Explicit Feature and Edge Insertion for Improved Analog Layout Generators in Advanced Semiconductor Technologies

Benjamin Prautsch[*], Uwe Eichler[*], Torsten Reich[*], Jens Lienig[†]
[*]Fraunhofer IIS/EAS, Fraunhofer 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 predominantly manual task that is extremely difficult, time consuming, and costly. The so-called generator-based design methodology is one possibility to reduce the manual effort by substituting design steps with procedural automation. Recent work already discussed a high degree of technology-independence of procedural generators. However, same generator code creates always the same *structure* which reduces flexibility. Moreover, recent generators behave like black-boxes with *implicit* behavior. This work utilizes an abstract layout placement graph in order to include layout relations and/or layout shapes automatically within a post-processing step. As the result, abstract generator descriptions are much more robust and handle the great amount of advanced process design rules which is of high practical relevance when targeting multiple technologies. Both the degree of technology independence and layout quality are therefore increased in an *explicit* way for the first time – without changing any generator code.

1 Introduction

The design of analog integrated circuits is still a greatly manual procedure. While digital circuit problems can be subdivided and abstracted more easily and a variety of “digital” algorithms are focused on this *scaled/quantitative* problem, the analog domain deals with *sensitive interactions* and *many dependencies* which can not yet be handled entirely by current automation approaches [1, 2, 3]. One further challenge is that even comparably small analog design problems lead to a large number of constraints [4]. Even experienced designers cannot be aware of this quantity of considerations at a time. They are more likely to use symmetry and aesthetics as (poor) measure of layout quality [3]. In addition, analog designers often do not focus on the many constraints at all which are, however, important to consider [5, 6, 7, 8, 9]. In order to automate the analog design problem, mainly two approaches are being followed, namely optimization-based [10, 11, 12, 13] and procedural generator-based [14, 15, 16, 17, 18, 19]. The first approach is well-known for its *generic* nature due to the capability to adapt an optimization method to a wide range of design problems. However, with growing problem sizes this approach demands more powerful algorithms. Especially utilizing advanced technologies where very complex and dependent design rules are to be considered optimization tends to be too computation expensive when taking detailed layout tasks into consideration [20]. In the second approach, procedural generators are used to create layouts (also schematics [21] or the entire set of required views [14, 18, 19]) very fast even in advanced processes [19]. The reason for the fast execution speed is the *detailed procedural description* of the solution to be generated (expert knowledge) which, at the same time, is the main drawback: procedural generators are structurally very static.

1.1 State of the Art

Roughly speaking, optimization-based approaches are mainly applied in academia while generator-based approaches are already used in the industry. In order to address future industrial analog design automation, a combination of both principles is promising [2]. However, both approaches are still considered separately today.
Procedural descriptions of layout generators are often defined by relative positions using design rule variables from a technology database such as e.g. in [14, 16, 17, 22] or as in the geometric templates in [10] and [23]. This means that vectors are calculated within the procedural generator which are then used to change the position of a layout element. In “large” technologies (approx. > 180 nm) this method works properly. However, such design rule variables are hardly sufficient in processes below 180 nm and no more sufficient in advanced nodes

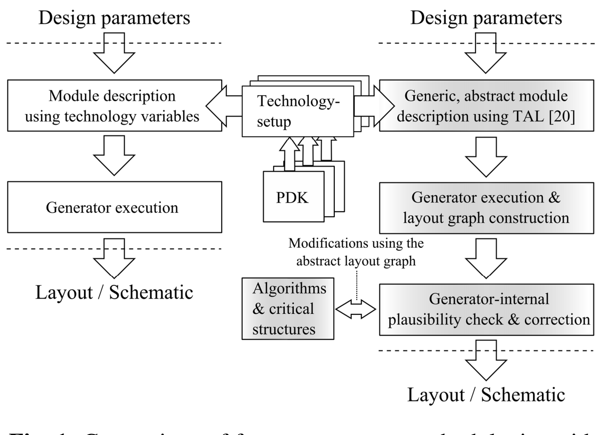
Design parameters
Module description using technology variables
Generator execution
Layout / Schematic
Technology-setup
PDK
Modifications using the abstract layout graph
Algorithms & critical structures
Design parameters
Generic, abstract module description using TAL [20]
Generator execution & layout graph construction
Generator-internal plausibility check & correction
Layout / Schematic

Fig. 1 Comparison of former generator methodologies with no possibility of post-processing (left) and our new generator approach with layout graph construction and graph post-processing (right). Our generators are “transparent”. Therefore, algorithms can be applied directly in order to adapt the generated result rather than changing design parameters.

far below 90 nm since design rules become increasingly layout-dependent. Therefore, in [19, 20, 24] dependent design rules (depending on size and relative position of related layout elements) are calculated and applied during generator runtime based on generator commands which allow a description of abstract placement *relations*. In [20] both calculation and placement are combined into abstract placement commands which, in addition, create nodes and edges of an abstract layout placement graph through the generator code.

1.2 Our Contribution

According to [2], we focus on the combination of both aforementioned automation approaches. Therefore, we follow a methodology to qualify analog generators better for optimization-based approaches in order to reuse the expert knowledge described in the procedural generator code. Thus, in this work we propose a new insertion technique applying the idea of improved accessibility which improves the flexibility of otherwise rather static (meaning *structurally* fixed) layout generators. After the procedural layout generation, all layout relations and layout elements are stored in an abstract layout graph according to [20]. This graph is subsequently analyzed for layout structures out of a library of known critical structures (see **Fig. 1**). If such structures are found in the graph, which is either critical in general (problematic generator code) or relates to a design rule from the technology (e.g. dependent rule), they are resolved by means of insertion of additional features such as edges or both nodes and edges into the layout graph. The goal of this methodology is to improve both technology independence and layout quality of procedural layout descriptions which is of high practical relevance. Moreover, the capability to apply post-processing directly on a generated layout through the generator itself allows the direct connection of generators and optimizers. This means that an optimizer is not limited to vary generator parameters only. Direct adaption of the generated layout result is possible as well by reusing the contained expert knowledge through the placement graph. To the best of our knowledge such insertion technique has not been presented before for procedural layout generators.

Our particular contribution is summarized as follows:

- We propose a new insertion technique which allows post-processing and adaption of procedural analog layout generators
- We present an analysis of layout structures to be considered during this post-processing step
- Algorithms are given which are used during the post-processing step in order to improve the quality of procedurally generated layouts

2 Problem Description

In procedural layout generators $PG_L(P, E, C, t)$, a layout L consisting of layout elements $e \in E$ (such as shapes or figures) is described utilizing variable parameters $p \in P$ (e.g. transistor dimensions). The generator can be executed for each technology t out of the set of supported technologies T through generic and ordered commands $c \in C$ with $C \subseteq C_{API}$. The set C_{API} depends on the programming interface (API) which is used. All c utilized in PG_L describe instantiation, parameterization, rotation, and position or relation of each $e \in L$. Due to the strong connection between P, E, C, t, and L, C_{API} greatly influences the degree of technology independence. In Section 1.1 it is discussed that many former methods only support the description of positions rather than *relations*. In addition, since PG_L can be programmed by an arbitrary series of commands c, generators cannot ensure layout correctness intrinsically and, hence, much generator verification is to be performed.

Therefore, an automated post-processing step with verification and correction is required. In order to extract the command series of PG_L, each c creates either an edge or a node in a layout graph G_L which describes L in an

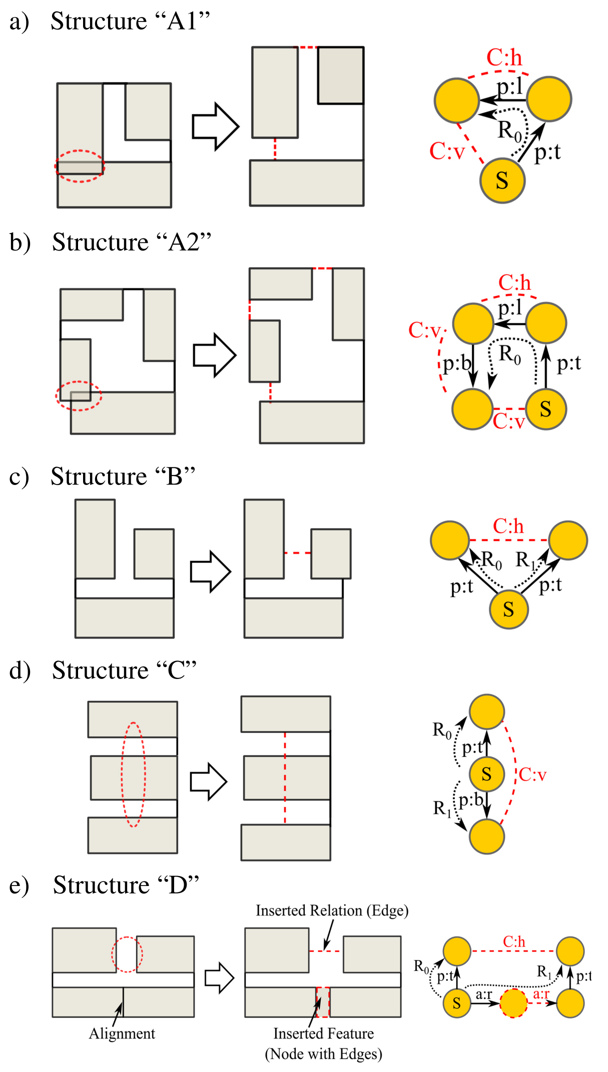

Fig. 2 A subset of critical layout structures which might be defined by a generic generator description. On the left side the generated layout is illustrated, while the corresponding placement graph is shown on the right side. Nodes containing letter S are the starting point of the analysis. Dashed elements are added by the solution strategy.

abstract way (see [20]). Once all commands $c \in C$ were executed, critical structure definitions CS_{def} out of a library L_{CS} must be searched in G_L in order to apply a related solution CS_{sol} which resolves CS_{def}. This means that once a problematic command series is found, it is resolved in an *explicit* way which is required to introduce more flexibility into otherwise structurally static generators.

3 Critical Layout Structures

Due to the freedom of the programmer, layout descriptions may be implemented which lead to DRC violations. In order to improve DRC correctness, critical structures need to be recognized and must be handled using appropriate solutions.

Each node in the placement graph represents a layout element while each edge represents a detailed description of the relation between two layout elements. An edge can either imply an alignment *a*, a placement *p* or decompaction *d*. Alignment means that the borders of two layout elements are placed exactly onto each other, placement means that two layout elements are placed with respect to each other using the minimal possible distance or a greater user-defined spacing, and decompaction means that the spacing between layout elements is increased. Each relation can be defined in the direction dir of all sides left *l*, right *r*, top *t*, and bottom *b* (additionally, reference points are defined which is not discussed in this work for reasons of clarity; see [20]):

$$dir \in \{l, r, t, b\}$$

In this work, a primitive placement relation step RS is defined by a placement relation $r \in \{p, a, x, d\}$ with $x = p \lor a$ and a related direction dir (to be read "r from dir"):

• Alignment:	$a: dir$
• Placement:	$p: dir$
• Placement *or* Alignment:	$x: dir$
• Decompaction:	$d: dir$

3.1 Classification of Critical Structures

In order to analyze the placement, a library L_{CS} containing critical structures CS was defined. This library can be extended fast by new critical structures once new structures were recognized. A critical structure CS contains both the definition of the critical layout structure definition CS_{def} and a related solution CS_{sol}. Thus, it is a tuple $CS = (CS_{def}, CS_{sol})$. Each i-th critical structure definition is a tuple of $n(i)$ relations R which define a path within the abstract layout graph. In order to define paths, each j-th relation is a tuple of $m(j)$ primitive relation steps RS which contain the actual placement relation and placement direction (data of edges) of the critical structure. Each relation R starts from the start node S – the node currently analyzed during the iterative search (see Section 4). Generalized, a critical structure CS is given as follows:

$$CS = (CS_{def}, CS_{sol}),$$
$$CS_{\mathrm{def}\,i} = (R_0, R_1, \dots, R_{n(i)}),$$
$$R_j = (RS_0, RS_1, \dots, RS_{m(j)}),$$
$$RS = r: dir$$

The solution CS_{sol} to resolve a critical structure CS contains information about new (inserted) edges and/or nodes. Such edges and nodes are inserted and/or overridden. Again a relation R is used in order to describe the location of edge and node insertion.

In **Fig. 2** critical structure definitions out of L_{CS} are illustrated in black/solid while the related solution is shown in red/dashed. Relations are illustrated as dotted arrows. As an example, in Fig. 2a) the critical layout structure definition is $CS_{def} = (\,R_0\,)$ with $R_0 = (\,x: t, x: l\,)$. The related solution CS_{sol} is applied from the node found along R_0. This means that the inserted edge is starting from the upper left node facing node S. "C:v" indicates that the new edge includes a vertical constraint. In this particular case, first the upper left element is decompacted upwards by moving the node located at R_0 relative to S. Following this, placement or alignment of the upper right element (found at $R = (x: t)$) is applied from the right w.r.t the upper left element to maintain the related edge. This solution is written as follows:

$$CS_{sol} = \big((R_0, (d: t)), (R_0, R, (x: r))\big)$$

Since real designs contain several levels of hierarchy as well as so-called figure groups, which realize a logical

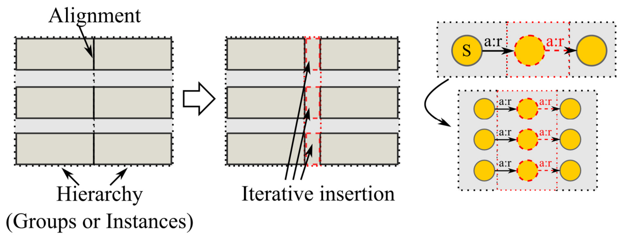

Fig. 3 Hierarchical application of the analysis of critical structures (dotted rectangles mark hierarchical elements such as figure groups or instances). On the left side a layout is illustrated while on the right side the corresponding graph representation is given (dashed elements are added).

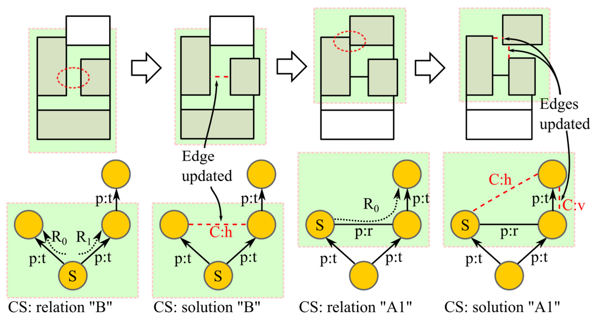

Fig. 4 Iterative application of the feature insertion technique. Once a critical structure is found, the related solution is applied. If nodes and/or edges are inserted, nodes are added to the list of starting nodes and edges are considered during the ongoing structure search.

hierarchy, solutions are to be propagated top-down as it is illustrated in **Fig. 3**. In this example, a node and an edge are inserted at position $R = (a\colon r)$. Since the starting node is a figure group, the solution is propagated into this figure group where it is applied to each primitive shape which is aligned on the right figure group border (dir equals r).

In order to apply corrections on the whole layout, the starting node S "moves" through the entire abstract layout placement graph as it is illustrated in **Fig. 4**. This is discussed in more detail in Section 4.

3.2 Isomorphisms of Critical Structures

In order to reduce the number of defined critical structures and to increase the reliability of the library of critical structures, each structure is transformed such that rotation and mirroring are considered. This transformation is necessary due to the variety of possible orientations of the analyzed layout. Applied transformations are, therefore, defined as set of cyclic permutations (symmetric groups, such as clockwise rotation and mirroring). The following $N_{SG} = 7$ symmetric groups are applied on each element out of the library of critical structures L_{CS} in order to extend the number of structures which will be searched in the layout placement graph:

- Rotation, 90°: R90 = (t l b r)
- Rotation, 180°: R180 = (t b)(r l)
- Rotation, 270°: R270 = (t r b l)
- Mirrored, x-axis: MX = (t b)(r r)(l l) = (t b)
- Mirrored, y-axis: MY = (t t)(r l)(b b) = (r l)
- Composition R90∘MX = (t l b r)∘(t b) = (t r)(l b)
- Composition R90∘MY = (t l b r)∘(r l) = (t l)(b r)

As an example, the direction dir of the placement step $RS = p\colon l$ is transformed into each isomorphism resulting in a transformed set I_s (the identity function applies first and multiple appearances are allowed; $|I_S| = N_{SG} + 1$):

$$I_s = \{p\colon l, p\colon b, p\colon r, p\colon t, p\colon l, p\colon r, p\colon b, p\colon t\}$$

This transformation is also to be done for each placement step within a relation R such that a set of new relations is created. For example, a relation R_0 is $(p\colon t, p\colon l)$ which evaluates to the following set of transformed relations T_R:

$$T_R = \{\ (p\colon t, p\colon l), (p\colon l, p\colon b), (p\colon b, p\colon r), (p\colon r, p\colon t), \\ (p\colon b, p\colon l), (p\colon t, p\colon r), (p\colon r, p\colon b), (p\colon l, p\colon t)\ \}$$

Applying this transformation for CS_{def} and CS_{sol} out of each critical structure CS as well, the library of critical structures contains all transformations of each critical structure. Multiple appearances of same isomorphisms are now ignored in order to search each structure only once.

4 Algorithms to Find and Resolve Critical Layout Structures

The critical structures discussed in Section 3 are now to be searched in the abstract placement graph which is generated during runtime of the generator. Former generators would simply be finished after all commands

```
check_cs_from_startNode($S, L_{CS\,iso}$):
fitLst = {}
nodeEdges = S.get_edges_outgoing()
∀ cs ∈ $L_{CS\,iso}$:
    allFit = True; ignEdge = {}; edgeList = {}
    ∀ R ∈ cs:
        fit = False
        ∀ edge ∈ nodeEdges|ignEdge:
            (edgeSer, success) = getFits(R, edge)
            if success:   // R fits at edge
                ignEdge = ignEdge ∪ {edge}
                edgeList = edgeList ∪ {(R, edgeSer)}
                fit = True
        if not fit:   // R fits at no edge
            allFit = False; break
    if allFit:   // all R out of cs fit node S
        fitLst = fitLst ∪ {(cs, edgeList)}
return fitLst
```

Fig. 5 Algorithm to find a critical structure out of the set of transformed critical structures $L_{SC\,iso}$ starting from node S. $getFits()$ implements a search starting from $edge$ to check if relation R can be found. If so, the series of edges is returned. Each critical structure cs which was found is stored in $fitLst$ together with the relation R, and the related list of edges. Finally, $fitLst$ is returned.

were executed. In contrast, in the presented work a post-processing step is applied in order to check and correct the resulting layout.

4.1 Search of Critical Structures

First, the critical structures $CS \in L_{CS}$ are transformed as given in Section 3.2 which results in a set of transformed critical structures $L_{CS\,iso}$ containing more critical structures since all isomorphisms are considered. For each node S (current starting node) in the abstract layout graph, it is checked whether or not a critical structure out of $L_{CS\,iso}$ matches starting from S (see **Fig. 5**). This means that it is checked if the edges starting from S fit the relations R out of the critical structure definition CS_{def} which is contained in the critical structure CS. Once a relation is found at node S, the related starting edge $edge$ is ignored during the next iteration using $ignEdge$ (heuristic approach to reduce the number of searches; it is assumed that at most one critical structure can be found per edge which means that L_{CS} must be defined properly).

```
apply_cs_solutions($S, fitLst$):
reslvNodes = {}
∀ (cs, edgeLst) ∈ fitLst:
    $cs_{sol}$ = cs.get_solution()
    ∀ solution ∈ $cs_{sol}$:
        movNode = applyPlc(solution, edgeLst, S)
        reslvNodes = reslvNodes ∪ {movNode}
return reslvNodes
```

Fig. 6 Algorithm to resolve the critical structures which were previously found starting from node S. $applyPlc()$ implements the actual application of each resolved new or updated placement command which is located defined by relation R (contained in $solution$). Once each solution is applied, all resolved nodes (= layout elements) are returned.

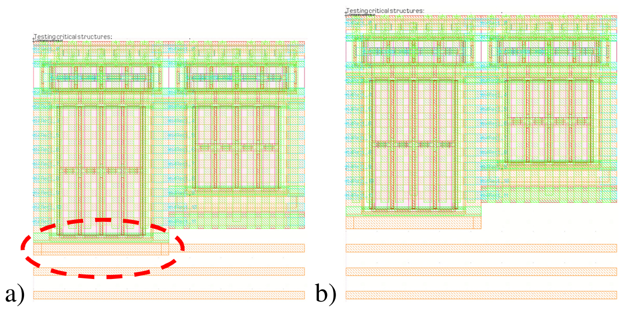

Fig. 7 A part of a test generator showing the critical structure “A1” out of the library L_{CS}. Without edge-insertion an overlap error occurs (a), while the same structure is generated correctly using our edge-insertion technique (b).

4.2 Resolving Critical Structures

After the algorithm from Section 4.1 is completed, the returned value $fitLst$ contains the information about critical structures which were found starting at node S. This information is now used to resolve the critical structures (see **Fig. 6**) such that after the iterative post-processing step the overall layout is corrected.

5 Experimental Results

In order to test the capability of the proposed method and algorithms, two different test generators were developed. The first test generator implements the critical structures defined in the critical structure library L_{CS}. It verifies that each of those critical layout structures are found and resolved correctly. The second test generator creates an arbitrary series of placement commands. This second test is used both to find further critical structures and to evaluate the capability of the search algorithm.

5.1 Test Generator for the Generation of Known Critical Structures

In order to check proper recognition and resolving of known critical structures, all critical structures are created in a dedicated test generator. **Fig. 7** shows a part of the layout result from this test generator. While the critical structure is not resolved in Fig. 7a), in Fig. 7b) the layout issue was resolved by the edge insertion technique. It is important to mention that both layouts result from identical generator code. The only difference in Fig. 7b) is that the edge insertion was applied in a *post-processing step* which first analyzed the layout using the abstract placement graph and subsequently applied an *explicit* solution provided by the critical structure library L_{CS}.

5.2 Test Generator for the Generation of Pseudo-Random Patterns

In addition to the generation of the known patterns, a pseudo-random generation was applied as well. This means that a second test generator was developed which executes a large number of instantiations of shapes followed by random placement and align commands from multiple directions. This variety is used to test the capability of our algorithms (see **TABLE 1**).

TABLE 1 Search of critical structures in a pseudo-randomly generated layout.

No. of nodes	**20**	**40**	**80**	**160**
No. of structures	*2*	*5*	*10*	*19*
CPU runtime	*0.10 s*	*0.21 s*	*0.51 s*	*1.74 s*

It can be seen that currently the CPU runtime t depends slightly quadratic on the number of layout elements N ($t \approx (5 + 0.1 * N + 0.006 * N^2) * 10^{-2}$s). More than 1000 layout elements are required in order to reach a runtime of a minute which would still be reasonable. However, since analog cells often contain a comparably low number of layout elements (< 100), the runtime is very short (< 2 s). Independently, we expect that both the implemented graph database and the applied algorithms can be optimized further for runtime. The graph database could be substituted by a more performant open source implementation and improved checks for sub graph isomorphism may speed up the algorithms. Additionally, the procedures for placement graph construction and related checks can be improved.
Furthermore, it should be stated that the post-processing step can be deactivated at any time. In a real design flow (or synthesis), one may (partially) skip this step during layout-aware sizing of all involved modules. In the final step, the post-processing would be activated again in order to improve the generated layouts efficiently.

6 Discussion of the Proposed Insertion Technique

In contrast to former generator-based approaches such as [14, 15, 16, 17, 18, 24], our method includes a post-processing step. The experimental results show that this post-processing step following the generator code execution is capable of improving the generated layout in an *explicit* fashion without changing any line of generator code. Such capability is a prerequisite in order to achieve *real* technology independence of generators. Dependent spacing rules and density rules are examples for considerations which would otherwise require technology dependent generator code. In addition, our method can be extended towards two main directions.
First, the proposed method can be used to apply a built-in verification step after generator execution. This means that *logical errors* in the generator code are immediately found which both improves generator robustness and decreases generator development time by fast debugging.
Second, instead of searching and resolving known critical layout structures out of a fixed library L_{CS}, further algorithms can be developed which are capable of finding critical layout structures automatically. Such algorithms are not limited to consider DRC issues only. Also further considerations such as constraints regarding symmetries between layout elements or coupling constraints can be

addressed. This means that such algorithms can either again verify the quality of a generator or they can directly apply changes in order to improve the resulting layout quality by means of (explicit) constraints which are provided to the generator.
Moreover, one can imagine that the link “into” the generator can be used for a wide range of further algorithms. We believe that this insight and the capability of applying algorithms will allow merging expert knowledge programmed in generators and optimization. This means that an optimizer can search a solution in a more directed way (cf. Fig. 1, right) instead of just defining parameters and evaluating a black-box behavior (cf. Fig. 1, left) iteratively.

7 Summary and Outlook

In this paper a new methodology to improve the technology independence and layout quality of procedural analog circuit generators was presented. An abstract placement graph which is extracted from the generator code during runtime is utilized during a post-processing step to resolve layout issues *explicitly* using a library of critical layout structures. Applying test generators, it was shown that the principle can actually be used to modify the layout result of a generator without changing any line of generator code. Our method not only improves the quality of generated layouts but it also improves the technology independence of procedural generators in advanced processes since design rules can be considered holistically by applying appropriate algorithms which utilize the abstract placement graph.
The new post-processing step will improve the correctness of generated layouts since the consideration of critical structures is no more to be defined only within the generator code *implicitly* but also by a separate library of *explicit* critical structures. Once a new structure is found, it can be added to this library to improve layout quality further.
Moreover, the post-processing step utilizing the abstract layout graph can be used to apply arbitrary algorithms such as for verification or for layout changes directed by *explicit* constraints. Future analog design automation methods can, thus, link generators and optimization directly in order to combine the advantages of both approaches.

Acknowledgements

The presented work was partly supported by the European Union and the Free State of Saxony within the project THINGS2DO (Ref. No. 16ES0240).

References

[1] R. A. Rutenbar and J. M. Cohn, "Layout Tools for Analog ICs and Mixed-Signal SoCs: A Survery," *Proc. of the Int. Symp. on Physical Design ISPD,* pp. 76–83, 2000.

[2] 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.

[3] R. A. Rutenbar, "Analog Synthesis (and Verification) Revisited: Whats's Missing?," *Int Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design, SMACD,* Sept. 2012, http://rutenbar.cs.illinois.edu/publication/. [Accessed July 2016]

[4] H. Graeb, S. Zizala, J. Eckmueller and K. Antreich, "The Sizing Rules Method for Analog Integrated Circuit Design," *Proc. of the 2001 IEEE/ACM Int. Conf. on Computer-aided design,* pp. 343–349, Nov. 2001.

[5] A. Krinke, M. Mittag, G. Jerke and J. Lienig, "Extended Constraint Management for Analog and Mixed-Signal IC Design," *Proc. of the 21st European Conf. on Circuit Theory and Design (ECCTD),* pp. 1–4, Sept. 2013.

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

[7] 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, Aug. 2011.

[8] G. Jerke and J. Lienig, "Constraint-driven Design — The Next Step Towards Analog Design Automation," *Proc. of the 2009 Int. Symp. on Physical design (ISPD'09),* pp. 75–82, 2009.

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

[10] 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.

[11] 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, Nov. 2013.

[12] R. Martins, N. Lourenco, S. Rodrigues, J. Guilherme and N. Horta, "AIDA: Automated Analog IC Design Flow From Circuit Level to Layout," *Proc of the Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD),* pp. 29–32, 2012.

[13] H. Graeb, F. Balasa, R. Castro-Lopez, Y.-W. Chang, F. V. Fernandez, P.-H. Lin and M. Strasser, "Analog Layout Synthesis - Recent Advances in Topological Approaches," *Proc. of the Conf. on Design, Automation and Test in Europe,* pp. 274–279, Apr. 2009.

[14] IPGen 1Stone Developer, [Online]. Available: http://www.ipgenme.de/eda-and-ip-products/1stone-developer.html. [Accessed July 2016].

[15] S. Youssef, F. Javid, D. Dupuis, R. Iskander and M.-M. Louerat, "A Python-Based Layout-Aware Analog Design Methodology for Nanometric Technologies," *Proc. of the IEEE 6th Int. Design and Test Workshop (IDT),* pp. 62–67, Dec. 2011.

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

[17] 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," *Proc. of ANALOG 2013 – Entwicklung von Analogschaltungen mit CAE-Methoden,* Mar. 2013.

[18] J. Crossley, A. Puggelli, H.-P. Le, B. Yang, R. Nancollas, K. Jung, L. Kong, N. Narevsky, Y. Lu, N. Sutardja, E. An, A. Sangiovanni-Vincentelli and E. Alon, "BAG: A Designer-Oriented Integrated Framework for the Development of AMS Circuit Generators," *Proc. of the 2013 IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD),* pp. 74–81, Nov. 2013.

[19] B. Prautsch, U. Eichler, S. Rao, B. Zeugmann, A. Puppala, T. Reich and J. Lienig, "IIP Framework: A Tool for Reuse-Centric Analog Circuit Design," *Proc. of the 13th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, June 2016.

[20] 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-Symp. Reliability by Design,* pp. 1–6, Sept. 2015.

[21] D. Marolt, M. Greif, J. Scheible and G. Jerke, "PCDS: A New Approach for the Development of Circuit Generators in Analog IC Design," *22nd Austrian Workshop on Microelectronics (Austrochip),* pp. 1–6, Oct. 2014.

[22] N. Jangkrajarng, S. Bhattacharya, R. Hartono and C.-J. R. Shi, "IPRAIL—intellectual property reuse-based analog IC layout automation," *Integration, the VLSI Journal,* vol. 36, no. 4, pp. 237–262, 2003.

[23] R. Castro-López, F. V. Fernández, F. Medeiro and A. Rodriguez-Vazquez, "Generation of Technology-Independent Retargetable Analog Blocks," *Analog Integrated Circuits and Signal Processing,* vol. 33, no. 2, pp. 157–170, 2002.

[24] Synopsys, "PyCell Studio," [Online]. Available: https://www.synopsys.com/TOOLS/IMPLEMENTATION/CUSTOMIMPLEMENTATION/Pages/pycell-studio.aspx. [Accessed July 2016].