A High Level Language for Pre-Layout Extraction in Parasite-Aware Analog Circuit Synthesis*

Raoul F. Badaoui, Hemanth Sampath, Anuradha Agarwal and Ranga Vemuri
University of Cincinnati
{rbadaoui,hsampath,agarwala,ranga}@ececs.uc.edu

ABSTRACT

This paper presents a high-level language MSL, for the specification of parameterized, topology-specific circuit extractors. Upon compilation, the MSL program yields an executable module which generates the extracted circuit containing parasitics, passive and active devices when given specific sizes. In contrast to traditional post-layout extraction, this is done *without* ever generating a layout. We call this *pre-layout extraction*. Pre-layout extraction is much faster than post-layout extraction and is highly suited for use in layout-aware circuit sizing programs. MSL can also be used for the specification of parameterized layout generators. Thus, although a concrete layout is never generated during pre-extraction, the extracted circuit is very much influenced by the symbolic placement and routing specified in the layout generation part of the MSL program. This ensures that the pre-layout extraction process yields the same results as post-layout extraction. Being a high-level language based approach, users can tune pre-layout extraction to a desired level of accuracy by modeling selected parasitics and ignoring others. This ability helps further speed up the circuit sizing process up to a factor varying from 2.5 to 4.5 compared to layout-inclusive synthesis methodologies.

Categories and Subject Descriptors: J.6 Computer-Aided Engineering: Computer-Aided Design

General Terms: Design, Languages.

Keywords: Pre-Layout Extraction, Analog VLSI, Parasitics, MSL.

1. INTRODUCTION

Layout-induced parasitics have significant effects on the performance of high-frequency analog circuits. To achieve parasite-inclusive performance-closure, layout-aware circuit synthesis methodologies are beginning to emerge. In layout-in-the-loop synthesis methodologies [9, 11, 3], illustrated in Figure 1, performance analysis is based on generation of a concrete layout for the explored circuit sizes. A parasite-inclusive circuit is extracted from the layout using a standard extractor and is analyzed using a simulator to determine whether the constraints are met.

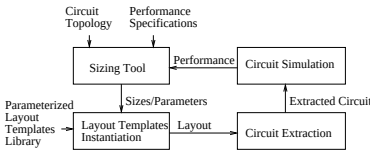

Figure 1: Layout-in-the-loop synthesis methodology

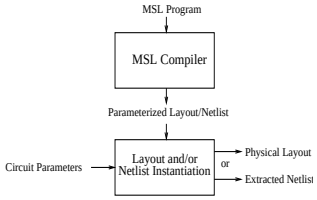

Figure 2: MSL Program Usage Flow

Table 1 shows typical synthesis times for some examples using a layout-in-the-loop methodology. The methodology uses a simulated annealing algorithm for circuit sizing. The constraints in each case are set to be moderately difficult to achieve. The time taken for layout generation and circuit extraction within the synthesis process is shown both as an absolute value and as a percentage of the total synthesis time. The purpose of layout generation during the synthesis process is solely to determine the layout-induced effects in terms of device and interconnect parasitics in the extracted circuit in order to perform accurate, layout-aware performance analysis. If the parasitics could be estimated or determined otherwise, there would be no need for layout generation.

Various approaches of estimating parasitics without generating a layout have been proposed [1]. Malavasi's work [6, 7] uses its built-in tool [2] to generate the sensitivities of performance parameters to existing parasitics. It checks the performance deviations from the initial simulation in each of the synthesis iteration. The performance characteristic that deviates the most has its weight in the cost function recalculated. Then, other iterations keep running until all performance deviations are minimized. Gielen's work in LAYLA [4, 5] is a performance-driven search algorithm. Its cost function includes performance sensitivities for each net and an estimation of the performance degradation due to the addition of the remaining predicted part of that net. All of these methods attempted to estimate parasitics of the layout without generating it. They lack

*This work is sponsored in part by the the DARPA/MTO NEOCAD program under contract number F33615-01-C-1977 monitored by the Air Force Research Laboratory

GLSVLSI'04, April 26–28, 2004, Boston, Massachusetts, USA.

Module	Synthesis Time (s)	Layout Time (s)	Percentage
Single Ended Opamp 1	15007	12155	73.78
Single Ended Opamp 2	15054	12120	73.245
Two Stage Opamp	10335	6678	63.387

Table 1: Layout Time as a percentage of Synthesis Time in Layout-Inclusive Synthesis

the correctness that would only come from examining the layout itself.

Dessouky [3] proposed a novel method for direct generation of the extracted circuit without generating a layout. His method uses a library of layout templates for basic analog modules. A procedure, in C++, for calculating parasitics is associated with each layout template. These procedures can be executed without the need for layout elaboration. An extracted circuit including parasitics can be generated using these procedures without generating a layout during the circuit sizing. Layout elaboration takes place once and only after the circuit sizing is completed. A user can not easily customize one of those built-in extraction procedures or add new ones.

Our work follows the same general idea of Dessouky's work with the following important distinctions: (1) Our high-level language layout specification characteristic is hierarchical in its nature; the user can build as high a hierarchy depth as desirable by the design process. (2) The placement of blocks in the parameterized layout generators is not bound by specific channels or columns. (3) We use MSL for specifying circuit extractors. It provides constructs for concise specification of how the extracted circuit should be generated. (4) MSL allows mixing of pre-layout and post-layout extraction techniques by selective layout generation for desired cells in the circuit.

The rest of this paper is organized as follows: Section 2 describes the MSL language using illustrative examples. Section 3 presents example circuits demonstrating the usage of MSL, how to control the accuracy level of an extracted circuit, and the compliance of the extracted circuits with post-layout extraction. Section 4 discusses the use of MSL extractors in synthesis and compares the use of pre-layout, post-layout and mixed extraction techniques in synthesis. We also show how MSL can be used to focus only on the significant parasitics and ignore others during early synthesis runs. Section 5 contains concluding remarks and future work.

2. THE MSL LANGUAGE

MSL or Module Specification Language is a language for the specification of parameterized layout generators and circuit extractors. As shown in Figure 2, an MSL program is written by the designer and then compiled. The result is an executable module that takes in parameters as input and outputs a physical layout or an extracted circuit net-list. This enables the designer to build layout templates to be used in a synthesis tool.It is important to note though that no layout is ever generated if the user chooses to use the executable module to output an extracted circuit net-list.

The main advantages of MSL are the re-usability of its modules through hierarchical designs, the easiness of interfacing with external routers and tools, and the possibility to use external C++ functions within the definition of an MSL program.

An MSL program is comprised of several sections: (1) Attributes section, (2) Variables section, (3) Types Section, (4) Rules Section, and (5) Extraction section.

The attributes section defines the attributes of the module which are the parameters that would be input by the user at the instantiation step. These attributes are the elements that make an MSL module completely parameterizable.

```
attributes
   trlength, totalwidth: int;
   diffusionLayer, conLayer, selLayer: layer;
   numFingers: int;
   fingerLimit: int:=50*lambda;
   spacing:int:=0;
   polyOh: int:=2*lambda;
   diffOh: int:=13*lambda;
   selEnc: int:=0;
   maxHeight: int:=18*lambda;
end attributes
types
   devices: mosDevice[numFingers];
   innerconnectors: terminal[numFingers];
   outerconnectors: terminal[numFingers];
   gateconnectors: terminal[numFingers];
   inner, outer, gate : rect;
end types
variables
   individualWidths: int[numFingers];
   arrIndex : int;
   innerIndex, outerIndex : int:=0;
   ySpacing : int:=0;
   outerSpace : int:=0;
   innerSpace : int:=0;
   gateSpace : int:=0;
   deviceSpace : int:=0;
end variables
rules
[...]
   innerconnectors[innerIndex].stemHeight = devices[arrIndex].con2.bl'second - inner.bl'second;
   innerconnectors[innerIndex].stemLayer = layer9;
   innerconnectors[innerIndex].conLayer = layer10;
   innerconnectors[innerIndex].tl = devices[arrIndex].con1.bl; innerIndex = innerIndex + 1;
[...]
end rules
```

The program extract above shows an example of an MSL program for the layout generation of a fingered transistor. A set of attributes are defined in the section 'attributes.' For example, the length, the width of the transistor and the layer type of the various rectangles comprising the layout are attributes of that fingered device.

The 'types' section defines the objects forming that module. They are instantiations of previously defined MSL modules or basic rectangles with their specific layer attribute. This makes hierarchical composition of modules possible.

In the same example shown above, a number of devices has been declared (these devices are specified as other MSL modules in sep-

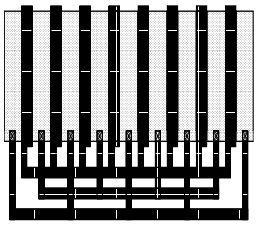

Figure 3: Instantiated Fingered Transistor Layout from MSL Program

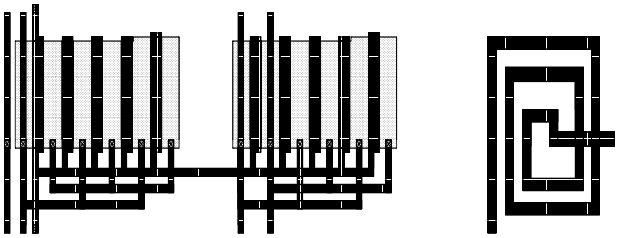

Figure 4: Gate-Connected Pair and Inductor Layouts generated using an MSL module

arate files). In addition, various interconnections and rectangles (which are the built-in modules provided by MSL) are declared. The number of those devices and rectangles is parameterized and dependent on the number of fingers the transistor in question has. It is dynamically instantiated during the elaboration of a specific fingered device, after compilation of the MSL program into an executable module.

The objects declared in the 'types' section are parametrically sized and placed relatively to each other in a third section called the 'rules' section. A set of positional built-in attributes for each instance of an object such as "br"(bottom right) or "lc"(left center) makes the relative positioning of blocks and their sizing easy. External functions written in C++ can be linked and used in the rules section to perform calculations that might not be possible to write by just using the constructs of the MSL language only. If such functions were to be used, they would need to be specified in a separate section called the headers section.

MSL allows the use of internal variables in the rules section to make calculations easy to write and code. These variables have to be declared in the 'variables' section of the program.

A specific instantiation of the layout of the fingered device described in the program above is shown in Figure 3.

The sections described above are sufficient to enable the design of parameterized layout generators. [10]

As per the main advantage of MSL in layout-aware circuit synthesis, the 'extraction' section of an MSL program makes it possible for extraction rules to be specified as a part of an MSL module.

The section starts with a declaration of the electrical nodes present in the circuit. These nodes would be instantiated at runtime. An example of an MSL 'extraction' section is shown below:

```
extraction
   node1,node2,node3[5],node4,node5[4];
   resistor: R3(node1,node2) = var1;
   N-Transistor: M1[var4]( node1, node3[2], node2, node4 )
wtran = var2 ltran = var3;
   P-Transistor: M2(node1,node3[2],node5[0],node4) wtran
= var6 ltran = var7;
   capacitor: C1(node4,node2) = var8;
   node1 = rect2.node3;
   node3 = module2.node2;
end extraction
```

```
rules
[...]
   var2 = tranw / numfingers;
   var3 = inputTranL;
[...]
end rules
```

What follows the nodes declaration line is a definition of the circuit elements in that circuit. Variable numbers of a circuit element are allowed as shown in line 3 of the section defining an array of N-transistors as a fingered transistor. The variable 'var4', and any other variable used, are declared and defined in the rules section. This allows the parameterization of the number of electrical nodes, circuit elements and their assigned values. The allowed circuit elements declarations in MSL are resistors, capacitors, inductors, N-Transistors and P-transistors.

In each element declaration, an assignment of that circuit elements' physical value(s) is(are) defined at the end of the line. Those values can be assigned variables that would be declared and defined in the 'rules' section. An example of the grammar of the circuit elements definitions is as follows:

$$resistor \rightarrow \quad \text{“resistor:”}\langle element\_name\rangle \\ (\langle node\_name\rangle, \langle node\_name\rangle)\text{“=”}\langle value\rangle\text{“;”}$$

The last part of the 'extraction' section is the list of assignments shown after line 6. These lines define the connectivity of the electrical nodes allowing the hierarchical use of MSL modules. Each assignment statement connects one of the declared electrical nodes in line 1 of the 'extraction' section to a node of the instantiated module objects defined in the 'types' section. This ensures the capability of full connectivity through a hierarchical module. The grammar of the connectivity section is described below:

$$\langle circuit\_connectivity\_line\rangle \rightarrow \langle node\_name\rangle \text{“}:''\\ \langle type\_name\rangle \text{“}.''\langle node\_name\rangle \text{“;”}$$

The use of such a program written in MSL has been shown in Figure 2. An MSL program is written and then compiled to produce a parameterized layout and circuit block. This block takes in the parameters of the circuit as input and outputs a physical layout or a sized circuit net-list. This generation of a sized circuit net-list allows it to be used in a circuit synthesis flow as will be described in Section 4

Two more examples of modules written in MSL, an inductor and a gate-connected pair, are shown below along with their corresponding generated layouts in figure 4.

```
attributes
   inductancevalue: int;
   mulfactor: int;
   turnlayer, conlayer:layer;
   numturns: int:=0;
   innerdia: int:=10*lambda;
end attributes
types
   topturn: rect[numturns];
   bottomturn: rect[numturns];
   leftturn: rect[numturns];
   rightturn: rect[numturns];
   contacts: rect[2];
end types
variables
   arrIndex: int;
   terminalHeight: int:=8*lambda;
end variables
rules
[...]
   for(arrIndex=0; arrIndex ¡ numturns; arrIndex = arrIndex +
1 )
   {
      leftturn[arrIndex].h = bottomturn[arrIndex].l;
      leftturn[arrIndex].l = turnwidth;
      leftturn[arrIndex].bl = bottomturn[arrIndex].bl;
   }
[...]
end rules
```

```
attributes
   tranlength, tranwidth: int;
   diffusionLayer, conLayer, selLayer: layer;
   wellLayer, wellContactLayer: layer;
   transistorSpacing: int:=0;
   inductancevalue: int;
end attributes
types    transistor1 : fingeredDevice;
   transistor2 : fingeredDevice;
   downSource, upSource : terminal;
   upFirstDrain, downFirstDrain: terminal;
   upSecondDrain, downSecondDrain: terminal;
   upGate, downGate : terminal;
end types
rules
[...]
   transistor1.trlength = tranlength;
   transistor1.totalwidth = tranwidth;
   transistor1.diffusionLayer = diffusionLayer;
   transistor2 = transistor1;
   transistor1.bl = gateConnectedPair.bl + ¡wellEnc, terminal-
Height¿;
[...]
end rules
```

MSL also gives the user an ability to parameterize not only the circuit elements and their values but also the accuracy inherent to said circuit. The inductor MSL program shown above has been used to elaborate this ability. The program extract shown below depicts the extract section that was added to the inductor's MSL program.

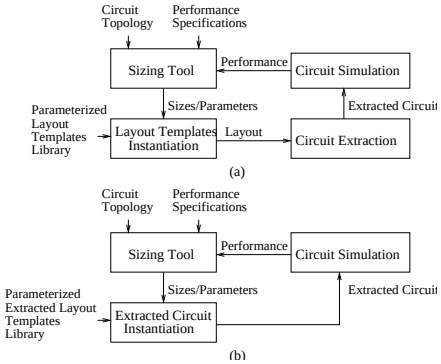

Figure 5: Layout-in-the-loop (a) and pre-layout extraction synthesis methodologies (b)

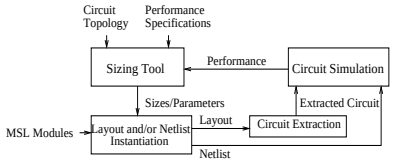

Figure 6: Combination of both synthesis methodologies

```
rules
[...]
   var2 = var2 * ACCURACY;
   var3 = var3 * ACCURACY;
end rules
extraction
   node1, node2, node3, GND
   inductor: L1(node1,node2) = var1;
   resistor: R1(node2,node3) = var2;
   capacitor: C1(node2, GND) = var3;
end extraction
```

The variables 'var1', 'var2', and 'var3' that are used to assign values to the circuit elements would be calculated in the rules section. As shown in said section, 'var2' and 'var3' have been multiplied by a parameter value 'ACCURACY'. If that constant is set to '0' at instantiation, those variables take '0' as a value too. Thus, the resistor and the capacitor do not appear in the final extracted circuit at instantiation and the parameter 'ACCURACY' is a boolean value that sets the level of accuracy of the circuit.

Another method of controlling the accuracy of a more complicated module would be to write two separate 'extraction' sections with different levels of parasitics details for the same module. The use of such various levels of detailing is discussed further in Section 4 in a circuit synthesis process.

3. EXPERIMENTAL CIRCUITS

The MSL language has been implemented and written in ANTLR and C++. A multitude of analog and digital modules has been developed to present the various features of the language. Table 3 shows various examples along with their characteristics emphasiz-

Performance Attribute	Ext.Models: HIGH DETAIL		Ext.Models: MED.DETAIL		Ext.Models: LOW DETAIL	
	Predicted	Actual	Predicted	Actual	Predicted	Actual
Gain (dB)	48.067	47.8	52.34	46.5	54.67	47.6
Output Offset (V.)	0.082	0.081	0.085	0.0810	0.083	0.080
Phase Margin (degrees)	42.96	41.34	45.45	41.87	46.45	40.67
3dB Freqquency (MHz)	3.57102	3.4587	3.9243	3.4178	4.5434	3.5690
Unity Gain Frequency (MHz)	479.194	475.67	487.775	474.598	502.734	460.35
Synthesis Time (s)	11550		10339		9852	

Table 2: Single-Ended Opamp Synthesis Results

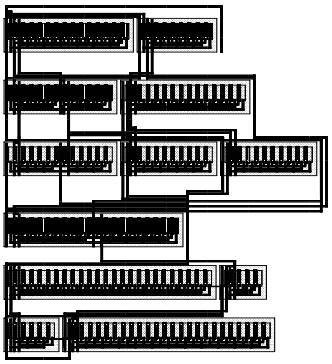

Figure 7: Synthesized Single-Ended Opamp Layout

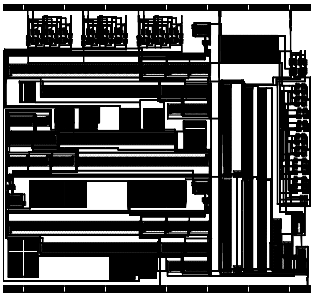

Figure 8: Synthesized Third-Order Delta-Sigma Modulator Layout

ing on a comparison between the number of lines used to create the module in MSL and that number if C++ was to be used.

To most of these modules, an extraction section was written to define the way each circuit should be extracted. In an attempt to validate the correctness and accuracy of the proposed pre-layout extraction, experiments have been executed using both pre-layout extraction as described and post-layout extraction through the MAGIC extractor.

One of the main goals reached by using our proposed approach is the ability that the user gets in having circuits being extracted into widely distributed net-lists. As shown in Table 4, the bigger number of extracted components obtained in the pre-layout extraction method shows the high level of extraction detail that the designer can reach by using those MSL written extraction specific modules. Added to that is the speedup in time that the process can achieve.

Those fully parameterized modules described above can be used in a synthesis loop as will be shown in section 4.

4. CIRCUIT SYNTHESIS USING MSL

A set of MSL modules presents the designer with the ability to run circuit synthesis [8]. Those modules can be used for three types of synthesis:

* Synthesis using exact layout elaboration
* Synthesis using pre-layout extracted circuit elaboration
* Synthesis through combination of both methods

As shown in Figure 5, synthesis using exact layout elaboration generates the layout in each iteration and performs a circuit extraction on that layout. On the other hand, synthesis using the pre-layout extracted circuit elaboration is based on the proposed approach; the extracted circuit to be simulated is generated using the parameterized MSL generated extraction modules in each iteration.

The combination of those two approaches is one of the advantages of introducing the proposed pre-layout extraction approach. As shown in Figure 6, parts of the circuit can be chosen to be extracted using the MSL system pre-layout extractor while the remaining parts are extracted using a regular extractor such as the MAGIC extractor. The incentive behind such combination is the desired difference of accuracy in the extraction process through different parts of the circuit, or even different steps of the design process.

A single ended opamp has been built hierarchically, it has been synthesized using the methods shown in Figure 5. Three different versions of the opamp modules have been used; each version's depth of detail in its extraction section is different from the others as was discussed at the end of Section 3. The synthesized results obtained are shown in Table 2. As shown in the table, when the models with the least levels of extraction details were used, the synthesis time reduced significantly while the error between predicted and actual results was the highest. A tradeoff between accuracy of extraction and exactness of predicted performance values is what controls the choice of library modules the user makes. It is important to note that the time it took to complete a synthesis run with a complete layout elaboration and extraction inside the loop for that same opamp took 26564s. which shows a minimum speedup factor of 2.5 using the proposed approach. The final layout of the medium-level detail version of the opamp is represented in Figure 7.

Furthermore, a two-stage opamp has been also built hierarchically using MSL. The actual synthesized results obtained are shown in the table below. This synthesis yielded a speedup of 2.9x compared to a synthesis using exact layout elaboration and extraction techniques.

Module	MSL Lines	C++ Lines	Hier.Levels
MOS Tran	71	718	1
Fingered MOS	178	1174	2
Diff. Pair	202	1460	3
Interdigitated Diff. Pair	239	1540	3
2x1 Mux	285	2161	3
SR Latch	129	1231	3
JK Flip-Flop	254	1965	4
Single Ended Opamp	394	5157	4
Two-stage Opamp	321	3603	4

Table 3: Various MSL Modules

Module	Parasitics (Pre/Post)	Active (Pre/Post)	Speedup
MOS Device	15/0	1/1	4.5x
Fing.Transistor	33/0	3/3	4.5x
Diff. Pair	160/15	10/10	3x
Gate-conn. Pair	150/12	6/6	3x
Two-Stage Op.Amp.	360/35	71/71	2.5x

Table 4: Comparison between Pre-Layout and Post-Layout Circuit Extraction

Performance Characteristic	Value
Gain (dB)	30.9151
Phase Margin (degrees)	70.3341
3dB Frequency (MHz)	1.386
Unity Gain Frequency (MHz)	48.177

Finally, a delta-sigma modulator has been built using various components built and synthesized using MSL and the proposed approach. The final system routing was done using a regular maze router. The final layout of that delta-sigma modulator is shown in figure 8.

5. CONCLUSION AND FUTURE WORK

This paper presented MSL, a high-level language for the specification of parameterized layout generators and circuit extractors, focusing on the use of the pre-layout extraction capability that the language offers to circuit synthesis. It allows the circuit designer to build parameterized circuit net-lists for corresponding parameterized physical layouts. It introduced a synthesis methodology that uses those parameterized circuit extractors, with different degrees of accuracy on the same topology, to be used in consecutive steps of the design process. The use of such parameterized circuit net-lists in a synthesis methodology was shown to have a speedup factor varying from 2.5 to 4.5 in comparison with layout-inclusive synthesis methodologies. This method shall be referred to as Pre-Layout Extraction. Its novel idea could be used as basis for an automatic generation of such extract sections in future MSL-like languages. The work presented was part of the SHARC tools suite developed at the University of Cincinnati.

6. REFERENCES

[1] A.Agarwal, H.Sampath, V.Yelamanchili, and R.Vemuri. Accurate estimation of parasitic capacitances in analog circuits. In *Proc. of IEEE/ACM DATE Conf.*, February 2004.

[2] U.Choudhury and A.Sangiovanni-Vincentelli. Constraint generation for routing analog circuits. In *27th ACM/IEEE Design Automation Conference*, pages 561–566, 1990.

[3] M.Dessouky, M.-M. Lourat, and J.Porte. Layout-oriented synthesis of high performance analog circuits. In *Proc. of IEEE/ACM DATE*, pages 53–57. ACM Press, 2000.

[4] K.Lampaert, G.Gielen, and W.Sansen. Analog routing for manufacturability. In *IEEE Custom Integrated Circuits Conference*, pages 175–178, 1996.

[5] K.Lampaert, G.Gielen, and W.Sansen. *Analog Layout Generation for Performance and Manufacturability*. Kluwer, 1999.

[6] E.Malavasi, U.Choudhury, and A.Sangiovanni-Vincentelli. A routing methodology for analog integrated circuits. In *Proc. IEEE ICCAD*, pages 202–205, November 1990.

[7] E.Malavasi and A.Sangiovanni-Vincentelli. Area routing for analog layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1186–1197, August 1993.

[8] M.Ranjan, W.Verhaegen, A.Agarwal, H.Sampath, G.Gielen, and R.Vemuri. Fast layout-inclusive analog circuit synthesis using pre-compiled parasitic-aware symbolic performance models. In *Proc. of IEEE/ACM DATE*, February 2004.

[9] H.Onodera et.al. Operational amplifier compilation with performance optimization. *IEEE JSSC*, 25(2):466–473, April 1990.

[10] H.Sampath and R.Vemuri. MSL: A high-level language for parameterized analog and mixed-signal layout generators. In *Proc. of IFIP 12th International Conference on VLSI*, March 2003.

[11] P.Vancorenland, G.V. der Plas, M.Steyaert, G.Gielen, and W.Sansen. A layout-aware synthesis methodology for rf circuits. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 358–362. IEEE Press, 2001.