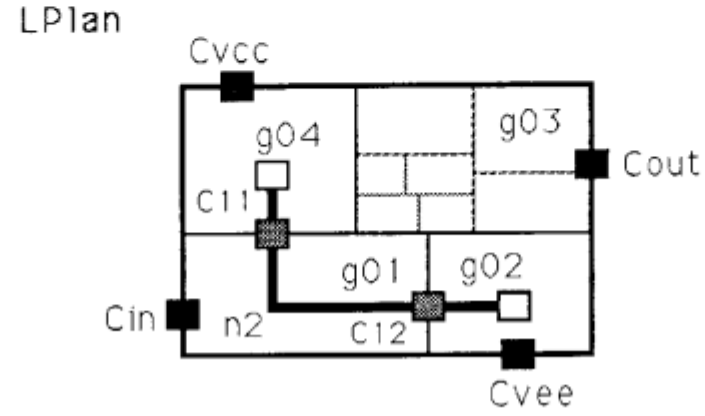
```
solve_a_layoutproblem(CirNet,LPlan,Proc,Constr):-
        leaf(CirNet),!,
        use_library(CirNet,Proc,LPlan).
solve_a_layoutproblem(CirNet,LPlan,Proc,Constr):-
        not(leaf(CirNet)),!,
        generate_subproblems(CirNet,SonsCirNets,Constr),
        check_module_allocations(SonsCirNets,Constr),
        layout_all(SonsCirNets,SonsLPlist,Proc),
        check_shape_and_wire_abutment(SonsLPlist),
        aggregate_subproblems(SonsLPlist,LPlan).
layout_all([],[],_).
layout_all([Hcs|Tcs],[Hls|Tls],Proc):-
        solve_a_layoutproblem(Hcs,Hls,Proc),
        layout_all(Tcs,Tls,Proc).
```

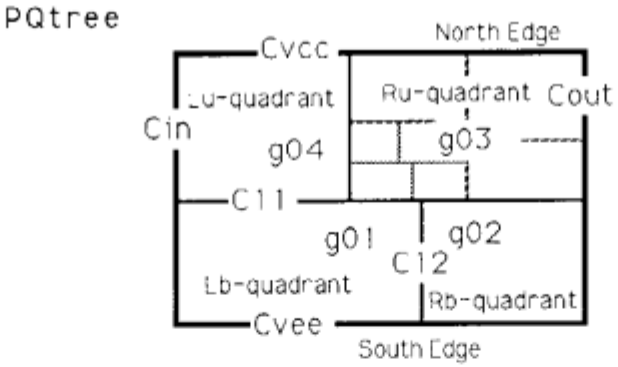
2.4. Problems

• Large computation time

The *generate_subproblems(...)* consumes much time in dividing the *CirNet* into *SonsCirNets*. The recursion by







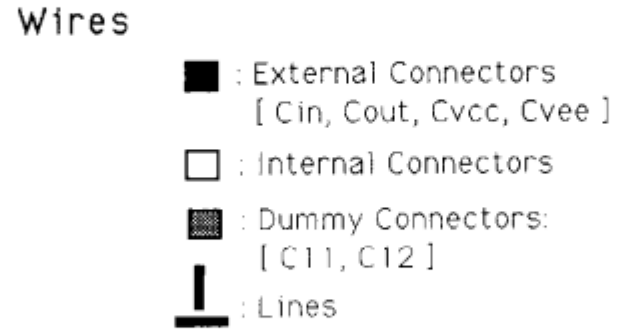


Figure 2.1. Slicing Structure Representation of Layout

layout_all(...) are also time consuming. So, backtrackings by either *check_module_allocations(...)* or *check_shape_and_wire_abutment(SonsLPlist)* become inadmissible.

• Large chip area

The *check_shape_and_wire_abutment(...)* for chip area reduction is difficult to be satisfied by a simple *layout_all(...)* that lacks the ability of shape- and wire-abutment among sub-circuits.

Unless these defects are overcome, the specification has no practical value, however elegant it might be.

2.5. Approaches

• Pre-compilation of circuit net into a quadtree

We pre-compile the given *CirNet* into a quadtree [9] named *CirTree*. Modules that should be placed near are gathered in the lower node of the *CirTree* so as to suffice the *Constr* and avoid *check_module_allocations(...)*.

• Vertical coordination of module shapes

The patent rectangle is sliced and sub-circuits are embed in them by *generate_subproblems(...)*. In *layout_all(...)*, each son tries to obey the slice shape as a guideline or a budget. So, even if subproblems were solved in parallel, they can have nearly abutted shapes.

• Horizontal co-operation in wiring

Wire abutment among processes running in parallel is an unsolved problem in VLSI layout design automation. we make the recursive and parallel specification workable by way of runtime co-operations among processes.

The dummy connectors defined before are used as a global mail box among subproblems. At any time, current existence range of the dummy connector is written on the mail. The *generate_subproblems(...)* reads the range and tries to narrowed it in view of its inner net profile. The narrowing action is made among face-to-face modules in concurrent mode. Round-robin trial is made to avoid deadlocks.

3. SPECIFICATIONS OF co-HLEX

An overview of **co-HLEX** is given in Figure 3.1. The Main components of **co-HLEX** include: a set of original data, I/O functions, a static and a dynamic memory, a problem solving kernel, and layout primitives. They are detailed in this section. To give both clear and brief specifications, following GHC-like description is used [8].

Head:-Body_guards|Body_goals.

Unless both Head and Body_guards succeed, the clause cannot pass the commit bar; '|'. This means that this clause is suspended and cannot commit to the reduction of the parent goal. For brevity of expressions, we assume a non-flat GHC; any complex guard-predicates can be used.

3.1. The Goal and Related Terms

• The Goal

?-solve_a_layoutproblem(CirNet,LPlan,Proc,Constr).

• Circuit network representation

CirNet:=[Modules,Nets].
Modules:=[module(Name,Type,ConnectNetList,Comments)|Rests].
Nets:=[net(Name,Type,ConnectModuleList),Comments)|Rests].

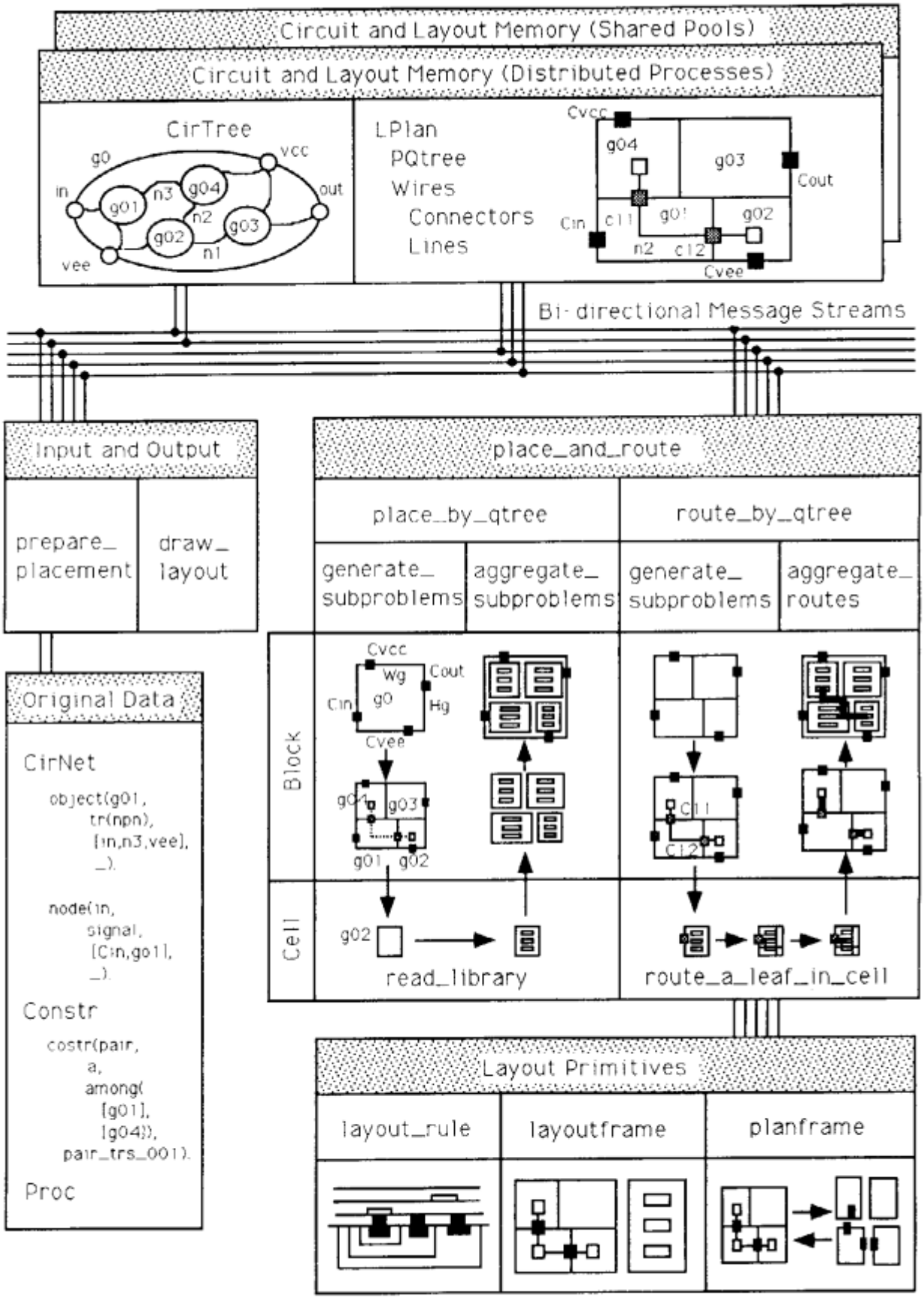


Figure 3.1. Overview of co-HLEX

- **Constraints and fabrication process representation**

Constr:=[Nearnesses,TopLplan].
Nearnesses:=[constr(Type,Level,Modules,_,ConstraintName)|Rest].
*[constr(pair,a,[g01,g04],_,pair_trs_001)]:=*An example specification of Level-*a* pair among modules; *g01* and *g04*.
TopLplan:=[size(Wp,Hp),Pad_placements].
Pad_placements:=[con(Name,Range,Type,Layers,_,Prop,_)|Rest].
*Proc:=*A string, for example: *process100*.

- **Layout plan representation**

LPlan:=[PQtree,Wires].
PQtree:=[Shape,Properties,SonsPQtrees].
Shape:=[X0,....X3,Y0,....Y3,....,RX0,....,RX3,RY0,....,RY3].
*Xi,Yi/RXi,RYi:=*Slicing points in local/global coordinate of the chip.
Properties:=[circuit(CirName,Level,Status),LFName,Area,DeadSpace,
AspectRatio,NetList,_].
Level:=block|cell|block_in_cell|leaf_in_cell.
Status:=free|fixed. % Denotes if the slice has free or fixed layout.
*LFName:=*The layoutframe name adopted by this slice.
NetList:=[[MyName,MyPattern,MyNWSEConnectors,MyInnerConnectors]|_].
*MyPattern:=*The wiring pattern name adopted by the net in this slice.
*MyNWSEConnectors:=*External connector names of the net
*MyInnerConnectors:=*Inner connector names of the net.
SonsPQtrees:=[Lus,Lbs,Rbs,Rus].
Lus, etc.:=Quadtree of the son in the left upper or north west slice, etc.
Wires:=[[Connector|_],[Line|_]].
Connector:=[Name,[NetName,Range,Type,Layers,NWSELines,
Props,RangeFlag]].
Range:=[Xfrom,Yfrom,Xto,Yto].
NWSELines:=[[NlineName,Other]|_].
Props:=[[NlineWidth,NlineLayer]|_].
RangeFlag:=[MyFlag,YourFlag].
MyFlag/YourFlag:=void|inspected|narrowed|finished.
Line:=[MyName,[NetName,Width,Layer,StartContName,EndContName,Props]]

3.2. Topmost Clause

3.2.1. Global coordinator

Topmost clause; *solve_a_layoutproblem(...)* coordinates the overall layout problem solving steps.

solve_a_layoutproblem(CirNet,LPlan,Proc,Constr,PEs):-true|
prepare_placement(CirNet,CirTree,LPlan,Constr),
create_process_net(CirTree,LPlan,CNstr,LPstr,PEs),
place_and_route(CNstr,LPstr,Proc),
remove_process_net(CNstr,LPstr,CirTree,LPlan),
draw_layout(LPlan). *prepare_placement(CirNet,CirTree,LPlan,Constr):-*
% Compile the *CirNet* into a quadtree; *CirTree*, so as to satisfy constraints specified in *Constr*. Further, Generate a homologous quadtree skeketon as *LPlan*. Register the chip layout plan in *Constr* on *LPlan*.
create_process_net(CirTree,LPlan,CNstr,LPstr,PEs):-
% Make two hierarchical process networks corresponding to *CirTree* and *LPlan* on processors; PEs. Define relevant message streams; *CNstr* and *LPstr*.
remove_process_net(CNstr,LPstr,CirTree,LPlan):-
% Remove nets and read out layout data into *LPlan*.
draw_layout(LPlan):-% Output layout figure on a display.

3.2.2. Placement and routing coordinator

The *place_and_route(...)* is a topmost clause of the algorithm **HRCTL**. After the module placement by *place_by_qtree(...)*, placement coordinates are transformed into global or world coordinates of the chip by *prepare_routing(...)* to make the precise wiring by *route_by_qtree(...)* possible. To get a good; short wired placement, the *place_by_qtree(...)* plans rough wires using many functions of *route_by_qtree(...)*.

place_and_route(CNstr,LPstr,Proc):-true|

```
        place_by_qtree(CNstr,LPstr,Proc),
        prepare_routing(CNstr,LPstr),
        route_by_qtree(CNstr,LPstr,Proc),
        repair_layout(CNstr,LPstr),
        proute_by_qtree(CNstr,LPstr,Proc).
repair_layout(CNstr,LPstr):-% Repair bad wires.
proute_by_qtree(CNstr,LPstr,Proc):-% Wire power lines.
```

3.3. Hierarchical Recursive Placement

3.3.1. Parallel recursive inference

The *place_by_qtree(...)* and the *route_by_qtree(...)* are the kernel clauses of **HRCTL**. Each of them takes a form of recursive divide-and-conquer.

The first clause is for void sons of a parent with less than four sons. The second clause is for terminal modules where a cell, which matches the circuit and the *Proc* condition, is recalled from a library.

The third clause applies to divisible, block-level modules. After choosing a good slicing template named *layoutframe* by *choose_a_layoutframe(...)*, the parent slice is divided into usually four sons by *generate_subproblems(...)*. Then *place_all_sons(...)* solves them in recursive mode. The *aggregate_subproblems(...)* gathers sons layouts to form the parent layout. Notice the use of short-circuits of Takeuchi [8] to spawn four sons in parallel and detect the overall completion.

```
place_by_qtree(CNstr,LPstr,Proc):-void(CNstr)|true.
place_by_qtree(CNstr,LPstr,Proc):-cell(CNstr)|
        read_library(CNstr,LPstr,Proc).
place_by_qtree(CNstr,LPstr,Proc):-not(cell(Cn))|
        choose_a_layoutframe(CNstr,LPstr,Proc),
        generate_subproblems(CNstr,LPstr,Proc),
        place_all_sons(CNstr,LPstr,Proc),
        aggregate_subproblems(CNstr,LPstr,Proc).
place_all_sons(CNstr,LPstr,Proc):-
        get_streams(sons_streams,CNstr,[CNstr1,CNstr2,CNstr3,CNstr4],[]),
        get_streams(sons_streams,LPstr,[LPstr1,LPstr2,LPstr3,LPstr4],[])|
        place_by_qtree(CNstr1,LPstr1,Proc,Ei,E1),
        place_by_qtree(CNstr2,LPstr2,Proc,Ei,E2),
        place_by_qtree(CNstr3,LPstr3,Proc,Ei,E3),
        place_by_qtree(CNstr4,LPstr4,Proc,Ei,E4),
        judge_end([E1,E2,E3,E4],Eo).
get_streams(sons_streams,CNstr,[CNstr1,CNstr2,CNstr3,CNstr4],[]):-
        % Get streams to four processes of sons through the parent stream; CNstr.
judge_end([E1,E2,E3,E4],Eo):-E1=end,E2=end,E3=end,E4=end|Eo=end.
```

3.3.2. Embedding sub-circuits into slices using a layout frame

By *choose_a_layoutframe(...)*, the parent rectangle is divided into usually four slices and sons are embed onto them. The parameterized slicing template; *layoutframe*, is used as a division tool. All the possible *layoutframes* and embeds are tried.

```
choose_a_layoutframe(CNstr,LPstr,Proc):-
        get_and_put(parent_plan,LPstr,OmyPlan,NmyPlan,LPstr1),
        get_and_put(sons_stream,LPstr1,SonsLPstr1,SonsLPstr2,LPstr2),
        get(sons_nets,CNstr,SonsNets,[])|
        separate_nets(LPstr2,NWSEnets),
        try_all_layoutframes(NWSEnets,OmyPlan,SonsLPstr1,SonsNets,Triedstr),
        find_a_best_layoutframe(Triedstr,NmyPlan,SonsLPstr2).
get_and_put(parent_plan,LPstr,OmyPlan,NmyPlan,LPstr1):-
        % Get the parent planned layout OmyPlan via the LPstr stream and put a new plan NmyPlan instead.
        Prepare a tail stream for later use.
get_and_put(sons_stream,LPstr,SonsLPstr1,SonsLPstr2,LPstr1):-
        % Get a stream SonsLPstr1 to the LPlan-processes of sons via LPstr.
        Put a new stream; SonsLPstr2 and prepare a tail; LPstr1, for later use.
separate_nets(LPstr1,NWSEnets):-
```

% Separate the parent net into north, west, south, and east ones.

try_all_layoutframes(NWSEnets,OmyPlan,SonsLPstr1,SonsNets,Triedstr):-

% Generates a list of all the possible combinations; *Candstr*, of usable *layoutframes* and sons circuit embed permutations, then all the candidates are tried to give simulated layouts.

find_a_best_layoutframe(Triedstr,NmyPlan,SonsLPstr2):-

% The best layout-frame and relevant embed plan is chosen as *NmyPlan*.

3.3.3. Generation of dummy connectors around sub-circuits

By using the chosen *layoutframe* and the relevant sons embed into slices, the inter-sons rough wires are planned by *plan_wirings(...)* to introduce dummy connectors on each crossing point of wires and inner edges of the slice. They are descended to sons by *propagate_plans(...)* as their external connectors. Now, each son has a planned layout homologous to the patent's, they can be solved recursively.

```
generate_subproblems(CNstr,LPstr,Proc,):-true|
        prepare_generate_sub(CNstr,LPstr,NetsList),
        plan_wirings(LPstr,NetsList,NetsList1,Proc),
        propagate_plans(LPstr,NetsList1).
prepare_generate_sub(CNstr,LPstr,NetsList):-
        % Gather external and internal net names to form NetsList.
propagate_plans(LPstr,NetsList1):-
        % Dummy connectors are descended to sons as their external connectors.
```

• Wiring nets

The *plan_wirings(...)* first estimates wiring capacity of all the edges of the slice as well as the via-placement capacity of each slice. The result is memorized in a vector form by a process having a message stream; *GoalVecstr*.

```
plan_wirings(LPstr,NetsList,AnsNetsList,Proc):-
        get(patent_shape,LPstr,Shape,LPstr1),
        get(layoutframe,LPstr1,LFName,LPstr2),
        get(circuit_name,LPstr2,CName,LPstr3)|
        planframe(LFName,generate_goal_vector,LPstr3,GoalVecstr,
                InitVecstr,InitCoststr,Proc),
        generate_wiring_manager(Netsstr),
        wire_all_nets(CName,NetsList,Netsstr,Shape,LFName,GoalVecstr,
                InitVecstr,InitCoststr,AnsNetsList).
generate_wiring_manager(Netsstr):-
        % Generate a wiring manager to command the wiring activities performed by wire_all_nets(...).
```

The *wire_all_nets(...)* first examines the profile of the internal connectors of the net; *InNetProfile*, by *get_inner_net_profile(...)*. Then streams to external connectors of the net is gathered by *get_pericont_stream(...)* to form the *NWSEcontsstr*. Then the process; *wire_a_net(...)* is called to make the wire for the net.

```
wire_all_nets(_,[],Netsstr,_,_,GVstr,IVstr,ICstr,AnsNetsList):-true|
        close([Netsstr,GVstr,IVstr,ICstr,AnsNetsList]).
wire_all_nets(CName,[Net|Rest],Netsstr,Shape,LFName,GVstr,IVstr,
                ICstr,AnsNetsList):-true|
        Netsstr=[Netstr|Tail1],
        AnsNetsList=[AnsNet|Tail2],
        get_inner_net_profile(Net,InNetProfile),
        get_pericont_stream(Net,NWSEcontsstr),
        wire_a_net(CName,Netstr,Net,InNetProfile,NWSEcontsstr,Shape,
                LFName,GVstr,IVstr,ICstr,AnsNet),
        wire_all_nets(CName,Rest,Tail1,Shape,LFName,GVstr,IVstr,ICstr,Tail2).
```

The *wire_a_net(...)* finds a good spanning pattern that can connect all the inner connectors of the net being wired. At first, each external connector of the net is pulled into an appropriate slice and appended with original inner connectors. Then inter-slice wires are made using route-patterns among these connectors.

Receiving the message; *finished*, the first clause returns the wiring result as *AnsNet*. For *narrow*, the second and the third clause applies.

The third, for non-wirable cases due to the incomplete feed-through; defined as a net having neither inner net and pulled-in external connectors. The *renew_cont_rangeflag(...)* sets all the range-flags of the external connectors to *inspected* releasing the pull-in actions of neighboring modules. At the same time the answer to the *wire-manager(...)* is set as *inspected* to get a next retry command; *narrow* , from it.

The second clause is for wirable cases. After external connectors of a net are pulled-in, the *find_a_good_route_pattern(...)* chooses a good route pattern to connect them. At crossing points of the pattern and the slice edges, dummy connectors are introduced.

```
wire_a_net(_,[finished],Net,_,NWSEContsstr,_,_,GVstr,IVstr,ICstr,AnsNet):-true|
    AnsNet=Net,close(NWSEContsstr,GVstr,IVstr,ICstr).
wire_a_net(CName,[{narrow,Ans}|TailMes],Net,InNetProfile,
        NWSEContsstr,Shape,LFName,GVstr,IVstr,ICstr,AnsNet):-
    get_and_put(cont_range_flag,NWSEContsstr,NWSEConts,NWSEContsstr1),
    wait_neighbours_action(CName,NWSEConts),
    or(exist_inner_net(InNetProfile),wirable_feed-through(NWSEConts))|
    renew_cont_range_flag(narrowed,NWSEConts),
    get_and_put(vector,GVstr,GV,GVstr1),
    get_and_put(vector,IVstr,IV,NewIV,IVstr1),
    get_and_put(vector,ICstr,IC,NewIC,ICstr1),
    planframe(LFName,get_route_patterns,RPatterns),
    pull_in_connectors(NWSEConts,Shape,{InNetProfile,IV,IC},
        NewProfileVec),
    find_a_good_route_pattern(RPatterns,NewProfileVec,GV,
        NewIV,NewIC,RPattern,NewDummyConts),
    add_crossing_points_on_netdata(Net,NewDummyConts,RPattern,NewNet),
    Ans=finished,
    wire_a_net(CName,TailMess,NewNet,InNetProfile,NWSEContsstr1,
        Shape,LFName,GVstr1,IVstr1,ICstr1,AnsNet).
wire_a_net(CName,[{narrow,Ans}|TailMes],Net,InNetProfile,
        NWSEContsstr,Shape,LFName,GVstr,IVstr,ICstr,AnsNet):-
    otherwise,
    get_and_put(cont_range_flag,NWSEContsstr,NWSEConts,NWSEContsstr1)|
    renew_cont_rangeflag(inspected,NWSEConts),
    set_old_range(NWSEConts),
    Ans=inspected,
    wire_a_net(CName,TailMes,Net,InNetProfile,NWSEContsstr1,Shape,
        LFName,GVstr,IVstr,ICstr,AnsNet).
get_and_put(cont_range_flag,ParentBlock,NWSEContsstr,
        NWSEConts,NWSEContsstr1):-
    % Get current range and the range-flags of all the external connectors of a net through the stream;
    NWSEContsstr as NWSEConts.
wait_neighbours_action(CName,NWSEConts):-
    % Wait until some neighbour module performes a pull-in action.
find_a_good_route_pattern(RPatterns,NewProfileVec,GV,
        NewIV,NewIC,RPattern,NewDummyConts):-
    % Choose a best candidate wiring pattern that can span inner connectors of the parent as RPattern.
    Relevant new dummy connectors, induced on the crossing points of the chosen pattern and slice
    borders, are returned as NewDummyConts. The NewIV and NewIC denotes the renewed resource
    consumption vector and the resource usage cost, respectively.
```

The *wirable_feed-through(...)* guards the second clause of *wire_a_net(...)*. It can succeed if either *i_can_pull_in(...)* or *i_should_pull_in(...)* succeeds. The former can succeed after all neighboring modules have performed pull-in actions for external connectors, i.e. if all "your" flags; *Nyf*, *Wyf*, *Syf*, and *Eyf* suffice the *all_are_members(...)* condition. The latter can succeed if all the connectors, which "I" have abandoned to pull-in before, are also abandoned by "you". This fact can be detected by checking if all the connectors "I" have abandoned before have been *inspected* by "you".

```
wirable_feed-through(NWSEConts):-
    get_pullin_flags(NWSEConts,NWSEflags),
    or(i_can_pull_in(NWSEflags), i_should_pull_in(NWSEflags))|true.
```

```
i_can_pull_in([{Nmf,Nyf},{Wmf,Wyf},{Smf,Syf},{Emf,Eyf}]):-
        all_are_members([Nyf,Wyf,Syf,Eyf],[void,finished,narrowed])|true.
i_should_pull_in(NWSEflags):-
        flags_i_have_delayed_before(NWSEflags,Delayedflags),
        you_have_tried_all(Delayedflags)|true.
you_have_tried_all(Delayedflags):-
        all_are_unifiable({_,inspected},Delayedflags)|true.
```

External connectors are pulled into some slices and appended with inner connectors by *pull_in_connectors(...)*. The first clause returns the appended inner connector profile; *AnsInNetProfile*. The second clause pulls-in a connector with an already reduced existence range; *ORange*. The pulled-in connector is appended to current *InNetProfile* defining a *NewInNetProfile*.

The third applies to ambiguous cases when *ORange* has overlaps with two slice borders. The *possible_pull_in_slices(...)* examine all the candidate slices as *PSlices*. Then the *PossibleInNetProfiles;* the list of all the candidate appended inner net profiles, is generated by *possible_innet_profiles(...)*.

```
pull_in_connectors([],_,InNetProfile,AnsInNetProfile):-true|
        AnsInNetProfile=InNetProfile.
pull_in_connectors([{{ORange,NRange},RFlag,Layer}|Rest],Shape,
                InNetProfile,AnsInNetProfile):-
        have_narrow_range(ORange,Shape),
        determine_pull_in_slice(ORange,Shape,Slice,NRange)|
        renew_innet_profile(InNetProfile,Slice,Layer,NewInNetProfile),
        pull_in_connectors(Rest,Shape,NewInNetProfile,AnsInNetProfile).
pull_in_connectors([{{ORange,NRange},RFlag,Layer}|Rest],Shape,
                InNetProfile,AnsInNetProfile):-
        not(have_narrow_range(ORange,Shape)),
        possible_pull_in_slices(ORange,Shape,PSlices)|
        possible_innet_profiles({{ORange,NRange},RFlag,Layer},
                InNetProfile,PSlices,PossibleInNetProfiles),
        pull_in_connectors(Rest,Shape,PossibleInNetProfiles,AnsInNetProfile).
```

3.3.4. Aggregate sons realized layouts

The layout-frame specific *planframe(...)* is used to aggregate sons finished layouts into a realized layout of the parent; *RealShape*. If the planned *layoutframe* name; *PlanLFName* happened to become inappropriate due to the gap between the plan and the realization, it is changed to a new *RealLFName*. All the dummy connectors introduced before are thrown away as they become obsolete.

```
aggregate_subproblems(CNstr,LPstr,Proc):-
        get_and_put(shape,LPstr,PlanShape,RealShape,LPstr1),
        get_and_put(layoutframe,LPstr1,PlanLFName,RealLFName,LPstr2),
        get(sons_shapes,LPstr2,SonsShapes,[])|
        planframe(PlanLFName,aggregate_sons_shapes,PlanShape,
                SonsShapes,RealShape),
        renew_lfname(PlanLFName,RealShape,RealLFName).
```

3.4. Hierarchical Recursive Wiring

3.4.1. Wiring preparation

The local coordinates used by slices thus far are transformed into a global or world coordinate to avoid repeated transformations of the existence ranges of pulled-in connectors, performed by slices having mutually different local coordinates. The *{0,0}* denotes the global origin; the north west corner of the chip.

```
prepare_routing(CNstr,LPstr):-true|
        get_topmost_plan(LPstr,TopShape,ExternalConnectors),
        generate_global_placement({0,0},TopShape,LPstr),
        set_topmost_plan(LPstr,TopShape,ExternalConnectors).
```

3.4.2. Precise wiring

• **Recursive inference for wiring**

Precise wiring is made by *route_by_qtree(...)*. It has a recursive structure similar to *place_by_qtree(...)*. The *generate_subproblems(...)* used before is reused. However, the *choose_a_layoutframe(...)* is skipped as placements are already given. The recursion continues untill the indivisible slice, named *leaf_in_cell*, is reached.

The first clause is the terminal case where only one net can appear in the slice. The *generate_subproblems(...)* first sets all the range flags of external connectors as *finished*, to release pull-in actions of neighboring slices, then chooses a wiring pattern that can span these external connectors. The second clause performes recursions.

```
route_by_qtree(CNstr,LPstr,Proc):-true|
        get(circuit_property,LPstr,level(leaf_in_cell),LPstr1),
        generate_subproblems(CNstr,LPstr1,Proc),
        route_a_leaf_in_cell(CNstr,LPstr1,Proc).
route_by_qtree(CNstr,LPstr,Proc):-
        get(circuit_property,LPstr,Level,LPstr1),
        or(Level=level(block),Level=level(block_in_cell))|
        generate_subproblems(CNstr,LPstr1,Proc),
        route_all_sons(CNstr,LPstr1,Proc),
        aggregate_routes(CNstr,LPstr1,Proc).
route_all_sons(CNstr,LPstr,Proc):-
        get_streams(sons_stream,CNstr,[CNstr1,CNstr2,CNstr3,CNstr4],[]),
        get_streams(sons_stream,LPstr,[LPstr1,LPstr2,LPstr3,LPstr4],[])|
        route_by_qtree(CNstr1,LPstr1,Proc,Ei,E1),
        route_by_qtree(CNstr2,LPstr2,Proc,Ei,E2),
        route_by_qtree(CNstr3,LPstr3,Proc,Ei,E3),
        route_by_qtree(CNstr4,LPstr4,Proc,E4,Eo),
        judge_end([E1,E2,E3,E4],Eo).
```

• **Line segment generation in leaf slices**

First, the *NetList* containing external connector placement plans are read out. Then lines are generated among them.

```
route_a_leaf_in_cell(CNstr,LPstr,Proc):-
        get_and_put(netlist,LPstr,NetList,[])|
        generate_lines(NetList,Proc).
```

Line are generated by *generate_lines(...)*. The first argument is an input list of nets made by *generate_subproblems(...)*. A net data includes; a net name; *Net*, a route pattern adopted; *RPattern*, and streams to both four external- and one internal-connectors; *NCstr,...,ICstr*. By *get_and_put(...)* a line skeleton; *LineSkeleton*, which spans external connectors are read out and send to *make_lines(...)*. The *make_line(...)* that unifies with the given *RPattern* and the *LineSkeleton* generates *RealLines*.

```
generate_lines([],_):-true|true.
generate_lines([{Net,RPattern,NCstr,WCstr,SCstr,ECstr,ICstr}|Rest],Proc):-
        get_and_put(cont_data,[NCstr,WCstr,SCstr,ECstr|ICstr],
                LineSkeleton,RealLines,AllContsstr)|
        make_lines(RPattern,LineSkeleton,RealLines,AllContsstr,Net,Proc),
        generate_lines(Rest,Proc).
make_lines(RPattern,LineSkeleton,RealLines,AllContsstr,Net,Proc):-
        % Generate line segments data among connectors and
        create line processes.
```

3.5. Libraries

3.5.1. Plan frames

Template dependent problem solving clauses are gathered to form a library named planframes to enhance the maintainability of **co-HLEX**. A few representatives will be shown here.

- **Shape evaluation**

planframe(LFName,evaluate_shape,OmyPlan,SonsPlace,
NmyPlan,NSonsPlace,NDeadsp,NAspect):-
% Determine slicing point parameters; *NmyPlan* so as to suffice sons area constraints under the given parent's planned layout. Estimated dead space and aspect ratio are reported as *NDeadsp* and *NAspect*.

- **Wiring evaluation**

planframe(LFName,evaluate_wires,NWSEnets,SonsPlace,XwCost,YwCost):-
% Given External nets; *NWSEnets* and sons placement in four slices, the number of feed-through-wires in horizontal and vertical directions are estimated as *XwCost* and *YwCost*, respectively.

- **Wiring resource vector generation**

planframe(LFName,generate_goal_vector,LPstr,
GoalVecstr,InitVecstr,InitCoststr,Ei,Eo):-
% Generate a wiring resource vector; *GoalVector* for a parent layout slice under the given slicing parameters, usable layers, connector width, and inter-connector distance.

- **Connector distribution**

planframe(LFName,distribute_connectors,WirePlan,
Luwp,Lbwp,Rbwp,Ruwp):-
% The dummy connectors on a parent layout are delivered to relevant sons as their external connectors. Each connector has a message stream that will be used in pull-in co-operations.

- **Aggregation of sons layouts**

planframe(LFName,aggregate_sons_shapes,PShape,
[{LuW,LuH},{LbW,LbH},{RbW,RbH},{RuW,RuH}],RShape):-
% Realized shapes; *{LuW,LuH},....,{RuW,RuH}*, of four sons are aggregated to form a parent shape in *RShape*.

- **Layout template retrieval**

Various *layoutframe*-specific templates required in layout tasks are retrieved by *planframe(...)*. Here are two examples.

```
planframe(LFName,get_route_patterns,RoutePatterns):-true|
    layoutframe(LFName,_,_,_RoutePatterns,_,_,_,_).
planframe(LFName,tree_skeleton,TreeSkeleton):-true|
    layoutframe(LFName,_,_,_,_,_,_,TreeSkeleton,_).
```

3.5.2. Layout frames

Layoutframes are definitions of both geometrical and electrical properties of technology dependent layout primitives. The *LFName* is the retrieval key.

layoutframe(LFName,Blocks,Connectors,Lines,RoutePatterns,
RouteVector,Obstacles,TreeSkeleton,Others):-
% Define a module named *LFName* in forms of Blocks, Connectors, and *Lines*. *RoutePattern* is the usable route patterns. *RouteVector* is the wiring resource vector template. *Obstacles* give wiring obstacles. *TreeSkeleton* is the quadtree slicing of the *layoutframe*.

3.5.3. Layout rules

The followings are a few example layout rules currently held in **co-HLEX**. Most of them are specific to bipolar analog devices.

- **Usable wiring layers**

```
layout_rule(usable_layers,process100,Layers):-true|
    Layers=[layer(1),layer(2),layer(3)].
```

- **Distances among objects**

```
layout_rule(min_line_width,process100,signal,_,_,Width,_,_):-true|Width=4.
layout_rule(min_line_distance,process100,_,_,signal,signal,Dist):-true|Dist=4.
layout_rule(half_iso_width,process100,_,_,Width):-true|Width=4.
```

• **Connectors**

```
layout_rule(connector_params,process100,Layers,pad,Holes,Pads):-
        true|Holes=[],Pads=[pad(layer(1),6,6)].
layout_rule(connector_params,process100,Layers,throughole,Holes,Pads):-
true|Holes=[],Pads=[pad(layer(1),6,6)].
```

• **Transistors**

The planer structure of a simple npn transistor is defined by a collector, a base, and an emitter diffusion rectangles as well as relevant contacts.

```
layout_rule(tr_parameter,process100,npn,simple_tr,Diffusions,
        Contacts):-true|
    Diffusions=[diff(collector,n_diff,rect(0,0,24,48),_),
        diff(base,p_diff,rect(4,16,14,28),_),
        diff(emitter,n_diff,rect(7,33,8,8),_)],
    Contacts=[cont(collector,type1,center(12,7),[],[pad(layer(1),6,6),_),
        cont(base,type1,center(12,23),[],[pad(layer(1),6,6),_),
        cont(emitter,type1,center(12,37),[],[pad(layer(1),6,6),_)].
```

• **Resistors**

The planer structure of a simple base-diffusion-resistor is defined by components; a resistor rectangle on diffusion layer and two contacts for anode and cathode.

```
layout_rule(res_parameter,process100,base_res,simple_res,
        Diffusions,Contacts):-true|
    Diffusions=[diff(base,p_diff,rect(0,0,8,48),_)],
    Contacts=[cont(anode,{normal,type1},center(4,4),[],[pad(layer(1),6,6)],_),
        cont(cathode,{normal,type1},center(4,44),[],[pad(layer(1),6,6)],_)].
```

3.6. Load Balancing

• **Creation of tree processes**

The static *CirTree* and *LPlan* are compiled into distributed process networks by the following program:

```
create_process_net(CirTree,LPlan,CNstr,LPstr,PEs):-true|
    LPlan=[PlaceTree,Wires], Wires=[Connectors,_],
    create_qtree_net(CirTree,PlaceTree,CNstr,LPstr,PEs),
    create_wirenet(Connectors,LPstr).
create_wirenet(Connectors,LPstr):-
    % Create process network representing line segments of wires.
```

The *create_qtree_net(...)* generates data processes for circuit and layout data. *CNstr* and *LPstr* are message streams to them, respectively. The first clause divides a cell into a quadtree using the cell's *layoutframe*-specific *TreeSkeleton*. The second one is for terminal nodes where process generations are stopped and streams are closed. The third one is for other cases where circuit- and placement-processes are generated. Message streams to sons are also taken out as *SonsCNstr*, etc. Then sons of the tree node are recursively processed by *create_all_sons(...)*.

```
create_qtree_net(CirTree,PlaceTree,CNstr,LPstr,PEs):-
    get(circuit_data,CirTree,CircuitData),
    get(place_data,PlaceTree,PlaceData),
    get(circuit_property,PlaceData,level(cell)),
    get(layoutframe,PlaceData,LFName)|
    planframe(LFName,tree_skeketon,TreeSkeleton),
    generate_local_qtree(TreeSkeleton,PlaceData,LQtree1),
    generate_local_ctree(TreeSkeleton,CircuitData,LCtree1),
    create_qtree_net(LCtree1,LQtree1,CNstr,LPstr,PEs).
create_qtree_net(CirTree,PlaceTree,CNstr,LPstr,PEs):-
    terminal(CirTree)|close([CNstr,LPstr]).
create_qtree_net(CirTree,PlaceTree,CNstr,LPstr,PEs):-
```

```
        otherwise|
        get(circuit_data,CirTree,CircuitData),
        get(place_data,PlaceTree,PlaceData),
        create_circuit_processes(CircuitData,CNstr,SonsCNstr),
        create_place_processes(PlaceData,LPstr,SonsLPstr),
        create_all_sons(CirTree,PlaceTree,SonsCNstr,SonsLPstr,PEs).
```

Four sons are forked on processors; *LuPE*, *LbPE*, *RbPE* and *RuPE* defined by *assign_processor(...)*. The *Goal@processor(PEaddr)* notifies that the *Goal* should be spawned on the processor; *PEaddr*.

```
create_all_sons(CirTree,PlaceTree,SonsCNstr,SonsLPstr,PEs):-true|
        SonsCNstr=[LuCNstr,LbCNstr,RbCNstr,RuCNstr],
        SonsLPstr=[LuLPstr,LbLPstr,RbLPsre,RuLPstr],
        get(sonsCirTrees,CirTree,[Sct1,Sct2,Sct3,Sct4]),
        get(sonsPlaceTrees,PlaceTree,[Spt1,Spt2,Spt3,Spt4]),
        get(sons_areas,PlaceTree,SonsAreas),
        assign_processors(PEs,SonsAreas,[LuPEs,LbPEs,RbPEs,RuPEs],
              [LuPE,LbPE,RbPE,RuPE]),
        create_qtree_net(Sct1,Spt1,LuCNstr,LuLPstr,LuPEs,Ei,E1)
              @processor(LuPE),
        create_qtree_net(Sct2,Spt2,LbCNstr,LbLPstr,LbPEs,Ei,E2)
              @processor(LbPE),
        create_qtree_net(Sct3,Spt3,RbCNstr,RbLPstr,RbPEs,Ei,E3)
              @processor(RbPE),
        create_qtree_net(Sct4,Spt4,RuCNstr,RuLPstr,RuPEs,Ei,E4)
              @processor(RuPE),
        judge_end([E1,E2,E3,E4],Eo).
```

• Assignment of created processes on PEs

The *assign_processor(...)* divides the available processor set; *PEs* of the patent into at most four subsets on accordance with task volumes of sons. The chip area is used as a task volume approximation. One of the element is chosen from the subset to form a list; *SonsPEs* on which sons are spawned. After the processor set *PEs* become indivisible, the current processor; *CPE* of the patent is used by all sons.

```
assign_processors(PEs,SonsAreas,SonsPEs,SonsCurrentPEs):-
        area_ratio(SonsAreas,SonsAreaRatioList)|
        divide_pes(PEs,SonsAreaRatioList,SonsPEs),
        current_pe(CPE),
        define_sons_current_pe(CPE,SonsPEs,SonsCurrentPEs).
define_sons_current_pe(CPE,[[PE|_],[],[],[]],[SPE1,SPE2,SPE3,SPE4]):-true|
        SPE1=PE,SPE2=CPE,SPE3=CPE,SPE4=CPE.
define_sons_current_pe(CPE,[[PE1|_],[PE2|_],[],[]],[SPE1,SPE2,SPE3,SPE4]):-true|
        SPE1=PE1,SPE2=PE2,SPE3=CPE,SPE4=CPE.
```

4. ANALYSES OF HRCTL

4.1. Layout Capability of HRCTL

• Definitions.

Let PrBPT(R,N) denotes a layout problem in the form of a balanced tree of height N having R sons at each node.

• Suppositions.

The placement and the routing problem can be represented by PrBPT(4,N) and PrBPT(4,M), respectively. The former terminates in a *cell* but the latter terminates in a *leaf_in_cell* which is an indivisible slice in a cell. In other words, M>N and the difference M-N comes from the introduction of a local quadtree; PrBPT(4,M-N+1), in each cell for precise wiring.

• Theorem 1.

HRCTL can solve the placement problem for PrBPT(4,N).

- **Proof.**

For N=1,2, the first two clauses of *place_by_qtree(...)* solves the PrBPT(4,N). Suppose PrBPT(4,N) can be solved. Then PrBPT(4,N+1) can be solved thus. The third clause of *place_and_route(...)* divides it into four PrBPT(4,N)s by *generate_subproblems(...)*. Then place_all_sons(...) tries to solve them. But they are solvable by the assumption. Finally, *aggregate_subproblems(...)* gathers these sub-solutions to form the solution for PrBPT(4,N+1). **QED.**

- **Theorem 2.**

HRCTL can solve the routing problem for PrBPT(4,M).

- **Proof.**

Through similar inductions using *route_by_qtree(...)* clauses. **QED.**

- **Discussions.**

Only logical features of **HRCTL** were analyzed. Behavioral features such as non-deadlock termination of parallel co-operate placement and wiring were avoided.

4.2. Computational Complexity of HRCTL

- **Definitions.**

Let PrBPT(R,N) denotes a layout problem as above. Let leaf(PrBPT(R,N)) denotes the number of leaf nodes of PrBPT(R,N). Let no(PEs) denotes the number of parallel processing elements on which the problem PrBPT(R,N) is solved by the **HRCTL** algorithm.

- **Suppositions.**

All the nodes of PrBPT(R,N) consume the same computation power. Instantaneous communication among processes is possible without any computation load. The total elapsed time of processing on one PE is proportional to its total computation load.

- **Theorem 3.**

For PrBPT **HRCTL** has the time complexity of either

O(log(leaf(PrBPT(R,N)))) or O(log(no(PE))+leaf(PrBPT(R,N))/no(PE)). The latter is the usual case where large problem is solved on limited PEs.

- **Proof.**

Case1. no(PE) >=leaf(PrBPT(R,N)): The president PE is the neck processor which receives the topmost node of PrBPT(R,N). It processes maximum number of nodes among PEs. The maximum number is log(leaf(PrBPT(R,N))).

Case2. no(PE) =< leaf(PrBPT(R,N)): The president PE is also the neck processor. Until the depth of log(no(PE)) is reached on PrBPT(R,N), case1 applies. After it, each PE is obliged to solve all the unsolved nodes in pseudo parallel mode. Here, the number of unsolved nodes is leaf(PrBPT(R,N))/no(PE). As the president PE faces the two situations sequentially, they should be added to give the log(no(PE)) + leaf(PrBPT(R,N))/no(PE) complexity. **QED.**

5. SYSTEM DEVELOPMENT AND LAYOUT EXPERIMENTS

5.1. System Development

• Environment

Multi-PSI system with 16 PEs (Processor Elements in 4 by 4 array, each PE has 12 MW memory) is used.

• Program development

The KL1 (Kernel Language 1) was used in the implementation. Due to the recursive nature of **HRCTL**, program size of about 6,000 lines could be attained at present (4,500 for **HRCTL**, 1,500 for I/O). So far, research priority has been placed mainly on parallelism and concurrency rather than high quality but domain-specific layout generation.

• Memory architectures

Two versions of **co-HLEX** has been developed; **co-HLEX<v.1>** and **co-HLEX<v.2>**. Each adopts a shared- and a distributed-memory architecture, respectively.

5.2. Experiment Design

• Main objectives

The main objectives of the experiment are the verifications of:

OE1. Parallel placement capability.

OE2. Both wire length and chip area reduction by vertical coordination.

OE3. Parallel co-operative wiring capability.

OE4. The performance dependency on the memory architecture.

• Circuit

Simple bipolar analog circuit shown in Figure 5.1 (see N.W. part) is used. It is an analog amplifier having 16 modules; 8 npn transistors and 8 resistors. The 2 input-, 2 output-, 1 vcc-, and 1 gnd-connector are external connectors. All the 8 registers are transformed into equivalent parallel resistors to align their shapes with those of the transistors. Thus, the modified circuit become to have 24 modules.

• Shortcuts

In respect of the system development policy mentioned above, various shortcuts are made. Module shapes are aligned by changing device parameters. Exact pair formation, device isolation, wiring layers, power cable treatments and exact layout rules are postponed for later considerations.

• Data generation

Circuit quadtree is made manually. Part of the quadtree representations is shown in Figure 5.1.(S.W. part) Pair conditions for transistors q1, q2 forming a differential amplifier, etc. are taken into consideration in this task. An automatic quadtree explosion program is developed to generate large quadtrees from the 24-moduled seed.

5.3. Experiment Results and Observations

• Results

The topmost chip layout plan is displayed in Figure 5.1.(N.E. part) with the generated layout figure (S.E. part). Figure 5.2 shows a layout for 384 modules. Figure 5.3 shows performance curves of the two **co-HLEX** versions.

• Overall performances

Both parallel placement and parallel wiring are realized by **co-HLEX**. The **co-HLEX<1>** failed to solve

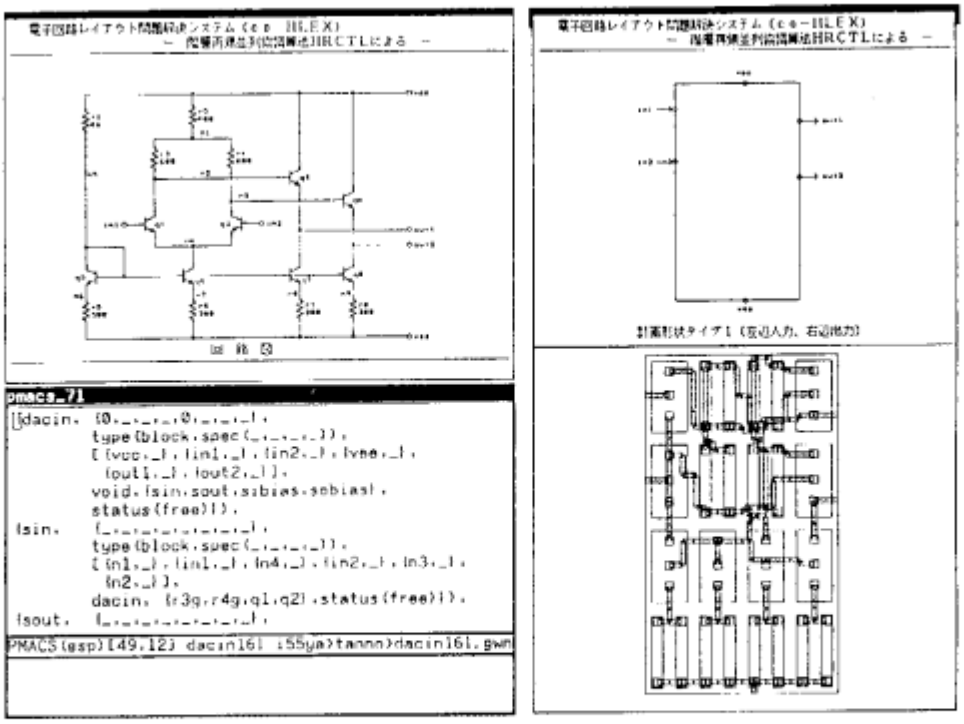


Figure 5.1. Window Image of co-HLEX

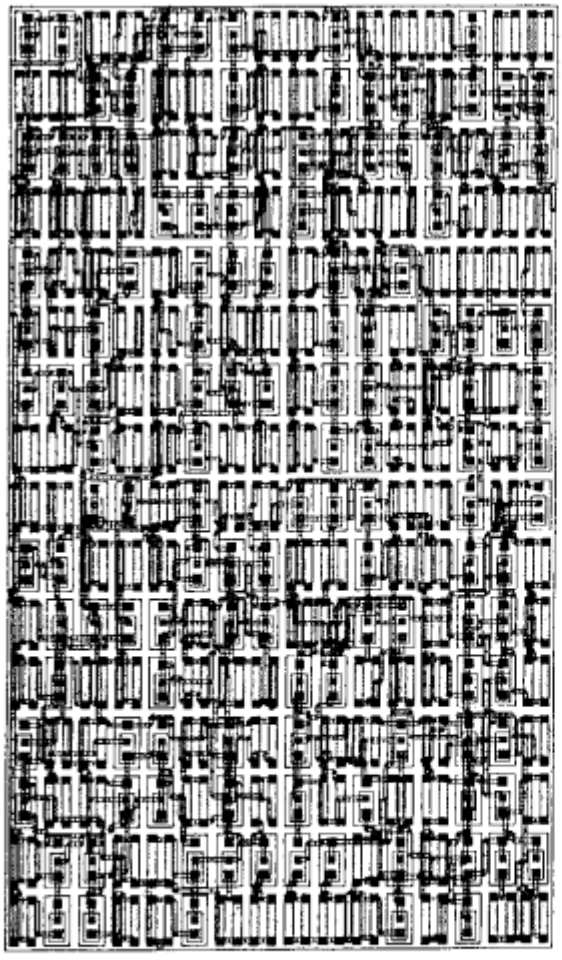


Figure 5.2. Example Output

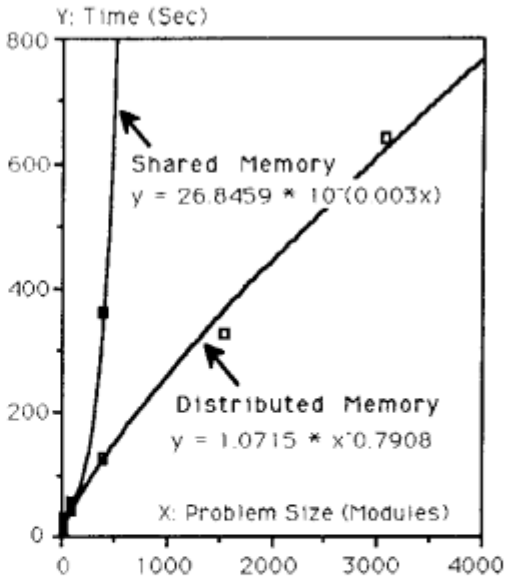


Figure 5.3. Performance

more than 400 module circuit.

We suppose the Neumann's Bottleneck on the message stream; connecting parallel processes and a shared memory, might be the cause. The **co-HLEX<2>** remarkably outperformed the former. It has a time complexity of $O(N^{0.8})$ on 16 PEs in the experimented range. This illustrates the power of the streamed-parallel and distributed-memory architecture. For the 3,000 module circuit, it took 600 sec to generate a layout. In this case nearly 50,000 parallel processes are estimated to be running on 16 PEs.

• Quality of placement

For the experimental circuit with aligned shaped modules, dead space free layouts could be generated. This illustrates the area compaction capability of **co-HLEX** whenever possible. Moreover, modules are placed so as to shorten wires among them.

• Quality of wiring

Useless wire bends could be avoided due to the runtime wire abutments among processes. Useless channels can be reduced and overall chip area could be reduced.

• Solving an unsolved problem

As far as we know, **co-HLEX** is the first system adopting the hierarchical divide-and-conquer that can abut wires among dividends by co-operations.

• Program size

The 6,000 lined **co-HLEX** remarkably outperforms the $O(10^5)$-$O(10^6)$ sized traditional implementations.

• Ease of program maintenance

Memory architecture modification of the program through introductions of distributed data processes and relevant streams was easy. This is mainly due to the inherent distributed nature of KL1 and **co-HLEX.**

6. CONCLUSIONS

6.1. Results

• A co-operative hierarchical layout problem solver named **co-HLEX** is proposed. Specification of **co-HLEX** was given using GHC-like formalism.

• The kernel algorithm of **co-HLEX** is **HRCTL** which is a hierarchical recursive concurrent theorem prover for layout. The missed links among subproblems due to hierarchical problem divisions are recovered by runtime process co-operations.

• The unsolved problem of module shape and wiring connector abutment among parallel processes could be solved by **co-HLEX.**

• **co-HLEX** was implemented on ICOT's parallel inference machine; Multi-PSI. Due to the recursive nature of **HRCTL**, current program size is nearly 6,000 lines in KL1 (4,500 for HRCTL, 1,500 for I/O) which remarkably outperforms the traditional $O(10^5)$-$O(10^6)$ sized implementations.

• The **co-HLEX** implementation in streamed-parallel and distributed-memory architecture currently attaines $O(N^{0.8})$ performance on 16 PEs. For experimental circuit with 3,000 modules, it took 600 sec to generate a layout. In this case nearly 50,000 parallel processes are running on 16PEs.

6.2. Next Targets

• Functional enhancements of **co-HLEX** are required for good layout generation.

• Large scale problem solving should be tried on Multi-PSI with more PEs and finally on PIM.

• Program maintainability should be illustrated through applications of **co-HLEX** to various devices and processes.

ACKNOWLEDGEMENTS

We acknowledge Dr. K. Furukawa, Dr. K. Nitta, and other ICOT members for their supports of this research, Prof. M. Iri of the University of Tokyo for his advices on large scale problem solving algorithm design, Mr. S. Domen; the general manager of the Systems Development Lab. of Hitachi, Ltd. for giving us the chance of the research, and finally Mr. M. Oda, Mr. H. Yamaguchi and other members of IBS Comserve Ltd. who assisted program developments.

REFERENCES

[1] Poincare, H., La Science et l'Hypothese, 1902.

[2] Kleene, S.C., Introduction to Metamathematics, Van Nostrand,1952.

[3] Mandelbrot, B., The Fractal Geometry of Nature, W. H. Freeman, 1982.

[4] Six, P., Claesen, L., Rabaey, J., and De Man, H., An Intelligent Module Generator Environment, Proc. 23rd DAC, pp730-735, 1986.

[5] Watanabe, T. and Shinichi, H., Modelling Layout Problem Solving in Logic, in CAD Systems using AI Techniques, G. Odawara(ed.), pp181-188, North-Holland.

[6] Kowalski, R.A., Logic for Problem Solving, North-Holland, 1979.

[7] Sterling, L. and Shapiro, E., The Art of Prolog, The MIT Press, 1986.

[8] Shapiro, E. (ed.), Concurrent Prolog: Collected Papers, (Vols.1 and 2), The MIT Press, 1987.

[9] Samet, H., Region Representation: Quadtrees from Boundary Codes, C.ACM, Vol.23, No.3, pp163-170, March, 1980.

[10] Samet, H., The Quadtree and Related Hierarchical Data Structures, Computing Survey, Vol.16, No.2, pp187-260, June, 1984.

[11] Otten, R, Automatic Floorplan Design, Proc. 19th DAC, pp261-267, 1982.

[12] Luk, W.K., Tang, D.T., and Wong, C.K., Hierarchical Global Wiring for Custom Chip Design, Proc. 23rd DAC, pp481-489, 1986.

[13] Robinson, J.A., LOGIC: Form and Function., Edinburgh University Press, 1979.

[14] Chang, C.L. and Lee, R.C.T., Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

[15] Lloyd, J.W., Foundations of Logic Programming, Springer-Verlag, 1984.