MESH: Explicit and Flexible Generation of Analog Arrays

Benjamin Prautsch*, Uwe Eichler*, Torsten Reich*, Jens Lienig†
*Fraunhofer IIS/EAS, Institute for Integrated Circuits, Division Engineering of Adaptive Systems, Dresden, Germany
{benjamin.prautsch, uwe.eichler, torsten.reich}@eas.iis.fraunhofer.de
†Dresden University of Technology, Dresden, Germany; jens@ieee.org

Abstract—**Analog layout design is a costly and error-prone task since analog synthesis is still far from applicable. It is expected that procedural bottom-up generators managing constructive tasks will be part of future synthesis flows. Generators contain expert knowledge *implicitly* within complex and hard-to-understand source code. Due to a lack of explicit layout definition and uncaptured design intent, generator layouts can hardly be adapted by constructive algorithms directly. Thus, synthesis flows need to adapt layout blocks by varying generator parameters which results in computation-expensive optimization. This paper introduces MESH—a software structure to define detailed and flexible layout generators *explicitly*. Using MESH, just a few lines of code describe complex layouts while all relations and design intents, such as element positions and routing styles, are captured through abstract commands. As a result, generators are created fast with less programming errors, and constructive algorithms can modify the generator *structure* directly.**

Keywords—Layout; Analog Design Automation; Generator; Technology Independence; Reuse

I. INTRODUCTION

Analog EDA is struggling for a long time to gather pace in the analog design community in which designers are still mostly following an almost entirely manual design approach. Especially manual layout design is an iterative long-lasting and error-prone process causing high cost and design risk. Over the last decades, thus, a multitude of approaches has been proposed to tackle different aspects of the analog design problem.

A. State of the Art

Focusing on the creation of analog layouts, one can divide approaches into bottom-up generator-based [1, 2, 3, 4], template-based [5, 6], and top-down optimization-based [7, 8, 9, 10] all of which may interact with each other. The high *structural flexibility but long run time* of optimization-based approaches is well known, as is the *reduced structural flexibility but short run time* of generator-based methodologies [11]. Optimization can handle complex dependencies and constraints [10, 12, 13] *explicitly* while procedural generators implement detailed and valuable expert knowledge *implicitly*. Explicit refers to being accessible to algorithms while implicit states a lack of clear expression. Combining both approaches is promising in order to achieve valuable automation [14].

A recent placement approach utilizes swarm intelligence in order to arrange procedurally generated blocks [15]. Different hierarchy levels are used to encapsulate dedicated constructive tasks and to organize recursive placement cycles while considering constraints. Another approach utilizes local automatic placement within generators [16]. There, the variety of constructive commands which define layout relations is stored in an abstract placement graph. The graph is subsequently analyzed to adapt the layout by constructive algorithms in an explicit way for the purpose of layout optimization.

B. Our Contribution

Our work supports closing the gap between *implicit procedural layout generation* and *explicit algorithmic layout optimization* by providing an explicit bottom-up generator interface. The goal is to describe generators abstractly and *explicitly* based on the expert's intent rather than defining complex, hard-to-understand procedural generator code containing expert knowledge *implicitly*.

Our approach is inspired by template-based methods which provide a symbolic high-level definition of the desired layout floorplan. In order to consider routing as well, the template approach was also used to define detailed routing constraints [17]. Such approaches are very general, but for regular

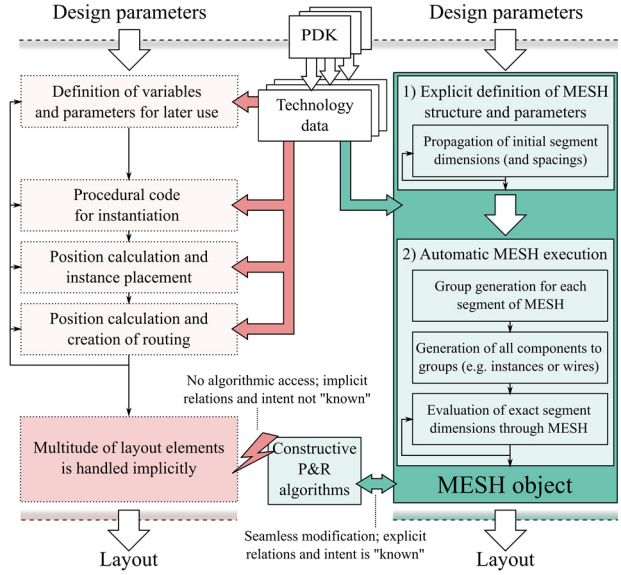

Fig. 1 Comparison of "classical" generator programming (left) and structured generator description using MESH (right). Classical generators may use algorithms *implicitly*; explicit adaption is not designated. With our approach, however, definition of parameters and layout structure is *explicit* through MESH methods. This way, the whole structure and all elements are addressable by constructive commands and algorithms.

Please quote as: B. Prautsch, U. Eichler, T. Reich, J. Lienig "MESH: Explicit and Flexible Generation of Analog Arrays," Proc. of the Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD 2017), Giardini Naxos - Taormina, Italy, June 2017.

structures, which appear in many analog layouts, the required optimization tends to be oversized. Dedicated array generation is likely to reduce this runtime.

With our new approach we intend to overcome very complex procedural generator description (see Fig. 1). For this, we introduce *MESH*—Matrix-arranging explicit generator structure with object-oriented class hierarchy—for explicit, efficient, and correct-by-construction block planning and bottom-up creation of analog layouts. MESH is an extension to procedural layout generators and dedicated to create flexible arrays including routing.

Our contributions are:

- A flexible array arrangement to address a variety of analog block layouts including routing is proposed. Particular arrays are defined *explicitly*.
- In order to overcome the limits of strictly procedural generators, our approach allows *adapting all array elements in any order*.
- The array structure can be used *both* as frame for dedicated procedural layout description and to apply constructive layout algorithms.

II. The Structure of MESH

MESH is organized in a chessboard-like arrangement. Elements of this arrangement, hereinafter referred to as *segments*, represent instances of sub-blocks or devices, vertical or horizontal routing channels, or switch boxes for the interconnection of the routing channels. MESH represents these segments in an abstract (or "symbolic" as called in some template approaches) way, i.e. the structural definition contains no constructive layout generation code but describes the position (meaning row and column in MESH), orientation, and content of the segments. At first this abstract view is "empty", i.e. no abstract primitive layout elements are defined. By default, each instance is surrounded by routing channels and switch boxes. Depending on the target application, this structure is adapted to the final layout structure (see Fig. 2).

During this adaption step, the abstract view is "filled" with information on abstract primitive layout elements using the object-oriented and explicit MESH interface. All data stored is used to create the "real" layout within the subsequent layout generation step discussed in Section II.C.

A. Object Hierarchy of MESH

The intent of MESH is 1) to provide an interface to ensure a fast-to-implement generator structure and 2) to encapsulate the entire abstract design definition in a container object to which constructive algorithms have access. The MESH object contains all segments such as instances (*MInstance*), switch boxes (*MSwitchBox*), and vertical and horizontal routing channels (*MChannel,* with orientation). These segments contain abstract primitive elements (still symbolic) and element-related information such as wires with wire layer and size or instances with instance name and pin information. The class hierarchy is depicted in Fig. 3.

B. Explicit and Abstract Layout Description

In the "classical" and implicit way of procedural generator description, a lot of complex and branched code (few thousand lines) is written which describes the layout of an analog block in a strictly procedural fashion. A comparable approach can be applied to the presented matrix structure. However, the proposed structure is editable at any segment in any order. This way, problems with "locks" or programming order are overcome. An example of such case is given by a row of layout elements *(elemA-elemB-elemC)* in Fig. 4.

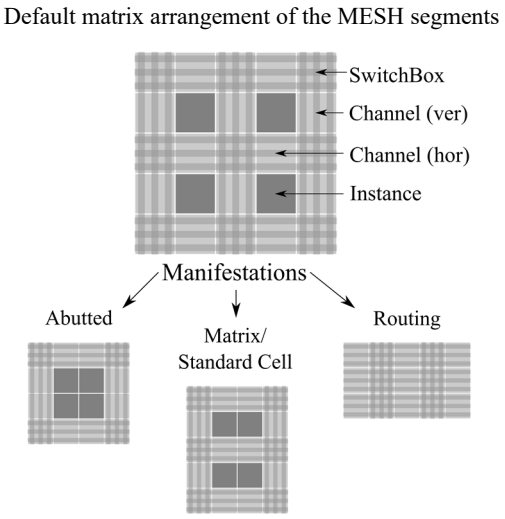

Fig. 2 Exemplified structure of MESH. Initially, ("empty") segments are predefined. Around each instance, horizontal and vertical routing channels are located with switch boxes at the corners. This structure repeats for the given number of rows and columns of instances (here: 2×2). Based on this structure, arbitrary rows and columns can be "filled" with abstract layout elements to achieve the desired layout structure (manifestation).

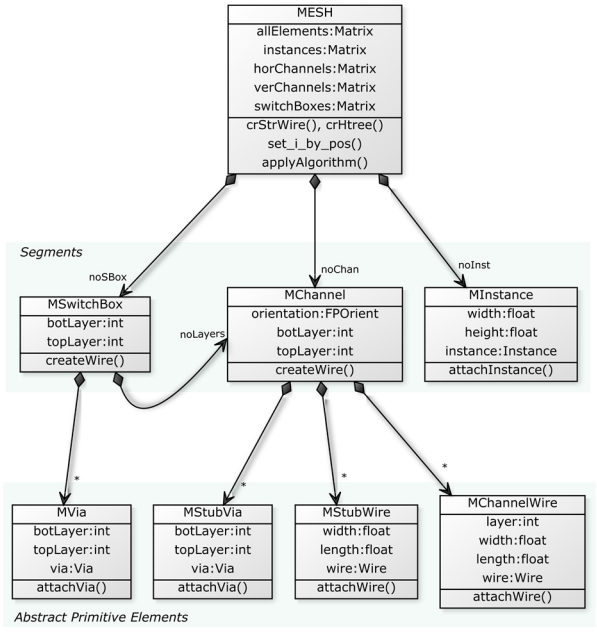

Fig. 3 Simplified UML diagram of the implemented class hierarchy. The MESH object is a container for all segments of the MESH structure. Through this object, each segment and each layout element is accessible. MESH also implements an interface to constructive algorithms utilizing the access methods.

```
Algorithm 1: Procedural "lock" caused by dependency
1  … procedural code causing elemC placement first …
2  elemC.placeRightOf(elemA, <alignment of C to A>)
3  elemB.placeRightOf(elemA, <alignment of B to A>)
4  // the following line affects both elemB and elemC
5  elemB.placeLeftOf(elemC, <alignment of B to C>)
6  … procedural code to handle re-arrangement of
   elemB and elemC …
```

Fig. 4 Exemplified pseudo code of a strictly procedural description with dependency (using detail placement as e.g. in [19, 20]). Due to further layout elements it can happen that the intended layout structure cannot be defined strictly procedurally (e.g. A-B-C). Since in the example *elemB* must be placed last, the intended order has to be corrected either using further procedural code (which may again show problems) or by placement or correction algorithms (such as in [15, 16], resp.).

Our proposed approach encapsulates all layout elements such that explicit and constructive algorithmic arrangement is handled hierarchically and all layout elements are addressable for modification. This way, independently of other layout elements, the intended order *(elemA-elemB-elemC)* is defined at each segment separately within the matrix using ordered *add()* commands or a shorter ordered *addList()* command:

```
meshElem = MESH.get_elem_by_pos( row, col )
meshElem.addList( (elemA, elemB, elemC) )
```

Due to the structure of MESH, we overcome implicit consideration of dependencies. If *elemA* would be a part of an H-tree routing path, other related parts must be created accordingly. Many nets with a multitude of connections to many instances would cause complicated and very branched procedural generator code. In contrast, MESH splits the layout into separate abstract segments which are modified locally. Dependencies are resolved subsequently by the MESH object using the entire context of segments and considering orders.

C. Layout Generation

From the MESH description, the final ("real") layout is generated. In advance to this step, height and width of each segment are propagated vertically and horizontally to arrange all segments properly. This step includes arranging all wires such that they align with related wires in neighboring segments. Moreover, technology-specific rules apply (see [19]) which will affect exact element size and position as well as segment sizes.

After all sizes and positions are specified, a layout group is created for each segment into which the "real" layout entities (i.e. instances, wires, etc.) are created. Finally, all groups are placed according to the abstract matrix position and their individual size.

III. Design Example

The proposed method was used in our IIP Framework [3] in order to create parts of the layout of a current-steering digital-to-analog-converter (CS-DAC). This converter contains sub blocks for both current mirrors and switches. H-tree routing is required for critical nets (for supply to current mirrors, clock, and analog output [18]).

A. Detailed and Algorithmic Constructive Commands

In the first step, MESH was used to explicitly define the layout structure in a detailed but guided way where the positions of all instances and wires were defined w.r.t. the segments. Since for all structural descriptions the MESH interface was used, the resulting constructive algorithms could be added to the MESH object. As the result, a set of constructive algorithms is now available. They currently encapsulate placement as well as complex layout creation such as instance arrangement and the generation of simple H-tree routing (for matrices and transposes of size $2^n \times 2^{kn}$ while $n \in N_{>0}, k \in \{1, 2, 3\}$), respectively. All constructive procedures which were added to MESH are available also to any other (older or future) generator that utilizes MESH. The effort of applying procedural layout description is reduced to a single method call per (complex) constructive step and, additionally, no dependencies must be considered. Hence, with the presented approach, generators can be implemented efficiently and robust. Furthermore, the development of new versions of the generator to address new applications is simplified.

B. Layout Description of CS-DAC Sub Blocks

MESH-based layout description is implemented in three major steps. First, all instances are attached to the intended MESH segment. In order to optimize the placement of current mirrors, we have used a common-centroid algorithm *getCC()*. Second, routing is added by means of constructive MESH methods *crStrWire()* and *crHtree()*. The latter implements parameterizable H-tree routing. This constructive procedure is part of MESH and accessible by a single MESH command

```
Algorithm 2: Layout implementation scheme using MESH
1   // Initialize MESH and create a rows*cols matched
2   // instance matrix using a device counts list devList
3   Mesh = new MESH(rows, cols)
4   ccMtrx = Matching.getCC(rows, cols, devList)
5   // Define orientation by position and add the
    arrangement to MESH (w.r.t. instances, thus "i")
6   for(row, 0, rows-1):
7       for(col, 0, cols-1):
8           instance = ccMtrx.get_i_by_pos(row, col)
9           orient = <calculate by (row, col)>
10          Mesh.set_i_by_pos(row, col, instance, orient)
11  // Create H-tree routing for critical nets
12  Mesh.crHtree(<net name, wire sizes, layers, …>)
13      …
14  // Create straight wires
15  Mesh.crStrWire(<net name, start, end, size, …>)
16      …
17  // Create straight wire buses
18  Mesh.crStrWire(<bus name, start, end, sizes, …>)
19      …
20  // Draw layout
21  createLayFromMESH( Mesh, position )
```

Fig. 5 Pseudo code of the MESH-based implementation scheme which was used for the CS-DAC sub blocks. This structure can be applied to any layout block. First, MESH is instantiated and the placement pattern is defined. Then, all wires are added to MESH while sophisticated structures such as H-trees are defined with one single command. Finally, the layout is created.

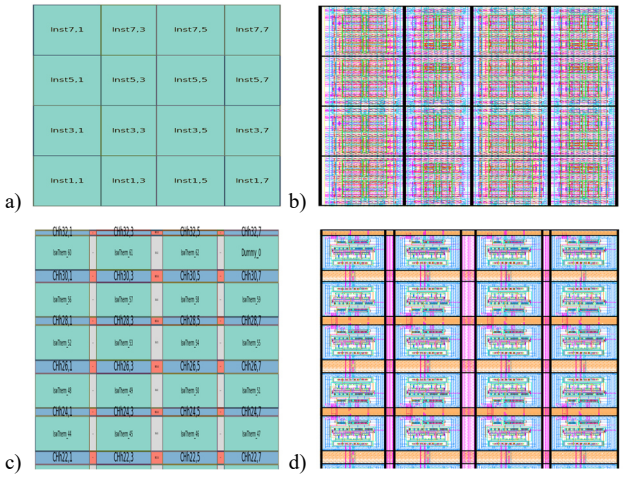

Fig. 6 Illustration of the design example. It can be seen that depending on the defined layout structure MESH segments are either ignored or "filled". In (a), the abstract placement view is given for abutted cells, while (b) depicts the related layout result. Fractions of both abstract view and layout of the routed CS-DAC switch block are depicted in c) and d), respectively.

(with no dependencies to consider) to any MESH-based generator. The creation step is the third and final step of layout generation through MESH. This final command draws the abstract MESH structure to a real layout. The core code structure of the MESH flow is shown in Fig. 5.

Fig. 6 depicts abstract views and layout samples generated using the presented method. The first example shows abutted cells as used in the current mirror block (4×4 instances; routing segments are "empty") generated in about 5 seconds while the second shows a part of the CS-DAC switch block (16×4 instances) including both H-tree and signal routing and supply distribution which required about 150 seconds. The latter example was described with only about 200 lines of source code. The algorithm for H-tree generation consists of about 250 code lines and shows a lot of structural interaction (such as setting positions and sizes) of instances and wires which is handled by MESH.

Note that implementing the entire layout procedurally might cause serious effort while MESH encapsulates the dependencies. The CS-DAC switch block contains three H-tree nets as well as three further nets to each instance (= 3+192 nets). Based on this quantity including related dependencies and considering other "classical" procedural matrix generators we have already implemented, we estimate that at minimum 2000 lines of complex "classical" procedural generator code are required to create the same layout result. MESH-based generators are much more compact while reducing the probability of code errors. Moreover, the layout structure is defined and stored *explicitly*.

IV. Summary and Outlook

Former layout generator approaches are based on complex procedural layout description with plenty of *implicit* considerations and dependencies. This paper introduced MESH which provides abstract methods for defining matrix-arranged layout structures *explicitly*. In contrast to former procedural generators, layout description is implemented in a structured way which is predetermined by the MESH interface. Further problem-specific and abstract layout description is added quickly to the set of constructive MESH methods. This code can then be reused by any other generator resulting in reduced implementation effort.

The presented design example of CS-DAC sub blocks shows that MESH allows efficient and robust implementation of layout generators for fast creation of very dedicated layouts. Complex layouts, such as matched arrangements with H-tree routing, are defined with only a few lines of generator code.

Future work will apply optimization on MESH. Also, the strict tiling will be diminished by hierarchical decomposition.

Acknowledgements

This work was partly supported by the European Union and the Free State of Saxony within the projects THINGS2DO (Ref. No. 16ES0240) and PRIME (Ref. No. 16ESE0110S).

References

[1] J. Crossley, er al., "BAG: A Designer-Oriented Integrated Framework for the Development of AMS Circuit Generators," *2013 IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD),* pp. 74–81, 2013.

[2] A. Graupner, R. Jancke and R. Wittmann, "Generator Based Approach for Analog Circuit and Layout Design and Optimization," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, IEEE, 2011, pp. 1–6.

[3] B. Prautsch, et al., "IIP Framework: A Tool for Reuse-Centric Analog Circuit Design," *13th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD),* June 2016.

[4] T. Reich, U. Eichler, K.-H. Rooch and R. Buhl, "Design of a 12-bit Cyclic RSD ADC Sensor Interface IC Using the Intelligent Analog IP Library," *ANALOG 2013 – Entwicklung von Analogschaltungen mit CAE-Methoden,* March 2013.

[5] R. Castro-López, O. Guerra, E. Roca and F. V. Fernández, "An Integrated Layout-Synthesis Approach for Analog ICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 7, pp. 1179–1189, July 2008.

[6] A. Unutulmaz, G. Dündar and F. Fernández, "LDS based Tools to Ease Template Construction," *Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), 2012 Int. Conf.,* pp. 61–64, 2012.

[7] R. Martins, et al., "AIDA: Automated Analog IC Design Flow from Circuit Level to Layout," in *Proc. of the Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2012.

[8] H. Graeb, et al., "Analog Layout Synthesis - Recent Advances in Topological Approaches," *Proc. of the Conf. on Design, Automation and Test in Europe,* 2009.

[9] R. Martins, N. Lourenco and N. Horta, "LAYGEN II—Automatic Layout Generation of Analog Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* pp. 1641–1654, 2013.

[10] H. Habal and H. Graeb, "Constraint-Based Layout-Driven Sizing of Analog Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 30, no. 8, pp. 1089–1102, 2011.

[11] G. G. E. Gielen and R. A. Rutenbar, "Computer-Aided Design of Analog and Mixed-Signal Integrated Circuits," *Proc. of the IEEE 88.12,* pp. 1825–1854, December 2000.

[12] A. Krinke, G. Jerke and J. Lienig, "Constraint Propagation Methods for Robust IC Design," *Proc. of ZuE 2015; 8. GMM/ITG/GI-Symp. Reliability by Design,* pp. 1–8, 2015.

[13] A. Nassaj, J. Lienig, G. Jerke, "A New Methodology for Constraint-Driven Layout Design of Analog Circuits," *Proc. 16th IEEE Int. Conf. on Electronics, Circuits and Systems,* pp. 996–999, 2009.

[14] J. Scheible and J. Lienig, "Automation of Analog IC Layout – Challenges and Solutions," *Proc. of the 2015 Int. Symp. on Physical Design,* pp. 33–40, 2015.

[15] D. Marolt, J. Scheible, G. Jerke and V. Marolt, "SWARM: A Self-Organization Approach for Layout Automation in Analog IC Design," *Int. Journal of Electronics and Electrical Engineering (IJEEE),* vol. 4, no. 5, pp. 374–385, 2016, doi: 10.18178/ijeee.4.5.374-385.

[16] B. Prautsch, U. Eichler, T. Reich and J. Lienig, "Explicit Feature and Edge Insertion for Improved Analog Layout Generators in Advanced Semiconductor Technologies," *Proc. of ANALOG 2016,* pp. 22–27, September 2016.

[17] A. Unutulmaz, G. Dündar and F. V. Fernández, "A Template Router," *Circuit Theory and Design (ECCTD), 2011 20th European Conf.,* pp. 334–337, 2011.

[18] G. V. d. Plas, J. Vandenbussche, G. Gielen and W. Sansen, "Mondriaan: a Tool for Automated Layout Synthesis of Array-type Analog Blocks," *IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE,* pp. 485–488, 1989.

[19] B. Prautsch, U. Eichler, T. Reich, A. Puppala and J. Lienig, "Abstract Technology Handling for Generator-Based Analog Circuit Design," *Proc. of ZuE 2015; 8. GMM/ITG/GI-Symposium Reliability by Design,* pp. 1–6, 2015.

[20] Synopsys, "PyCell Studio," [Online]. Available: https://www.synopsys.com/cgi-bin/pycellstudio/req1.cgi. [Accessed March 2017].

Please quote as: B. Prautsch, U. Eichler, T. Reich, J. Lienig "MESH: Explicit and Flexible Generation of Analog Arrays," Proc. of the Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD 2017), Giardini Naxos - Taormina, Italy, June 2017.