MOSAICO: User's manual

M. Beardslee, J. Burns, A. Casotto, M. Igusa,
F. Romeo, A. Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley

August 24, 1990

Contents

Chapter 1

The MOSAICO system

1.1 Introduction

This manual describes features and use of `MOSAICO`, a complete set of tools for floorplanning, placement, and routing of macro-cells. Macro-cells are functional blocks that are created by module generators, manual layout, or any number of other means. Since `MOSAICO` is tightly coupled to Berkeley's logic synthesis system [5] macro-cell functionality can be optimized with the characteristics of the target implementation style in mind. The `MOSAICO` floorplanner can then be used to produce specifications for the aspect ratios and pin positions of each macro-cell. The combination of the optimized logic equations and the geometric information from the floorplanner comprise the input to module generators; the macro-cells output are then used in the placement phase of `MOSAICO`.

To achieve design-rule independence the entire `MOSAICO` system operates at the symbolic-layout level, rather than at the physical-layout level. A spacing step is carried out prior to mask generation to ensure that the layout is design-rule correct.

1.2 Pipeline Structure

`MOSAICO` consists of five basic steps executed in sequence as a *pipeline*. The pipeline is represented in Figure 1.1 together with the sequence of symbolic views that are stored at each step in the OCT data manager [4]. Each step is described in detail in the following sections.

Figure 1.1 illustrates one of the most important features of `MOSAICO`: its modularity. Modularity is achieved by ensureing that all the information produced at each stage of the pipeline is stored in the OCT data base. Each of the tools in the pipeline reads its input data from an OCT view in the data manager and produces as its output another view which contains the previous

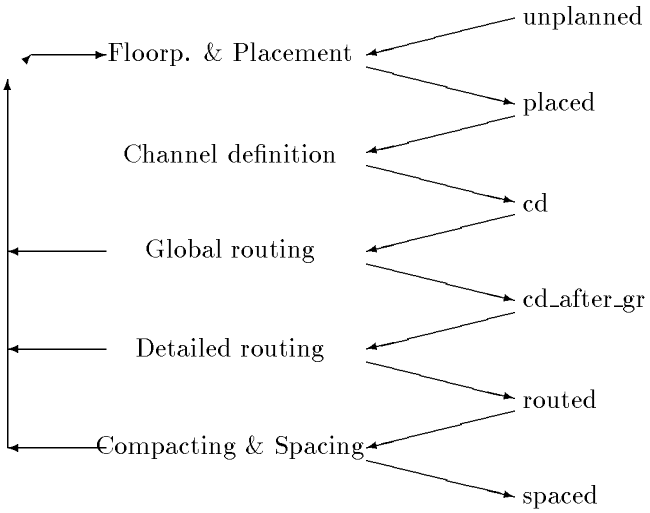

Figure 1.1: The MOSAICO pipeline

data updated with the information added by the tool. Of course, new views need not be created; the original view can be annotated in place by each tool if desired.

1.3 Floorplanning and Placement

In the `MOSAICO` *environment, the activity of floorplanning and placement (F&P) is handled in a unified framework; in fact there is no distinction between floorplanning and placement at the algorithmic level. At the floorplanning stage some of the parameters of the macro-cells (aspect ratio, pin positions) may be varied; at the placement level all the macro-cell parameters are fixed and the only remaining degrees of freedom are the positions and orientations of the cells. The F&P phase maintains a global view of the chip and directs the interactions with the module generators and the timing verifiers.*

The F&P iterations are started by an initial floorplanning operation in which the most detailed description available for each cell is retrieved from OCT. For some macro-cells the complete layout may be known. For others only a set of constraints, such as requirements on pin positions or aspect ratio, may exist. Other cells may be described at the logical or behavioral level. For such cells, called soft cells, an area estimation is performed, based on the complexity of the cell and on the features of the module generator that will be used to produce it.

The terminals of a soft cell are free to move about the border of the cell. We call them floating terminals.

After each iteration the floorplanner produces as output the position and orientation of each cell and also the shape and pin positions for those cells that were not completely specified. This information is passed to the module generators (e.g. Gem, Topogen [1], Wolfe [7]).

The input to the module generators comes from two sources, one being the floorplanner which specifies geometric constraints as described above. The other data is the logic description produced by the logic optimization tool Mis [2].

If a module generator is unable to satisfy all the constraints imposed on a particular cell, the actual shape of the generated cell is fed back to F&P and a new iteration is started.

The F&P module also interacts with the timing analyzer Hummingbird [8]. The purpose of the timing analyzer is to critique the floorplan. It performs two tasks: first it checks the arrival times of signals required for the correct functioning of the circuit, then it marks each net with information about the maximum and minimum delay through that net. If the requirements are not met, a new iteration of the floorplanner is necessary. The timing information is used to update the constraints imposed by F&P on the maximum or minimum length of the nets.

As the F&P task is repeatedly executed, more and more refined representations of the chip are generated. As the macro-cells gain their physical implementations, the information available to the F&P tool increases, and the floorplanning task becomes a placement task.

1.3.1 The FLOORPLAN bag

To indicate the fact that a cell is soft, a bag named "FLOORPLAN" must be attached to the interface facet of the cell. The FLOORPLAN bag contains constraints for floorplanner and module generators. The bag is always and only in the interface facet. It contains the following objects and properties:

A box This box has two meanings. For the floorplanner, the area of the box represents the area of the soft cell and its shape represents the desired aspect ratio of the cell. The aspect ratio can vary in the range specified by two properties **MIN_Y_OVER_X** and **MAX_Y_OVER_X** (see below). The box may be attached to a layer (for example layer PLACE) so that it can be seen in VEM, but such attachment is optional.

For a module generator, the box represents the area suggested by the floorplanner for the implementation of the cell. `MOSAICO` itself can be considered a module generator, and the box in the FLOORPLAN bag in the interface of the chip to be placed specifies the area available for floorplanning and placement.

MODULE_GENERATOR is a string property, naming the module generator which is supposed to implement the cell. This property is used by the floorplanner and by `alba`. Examples: "", "wolfe", "gem", "mosaico".

MIN_Y_OVER_X Minimum aspect ratio for a soft cell.

MAX_Y_OVER_X Maximum aspect ratio for a soft cell.

FORBIDDEN_EDGES (`PUPPY` only) Easy way to put edge constraints on the floating pins of the cell. Example = "LEFT TOP".

1.3.2 Tools

puppy does not consider soft cells with variable aspect ratio yet. See also "Puppy User's Manual".

alba is a program that performs a number of checks before and after floorplanning, allowing the user to change some of the information contained in the FLOORPLAN bag.

makeSoft is a program that creates the information necessary to consider a cell as a soft cell.

1.4 Channel Definition and Ordering

The channel definition task is to partition the available routing area into smaller regions for the subsequent global and detailed-routing phases. Once the regions are determined they have to be ordered so that each region can be routed in a sequence such that its width can be adjusted independently of all previously-routed regions.

Otten [6] presented a channel definition algorithm which produces a feasible routing order on a restricted class of placements called rectilinear slicing structures. The advantage of the slicing method is that it induces a simple hierarchical decomposition of the routing problem into independent subproblems. The chip is divided by the first slice into two blocks; the slicing procedure is then recursively repeated on the sub-blocks. Each slice represents a single routing area and the order in which the slices are found is the reverse of the routing order.

Given a placed macrocell layout, the channel definition program `atlas` will figure out where the routing channels are located and the order in which to route them. Channel definition is executed prior to global routing. Formal terminals (IO's) are expected to reside on the outer perimeter (in a rectangle) of the macroblock or chip. For each side, top, bottom, left and right, the formal terminals are merged together into one instance consisting of a strip of terminals. `Atlas` will try to use a slicing tree for the channel definition. When a slicing tree is impossible to create, `atlas` will use irregular channels, i.e. "Z" shaped channels. Currently, "Z" shaped channels are allowed when using the -Z option in mosaico. We suggest that the user use `atlasRpc` to visually inspect the channel definition and ordering. If the channel definition is not what is desired, then the user can interactively change the placement and redo the channel definition within the rpc applicaion.

The input channel definition program is a placed view. Polygons of the instance's master's PLACE layer are used to abstract the outline of the instances. These polygons are used by the channel definition to see where the routing channels reside. If the PLACE layer does not exist, then the bounding box is used as the outline of the instance. If the outlines of the instances overlap, then atlas will automatically move (all) instances apart to resolve the overlap.

See also Appendix A.

1.4.1 Tools

atlas is the channel definition program.

atlasRpc is the RPC version of `atlas`. This program is useful for visually inspecting the channel definition using vem.

1.4.2 Properties

CDChanDef is a program defined bag which is the symbolic representation of a channel.

CDChanOrder is a program defined property attached to the CDChanDef bag whose value is the order in which the channel should be routed.

CDCorners is a program defined bag attached to each instance to tell the global router which boundaries of the instance lie on which channel.

CDCorner is a program defined property attached in a "linked" list with the CDCornerEdge bag whose value is the index into the PLACE polygon of the instance to denote the part of the instance polygon which is bordering the channel the CDCorner property is attached to. Hopefully no one should need to understand this property except the global router.

CDCornerEdge - see CDCorner propery.

CDOuterChannel is a property attached to the external channels.

1.5 Global Routing

After the channel definition and ordering step described in the previous section, the data must be prepared for the global router. The channel graph, with the *positions of the terminals on its edges, and a net list, are the inputs necessary for global routing. In the channel graph, nodes represent intersections between channels while edges represent channels or sections of them. Each edge in the graph is assigned a weight that represents the maximum number of tracks that can be accommodated in the channel. Once the channel graph is built, all the connected terminals of each macro-cell instance are projected to the closest edge in the graph. The weights on the edges are interpreted as capacity constraints. It is important to note that the input to the global router is completely symbolic and therefore totally independent of the layout style.*

When the global router finishes, each net is represented by of a sequence of subnets, each of which is assigned to one of the edges of the channel graph. When a net exits one channel to enter another one a pseudo-terminal is created.

1.5.1 Tools

cds is the program that interfaces the Channel Definition System (CDS) to the global router.

`cds` consists of two phases: The first phase prepares the data in the format suitable for the used global router, presently `TimberWolfMC`. In the second phase, `cds` reads the results produced by the global router and stores them back into OCT so that they can be used to complete the detailed routing by Spider.

1.5.2 OCT structure

After the global routing, each net consists of a number of segments, each of which is assigned to a channel. The connectivity among nets is obtained with pseudo-terminals located at the intersection of channels. These pseudo-terminals can be considered as place holders for the actual connectors that will be introduced by spider in the detailed routing phase. Niether the pseudo-terminals nor the subnets have, at this stage of the pipeline, any geometry attached to them. Their purpose is to combine the connectivity information with the path on the channel graph that has been selected by the global router for each net. This information is stored in the database as follows:

CDSubnet an OCT object of type net. It is contained by the relative net. Each subnet contains exactly two pseudo-terminals and actual-terminals.

CDPseudoTerm an OCT object of type bag. It is contained by exactly two CDSubnets. It doesn't contain any OCT object.

1.6 Detailed Routing

The channel server `Cprep` *analyzes information from the global router and the channel definition and ordering phase and produces channel specifications for* *the router server* `Spider`*.* `Spider` *performs the detailed routing of each channel using this information and the design rules for the fabrication technology.* `Spider` *can handle routing regions with irregular edges, and wires with different widths, which is necessary to route special nets like power and ground in the same channel with standard signal nets.*

`Spider` *serves as a data translator from OCT to a format usable by a channel router in the* `MOSAICO` *library (*`yacr`*,* `mighty`*) and then back to OCT. From the OCT channel description* `Spider` *extracts a non-uniform symbolic grid in which the grid lines are placed to obtain the best possible alignment with the actual locations of the pins. If the general-area router* `mighty` *is used, an estimate of the space necessary to complete the routing must first be computed.*

1.6.1 Tools

Cprep Pulls each channel out of the database in proper order and sets up routing information.

spider Interfaces non-OCT based routers to and from OCT.

YACR General two layer channel router (fastest and powerful, YACR almost always routes channels in the minimum number of wiring tracks).

mighty Maze routing based channel and switch box router (usually routes a channel in the minimum number of tracks but is slower that YACR)

1.7 Power and Ground Routing

The routing of the power and ground nets is more involved than the routing of signal nets because of problems related to voltage drops and maximum allowable current densities. As a consequence, power and ground nets may have variable wire-segment widths.

In modern technologies the presence of two layers of metal makes it unnecessary to require planar routing techniques for power and ground connections. The policy used in `MOSAICO` *is to use the same set of tools to route both the standard signals and power and ground. However, power and ground connections require some additional processing.*

Mosaico assumes that the amount of current that can flow through a macro-cell without damaging its internal power (ground) connection is unknown. Thus, none of the power(ground) nets are routed through any macro-cell.

After the global routing stage the current requirements of the power nets are found at the boundaries of the channels. This information is passed to the channel server `spider`*. After* `spider` *performs the symbolic routing of each channel, the segments of the power and ground nets in the channel are correctly sized.*

The nets with type "SUPPLY" or "GROUND" are treated specially by `MOSAICO`. The terminals on those nets are searched for the property PEAKCURRENT, and the width of the nets is computed in order to accomodate such current. The value of the maximum current density for each layer is currently stored in the `MOSAICOrules` file in the technology directory. The width of the power and ground wires in Lambda is quantized to the closest of the values in the sequence:

$$2, 3, 4, 5, 6, 8, 10, 12, 14, 17, 20, 25, 30, 35, 40, \ldots$$

1.7.1 Properties

NETTYPE : string valued property (*OCT Symbolic Policy*). Legal values are: `SIGNAL,SUPPLY,GROUND,CLOCK`.

TERMTYPE : string valued property (*OCT Symbolic Policy*). Legal values are: `SIGNAL,SUPPLY,GROUND,CLOCK`.

PEAKCURRENT : real valued property (*OCT Symbolic Policy*), expressing the peak current through the terminal in amperes. Example: `0.005`

1.7.2 Tools

PGcurrent must be run after global routing. Each P&G net must have only one source for PGcurrent to work properly. Power nets with more than one source must be partitioned before running PGcurrent. No tool will allow you to do this partitioning yet.

spider routes power and ground as signal nets initially, then uses information from PGcurrent to correctly size the wires.

1.8 Flattening

`MOSAICO` creates the routed view in a hierarchical way, one channel at a time. The hierarchy is then flattened using `octflatten`.

1.8.1 Tools

octflatten a program for flattening instasnce hierarchies.

nle a library of routines that allow flattening and merging of instances. Used by `octflatten`.

nleRpc RPC version of `nle`.

1.9 Via minimization

The input to the via minimizer `mizer` *is the flattened layout containing macro-cells and wiring. At this point, all horizontal wires are on one metal layer and all vertical wires are on the other metal layer. By reassigning some wire segments to the opposite layer, the via minimizer reduces the total number of vias. Note that the locations of the segments are not changed, only their layer assignment. The terminals of a macro-cell are usually implemented in one or more layer. These implementation layers determine which layers* `mizer` *may use to connect to that terminal.* `mizer` *does not check the actual geometrys of the macro-cell to determine this information. Because of this* `mizer` *can not handle over-the-cell routing (yet).*

1.10 Layout Spacing

All steps in the `MOSAICO` *pipeline are carried out at the symbolic-layout level. Symbolic layout spacing (or compaction) is thus an essential part of the system for several reasons. First, all designs must be spaced to ensure that they are design-rule correct; this has the advantageous effect of eliminating design-rule checking, since the layouts are correct-by-construction. Second, the use of generalized symbolic layout provides a mechanism for producing technology-independent designs. The spacing techniques used in* `MOSAICO` *are capable of updating the symbolic layout primitives themselves (e.g., transistors, contacts), as well as the spacings between them. Also, by the use of a variety of spacing techniques, various optimizations of the layout can be performed over a range of area/cpu-time trade-offs.* **Sparcs** *[3] is the spacer presently used in* `MOSAICO`*.*

1.10.1 Tools

sparcs a symbolic-layout compactor.

padplace with the -S option, `padplace` puts the proper constraints for the pads.

1.11 Verification and evaluation

1.11.1 mosaicoRC

`mosaicoRC` performs 4 different checks to asses the correctness of the routing process. It checks the consistency, from a net-list point of view, of two facets; it checks for disconnected terminals, disconnected paths, and possible violations of the symbolic policy with respect to the routing procedures.

The consistency check is performed in two steps. In the first step, the first facet is selected and all the nets attached to it are generated. For each one of the nets, all the terminals are generated. Every terminal is matched by name with a corresponding terminal on the second facet. If the matching fails, an error is recorded. The procedure is then repeated with the facets exchanged.

To check for disconnected terminals, mosaicoRC proceeds as follows: For each net, generate the first terminal attached to it and start to follow the path originating from it. The path is followed until either it ends on a terminal or it branches to two or more other paths. Each one of these is then followed. The procedure is repeated recursively. All the terminals are marked as soon as they are reached by one of the paths starting from the first terminal. When, for the net under consideration, all paths have been examined, all terminals attached to it must be marked. For each one of the terminals that is not marked, an error is detected and recorded.

The checking for disconnected paths is accomplished by generating all paths attached to the layers used to perform the routing. For each of the paths it is checked that connects to exactly two between terminals and connectors.

The checking for violations of the routing policies comes for free while the other checks are being performed.

The inconsistencies eventually determined are reported in two different formats: a textual one and a graphical one. For the textual messages a file named *circuitName*`_mosaicoRC.out` is created and all the violations discovered are reported in it in an textual format. The violations that are related to geometric entities are collected into a bag which is stored in OCT.

Once the execution of `mosaicoRC` is completed, the errors can be displayed graphically by means of Vem by the following sequence of operations:

run Vem;

open the contents view of the facet on which `mosaicoRC` has been run;

select the bag named ROUTING_ERRORS;

At this point the location where errors occurred will be highlighted. Each of the object that is highlighted will have a property named ROUTING_ERRORS attached to it. The value of the property is a string that describes the type of the error detected. Some of the errors, like for example differences between two facets, cannot be represented graphically. In this case the error is reported in the textual format only. As a consequence, the most complete report of the inconsistencies detected by `mosaicoRC` is contained in the file *circuitName*`_mosaicoRC.out`.

OCT Structure

ROUTING_ERRORS an OCT object of type bag. It is contained by the facet and contains the objects for which an error has been detected.

ROUTING_ERROR an OCT object of type property. It is contained by the bag ROUTING_ERRORS and by the object for which an error has been detected. It does not contain any object. The property is of type string. Its value is a short message describing the type of error detected.

1.11.2 chipstats

`chipstats` computes various statistics on an OCT view. It can be run on the routed view as well as on any other symbolic view. It computes the length of each net (or its bounding box, if there is no routing), the area of the chip, and the number of vias.

Chapter 2

How to run MOSAICO

This chapter gives a brief introduction on how one can use `MOSAICO` to place and route a chip.

2.1 Get a correct net list

The input for `MOSAICO` is a net list, that is, a description of the cells that constitute the chip with all the nets that interconnect terminals of the cells. This net list must be stored in the OCT database in a facet obeying all the rules of the *OCT Symbolic Policy*. Once you have a correct net list `MOSAICO` will do the placement, routing and compaction.

In order to get a net list you can use one of the following tools:

bdnet is a net list translator distributed with the octtools. See relevant man page.

Some special rules should be used when you set up a bdnet file to describe a chip for `MOSAICO`.

- Remember to attach the NETTYPE (`SUPPLY` or `GROUND`) property to your supply and ground nets;
- Remember to attach the TERMTYPE (`SUPPLY` or `GROUND`) property to all the supply and ground formal terminals in the interface facet of the chip.
- Do not worry about pads. Just provide one formal terminal for each pad. You can use `padplace`to implement the pads and to place them around the chip;
- You can use the bdnet file to add special constraints for the placement, like the FLOORPLAN bag (see section 1.3.1), or the LEGAL_TRANSFORMATIONS property for the instances.

tw2 provides an interface between the TimberWolfMC input format and OCT. See the man page for `tw2` for more details.

2.1.1 Editing the net list, merging and flattening

`NOTE: you can skip this section on your first reading.`

Suppose that you have a *correct* net list and that you would like to edit it. For example, you might want to partition your design into two or more smaller pieces. The basic tool to do this is the `nle` library. You can either use `nleRpc` to work interactively or `alba` to work in batch mode.

In order to define a partition with `alba`, you need to create the following structure in the contents facet of your chip:

- A bag whose name begins with the letters "`PARTITION`". The name can be longer. For example "`PARTITION_my_subchip`" is a legal name. The bag must contain all of the following:
- All instances that belong to the partition;
- property `PARTITION_NAME` whose value is a string which is the name of the new cell to be created;
- property `PARTITION_TYPE` with value `HARD` or `SOFT`, according to the type of merging that you want to perform (see man page for `nle`);
- optional property `MODULE_GENERATOR`, with value the name of the program that you want to use to process the partition. For example, this could be "wolfe" or "mosaico".

This structure is used by `padplace`to specify how the pads should be merged into pad blocks in order to simplify the job of the router.

2.2 Implement and place pads

Suppose you have your chip correctly described in the OCT view `chip:unplaced`. The next thing to do is to implement the pads. First of all, get a list of all the formal terminals and of their properties using `padplace`.

```
padplace -l chip:unplaced > chip.pads
```

The file `chip.pads` contains the list of all the terminals. The format of this file is such that it can be fed back into `padplace`. You can edit the file to add placement information like TERM_EDGE and TERM_RELATIVE_POSITION, TERM_ROW (Fig.2.1).

Then you can ask `padplace`to implement all the pads. Normally `padplace`comes with the following pad families in the *scmos* technology:

_2uFamily

_msuFamily

_cFamily

Suppose that for your chip you want to use the `_2uFamily`, and that you want `padplace`to use the information contained in `chip.pads` to decide where each pad should go. Then you can type:

```
padplace -u _2uFamily -D chip.pads -o chip:with_pads chip:unplaced
```

If you do not have a clear idea on where to place the pads, you can use the `-n` option of `padplace`to obtain a pad placement that attempts to minimize the length of the nets connected to the pads. This is especially useful with the `_cFamily`. Try:

```
padplace -u _cFamily -o chip:with_pads chip:unplaced
```

Use `vem` to check that everything is like you want it. Then go on to the next section.

2.3 Place the chip

First, it is useful to run `symclean` on your design. `symclean` checks an Oct facet for conformity to Oct symbolic policy and can try to clean up problems.

```
symclean [-duv] chip:placed
```

By default, `symclean` will not modify the cell. The `-d` flag allows `symclean` to make the necessary modifications. If the `-u` flag is specified, unimplemented formal terminals will not be deleted (formal terminals from MIS and BDNET do not have implementations). `Vulcan` will be run if any changes have been made. The `-v` flag turns off the use of `vulcan`.

Now, you can either run `MOSAICO` with the `-P` option, which will invoke `PUPPY` as a placement tool and then go on to the routing, or you can invoke `PUPPY` directly with:
`puppy -o chip:placed chip:with_pads`

2.4 Run MOSAICO

2.4.1 Easy to use

You now have a placed chip in `chip:placed`. To route the chip type:
`mosaico chip:placed`

```
#
# Formal Terminal description for ram:flat
# padplace recognizes the following keywords:
# TERMTYPE DIRECTION
# TERM_FLOATING_EDGES TERM_EDGE
# TERM_RELATIVE_POSITION TERM_RELATIVE_POSITION_STEP

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
TERMTYPE SIGNAL
DIRECTION INPUT
TERM_EDGE LEFT
FORMAL_TERMINAL shr
FORMAL_TERMINAL shl
FORMAL_TERMINAL clk
FORMAL_TERMINAL clear
TERM_EDGE RIGHT
FORMAL_TERMINAL data<2>
FORMAL_TERMINAL data<1>
FORMAL_TERMINAL data<0>
DIRECTION OUTPUT
TERM_EDGE TOP
FORMAL_TERMINAL char2_out<3>
FORMAL_TERMINAL char2_out<2>
FORMAL_TERMINAL char2_out<1>
FORMAL_TERMINAL char2_out<0>
TERM_EDGE BOTTOM
FORMAL_TERMINAL char4_out<3>
FORMAL_TERMINAL char4_out<2>
FORMAL_TERMINAL char4_out<1>
FORMAL_TERMINAL char4_out<0>
```

Figure 2.1: Example of a file created by `padplace`and edited

Done! The output view is `chip:placed.spaced`, but you can specify another output view with the `-o` option. Many intermediate files and directories are created by `MOSAICO`, these will deleted after they are no longer needed if you run `MOSAICO` with the addition of the `-c` option.

```
mosaico -c chip:placed
```

If you have already run `MOSAICO` without the `-c` option, you can clean the unnecessary files using the `-C` (capital C) option:

```
mosaico -C chip:placed
```

2.4.2 Time and space requirements

The whole process might take anywhere from 5 minutes for a small chip with less than 10 cells and about 20 nets, to a few hours for large chips with a few tens of cells and hundreds of nets. The two standard benchmarks (`ami33` and `xerox`) take about 30 CPU minutes each on a VAX 8800.

`MOSAICO` creates a number of intermediate files, directories and facets. The `RTstuff` directory contains all the channels before and after routing. It also contains the log files generated by spider. These files can be useful to the expert user for debugging, but under normal conditions they contain no useful information. Typically, each channel requires about 20 to 200 kBytes of disk storage, depending on the amount of routing. Large channels (hundreds of nets, 50 or more tracks) can grow up to half a MByte or more.

OCT cells with name of the form `<chip>.ioleft, <chip>.ioright, <chip>.iotop, <chip>.iobottom` contain the I/O pads, which are merged together in order to reduce the number of channels to be routed.

The file `mosaicoDemoRpc.seq` can be used together with the RPC application `mosaicoDemo` (see manual).

The file `<chip>_mosaicoRC.out` contains the output from `mosaicoRC` and it is created only if any error is detected in the routing.

The following views of the chip are also created:

`placed.cd` : containing the channel definition;

`placed.cd_after_gr` : containing the global routing information;

`placed.hier` : containing the complete hierarchy of channels;

`placed.flat` : which is the flattened version of `placed.hier`;

2.4.3 If something goes wrong

To be realistic, sometimes something goes wrong, often due to some misunderstanding or some missing property. To find out where the problem is, it is always convenient to redirect the output from `mosaico` to a file:

```
mosaico <chip>:placed >& mout &
```

Study this file (`mout`) to discover all the errors that have occurred. Then refer to the documentation relative to the appropriate tool to solve the problem.

2.4.4 Common problems

Overlapping instances: If there are overlapping cells, the definition of channels impossible. If the ovelapping is minor, `MOSAICO` will first try to expand the placement by a fixed factor of 1.1. If overlap persists after the expansion, the run will be aborted. It is up to you to solve the problem. If you used `PUPPY` for the placement, you can try to rerun it using a slightly higher weight for the overlap cost. Also, you could just try to rerun `PUPPY` keeping the same parameters. `PUPPY` uses a probabilistic algorithm and it will generally produce different placements in each run (however the cost of the final placement will be about the same).

Dropped nets:

Terminals too close: Terminals must be separated by a minimum amount that is dependent on the technology being used regardless of the layer(s) the terminals are on. `MOSAICO` (`spider` in particular) will complain somewhat like this:

```
ERROR - pins too close
        PIN VDD at loc 40620, 63300 width 400
        PIN HandXHS1 at loc 40450, 63760 width 60
```

This indicates that terminals VDD and HandXHS1 are too close and their locations and widths. This situation must be fixed because one of these terminals will be left unconnected.

Terminals on a wrong layer: known layers for standard scmos technology are: `MET1, MET2, POLY`. Terminals on other layers cannot be used.

Other routing errors: `MOSAICO` performs double checking of the routing using the program `mosaicoRC`. If an error is detected, you should check the file `<cellname>_mosaicoRC.out` for a textual description of the error. In the file you will also find instructions on how to use `vem` to see the location of the error.

Power and ground nets are not proper size: this is generally due to lack of properties in the right places. See section 1.7 for more details.

2.4.5 Known problems

Placement with overlapping instances: The algorithm used by `PUPPY` (Simulated Annealing) performs a minimization of a cost function, and it does not guarantee an overlap free placement. It is generally easy to obtain an overlap free placement by properly tuning the annealing parameters using the interactive graphical interface provided by `PUPPY`. See the *Puppy Users Guide* for more details.

Inefficient use of empty rooms: `MOSAICO` performs at its best on dense placements, with no empty rooms (large empty areas). If your placement has empty rooms, they might stay empty.

2.5 Floorplanning

Suppose that your chip has one cell which is not yet implemented. Suppose also that the module generator that you are going to use to implement the cell gives you some flexibility on the shape of the cell as well as on the position of its terminals. The `MOSAICO` system provides some support to decide which constraints to set for the implementation of the cell. This will be illustrated by an example.

The floorplanning tool in `MOSAICO` is `PUPPY` which cannot handle cells with variable aspect ratio (at least for the time being). Thus you have to decide the shape of the cell. Suppose that you want this shape to be a rectangle 100 by 300. Type the following command:

```
makeSoft -w 100 -h 300 MYSOFTCELL:someview
```

And this is all we can do for the aspect ratio of the cell.

`PUPPY` does handle floating pins on soft cells. Some terminals are only allowed to be floating on some of the edges of the cell, others can float anywhere. One easy way to tell `PUPPY` that the terminals are floating is to use `makeSoft`. For example:

```
makeSoft -F LR MYSOFTCELL:someview
```

specifies that all the terminals of the cell can float but only on the left or on the right edge (`-F LR`).

A more detailed representation of the constraints on the floating pins can be expressed using `padplace`:

```
padplace -l  MYSOFTCELL:someview > someFile
```

to get a list of all the terminals of the cell. Then you can edit the file `someFile` which might look something like Fig.2.2. The file says that the 3 input terminals can float only on top and left edges, while the output terminals can float only on the right edge. Give this information to `padplace`again:

```
# comments begin with a \# and end with newline

       TERM_FLOATING_EDGES TOP LEFT
FORMAL_TERMINAL in.1
FORMAL_TERMINAL in.2
FORMAL_TERMINAL in.3
       TERM_FLOATING_EDGES RIGHT
FORMAL_TERMINAL out.1
FORMAL_TERMINAL out.2
       TERM_FLOATING_EDGES RIGHT TOP
FORMAL_TERMINAL clock
```

Figure 2.2: Example of file to describe floating terminals. Indentation is used only for clarity. You can use this file also to specify other properties of the terminals, like DIRECTION and TERM_TYPE

```
padplace -D someFile  MYSOFTCELL:someview
```

2.5.1 Important note on floating pins

Floating terminals are very hard to deal with, especially if there are many (hundreds) of them. `PUPPY` finds it hard to decide on an orientation for a cell whose terminals are all floating because at high temperature almost all the orientations are equivalent. This means that for reasonable running times, the resulting placement might not be very good (it might even be unfeasible).

It is always better to constrain the floating terminals as much as possible. The possibility to do this depends, of course, on the specific problem. In some cases it makes sense to constrain the input terminals all on one side, while letting the output terminals float on the opposite side.

Depending on the structure of your chip, there might be some pad arrangements that make more sense than others. Although it would be desirable that the tools themselves automatically detect this condition, this is not yet the case. Locally, for each soft cell, `padplace`with its option `-n` can generally do a good job.

2.5.2 Wolfe and `MOSAICO`

If a chip contains standard cells like those in the MSU library, you should use `wolfe` to obtain a dense layout of these cells.

You should determine an optimal partitioning of all the standard cells in your chip. Then place and route each partition using `wolfe`, then use `MOSAICO` to wire all the pieces together. The process of determining an optimal partition is currently not supported by any tool (i.e. it must be done by hand, trying to be a little smart about it).

Please consider the following ideas:

- You can control the aspect ratio of a `wolfe` cell using the `-r` option (specifying the number of rows). Alternatively, you can use `makeSoft` to specify a target bounding box for the cell.
- You can control the position of each terminal in a wolfe cell using the `TERM_EDGE` and `TERM_RELATIVE_POSITION` properties. `padplace`provides an easy ASCII interface to enter such properties, and `PUPPY` can choose the optimal value for those properties based on its own floorplanning criteria.
- It is not a good idea to have a partition containing only one standard cell, but it is disastrous to have many such partitions. You can merge standard cells into the same partition using `nle` or `alba -p` (please refer to the appropriate manuals).

2.5.3 Hand layout and `MOSAICO`

Some rules should be followed to allow a painless integration of hand layout in a chip which has to be routed by `MOSAICO`. The rules are listed here ordered by importance:

- Do not allow terminals to overlap.
- Keep the terminals at least 7 lambdas apart (for scmos) [1]. One easy way to guarantee this result even after compaction is to implement all the terminals using MET1-MET2 vias.
- The layers connected to are MET1, MET2, and POLY. A terminal must contain at least one geometry on these layers or it cannot be connected by `MOSAICO`.
- Place all the formal terminals along the periphery of the cell so that they can be easily accessed from outside the cell.
- Try to use the same layer to implement all the terminals that appear on the same side of a cell. Example: make all the terminals on the TOP of the cell in MET1 and all the terminals coming out form the LEFT edge of the cell in MET2. The effect of applying this rule is that of producing "better looking" routing. This is by far the least important of all the rules.

[1] The recommended spacing between terminals is the minimum center to center distance of two MET1-MET2 vias

The restrictive rules mentioned above make it easier for MOSAICO. However, MOSAICO has been designed to handle much more complicated situations, like irregularly shaped cells, terminals internal to the cell, and overlapping terminals. Please be warned that each of these situations causes MOSAICO to go through some complicated heuristics to determine the best way to connect the problematic terminals.

Any internal terminal must be reachable from the outside using a straight segment in one of the layers used to implement the terminal. Unreachable terminals are generally detected by vulcan.

MOSAICO (and more precisely cds) uses the concept of "projection of a terminal onto a channel". There are 4 channels surrounding each cell, independent of the shape of the cell. Each terminal is projected onto one channel. Sometimes a terminal is projected into more channels to give more degrees of freedom to the global router.

There are a number of ways in which terminals interact among themselves. If two terminals overlap, they must be projected onto two different channels. Similarly, if the straight segment necessary to connect to an internal terminal "A" overlaps another terminal "B", then terminals "A" and "B" must also be projected onto different channels.

If it is not possible to map interacting terminals onto different channels, MOSAICO will not connect one of them. A loud warning is generated in such a case.

2.6 Example output from MOSAICO

This is the untouched output from MOSAICO running on a small test chip. Depending on the version of MOSAICO you are running some details might be different.

```
MOSAICO: ********************************
MOSAICO: * MOSAICO version 0.11 7/22/88 *
MOSAICO: ********************************
MOSAICO: Running on ck:puppy (in /net/eros/eros5/mosaico/examples/ck)
MOSAICO: Technology is: scmos
MOSAICO: Technology directory is: /octtools/lib/technology/scmos
MOSAICO: >>>>>> 1 1 1 1 <<<<<
MOSAICO: Starting channel definition step (atlas) ...
Channel definition and ordering (rev 1.7 88/05/07 19:47:51)
-- Start "Atlas" on Tue Jul 26 11:52:21 1988
Initial memory size = 0.31M
Merged ios - starting pass 2 of atlas ...

Total Cpu Time = 0.27 min
```

```
Maximum memory size = 0.95M
-- End "Atlas" on Tue Jul 26 11:52:47 1988
MOSAICO: >>>>>> 2 2 2 2 <<<<<
MOSAICO: Global routing (cds -> TimberWolfMC -> cds)

CDS VERSION 1.5 : Tue Jul 26 11:33:50 PDT 1988


CDS environment <- Global Router environment(TW)


Reading oct input ck:puppy.cd:contents:


****************  ck  ******************
Number of channels           = 48
Number of terms              = 85
Number of connected terms    = 78
Number of nets               = 31


Printing input files for TimberWolf
Starting the global router with the following command
---> TimberWolfMC TWck
Global router completed


CDS environment <- Global Router environment(TW)


Reading output files from TimberWolf
Storing informations back to OCT
Flushing data back to OCT
MOSAICO: >>>>>> 3 3 3 3 <<<<<
MOSAICO: Adjusting term/subnets attachements
MOSAICO: Cleaning up global router output
MOSAICO: >>>>>> 4 4 4 4 <<<<<
MOSAICO: Starting power and ground current calculations (PGcurrent)...
PGcurrent: found a source FT_VDD.VDD (instance terminal) for net vdd
of type SUPPLY
PGcurrent: found a source VSS (instance terminal) for net vss of type
GROUND
MOSAICO: Cleaning up previous router debug stuff (RTstuff directory)
...
```

```
MOSAICO: >>>>>> 5 5 5 5 <<<<<
MOSAICO: Starting routing step (cprep) ...
Router driver and channel preparation (rev 1.10 88/06/15 14:04:09)
-- Start "Cprep" on Tue Jul 26 11:54:27 1988
Initial memory size = 0.27M
There are 12 channels in cell ck puppy.cd_after_gr.
This will take a while.  Take a coffee break or something.
Channel router server is working on RTstuff/ck.ch0
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch0
Channel router server is working on RTstuff/ck.ch1
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch1
Channel router server is working on RTstuff/ck.ch2
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch2
Channel router server is working on RTstuff/ck.ch3
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch3
Channel router server is working on RTstuff/ck.ch4
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch4
Channel router server is working on RTstuff/ck.ch5
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch5
Channel router server is working on RTstuff/ck.ch6
WARN Top & bottom instances overlap
INFO Top instance will be moved 2060 to top
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch6
Channel router server is working on RTstuff/ck.ch7
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch7
Channel router server is working on RTstuff/ck.ch8
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch8
Channel router server is working on RTstuff/ck.ch9
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch9
Channel router server is working on RTstuff/ck.ch10
WARN Top & bottom instances overlap
INFO Top instance will be moved 2230 to top
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch10
```

```
Channel router server is working on RTstuff/ck.ch11
Running channel router interface (spider) using (yacr) ...
... done routing channel RTstuff/ck.ch11

Total Cpu Time = 4.52 min
Maximum memory size = 1.51M
-- End "Cprep" on Tue Jul 26 12:09:50 1988
MOSAICO: >>>>>> 6 6 6 6 <<<<<
MOSAICO: Flattening step 1 (mosaicoflat) ...
/octtools/bin/mosaicoflat: VERSION 2.01 : Tue Jul 26 11:49:15 PDT 1988
MOSAICO: >>>>>> 7 7 7 7 <<<<<
MOSAICO: Vias minimization (viamin) ...
via minimizer opening ck:puppy.flat
technology ~octtools/lib/technology/scmos
78 fixed met1 terminals
TOTAL 49 essential vias
85 vias needed
time 48.92
MOSAICO: >>>>>> 8 8 8 8 <<<<<
MOSAICO: Flattening step 2 (mosaicoflat) ...
/octtools/bin/mosaicoflat: VERSION 2.01 : Tue Jul 26 11:49:15 PDT 1988
MOSAICO: >>>>>> 9 9 9 9 <<<<<
MOSAICO: Double checking connectivity (mosaicoRC) ...
MOSAICO: >>>>>> 10 10 10 10 <<<<<
MOSAICO: Adding constraints for the compactor
/octtools/bin/padplace: Warning: No side assigned to pad VSS: cannot
add sparcs constraints
padplace: VERSION 3.0 : Tue Jul 26 11:08:55 PDT 1988
Placement of pads for ck:puppy.flat
MOSAICO: >>>>>> 11 11 11 11 <<<<<
MOSAICO: Starting global compacter (sparcs) ...
MOSAICO: >>>>>> 12 12 12 12 <<<<<
MOSAICO: Statistics on the final result
Tue Jul 26 12:17:30 PDT 1988
/octtools/bin/mosaicoStats: Version 1.0 : Chip  ck:puppy.spaced

Number of instances    = 16
Number of nets         = 31
Number of connectors   = 167
Number of vias         = 85

Average instance area  =       69625.00
St.dev. instance area  =       98471.98
Chip Area W x H        =   1467 x  1459
```

```
Chip Area              =     2140353.00 (100.00%)
Total Instance Area    =     1114000.00 ( 52.05%)
Routing + Empty Area   =     1026353.00 ( 47.95%)
Total   net area       =       87620.00
Total   net length     =       25502.00
Average net length     =         822.65
St.dev. net length     =         564.66

***** 6 LONGEST NETS
(---) NAME                  LENGTH (lambda)  AREA (lambda^2)
(  1) vdd                           2750.00          15473.00
(  2) vss                           2483.00          11340.00
(  3) net2                          1578.00           4734.00
(  4) net13                         1426.00           4278.00
(  5) net12                         1342.00           4026.00
(  6) net6                          1134.00           3402.00
MOSAICO: >>>>>> 13 13 13 13 <<<<<
MOSAICO: Vulcan on the output view ck:puppy.spaced
```

Chapter 3

Alphabetical Glossary

3.1 Floating terminals

To specify a floating terminal, attach the property "TERM_FLOATING_EDGES" to it in the interface facet. This is a string valued property consisting of a list of the edges in which the terminal is allowed to float. For example: "LEFT TOP". The empty string "" means that all edges are legal.

You can use the program `makeSoft` to make all terminals of a cell floating. If you want to be more detailed, use `padplace` with the `-l` and the `-D` options (see the man page for `padplace` for more details).

A cell with floating terminals is a SOFT cell. See also Sec.1.3.1.

3.2 Formal Terminals

Each formal terminal MUST HAVE A DISTINCT NAME.

All formal terminals in the masters must have an implementation, with the exception of floating terminals. No connection can be made to terminals without an implementation. Non-floating terminals without an implementation cause a fatal error for the routing system, although the placement tools can get away with that.

Minimum separation requirement: The terminals of the cells must be at least 11 lambdas apart when working with "scmos" technology. If they are closer than that, `MOSAICO` might not be able to connect to them.

Formal terminals of chip should be implemented, because they help defining the pads, and because that is the right way to do things anyway. The program `padplace` can help you give an implementation to formal terminals which lack one. See the manual page for `padplace` and in particular the option `-u`. See alsp Sec.2.1 and Sec.3.3.2

3.2.1 Properties

DIRECTION : string valued property: one of `INPUT, OUTPUT` or `INOUT` (*OCT Symbolic Policy*). This property is extremely important for power and ground terminals.

PEAKCURRENT : real valued property, expressing the peak current expected through the terminal in amperes for power and ground terminals only (*OCT Symbolic Policy*).

TERMTYPE : one of `SIGNAL, CLOCK, SUPPLY, GROUND` (*OCT Symbolic Policy*).

TERM_FLOATING_EDGES : list of edges on which a floating terminal is allowed to be placed. The empty string is the default, meaning that all edges are legal. Examples: `"LEFT", "LEFT RIGHT", "TOP BOTTOM", "LEFT RIGHT BOTTOM"` .

TERM_EDGE : one of `LEFT, RIGHT, TOP, BOTTOM`. This is the edge where the floorplanner recommends the terminal should be placed.

TERM_RELATIVE_POSITION is a real valued property in the range [0.0,1.0] representing the position of the terminal on the edge specified by "TERM_EDGE". For example, a value of 0.75 means that the terminal should be placed in a position corresponding to 75% of the edge, computed from left to right and from bottom to top.

TERM_ROW In the case in which you need more than 1 row of pads, this property specifies the row in which the pad should go. Row 0 is the outermost row, row 1 is the next row, and so on.

3.3 Instances

A chip has a number of instances. Every instance which is not a connector must have a distinct name. Every connector that implements a formal terminal should also have a distinct name.

MOBILITY (*OCT Symbolic Policy*) FIXED means that the cell's coordinates cannot be changed by `MOSAICO`. Default is "MOVABLE".

PLACEMENT_CLASS Used especially to recognize the pads. See section3.3.1.

LEGAL_TRANSFORMATIONS is a string valued property. The string is a list of words from the following vocabulary:

- "ALL" , the default value, means that all the transformations are legal;

- Any OCT transformation, without the prefix "OCT_": "NO_TRANSFORM", "MIRROR_X", "MIRROR_Y", "ROT180", "ROT90", "ROT270", "MX_ROT90", MY_ROT90";
- "THIS_ASPECT_RATIO" is a short hand for the list of 4 transformations which maintain the aspect ratio of the cell: "NO_TRANSFORM MIRROR_X MIRROR_Y ROT180";
- "NOT" means that all the following transformations are not legal.

Examples: "NOT THIS_ASPECT_RATIO", "ROT90 ROT270 NO_TRANSFORM".

The property can be attached to the interface of the master. If it is attached both to the instance and to its master, the instance property overrides.

3.3.1 Pads

Pads are special instances which want to be placed along the periphery of the chip. In general, pads are flagged by a special property, but in other cases pads are implicitly defined by the fact that they provide the implentation for the formal terminals of the chip.

PLACEMENT_CLASS This property is attached either to the instance or to the interface of the master. In both cases it has value "PAD".

3.3.2 Pad families

This section applies to version 3 of `padplace` and all later versions.

Pads are often designed in families, each containing a Vdd pad, a Gnd pad, an input pad, an output pad, and maybe others. Pads of the same family can abut and can create power and ground rings. In this section it is explained how a pad family can be described so that `padplace` can use the family to give a correct implementation to all the formal terminals of a chip.

Each pad in the family must be assigned a type and a direction. `padplace` will match type and direction of the formal terminal, to find the pad which is the proper implementation. `padplace` also will take into account power consumption of the pads to determine how many power and ground terminal to add to the chip bejond those already present in the chip.

A pad family can be of two types: instances (the "normal" type) or connectors.

A facet decribing the family must be of type SYMBOLIC and contain:

- One instance of each pad. The instances should be attached to INSTANCES bag or to the CONNECTORS bag according to the type of the family.

The instances should be rotated so that they can be placed on the top of the chip (inner edge facing down). Each instance should be annotated with the property DIRECTION (INPUT, OUTPUT, INOUT) and TERMTYPE (SUPPLY, GROUND, CLOCK , SIGNAL). Also each instance must have the real valued property PAD_VDD_GND_REQUIRED, expressing the amount of power required by the pin. For example: Suppose that the for a particular family, one should put one vdd/gnd pair for each 4 output pads. This will be expressed by attaching to the output pad the property PAD_VDD_GND_REQUIRED with value $0.25 = 1/4$.

- One net called PAD_BONDING_PINS containing all the bonding pins which will give the implementation to the formal terminals.
- One net called PAD_INTERNAL_PINS containing all the bonding pins which will give the implementation to the formal terminals.
- One net called PAD_SUPPLY_RING containing all the terminals that must be connected to form the supply ring for the pads. The net should also contain a path of the right width on the correct layer for the ring.
- One net called PAD_GROUND_RING containing all the terminals that must be connected to form the ground ring for the pads. The net should also contain a path of the right width on the correct layer for the ring.
- One property called PAD_MINIMUM_SEPARATION to specify the minimum distance between the pads. A separation of 0 means that the pads can be abutted.

Current limitations: Only 1 power voltage is supported. Tristate outputs are not supported yet.

3.3.3 Tools

padplace is the tool for pad placement. It understands which cells are pads, which side should face towards the inside of the chip, and can place the pads according to a number of different schemes. An instance is a pad if one of the following three conditions is satisfied:

1. The instance has the property PLACEMENT_CLASS with value PAD;
2. The master has the property PLACEMENT_CLASS with value PAD;
3. An actual terminal of the instance is attached to a formal terminal of the cell.

3.4 Layer PLACE

The layer "PLACE" is a dummy layer used by `MOSAICO`. Currently, the routing phase of `MOSAICO` (see 1.4) requires the layer to be present in the interface facet of the master of each instances. The geometry on the layer PLACE is used to define the shape of the cells.

The layer PLACE is now generated automatically by **vulcan**. *However, there are good reasons to change* **vulcan** *so that it does no longer generate the layer PLACE. We are currently changing* `MOSAICO` *so that it does not require the layer PLACE in the input facet. The layer PLACE will still be used internally by* `MOSAICO`.

The layer PLACE is also used to give shape and implementation to soft cells (see 1.3.1).

See also 1.4, 1.3.1.

3.5 Nets

Unique names for nets are NECESSARY.

NETTYPE one of `SIGNAL, CLOCK , SUPPLY, GROUND` (*OCT Symbolic Policy*).

NET_CRITICALITY integer valued property in the range [-1000,1000] (*OCT Symbolic Policy*). Criticality 0 is the default. The higher the criticality, the shorter the net should be.

NET_MAX_LENGTH integer valued property (`PUPPY` only) for the placement tools, this property suggests the maximum length of the net (in lambdas).

Chapter 4

RPC applications

Fig.4.1 shows how you can modify the `vem.bindings` file in order to include those RPC applications that have been developed with the `MOSAICO` system. Each application is briefly described in this chapter.

4.1 atlas: channel definition

The enables the user to visually check the channel definition and ordering. If the channel definition is not what is desired, then the user can alter the placement of instances using vem drag instance (don't use vem move instance) and do another channel definition.

4.2 nleRpc: net list editing

This RPC application gives direct access to the routines in the `nle` library. See Sec.5.2 for more detail.

```
REMOTE  nleRpc         localhost       ~octtools/bin/rpc/nleRpc
REMOTE  atlas          localhost       ~octtools/bin/rpc/atlas
REMOTE  mosaicoDemo    localhost
~octtools/bin/rpc/mosaicoDemo
INCLUDESTD
symbolic       nleRpc         MENU MOSAICO    1       ALIAS
symbolic       atlas          MENU MOSAICO    2       ALIAS
symbolic       mosaicoDemo    MENU MOSAICO    5       ALIAS
```

Figure 4.1: Example vem bindings file to include the `MOSAICO` RPC applications

4.3 mosaicoDemo: monitoring of `MOSAICO`

This application makes a great demo, but it can also be useful to check the behavior of `MOSAICO` in each phase of the pipeline, as well as in each channel. See the man page for more details.

Chapter 5

Miscellaneous libraries

5.1 adm:

The scope of the adm library is to provide a mechanism to record on an OCT facet the sequence of tools that have been used to generate the data that is presently stored in the facet. Furthermore, adm routines provide a way to check that dependency order among tools is satisfied.

Presently the adm library consists of 4 routines, namely

- admToolName(toolName, argc, argv)

  char *toolName;

  char *argv[];

  int argc;

- admLogInTool(OutFacet)

  octObject *OutFacet;

- admLogOutTool(OutFacet)

  octObject *OutFacet;

- int admCheckInput(InFacet)

  octObject *InFacet;

5.1.1 Routine Description

admToolName records the name of the tool, the command line used to run the tool. admToolName initializes also the procedure necessary to collect statistics on the OCT utilization. For this reason the routine must be called before the first call to an OCT routine. In any case it cannot be called after any other routine in the adm library. If this constraint is violated, the program calling any of the adm routines will terminate and an error message will be printed on stderr.

admLogInTool put a signature of the tool on the output facet. If any errors occur, a message is printed on stderr and the execution is terminated.

admLogOutTool completes a signature of the tool created on the output facet by admLogInTool with the information that the tool has successfully completed its job. The routine call must be the last thing before the facet is closed. If any errors occur, a message is printed on stderr and the execution is terminated.

admCheckInput checks that the precedence relations are satisfied before the tool is run. The routine call must follow immediately routine admLogInTool. It returns ADM_GO if the data in the input facet contains all the information required by to toolName. If not it retuns ADM_NO_GO. The relations used to check consistency are, in the present version, as follows:

- padplace: NO PRECEDENCE IS REQUIRED;
- puppy: NO PRECEDENCE IS REQUIRED;
- atlas: NO PRECEDENCE IS REQUIRED;
- cds: DEPENDS ON atlas;
- PGcurrent: DEPENDS ON cds;
- cprep: DEPENDS ON (cds OR PGcurrent);
- octflatten: DEPENDS ON cprep;
- mosaicoRC: NO PRECEDENCE IS REQUIRED;
- sparcs: DEPENDS ON octflatten;

5.1.2 OCT Structure

The data is stored in the facet in the following format.

MOSAICO_HISTORY octObject of type bag. Contains any number of octObjects of type bag, one for each of the tools that have been run on the facet. It is contained by the facet.

ToolName octObject of type bag. Contains 4 octObjects of type property. It is contained by the bag MOSAICO_HISTORY.

COMMAND_LINE octObject of type property. It does not contain anything. It is contained by the ToolName bag. The property is of type string. Its value is the command line that has been used to run ToolName.

TIME_LAST_CALL octObject of type property. It does not contain anything. It is contained by the ToolName bag. The property is of type string. Its value is the time and date in which ToolName was started.

TIME_LAST_CALL_SECS octObject of type property. It does not contain anything. It is contained by the ToolName bag. The property is of type integer. Its value is the time and date, expressed in seconds, in which ToolName was started.

COMPLETION octObject of type property. It does not contain anything. It is contained by the ToolName bag. The property is of type integer. Its value is 1 if ToolName completed successfully and is 0 if some error has been detected.

5.2 nle: net list editing

The `nle` library provides a set of routines for flattening, merging, and substitution of instances. The most difficult thing to understand is the concept of the "type" of a flattening or of a merging.

- HARD flattening means that any existing routing of the nets is preserved. HARD merging means that all the actual terminals of the instances to be merged are promoted to formal terminals if they are attached to a net that crosses the boundary of the partition. (clear ?) HARD merging is not well defined when routing is present. In any case, the routing will not be preserved. HARD merging and HARD flattening are the inverse operations of each other only if no routing is involved.

- SOFT flattening means that all the routing inside the instance to be flattened is ignored. SOFT merging means that the new cell are one for each of the nets that cross the boundary of the partition. Each of these terminals has no implementation. SOFT merging and SOFT flattening are the inverse operations of each other.

Appendix A

Appendix: Channel definition policy

```
Channels:

Channels are grouped in a binary tree as follows:

Each channel represents a submodule and a generalized partition of itself.
The partition represents the "channel" subdividing the submodule.

channel name = "CDChanDef" (type==OCT_BAG)
contains:
    if a real channel:
        channel order property
        sub channels
        sub nets
        instance corner edges
    if a leaf channel (no partition):
        cell instance
        surrounding global wire graph description
containers:
    channel (if not root channel)
    contents facet (if root channel)

Channels have a order property.  This property designates the order in which
the channels are to be routed.

channel order name = "CDChanOrder"   (type==OCT_BAG, prop.type==OCT_INTEGER)
    prop.value = order which the channels are to be routed (0 = 1st channel
                 to be routed, n = (n+1)th channel to be routed).
```

e.g.

The macro cell below is represented by the tree on the right:

```
+----------+.....+--------+                 ************
|          |     |        |                 * contents *
|          |     | Inst 2 |                 *   facet  *
|          |     |        |                 ************
|          |  C  +--------+                       |
|          |  h  .        .                  .....V.....
|  Inst 1  |  a  . Chan B .                  . Chan  A .
|          |  n  .        .                  ...........
|          |     +--------+                    |       |
|          |  A  |        |         ..........V..   ..V........
|          |     | Inst 3 |         . Leaf Chan .   . Chan  B .
|          |     |        |         .............   ...........
+----------+.....+--------+            |              |     |
                                 +-----V--+ ..........V.. ..V..........
                                 | Inst 1 | . Leaf Chan . . Leaf Chan .
                                 +--------+ ............. .............
                                                  |             |
                                              +---V----+  +-----V--+
                                              | Inst 3 |  | Inst 2 |
                                              +--------+  +--------+
```

Global route subnets:

Nets are grouped into channels as follows:

1) A subnet corresponding to channel is attached under the net
2) All aTerms in the net/channel are attached under the subnet.
   The original net - aterm attachments are untouched.
3) Pseudo terminals (terminals of the subnet representing net channel to
   channel interfaces) are created and attached to both the original
   net and the subnet.
4) Subnets are then attached under its corresponding channel.

channel subnet name = "CDSubNet"   (type==OCT_NET)

subnet contains:
    actual terminals   (real terminals)
    psuedo terminals   (channel to channel terminals)
subnet containers:
    original net
    channel subnet traverses

```
psuedo terminal name = "CDPseudoTerm" (type==OCT_BAG)

psuedo terminal contains:
    net width property
psuedo terminal containers:
    subnet
    original net
```

The net pin width is attach to pseudo terminals and actual terminals of special net pins that are not to be routed by a standard minimum width wire.

```
net terminal width name = "PorGWidth" (type==OCT_PROPERTY,
                                       prop.type==OCT_INTEGER)
```

e.g.

A global wire net below is represented by the tree on the right:

```
                                                            ..........
                                                            . Chan A .----+
                                                            ..........    |
                                                                          |
                                                            ..........    |
        .      .                                            . Chan B .    |
        .      .                                            ..........    |
        .Chan B.                         *************             |      |
        .      .                         *  Net      *----------+  |      |
  ATerm 3--+   .                         *************-------+  |  |      |
        .  |   .                          |  |  |  |         |  |  |      |
        .  |PTerm      ATerm     *********V* |  |  |         | .V..V..... |
     ......4.............2..     * ATerm 3 <-|--|--|---------|-. SubNet . |
           |             |       *********** |  |  |  +------|-.......... |
  Chan A   +-+-----------+                   |  |  |  |      |            |
             |                      .........V. |  |  |      |            |
     ........1..............        . PTerm 4 <-|--|--+      |            |
           ATerm                    ..........<-|--|----+    |            |
                                                |  |    |    |            |
                                       *********V* |    +---.V........    |
                                       * ATerm 1 <-|--------. SubNet <----+
                                       *********** |    +---..........
                                                   |    |
                                          *********V*   |
                                          * ATerm 2 <---+
                                          ***********
```

```
Global wire graph:

Surrounding global wire graph description

graph description name = "CDGraph" (type==OCT_BAG)
contains:
    1st point of surrounding polygon
containers:
    leaf channel

Surrounding polygon is an ordered list of vertices
that are connected together
with bags representing the graph edges.

The vertices are OCT points and the edges are OCT bags.

The graph edges represent global wiring channels.
The graph edges have an edge width property representing the width of the
global wiring channel.  The graph edges are attached to its corresponding
routing channel.

graph edge name = "CDGraphEdge" (type==OCT_BAG)
contains:
    points  (representing end points - last/closing endpoint is assumed to
             be the first point attached to the graph description bag and is not
             attached to that point).
    global wire channel width property
containers:
    channel (routing channel corresponding to global wiring channel - a
             many to 1 surjective mapping)

global wire channel width name = "CDEdgeWidth" (type==OCT_PROPERTY,
                                                prop.type==OCT_INTEGER)
    prop.value = average width of global wiring channel in database units


Corners:

Corners of instance boundaries designate each instance boundary segment to
a routing channel.  The boundaries of an instance are assumed to be either
clockwise or counterclockwise (i.e. the instance boundary
does not cross itself).
A boundary path delineating part of the routing channel defined by a bordering
instance is defined by two corners and a corner edge attached to the
routing channel.
Each corner is an index into an array of points (representing the boundary of
an instance).
```

```
corner bag name = "CDCorners" (type==OCT_BAG)
contains:
    1st corner of instance boundary
containers:
    instance

corner name = "CDCorner" (type==OCT_PROPERTY, prop.type==OCT_INTEGER)
    prop.value = index into oct point array of instance master "place" layer
                 polygon.  However if instance master "place" layer bounding
                 polygon is a box then:
                 0 represents the lower left corner of the master box,
                 1 represents the lower right corner of the master box,
                 2 represents the upper right corner of the master box,
                 3 represents the upper left corner of the master box.

corner edge = "CDCornerEdge" (type==OCT_BAG)
contains:
    corner property
containers:
    channel
```

Bibliography

[1] G. Adams, S. Devadas, K. Eberhard, C. Kring, F. Obermeier, P. Tzeng, R. Newton, A. Sangiovanni-Vincentelli, and C. Sequin. Module generation systems. Technical report, University of California, Berkeley, May 1986.

[2] R. Brayton, A. Cagnola, E. Detjens, S. Krishna, P. McGeer, L. Pei, N. Phillips, R. Rudell, R. Segal, T. Villa, A. Wang, R. Yung, and A. Sangiovanni-Vincentelli. Multiple-level logic optimization system. *Proc. ICCAD*, pages 356–359, 1986.

[3] J.L. Burns and A.R. Newton. Sparcs: A new constraint-based ic symbolic layout spacer. *Proc. CICC*, pages 534–539, 1986.

[4] D. Harrison, P. Moore, Rick L. Spickelmier, and R. Newton. Data management and graphics editing in the berkeley design environment. *Proc. ICCAD*, pages 24–27, 1986.

[5] A.R. Newton, A. Sangiovanni-Vincentelli, and C.H. Sequin. The berkeley synthesis project. Technical report, University Of California, Berkeley, May, 1986.

[6] R.H. Otten. Automatic floorplan design. *Proc. 19th Design Automation Conf.*, pages 261–267, 1982.

[7] R. Rudell. Wolfe: A standard-cell based module generator. Technical report, University Of California, Berkeley, May, 1986.

[8] N. Weiner. Aspects of pre-layout timing prediction and rectification. Technical report, University Of California, Berkeley, May, 1986.