Template-Driven Analog Layout Generators for Improved Technology Independence

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

Abstract

Analog generators, especially those used for automatic layout creation, are powerful tools to support the still largely manual analog design flow. The effort for generator development, however, is often found to be a bottleneck. Further, verification of generators is usually based on many cycles of generation, each requiring subsequent verification. This is often expensive in terms of computation effort. Up-to-date generators only allow to detect failures using post-layout checks such as DRC and LVS because they describe the designer's intent implicitly as a sequence of (layout manipulation) commands which cannot be verified directly. Also, sequential code often prevents the description of interdependent layout structures or forces the programmer to include extra code, which can again cause errors. In order to overcome these issues, we introduce a new approach to implement layout generators. In a first step, the layout is described as an abstract template. A second automatic step checks this template for structural errors and schedules the required procedural commands. As the result, layout generators are more compact, easy-to-read, and errors can be detected by formal checks of the template description. The new approach was applied to two technology nodes, 180 nm and 22 nm.

1 INTRODUCTION

Analog design automation is not yet common in industry, although a lot of research effort has been spent over the past decades. Most recent innovations are optimization-based layout-aware sizing tools [1, 2, 3] which incorporate generation of layout components [4, 5] in a top-down fashion. These approaches usually employ templates which represent component placement of the target layout in an explicit and abstract way [6]. Such templates are either predefined [7], extracted from existing designs [8], or generated dynamically [9]. Further, even fully-automated approaches not utilizing templates are possible [10]. Routers are used by such tools to physically connect placed components in the layout. These routers can be based on templates again [11] or they employ optimization. Based on the generated routing, parasitic effects can be considered during optimization. Thereby, constraints are used both for placement and routing to control the automatic processes [12, 13, 14, 15].
Besides optimization and templates, another approach utilizing procedural commands is followed, namely the generator-based approach [16, 17, 18, 19, 20, 21]. Generators are suitable for bottom-up creation of layouts. They describe layout details while containing expert design knowledge *implicitly* [22]. Generators can also be combined with top-down algorithms as e.g. for automatic placement of generated components [23]. Due to implicit code, experienced programmers spend significant effort on generator programming and verification. Although comprehensive tools exist that accelerate development of generator code and help debugging [24], generator correctness and independence of the process technology still cannot be guaranteed. An approach to *explicitly* capture the design intent through the generator code should be considered instead. Such an approach is a step to bridge the gap between implicit generators and explicit–thus algorithmically accessible–templates.

1.1 State of the Art of Generators

Types and complexity of generators are diverse. We define a generator as a piece of procedural code which executes commands sequentially (hereinafter called *procedural*; also see [22]). This code is executed in a framework environment such as [17, 18, 19, 20, 25] which provides the generator programming interface (generator API). Based

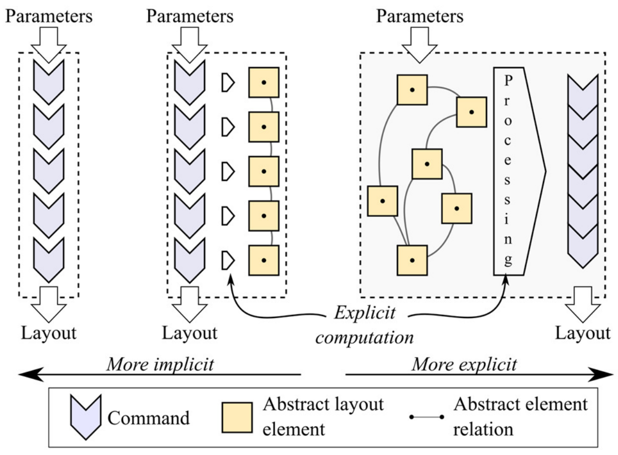


Figure 1 Comparison of previous procedural generators (left), procedural generators that capture explicit relations (middle), and the proposed template-driven generators with separate processing step (right). Procedural generators create layouts right away while abstract commands of template-driven generators can be handled by further algorithmic steps such as adaptation, scheduling, and checking.

TABLE 1 OVERVIEW OF BOTH GENERATOR AND TEMPLATE APPROACHES

Reference	Schematic/ Layout	Technology Handling	Uses Graph	Uses Templates	Comment
LDS [6]	-/x	Parameters	x	x	Script to define templates
IPRAIL [8]	-/x	Parameters	x	x	Structural templates for technology migration
(g)PCDS [17, 33]	x/-	Parameters	-	-	High-level schematic description
IPGen OneStone [18]	x/x	Parameters	-	-	Generator environment
BAG [19]	x/x	Interface (uses pyCells)	-	x, based on helper classes	Procedural templates and simulation
SWARM [23]	-/x	Parameters	-	-	Adaptive floorplanning by algorithms
pyCells [25]	-/x	Interface	-	-	Generator environment
IIP [20] and TAL [26] (our generator framework)	x/x	Interface	x	x (introduced by this work)	IIP is the environment, the new method was added to

on the generator code, schematics and/or layouts are generated (this paper is focused on layout generation). Parameters allow adaptation of dimensions but also topologies through the procedural code. Further, generators can be executed for multiple technologies. The extent of how many technologies are supported, strongly depends on the generator API. Often, *parameters* are used in the procedural code in order to map technology data. These parameters store *static* technology data only. Technology *interfaces* allow more independence of process technologies as they can handle complicated *context-dependent* design rules of advanced nodes [26]. Such design rules show rule values depending on an increasing count of measures of the layout context as minimum feature size decreases (see design rules of the FreePDK as an example [27]). An overview of generator and template approaches is given in **Table 1**.

1.2 Our Contribution

In contrast to implicit and procedural generators (which can also extract explicit layout relations as an additional result [26]), we propose an approach which allows layout description in an abstract way and processing of this description in an intermediate step prior to procedural layout generation (see **Figure 1**). Our approach adopts the idea of the recent MESH approach [28]. Based on an explicit interface, all structural information is stored prior to generator run in a dedicated object. While with MESH only array-arranged layouts can be described, our paper addresses the description of arbitrary layouts. Since the structural definition of the target layout is stored prior to execution, checking, ordering, and adaptation of the commands necessary for layout creation becomes possible. Therefore, also details of the layout description can be changed automatically which would otherwise violate design rules of the particular process technology the generator is run for. As an example, seemingly simple design rules such as maximum width or minimum area may force the *structure* of the layout to be changed accordingly (e.g. wires are sliced or shapes are added). Such adaptation, however, cannot be foreseen for each and every case in the generator code implicitly. Furthermore, updating all generators once design rules get more advanced is not efficient and error-prone. Thus, the generator code should be organized such that adaptation of layout details is possible automatically without changing the generator template. Adaptation of layout elements may require adaptation of related elements which needs to be handled, too.
Therefore, we propose to include the following new steps into generators which are also the contributions of this paper:

- Substitution of procedural generator code by a structural description. This way, the generator is turned into a template-driven description which implements relations of layout elements prior to layout creation.
- Implementation of rule checking to check all layout elements described and running strategies for layout adaptation if a design rule is violated.
- Scheduling of layout generation commands to create layout structures in a meaningful order and to satisfy context-dependent design rules.

2 PROBLEM DESCRIPTION

Former analog layout generators implement the layout description in a way that the final structure is already predefined. Depending on parameters, multiple structures (e.g. transistor with different gate count) are often implemented in a single generator. Further, generator descriptions are implemented procedurally, meaning that all commands and related layout generation are executed line by line. In case commands are based on layout data, which is not yet available (e.g. length of a wire along instances

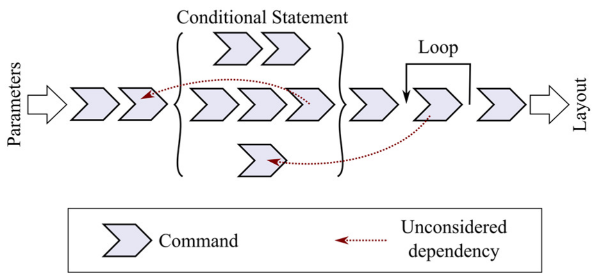


Figure 2 Scheme of a procedural generator without captured intent (running from left to right). Since commands run sequentially, effects on previous commands cannot be treated (unconsidered dependencies). Additionally (not treated in this paper), conditional statements and loops increase code complexity and worsen testability.

which were not yet placed), additional generator code is often included to pre-calculate layout positions. Such extra code for forecasting is error-prone, especially if context-dependent design rules of advanced processes must be fulfilled. Moreover, any dependency between commands is implemented *implicitly*, i.e. without being *explicitly* described e.g. using layout constraints [12] or abstract relations [26].

Therefore, as complexity and dependency of design rules increase in ever more advanced processes, an increasing count of layout structures generated through prior commands can get invalid. This principle is illustrated in **Figure 2**.

Finally, with new technologies new design rules will have to be considered during layout generation. They affect layout details of vias, wires, and other shapes as well as placement of instances and devices. As the result, in contrast to, for instance, simple spacing rules where only *dimensions* are adapted, advanced rules can require *structural* adaptation. Since, however, former generators implement detailed geometries without considering all individual types of design rules, such generators cannot ensure independence of technology if rule types change significantly. Therefore, every single generator must be adapted to meet new rule types.
Summarizing, an approach is required which is able to (1) consider dependencies of commands and (2) allow structural adaptation of layout details while the generator still creates the target structure according to the same underlying template description. This means that all elements of the generator description affected by structure-changing design rules must react automatically based on strategies. Such strategies can be implemented once and globally.

3 TEMPLATE-DRIVEN LAYOUT GENERATORS

As mentioned before, procedural layout generators lack flexibility in the description of layout structures because the layout is already predefined very detailed. Additionally, since generators work procedurally in the sense that each command runs line by line (with simultaneous layout generation), generators do not "remember" the commands run before. In other words: the description is implicit and there is no algorithmic access to the structure described by the generator code. Unfortunately, subsequent commands in the command series can cause design rule violations although former commands on their own had run correctly.
The following subsections discuss the problem of procedural generators based on an example followed by our new approach to overcome these limitations.

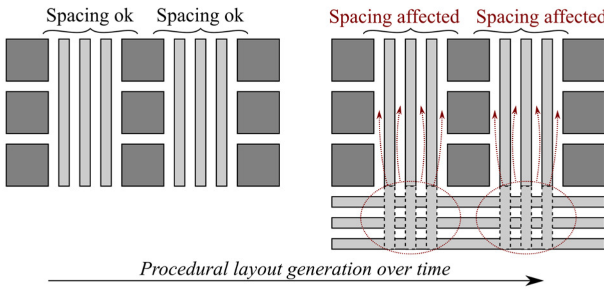
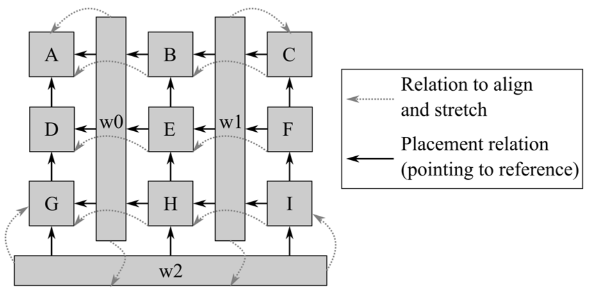


Figure 3 Sketch of the procedural generation of an array arrangement with context-dependent spacing rules. Intermediately, the placement of each individual instance (dark grey) and wire (light grey) is correct (left). When vertical wires are stretched to connect them to the bottom of the horizontal bus, their former spacing values must be recalculated due to a new layout context (right). For the sake of simplicity, wires to devices and vias are not illustrated here.



A
B
C
D
w0
E
w1
F
G
H
I
w2
Relation to align and stretch
Placement relation (pointing to reference)

Figure 4 Graphical representation of the template representing the array shown in Section 3.1 (simplified). Devices are indicated by letters while wires are indicated by "w" followed by a running number. Solid arrows represent placement relations while dotted arrows indicate relations for element alignment or stretching.

3.1 Backlashes of Context-Dependent Design Rules

In [26] and [30] context-dependent design rules can be handled for single placement tasks. If subsequent commands, however, change the layout context the former placement depended on, generated layouts can still violate context-dependent design rules. An example we observed in array structures is sketched in **Figure 3**. Multiple devices are to be placed and connected. The final lengths of all wires is unknown until the layout structure is completed. The reason is that the individual number of horizontal wires at the bottom including their individual context-dependent spacing rule has to be evaluated first. Later, the vertical wires are extended. Since the shape of the vertical wires is changed (new context as the size was increased; see [27]) but their distance was calculated based on a different former layout context, design rule violation can occur.

3.2 Template-Driven Layout Commands

The basic idea of templates is to define layouts in an abstract way. Based on structural commands which define placement relations of layout elements (e.g. instances or wires) as well as their relative alignment including abutment, a graph is built (in this work using [31]). Layout elements are represented by nodes and relations between them are represented by edges. Edges can represent both placement relations and further detailed relations for alignment or stretching. **Figure 4** shows the graphical representation of the template definition, with arrows indicating layout relations, which describes the prior array example (with just a single wire each for the sake of clarity).
At the time the template description is completed, no layout element has been created yet (as it would be the case in procedural generators). The next step analyzes the template

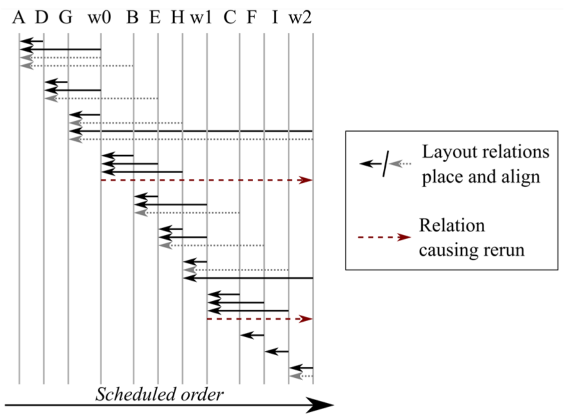


Figure 5 Illustration of the scheduled order of layout creation based on the explicit relations of the template shown in Figure 4. Arrows represent relations while vertical lines indicate layout elements according to their identifier on top. Based on the template, the resulting order of layout commands is determined. Solid and dotted arrows can be handled procedurally as they can be scheduled, while dashed arrows cause rerunning of commands.

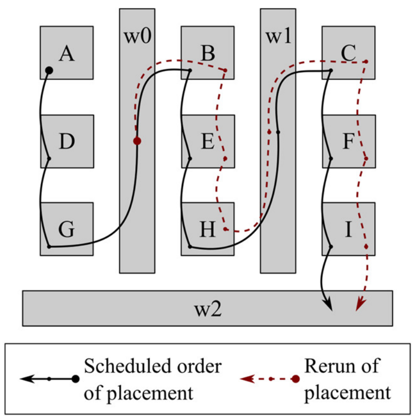


Figure 6 Graphical representation of the template with the related order of element placement indicated by arrows (cf. Figure 4 and Figure 5). The solid arrow represents the procedural order of layout creation while the dashed arrow represents the subsequent rerun step which re-executes parts of the layout placement. Rerunning is required in order to handle the dependencies both between w2 to w0 and w2 to w1.

description, runs generation of instances to gain information on instance sizes, may run a strategy for structural modification, and checks plausibility of the description.

3.3 Command Scheduling

Relations between layout elements defined by the template are represented by edges of the graph. Capturing layout relations was already introduced for generators in [26] but there the graph is an additional result of the generator created during execution (instead of an intermediate result) which allows post-processing of the run procedural generator code [32]. In this work, however, the graph is built prior to generator execution as an intermediate result. Thus, analyzing the dependencies of layout elements prior to layout generation gets possible. Such an approach was already realized in [28], but there, contrary to this work, array-arranged layouts can be defined only. One new step required for arbitrary layout definition is command scheduling. Every layout element's position is dependent on one or more other layout elements. This dependency must lead into a reasonable order of constructive commands to realize proper layout generation. While previous generators implement this command order implicitly, we can derive the dependency from the template definition. **Figure 5** shows these dependencies (already ordered).

Command ordering is based on iteratively eliminating the graph's nodes. At each iteration, the single node which has no dependency to other remaining nodes (node with no outgoing edge) is eliminated. The eliminated nodes are added to a list which is then used to define the command order once every node was eliminated. In case a node cannot be eliminated, the template contains a logical error, such as missing element relation (graph with unconnected node) or contradicting placement (e.g. an element being both left and right of another element).

Figure 6 illustrates the procedural path found through the template defined. Since *w0* and *w1* depend on *w2* (see Figure 4), a part of the sequence is executed again. For this, nodes with the minimum number of outgoing edges and with the maximum number of incoming edges (out of all nodes where all nodes have outgoing edges) are stored for re-scheduling. Please note that such iteration is hard to implement into implicit generators.

3.4 Consideration of Design Rules

Another effect which worsens technology dependence is seen when design rules contradict either other design constraints or the implicit and structural layout definition. Examples are:

- maximum width rule vs. minimum width due to current requirements,
- fixed gate orientation vs. flexible transistor placement,
- layout shape coloring (different "colors" of the same layer must alternate to avoid color conflicts [29]) vs. predefined arrangement of layout shapes,
- number of via cuts vs. shape and size,
- minimum area is violated and extra shapes are required (extra shapes may cause further conflicts),
- additional gate dummy poly affecting transistor placement and spacing.

Since the examples mentioned above would be again considered implicitly in recent generators, extra generator code is required in *each* generator to realize this *structural* adaptation. Depending on the effort spent to implement this implicit generator code and depending on verification, such rule violation might be avoided. However, it cannot be guaranteed, the complexity of every generator will increase significantly, and testability becomes worse.

From a general point of view, handling design rule violations first causes an action and second related layout adjustment of further layout elements might be required.

We extended the programming interface of our IIP Framework [20] in order to allow dynamic changes of layout details generated. This interface utilizes the former TAL approach (TAL: Technology Abstraction Layer) [26] for accessing technology data in a generic way. Our new template-driven approach now adds flexibility to the layout elements which are placed. Depending on design rules, ab-

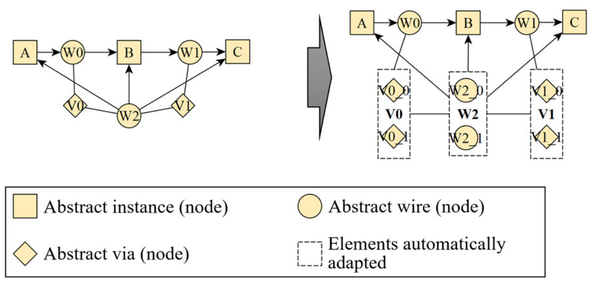


Figure 7 Principle of the dynamic adaptation of the template structure of which the related graphs are depicted (relations are depicted as lines). The basic abstract layout structure described by the template (left) is changed structurally due to design rules (right). Layout elements (here: wires and vias) are dynamically added or adapted (marked by dashed boxes).

stract layout elements can adapt themselves based on strategies globally available to all generators. This adaptation includes adding of layout elements and/or adaptation of dimensions.

Figure 7 shows an example where a maximum width rule constraint contradicts another constraint for the minimum amount of current (and thus minimum width) in wire *w2*. The abstract layout description constrains the horizontal metal to be on the particular layer defined. A technology-independent and globally available strategy was implemented into the abstract wire segment class which allows automatic wire slicing to fulfill both the constraint on wire width and the effective width to pass a current while the basic layout template is fulfilled, too. According to the structural change of the wire, each connected via defined by the underlying template reacts and will result in the creation of multiple vias.

As a summary, variability required in layout generation to improve technology independence has multiple appearances:

(1) Simple variability of dimensions with no further effect (e.g. size of a transistor),

(2) Variability of dimensions with further effect on other layout elements (e.g. size of a transistor or layout shapes triggering advanced design rules),

(3) Variability of structures affecting other elements, e.g. wire slicing, transistor rotation, obligatory dummy poly gates, discrete gate spacing, layout shape order due to coloring, or forbidden layout shapes. Such changes are directly triggered by rules. More complex rules such as antenna rules or density rules are not considered in this work, but we expect that related analyses and strategies could be implemented, too.

4 EXPERIMENTAL RESULTS

4.1 Generator Implementation

A simplified code example of a template-driven generator is given in **Figure 8**. First, all abstract layout elements are created (instances, wires, and vias) and then, relations between these abstract elements are set. Afterwards, the defined structure is checked for plausibility and finally the consolidated command order is executed (if no logical error is identified before).

```
# CREATE TEMPLATE CONTAINER
s = LayoutStructure("ArrayArrangement")

# CREATE instances, wires and vias based on (simplified) parameters
for instName in "ABCDEFGHI":
    s.add( AbstrInst(instData, width=i_width, height=i_height) )
for wireName, direction in ("w0", "ver"), ("w1", "ver"), ("w2", "hor"):
    # wire will be stretched later according to relations defined
    s.add( AbstrWire(wireName, width=w_width, height=w_height,
                     lay=w_lay, direction=direction) )
for viaName in "V0", "V1":
    s.add( AbstrVia(viaName, minCuts=no_cuts) )

# PLACE wire A, D, G, and w0
s.set("w0", "rightOf", ("A", "D", "G"))
s.set("D", "below", "A")
s.set("G", "below", "D")

# add layout RELATIONS with user-defined offset "par_overlapTop"
# even not yet placed wire w2 can be used as reference
s.defineRelation(editableValue="w0.upper", relation="=",
                 constraintValue="A.upper", deltaY=par_overlapTop)
s.defineRelation(editableValue="w0.lower", relation="=",
                 constraintValue="w2.lower")

[... further commands for B, C, E, F, H, I, w0, w1 ...]

# PLACE vias between wires (via layers are automatically evaluated)
s.setViaBetweenWires("V0", wireVer="w0", wireHor="w2")
s.setViaBetweenWires("V1", wireVer="w1", wireHor="w2")

#
# CHECK and ADAPT structure
s.checkPlausibility()

# EXECUTE procedural commands
s.executeGenerator()
```

Figure 8 Fragment of the simplified template-driven generator code. First, the structure is defined (template). Afterwards, the required commands are checked, adapted, scheduled, and finally run.

Depending on the technology, the resulting structure is adapted prior to execution. In the example given, the maximum width rule and the width requirement of the wires contradict each other. The implemented strategy of slicing the wires overcomes this problem (see graph in **Figure 9**). This strategy is not defined in the template but it is defined in the generator environment which handles the template. Therefore, the layout description is both short and robust and does not require implicit consideration of the technology. In case new design rule violations are found in new process technologies or in case new contradictions are found, further strategies can be added to the generator environment and no template must be updated.

4.2 Application to both 180 nm Bulk and 22 nm FD-SOI

The given example was run for both 180 nm and 22 nm in order to show the flexibility not only on the structural level of the graph representation. While for the first process technology fixed design rules are sufficient with no structural

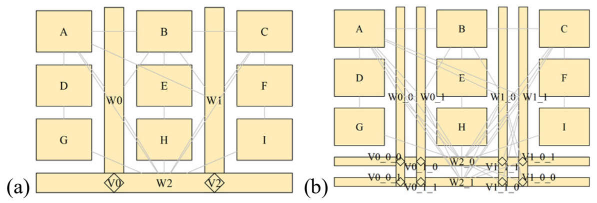


Figure 9 Graphs generated through the same template-driven generator description (with the shape of nodes being congruent to the elements). While (a) does not require structural adaptation due to design rules, (b) is adapted due to design rules constraining the maximum wire width; the basic layout structure is thereby retained.

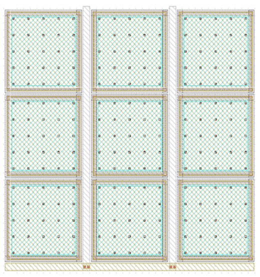
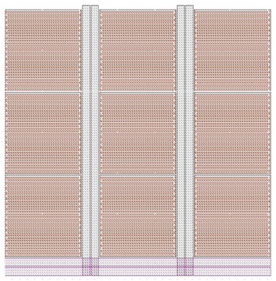
(a)  (b) 

Figure 10 Layout examples generated through the new template-driven generator approach. The layout description of both layouts is identical but results in different structures for (a) 180 nm and (b) 22 nm (via cuts are not visible due to their small size).

adaptation, the latter requires consideration of both context-dependent design rules and maximum width rules. The layout examples generated by the template-driven generator are depicted in **Figure 10**.

5 CONCLUSION AND OUTLOOK

Procedural generators with an implicit programming approach are hard to develop, they lack flexibility in case advanced design rules require structural layout adaptation, and their verification is time-consuming and incomplete. Our new template-driven approach improves development efficiency as well as layout correctness of generators. The explicit and non-procedural layout description eases generator development while logical errors of the generator description are automatically identified. Furthermore, since based on the template a graph representation is created as intermediate step, layout strategies such as wire slicing can automatically optimize details of the underlying template description. This way, we overcome time-consuming generator updates as no generator code must be changed and we ensure quality of the generated layout.

With our new approach we will accelerate generator development, especially for complex analog components. Next steps will concern further strategies of layout adaptation to improve technology independence. Furthermore, the new concept will be extended in order to allow interfacing between top-down automation and bottom-up generators.

ACKNOWLEDGEMENTS

The presented work was partly supported by the European Union and the Free State of Saxony within the project PRIME (Ref. No. 16ESE0110S).

REFERENCES

[1] N. Jangkrajarng, L. Zhang, S. Bhattacharya, N. Kohagen and C.-J. R. Shi, "Template-based parasitic-aware optimization and retargeting of analog and RF integrated circuit layouts," *IEEE/ACM Int. Conf. on Computer-Aided Design, 2006. ICCAD'06.,* pp. 342–348, Nov. 2006.

[2] N. Lourenço, R. Martins und N. Horta, „Layout-aware sizing of analog ICs using floorplan amp; routing estimates for parasitic extraction,“ *2015 Design, Automation Test in Europe Conf. Exhibition (DATE),* pp. 1156–1161, Mar. 2015.

[3] R. A. Rutenbar, "What's Up With Analog CAD…?," Okt 2016. [Online]. Available: http://rutenbar.cs.illinois.edu/wp-content/uploads/2017/01/rutenbar-tcace16.pdf. [Accessed Jul. 20, 2018].

[4] E. Yilmaz and G. Dündar, "Analog layout generator for CMOS circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* pp. 32–45, Jan. 2009.

[5] R. Martins, N. Lourenço and N. Horta, "LAYGEN II—automatic layout generation of analog integrated circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 32, no. 11, pp. 1641–1654, Nov. 2013.

[6] A. Unutulmaz, G. Dündar and F. V. Fernández, "LDS - a description script for layout templates," *2011 20th European Conf. on Circuit Theory and Design (ECCTD),* pp. 857–860, Aug. 2011.

[7] R. Castro-Lopez, O. Guerra, E. Roca and F. V. Fernandez, "An integrated layout-synthesis approach for analog ICs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, pp. 1179–1189, Jul. 2008.

[8] 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, pp. 237–262, Nov. 2003.

[9] R. Martins, A. Canelas, N. Lourenço and N. Horta, "On-the-fly exploration of placement templates for analog IC layout-aware sizing methodologies," *2016 13th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD),* pp. 1–4, Jun. 2016.

[10] H. Habal and H. Graeb, "Constraint-based layout-driven sizing of analog circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems,* vol. 30, no. 8, pp. 1089–1102, Aug. 2011.

[11] A. Unutulmaz, G. Dündar and F. V. Fernández, "A template router," *2011 20th European Conf. on Circuit Theory and Design (ECCTD),* pp. 334–337, Aug. 2011.

[12] A. Krinke, G. Jerke and J. Lienig, "Constraint propagation methods for robust IC design," *ZuE 2015; 8. GMM/ITG/GI-Symp. Reliability by Design,* pp. 1–8, Sep. 2015.

[13] A. Nassaj, J. Lienig and 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] G. Jerke und J. Lienig, „Constraint-driven design — the next step towards analog design automation,“ *Proc. of the Int. Symp. on Physical Design (ISPD'09),* pp. 75–82, Mar. 2009.

[15] J. Lienig, *Layoutsynthese elektronischer Schaltungen*, Springer, 2016.

[16] 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,* Mar. 2013.

[17] 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.

[18] A. Graupner, R. Jancke and R. Wittmann, "Generator based approach for analog circuit and layout design and optimization," *2011 Design, Automation & Test in Europe (DATE),* pp. 1–6, Mar. 2011.

[19] J. Crossley, et 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, Nov. 2013.

[20] 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," *13th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD 2016),* pp. 1–4, Jun. 2016.

[21] S. Youssef, D. Dupuis, R. Iskander and M. M. Louërat, "Automatic stress effects computation based on a layout generation tool for analog IC," *2010 IEEE Int. Behavioral Modeling and Simulation Workshop,* pp. 7–12, Sep. 2010.

[22] J. Scheible and J. Lienig, "Automation of analog IC layout – challenges and solutions," *Proc. of Int. Symp. on Physical Design (ISPD'15),* pp. 33–40, Mar. 2015.

[23] D. Marolt, J. Scheible, G. Jerke and V. Marolt, "Analog layout automation via self-organization: enhancing the novel SWARM approach," *2016 IEEE 7th Latin American Symp. on Circuits Systems (LASCAS),* pp. 55–58, Feb. 2016.

[24] Cadence, "Cadence PCell Designer For Cadence Virtuoso Users," [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/services/cadence-vcad-pcell-ds.pdf. [Accessed Jul. 20, 2018].

[25] S. Alassi and B. Winter, "Pycells for an open semiconductor industry," *Proc. of the 8th Eur. Conf. on Python in Science (EUROSCIPY 2015),* pp. 3–7, Jul. 2015.

[26] B. Prautsch, U. Eichler, T. Reich, A. Puppala and J. Lienig, "Abstract technology handling for generator-based analog circuit design," *GMM-Fachbericht 83, Reliability by Design (ZuE 2015), VDE Verlag,* pp. 56–61, Sep. 2015.

[27] North Carolina State University, "FreePDK - NCSU Electronic Design Automation (EDA) Wiki," [Online]. Available: https://www.eda.ncsu.edu/wiki/FreePDK. [Accessed Jul. 20, 2018].

[28] B. Prautsch, U. Eichler, T. Reich and J. Lienig, "MESH: explicit and flexible generation of analog arrays," *14 th Int. Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD),* pp. 1–4, Jun. 2017.

[29] Cadence, "A Call to Action: How 20nm Will Change," 2013. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.8961&rep=rep1&type=pdf. [Accessed Jul. 20, 2018].

[30] Synopsys, "Synopsys PyCell Studio," [Online]. Available: https://www.synopsys.com/cgi-bin/pycellstudio/req1.cgi. [Accessed Jul. 20, 2018].

[31] NetworkX, "NetworkX," NetworkX developers, 2018. [Online]. Available: https://networkx.github.io/. [Accessed Jul. 20, 2018].

[32] B. Prautsch, U. Eichler, T. Reich and J. Lienig, "Explicit feature and edge insertion for improved analog layout generators in advanced semiconductor technologies," *Analog 2016: Beiträge der 15. ITG/GMM-Fachtagung,* pp. 22–27, Sep. 2016.

[33] M. Greif, D. Marolt and J. Scheible, "gPCDS: An interactive tool for creating schematic module generators in analog IC design," *2016 12th Conf. on Ph.D. Research in Microelectronics and Electronics (PRIME),* pp. 1–4, Jun. 2016.