MPC

MULTI PROJEKT CHIP GRUPPE
BADEN-WÜRTTEMBERG

Herausgeber: Hochschule Ulm **Ausgabe:** 54 **ISSN** 1868-9221 **Workshop:** Ulm Juli 2015

Cooperating Organisation
Solid-State Circuit Society Chapter
IEEE German Section

Tagungsband zum Workshop der Multiprojekt-Chip-Gruppe Baden-Württemberg

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie.

Die Inhalte der einzelnen Beiträge dieses Tagungsbandes liegen in der Verantwortung der jeweiligen Autoren.

Herausgeber:

Gerhard Forster, Hochschule Ulm, Prittwitzstraße 10, D-89075 Ulm

Mitherausgeber (Peer Reviewer):

Jürgen Giehl, Hochschule Mannheim, Paul-Wittsack-Straße 10, D-68163 Mannheim

Frank Kesel, Hochschule Pforzheim, Tiefenbronner Straße 65, D-75175 Pforzheim

Axel Sikora, Hochschule Offenburg, Badstraße 24, D-77652 Offenburg

Diesen Workshopband und alle bisherigen Bände finden Sie im Internet unter:

http://www.mpc.belwue.de

Robert Bosch Zentrum für Leistungselektronik
Hochschule Reutlingen . Universität Stuttgart . Robert Bosch GmbH

CAPABLE: A Layout Automation Framework for Analog IC Design

Daniel Marolt, Jürgen Scheible, Göran Jerke, Vinko Marolt

***Abstract*—In practice, the use of layout PCells for analog IC design has not advanced beyond primitive devices and simple modules. This paper introduces a Constraint-Administered PCell-Applying Blocklevel Layout Engine (CAPABLE) which permits PCells to access their context, thus enabling a true "bottom-up" development of complex parameterized modules. These modules are integrated into the design flow with design constraints and applied by an execution cockpit via an automatically built layout script. The practical purpose of CAPABLE is to easily generate full-custom block layouts for given schematic circuits. Perspectively, our results inspire a whole new conception of PCells that can not only act (on demand), but also react (to environmental changes) and interact (with each other).**

***Index Terms*—Analog IC design, layout automation, parameterized cells, design constraints, bottom-up design.**

I. Introduction

Semiconductor products continue to revolutionize modern life in the 21st century, and analog IC content plays an essential role for the ongoing functional diversification of integrated circuits. Unfortunately, analog IC design still represents an economic bottleneck for the microelectronics industry. In particular, the design of physical layouts becomes more and more critical, facing the increasingly intricate challenges of advanced semiconductor technology nodes.

A. The Automation Gap in Analog Layout Design

In the digital domain, the layout creation task of integrated circuit design is highly automated. *Optimization algorithms* are successfully employed to place and route millions of devices per IC. In contrast, three decades of substantial research in Electronic Design Automation (EDA) have not yet achieved a large-scale adoption of such algorithmic approaches in the analog domain, where the number of devices is much smaller but the design requirements are significantly more complex. For that reason, optimization algorithms still struggle to find their way into industrial environments.

In practice, analog layouts are still handcrafted by expert designers in a time-consuming manual fashion today, with parameterized cells being the main source of automation. These so-called *PCells* represent layout *generators* and are fundamentally different from optimization-based approaches because they do not work in an algorithmic but in procedural way. PCells are mainly used to generate customizable layout variants of primitive devices such as transistors and resistors. From a scientific point of view, a PCell's automation abilities are comparably trivial, but for a design expert's daily layout work, PCells are indispensable. Thus it can be observed that – opposite to the ongoing pursuit of algorithmic solutions in academia – industrial flows rather drive the development of more powerful *module PCells* which are able to create layouts for entire circuits. In practice however, the advancement of PCells has not yet really proceeded beyond simple modules such as current mirrors and differential pairs.

The industrial reluctance to employ design automation for analog layout is rooted in several reasons. In particular, we consider the following three problems, which will be discussed in greater detail in Section III:

(A) The development and/or usage of automatisms is not intuitive: the setup of optimization algorithms is demanding due to their abstract nature; the usage of parameterized cells is more intuitive during design, but the programming of powerful hierarchical module PCells is quite challenging.

(B) The commonly available formal representations of design constraints are not suited to express complex functional circuit requirements with sufficient semantical conciseness.

(C) The inner workings of an automatism are usually not transparent to the user and give only little insight and control during its execution.

On the whole, all of these three problems add to one basic issue: analog layout automation does not adequately meet with the mentality of expert designers.

Daniel Marolt, daniel.marolt@reutlingen-university.de, Jürgen Scheible, juergen.scheible@reutlingen-university.de, Reutlingen University, Alteburgstraße 150, 72762 Reutlingen.

Göran Jerke, goeran.jerke@de.bosch.com, Vinko Marolt, vinko.marolt@de.bosch.com, Robert Bosch GmbH, Tübinger Straße 123, 72762 Reutlingen.

B. Our Contribution

This paper introduces a *Constraint-Administered PCell-Applying Blocklevel Layout Engine* for layout automation (CAPABLE). CAPABLE is a framework allowing layout engineers to easily combine various automatisms into dedicated, custom-made, cohesive, executable and traceable scripts which can be used to automatically create block layouts for given schematic circuits. While there are no restrictions concerning the nature of the includable automatisms (i.e., algorithmic or procedural), CAPABLE is primarily meant to realize a generator approach which focuses in particular on the application of PCells. With respect to this intention, CAPABLE decidedly targets the three particular problems mentioned above by implementing the following features, as will be covered in Section IV:

(A) CAPABLE facilitates a new style of hierarchical PCell composition to create higher-level modules. That way, the PCells can be successively imposed onto each other in a truly “bottom-up” fashion that is much closer to a layout engineer’s manual design style than the usual conception of module PCells which employ other sub-PCells internally.

(B) CAPABLE is integrated into the design flow via formally expressed constraints. For that purpose, new constraint types can be introduced to indicate the overall function of certain circuit structures. The inherent design requirements are then meant to be *implicitly* taken care of by respectively provided module PCells during the script execution.

(C) CAPABLE provides a convenient graphical user-interface (GUI) for the execution of the developed layout scripts. The GUI facilitates different pacing modes which allow the user to run a script step by step and thus to precisely track every single action that is thereby being performed in the layout.

Altogether, CAPABLE is a strongly designer-oriented engine that means to mimic a layout engineer’s manual design style as closely as possible. With this objective, the execution of a layout script in CAPABLE is supposed to give users the impression of replaying a “recorded session” of manual layout design.

Our paper is organized as follows: Section II discusses the characteristics of optimization algorithms and parameterized cells. Section III details the three limitations described in Section I.A, while Section IV illustrates the respective solutions (see above) as put into effect by CAPABLE. Section V demonstrates our approach with a practical example and finally, Section VI concludes with a summary and an outlook.

II. Related Work

A. Optimization Algorithms

Since layout design is – from a mathematical perspective – an optimization problem, optimization algorithms are a natural choice to address that problem for automation. Characteristically, optimization algorithms translate the problem into an abstract representation and cycle through a repetitive loop of optimization and evaluation [1] to find an optimal layout “solution” with respect to certain optimization goals. Due to the complexity of the layout problem, it is usually divided into several steps such that an optimization algorithm can focus on one specific design task. The two main tasks in layout design are placement and routing, both of which have put forth a vast variety of algorithmic approaches. Routing can be further split into two consecutive steps called global routing and detailed routing. While this has become common practice in the digital domain, the routing of analog circuits is rather performed in one single step called area routing. Two of the first developed area routers are Lee’s maze router [2] and Hightower’s line router [3]. Algorithmic placement is, due to the huge variability of the devices, enormously challenging in the analog domain, compared to the standard cell approach taken for digital systems. Popular placement algorithms are min-cut placement [4], force-directed placement [5] and the widely spread Simulated Annealing [6].

Around the 1990s, EDA research – inspired by the huge success of optimization algorithms in the digital domain – has led to a plethora of works in which algorithmic approaches were combined into full-fledged tools for automated analog layout synthesis at block level, such as ILAC [7], LADIES [8], ALSYN [9] and INALSYS [10]. However, none of suchlike tools is known to have found evident industrial acceptance.

Optimization algorithms have the characteristic ability to *explicitly* take design constraints into consideration, but to do so they require all these constraints to be comprehensively expressed in a *formal* way. Unfortunately, it is enormously difficult to express complex analog design requirements via formal expressions [11]. This restrains an incorporation of valuable expert knowledge into the automatism and represents a major weakness of algorithmic approaches.

B. Parameterized Cells

In contrast to optimization algorithms, PCells are not meant to self-intelligently *find* good layout solutions. Instead, a PCell executes a pre-defined series of operations in order to merely *reproduce* a customizable layout “result”. On this basis, a module PCell, designed to create a layout for a particular analog basic circuit, has the natural ability to *implicitly* consider all inherent design constraints without the need to formalize them. This characteristic trait allows PCells to encapsulate valuable expert knowledge in an *informal* fashion and to produce layouts in full-custom quality. PCells are especially feasible to implement layout modules for which best-practice layout solutions are already known from experience.

Powerful module PCells covering multiple hierarchy levels have already been presented, e.g. [12]. However, their development still implies a trade-off between module variability and programming effort which is far from viable for industrial demands. Over the past years, a couple of sophisticated commercial PCell programming tools such as 1Stone [13], PyCell Studio [14] and GOLF [15] have been developed. In particular, the PCell Designer tool [16] facilitates a visual programming approach resembling a layout expert's manual design style. The intention behind that approach is to provide an intuitive platform with which design groups can easily create their own appropriate automatisms. With such tools, simple module PCells like current mirrors and differential pairs have become state-of-the-art in the industry, but the development of more complex PCells is obviously still not profitable enough in terms of layout productivity.

III. Limitations Addressed By CAPABLE

In this section, each of the three limitations (A, B, C) briefly described in Section I.A is subsequently discussed in a subsection of its own.

A. *Hierarchical Module PCells*

As already stated, CAPABLE does not represent an algorithmic approach like the synthesis frameworks mentioned in Section II.A, but focuses on the application of PCells. For that reason, this subsection examines the common conception of PCells and the inherent difficulties when composing them into more complex hierarchical module PCells in the traditional way. Generally speaking, a module PCell is a PCell that internally instantiates other PCells, and then creates additional layout shapes according to its dedicated purpose. Concretely, the considerations in this paper focus on module PCells which instantiate a set of layout devices and connect them by creating wire shapes and vias which make up a module-specific routing. In Section V, other types of PCells will also be shown.

The usage of a PCell requires data to flow through various "channels" which are given by the design environment (see Fig. 1). In principal, each channel can provide write access to modify a design object, and read access to retrieve information from a design object. One such channel is the design editor itself, by which a user can instantiate and customize a module PCell in a design (a_1). Naturally, the design editor also allows for editing the context of such a PCell, i.e., the other objects around that PCell in its design (a_2). To generate a certain layout, a PCell has to be evaluated. During this evaluation, the PCell performs an internal series of operations, thereby accessing its internal device instances or routing objects in a programmatic way (b). For example, this can include measuring the distance between two instances or setting the width of a routing wire. In contrast to these *internal* operations,

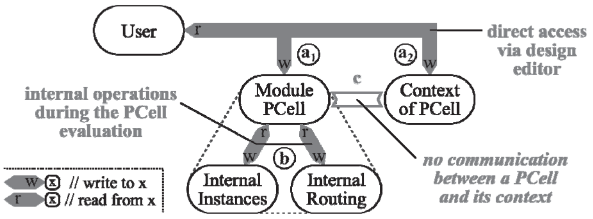

User
r
w
a_1
a_2
w
direct access via design editor
internal operations during the PCell evaluation
Module PCell
c
Context of PCell
r
r
w
b
w
Internal Instances
Internal Routing
no communication between a PCell and its context
w x // write to x
r x // read from x

Fig. 1: Data channels required during the usage of a module PCell.

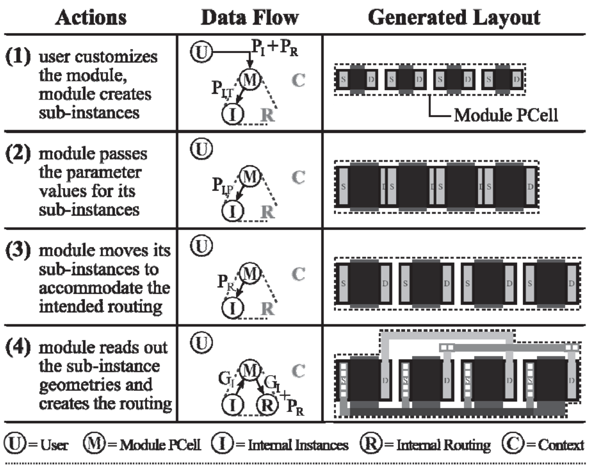

Actions
Data Flow
Generated Layout
(1) user customizes the module, module creates sub-instances
U
P_I+P_R
M
$P_{I,T}$
C
I
R
Module PCell
(2) module passes the parameter values for its sub-instances
$P_{I,P}$
(3) module moves its sub-instances to accommodate the intended routing
P_R
(4) module reads out the sub-instance geometries and creates the routing
G_I
G_I
P_R
U = User
M = Module PCell
I = Internal Instances
R = Internal Routing
C = Context

Fig. 2: Flow of data during the usage of a common module PCell.

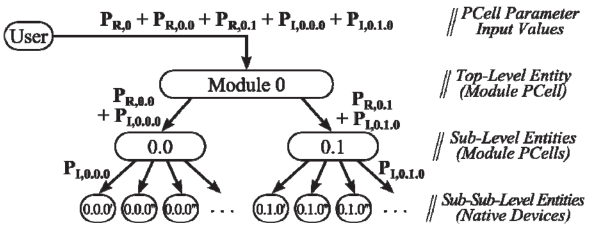

User
$P_{R,0}+P_{R,0.0}+P_{R,0.1}+P_{I,0.0.0}+P_{I,0.1.0}$
// PCell Parameter Input Values
Module 0
// Top-Level Entity (Module PCell)
$P_{R,0.0}+P_{I,0.0.0}$
$P_{R,0.1}+P_{I,0.1.0}$
0.0
0.1
// Sub-Level Entities (Module PCells)
$P_{I,0.0.0}$
$P_{I,0.1.0}$
0.0.0
0.0.0
0.0.0
0.1.0
0.1.0
0.1.0
// Sub-Sub-Level Entities (Native Devices)

Fig. 3: Passing of parameter values in a traditional module PCell.

a PCell – in its traditional conception – has no ability to access its *external* context (c). Hence, a module PCell cannot actively "communicate" with its surrounding design, which in turn implies that *all* data a PCell requires must be passed to it via PCell parameters (a_1).

Fig. 2 illustrates the usage and evaluation of a module PCell to examine the data flow in detail. First of all (step 1), the user U instantiates and customizes the module PCell M by setting its parameter values. The parameters of a module PCell, as focused on in this paper, can be logically divided into a parameter set P_I, which specifies the module's internal instances, and a parameter set P_R, which specifies its internal routing. P_I further consists of topology parameters $P_{I,T}$ which define the types and the number of the internal instances. With $P_{I,T}$ the module can initially create the internal instances I in their default configuration (1). P_I also contains the device parameters $P_{I,P}$ which are directly passed through to the internal instances to set their dimensions as desired (2). Now that the number, types, and sizes of the internal instances are set, they must be positioned in an arrangement that accommodates the intended routing. The arrangement is directly or indirectly defined by the routing parameters P_R, so this information must somehow be utilized to move each instance to its designated position (3). Then, the module can read-out geometrical data G_I about its instances (e.g., their pin positions) and use that information in conjunction with P_R to generate the appropriate routing R (4). During all these actions, the context C of the module PCell is never taken into account.

For simple modules, this common PCell conception is quite adequate, but it doesn't suit the traditional way of combining various PCells level by level to facilitate more complex hierarchical modules. As an example, two simple PCells can be implemented, each of which instantiates a couple of native transistors and connects them into a *current bank* or a *cascode*, respectively. A current bank and a cascode may be combined into a *cascode current mirror* PCell, which can then be employed for the realization of an *operational transconductance amplifier* (OTA). Such an OTA would then already span a total of four hierarchy levels.

At first glance, this approach is a natural strategy of using PCells as building bricks to form higher-level entities in a bottom-up fashion. But, since *every* parameter for *each* internal instance throughout the *entire* sub-hierarchy needs to be provided at the top-most module level, the flow of information in fact proceeds top-down. As shown in Fig. 3, all instance parameters P_I and routing parameters P_R for all internal entities must be given to the enclosing module by the user and are then internally distributed to the respective recipients. Technically, there is no limit to such a hierarchical composition of module PCells, but the cumulative amount of parameters at top level soon makes this approach virtually impractical. On one hand, the need to provide all parameters at module level escalates the development effort, and on the other hand, the unmanageable mass of parameters detracts from a PCell's usability. Furthermore, potential clashes of parameter names may require a cumbrous renaming scheme which in turn opposes the execution of device-specific validation mechanisms which are required to check and – if necessary – correct a user-entered parameter value. Even if all these problems can somehow be circumvented, the long-term maintenance of such a module PCell remains a critical issue. If, for example, a native transistor is equipped with a new parameter, then that parameter specification also needs to be added to the module PCell and to all its sub-modules throughout the module's entire hierarchy. Altogether, these drawbacks make the development of complex module PCells laborious, error-prone and inflexible.

B. Formulation of Design Constraints

From an abstract perspective, a design constraint is a piece of information that supplements a schematic circuit to help attain an electrically functioning layout design. Initially, design constraints are informal pieces of expert knowledge in a designer's perception, but to be explicitly considered by an automatism, they have to be expressed in a formalized way.

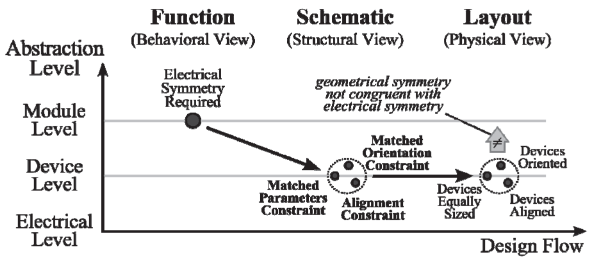


Fig. 4: Example of typical constraint usage in the design flow.

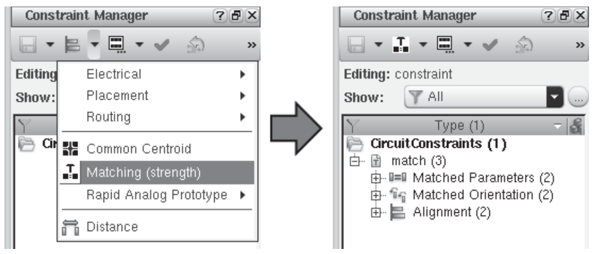


Fig. 5: In Virtuoso, matching is expressed with three constraints.

Architecturally, circuit designers tend to think not of individual transistors, but of larger circuit structures that constitute functional units, keeping in mind the essential layout requirements that must be satisfied to ensure their proper electrical functioning. One of the most fundamental duties in analog layout design is the achievement of *matching*. Matching denotes a symmetric placement of belonging-together layout devices to ensure functional robustness against process variations, parasitic effects and physical influences.

Equivalent to the depictions in [17], Fig. 4 shows the three views of the Y diagram [18] side by side and the different abstraction levels as parallel lines. From a functional perspective (a), a circuit designer may be well aware of the need for electrical symmetry concerning a certain circuit structure. For the schematic design (b), that matching requirement may be formalized, but usually this is done by resorting to an alternative set of more concrete constraints. For example, in the design environment Cadence Virtuoso, a *Matching* constraint is indeed available for assignment (Fig. 5, left), but eventually it produces a set of three different, geometrical constraints (Fig. 5, right): *Matched Parameters*, *Matched Orientation*, and *Alignment*. In the layout design (Fig. 4), compliance with these constraints is supposed to achieve the desired matching, but that goal is not necessarily achieved, because the geometrical symmetry imposed by the three concrete, formal constraints is not entirely congruent with the electrical symmetry denoted by matching.

On one hand, the three constraints are missing two further matching criteria: the distance between the devices and their interdigitation. Large spaces between the devices or an unbalanced interdigitation pattern can substantially compromise the overall matching even if the other constraints are satisfied. On the other hand, the static nature of the constraints is insufficient to express geometric variability. In particular, *Alignment* requires that all devices share a common edge. For a single-row layout of a current mirror, this constraint would be satisfied. However, the devices of a current mirror can just as well be placed in two rows. Such an arrangement can even improve the matching, but is not tolerated by the alignment constraint because the alignment edge of the top transistors is different from the alignment edge of the bottom transistors. Furthermore, the routing of the devices can also severely impair the matching, but is not at all covered by the formally expressed design constraints above.

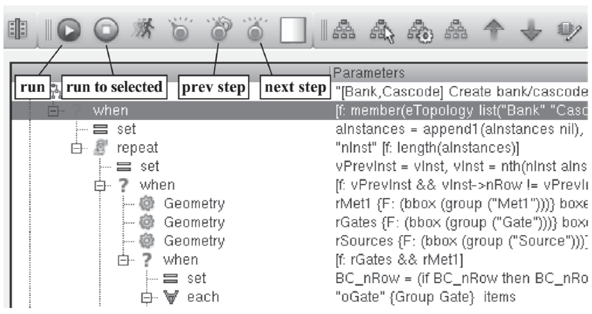
The advancement of *constraint engineering* in academia and in practice is promising, but it is still in its infancy, as shown by the above example. The choice of translating abstract design requirements (such as matching) into more concrete geometrical constraints is comprehensible because it allows for a formal verification of constraints and also makes them amenable to design automation. But, despite the ongoing developments on the consideration of constraints for verification and automation, the described shortcomings of contemporary constraint formulation represent a central limitation that still obstructs the evolvement of long-envisioned constraint-driven design flows.

C. *Opacity of Layout Automatisms*

The potential acceptance of analog layout automation concepts not only depends on their mere technical merit, but also on a human factor that should not be underestimated. In general, engineers are rather skeptical about processes they cannot easily retrace or influence. Unfortunately, this is the case with many layout automation approaches, taking a set of input values and producing a respective layout output with only little means (or none at all) to let a user follow or steer the course of the automatism's execution.

For algorithmic approaches, tracking an automatism is difficult anyway since an algorithm may perform millions of cycles of optimization and evaluation (e.g., the random perturbations in the placement algorithm Simulated Annealing). The stochastic nature and non-deterministic behavior found in the majority of algorithmic layout automatisms detach an understanding of their actions even farther from the human grasp.

In the digital domain, these problems are of no concern due to the more quantitative quality of the design problem. But in analog design, the success of algorithmic automation is bound to a comprehensive and precise description of all relevant design constraints, which is in turn enormously intricate if the relevance



```
                                    Parameters
                                    "[Bank,Cascode] Create bank/cascode
⊟ when                              [f: member(eTopology list("Bank" "Casc
   set                              aInstances = append1(aInstances nil),
  ⊟ repeat                          "nInst" [f: length(aInstances)]
     set                            vPrevInst = vInst, vInst = nth(nInst aIns
    ⊟ ? when                        [f: vPrevInst && vInst->nRow != vPrevIi
       Geometry                     rMet1 {F: (bbox (group ("Met1")))} boxe
       Geometry                     rGates {F: (bbox (group ("Gate")))} box
       Geometry                     rSources {F: (bbox (group ("Source")))
      ⊟ ? when                      [f: rGates && rMet1]
         set                        BC_nRow = (if BC_nRow then BC_nRo
        ⊟ each                      "oGate" {Group Gate}  items
```

Fig. 6: Execution modes in the Cadence PCell Designer tool.

of each individual constraint, as well as their aggregate effect on the overall course of the automatism cannot be easily and intuitively understood by the designer.

In contrast, the inner workings of a PCell are easier to comprehend than those of an algorithm because PCells implement a pre-determined series of layout operations. On that basis, many PCell development tools offer different PCell execution modes which allow PCell programmers to trace the evaluation of a PCell in detail. For example, the PCell Designer tool displays a PCell's series of operations as a tree and supports running the PCell up to a selected operation call, as well as a step-by-step execution (see Fig. 6).

During PCell development, stepwise PCell evaluations are helpful features, but unfortunately, they are not available for the actual PCell usage. During design, a PCell is always executed in one single stroke and immediately reflects any user-made modifications of parameter values. For simple device PCells, this is quite adequate, but for rather complex modules more detailed control over the evaluation of a PCell would be immensely helpful to understand the structure of the module and the influence of the individual PCell parameters on the finally generated layout result.

IV. The CAPABLE Approach

As will be individually discussed in each of the following subsections, CAPABLE specifically targets the three limitations (A, B, C) previously detailed in Section III. The combination of these efforts leads to a practical automation flow as depicted in Fig. 7.

Primarily, CAPABLE facilitates an *interface fabric* (A), with which design teams can implement *context-enhanced PCells*. Context-enhanced PCells are PCells that get equipped by the interface fabric with the ability to read and modify their design context. This allows module PCells to be hierarchically imposed onto each other in an intuitive, bottom-up fashion.

To generate the layout for a certain schematic circuit, CAPABLE provides a *constraint interpreter* (B) which allows designers to map a sequence of PCell-applying CAPABLE script commands to a particular constraint type. When assigning these constraints to components of a schematic circuit, the interpreter is

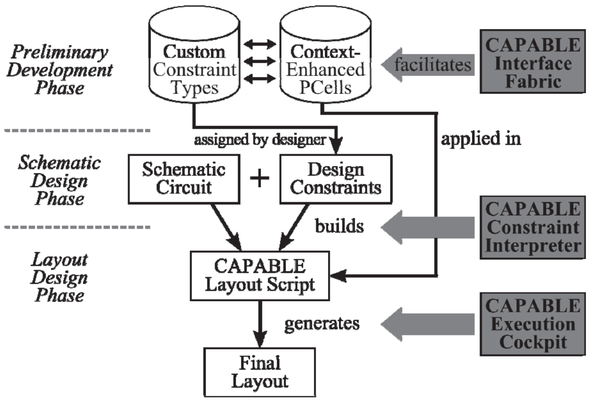


Fig. 7: Overview of the CAPABLE layout automation flow.

used to build the layout script from the respective script commands by which corresponding context-enhanced PCells are then automatically imposed on the constrained components in the layout. If desired, the layout script can also be manually edited and extended to perform further layout actions.

For carrying out a layout script, CAPABLE features a dedicated graphical *execution cockpit* (C). The execution cockpit is responsible for converting the used script commands into low-level code of the design environment's native programming language. To run the created code, CAPABLE's execution cockpit provides various different pacing modes which facilitate (amongst others) a stepwise execution of the script.

A. Interface Fabric

Reprising the illustration from Section III.A, Fig. 8 again shows the common data channels employed for the usage of PCells: the direct editing of a PCell (a_1) and its design context (a_2) as well as a PCell's access to its internal design entities (b). The previously missing communication abilities between a PCell and its context are now facilitated with CAPABLE's interface fabric (c), that can be considered as being "wrapped" around a PCell during its instantiation. This enables read and write access to the context and can be utilized by PCell developers via providing three dedicated context-related functions per PCell:

- An *adapt* function (mandatory) is required prior to the PCell instantiation to analyze the designated PCell context and turn it into parameter values that can then be passed by CAPABLE to the PCell when instantiating it. This allows a PCell to dynamically adapt itself to the design it is placed in.
- A *modify* function (optional) can be implemented in order to let a PCell alter its context after the initial PCell instantiation. This allows a PCell to modify its surroundings to make them suitable for the PCell's own design requirements.
- An *update* function (optional) is only necessary for PCells that modify their context, and can be used to trigger a re-evaluation of the PCell. This allows a PCell to update itself according to the previously modified context.

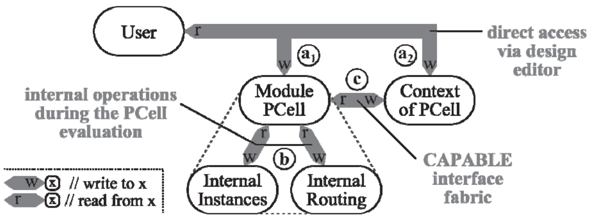


Fig. 8: Data channel enhancement via CAPABLE interface fabric.

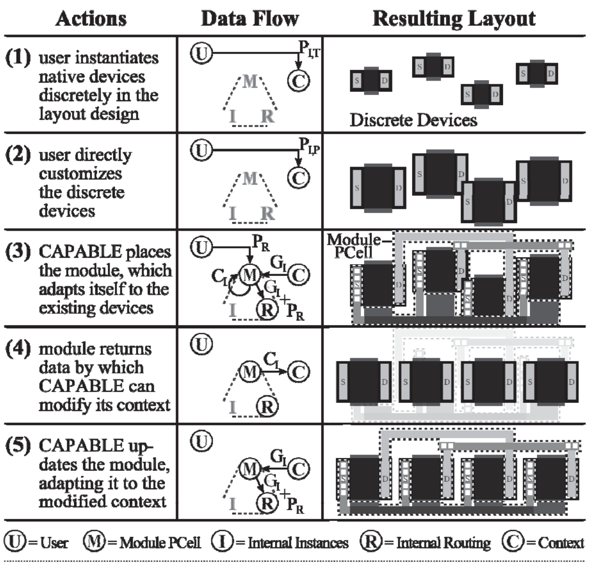

Actions
Data Flow
Resulting Layout
(1) user instantiates native devices discretely in the layout design
Discrete Devices
(2) user directly customizes the discrete devices
(3) CAPABLE places the module, which adapts itself to the existing devices
Module-PCell
(4) module returns data by which CAPABLE can modify its context
(5) CAPABLE updates the module, adapting it to the modified context
U = User
M = Module PCell
I = Internal Instances
R = Internal Routing
C = Context

Fig. 9: Flow of data during the usage of a context-enhanced PCell.

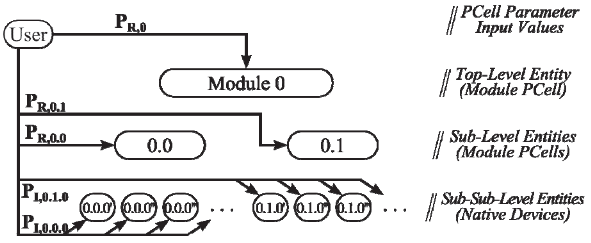

User
$P_{R,0}$
Module 0
$P_{R,0.1}$
$P_{R,0.0}$
0.0
0.1
$P_{I,0.1.0}$
$P_{I,0.0.0}$
// PCell Parameter Input Values
// Top-Level Entity (Module PCell)
// Sub-Level Entities (Module PCells)
// Sub-Sub-Level Entities (Native Devices)

Fig. 10: Passing of parameter values with context-enhanced PCells.

With the above functions, CAPABLE can feasibly split a PCell instantiation into three consecutive steps: adapt, modify, update. After the execution of these three steps, the PCell and its context are supposed to be in perfect conformance with each other.

Fig. 9 illustrates the use of CAPABLE's interface fabric with a practical module PCell equivalent to the example presented in Section III.A. In contrast to the traditional PCell approach, the native devices need not be placed internally by the module PCell, but can be discretely instantiated by the user in the layout design (step 1). This has the benefit, that the devices can also be *directly* customized to set their dimensions (2). With the native devices prepared, CAPABLE can be used to impose a context-enhanced module PCell on them (3): here, the *adapt* function reads out the pin positions of the discrete devices and passes this geometrical data G_I to the PCell via pre-defined parameters, so the PCell can generate the desired routing considering the routing parameters P_R. During this PCell evaluation, the PCell internally computes the correction data C_I, which – in this example – specifies a translational move for each individual device. The correction data is stored on the PCell itself and can thus be used after the initial PCell instantiation by the *modify* function to displace all devices such that they are in line with the PCell's intended best-practice arrangement (4). Finally, CAPABLE executes the *update* function so the module PCell adjusts its routing to the newly positioned devices (5). In the end, the resulting layout is equal to that of the traditionally implemented module PCell shown in Fig. 2.

The benefit of this approach becomes obvious when more complex modules are considered. As illustrated in Fig. 10, the interface fabric of CAPABLE eliminates the need to pass all parameter values for all of a module's internal instances throughout the entire subhierarchy of the module. Instead, the instance parameters P_I can be directly set on the native devices onto which the module (or, in this case: its sub-modules) are imposed. Equivalently, the routing parameters P_R for the module and its sub-modules are also directly customizable. In that way, each building brick of the overall module PCell can be individually customized *before* it is measured by an adapting PCell on the next-higher hierarchy level. This is not possible with the traditional PCell approach, where the internal instances are really nested inside other modules: it implies that an internal instance is entirely evaluated during the evaluation of its enclosing PCell, so the internal instance cannot be autonomously customized *before* its enclosing PCell begins its own evaluation.

B. Constraint Interpreter

The application of the context-enhanced module PCells is facilitated via design constraints. But, instead of resorting to low-level geometric constraints as described in Section III.B, CAPABLE employs dedicated higher-level constraints. These are supposed to be assigned to common circuit structures which represent functional entities and for which adequate module PCells are available on the layout side. Thereby it is the responsibility of the design team – depending on their focus (e.g., automotive applications) – to implement the desired module PCells, and also to specify the respective custom constraint types for a seamless integration into the CAPABLE flow. Of course, this presumes that the IC design environment allows for the specification of custom constraint types.

For every custom constraint type, a sequence of CAPABLE commands must once-only be declared in the constraint interpreter. Then, during the application of CAPABLE to a specific schematic circuit, the constraint interpreter turns every occurrence of that constraint type into a concrete call to that command sequence for applying it to the actual design entities the constraint has been assigned to. The final layout script results from the concatenation of the command sequences for all constraints encountered in the schematic design. The layout script can further be manually edited and extended, if desired.

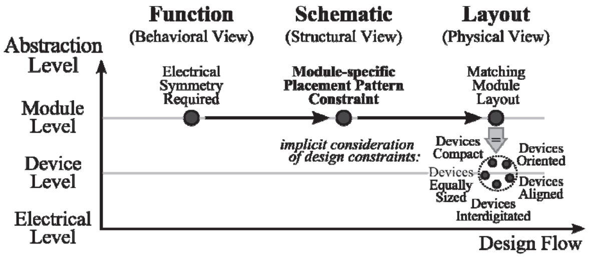


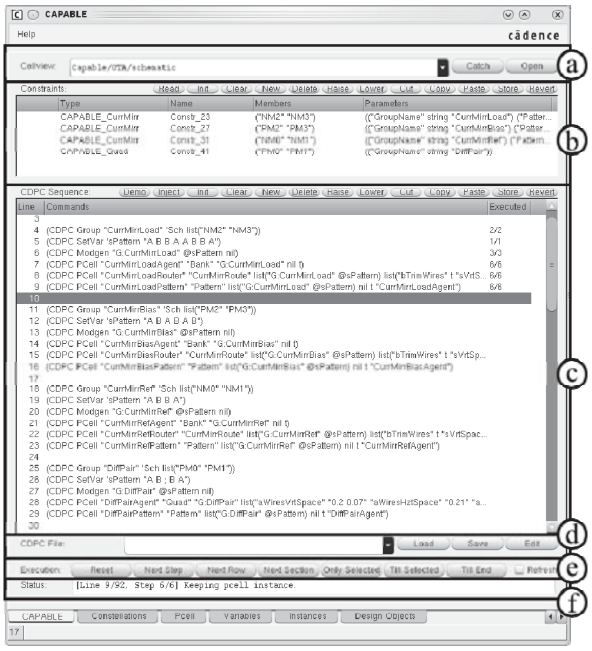
Fig. 11: Example of convenient constraint usage with CAPABLE.

CAPABLE provides a couple of script commands. The most important one of these is a *PCell* command that performs the instantiation of a context-enhanced module PCell according to the three-step approach described in Section A. Another command named *Group* allows to store a collection of design entities in a local variable. This is particularly useful if multiple PCells are to be imposed on the same set of devices. Further commands provided by CAPABLE are beyond the scope of this paper. Anyway, the set of provided commands is rather small, since the substantial automation powers are supposed to be covered by the PCells themselves, or other automatisms that may also be integrated into the overall CAPABLE framework (e.g., the *Modgen* tool, as will be shown in Section V).

As explained in Section II, PCells have the ability to consider intricate, low-level design constraints implicitly. Under that provision, the utilization of constraints is much more concise than in flows such as the one depicted in Section III.B. Fig. 11 shows how – instead of resorting to primitive geometric constraints – the need for electrical symmetry of a particular circuit structure can be simply expressed with a single custom constraint type in the schematic. That constraint should allow for the direct specification of a placement pattern by which the detailed device interdigitation can be explicitly defined. In the physical domain, a dedicated module PCell, specifically designed to respect the custom interdigitation pattern and all other requirements of the respective circuit structure, produces a module layout which achieves the desired matching (c). Thereby, the inherent low-level constraints are implicitly satisfied by the module PCell without the need to explicitly formulate them.

C. Execution Cockpit

The CAPABLE approach has been implemented for the Cadence Virtuoso design environment. The



Cellview: `Capable/OTA/schematic`

Type	Name	Members	Parameters
CAPABLE_CurrMirr	Constr_23	("NM2" "NM3")	(("GroupName" string "CurrMirrLoad") ("Patter...
CAPABLE_CurrMirr	Constr_27	("PM2" "PM3")	(("GroupName" string "CurrMirrBias") ("Patter...
CAPABLE_CurrMirr	Constr_31	("NM0" "NM1")	(("GroupName" string "CurrMirrRef") ("Pattern...
CAPABLE_Quad	Constr_41	("PM0" "PM1")	(("GroupName" string "DiffPair"))

Line	Commands	Executed
3		
4	(CDPC Group "CurrMirrLoad" 'Sch list("NM2" "NM3"))	2/2
5	(CDPC SetVar 'sPattern "A B B A A B B A")	1/1
6	(CDPC Modgen "G:CurrMirrLoad" @sPattern nil)	3/3
7	(CDPC PCell "CurrMirrLoadAgent" "Bank" "G:CurrMirrLoad" nil t)	6/6
8	(CDPC PCell "CurrMirrLoadRouter" "CurrMirrRoute" list("G:CurrMirrLoad" @sPattern) list("bTrimWires" t "sVrtS...	6/6
9	(CDPC PCell "CurrMirrLoadPattern" "Pattern" list("G:CurrMirrLoad" @sPattern) nil t "CurrMirrLoadAgent")	6/6
10		
11	(CDPC Group "CurrMirrBias" 'Sch list("PM2" "PM3"))	
12	(CDPC SetVar 'sPattern "A B A B A B")	
13	(CDPC Modgen "G:CurrMirrBias" @sPattern nil)	
14	(CDPC PCell "CurrMirrBiasAgent" "Bank" "G:CurrMirrBias" nil t)	
15	(CDPC PCell "CurrMirrBiasRouter" "CurrMirrRoute" list("G:CurrMirrBias" @sPattern) list("bTrimWires" t "sVrtSp...	
16	(CDPC PCell "CurrMirrBiasPattern" "Pattern" list("G:CurrMirrBias" @sPattern) nil t "CurrMirrBiasAgent")	
17		
18	(CDPC Group "CurrMirrRef" 'Sch list("NM0" "NM1"))	
19	(CDPC SetVar 'sPattern "A B B A")	
20	(CDPC Modgen "G:CurrMirrRef" @sPattern nil)	
21	(CDPC PCell "CurrMirrRefAgent" "Bank" "G:CurrMirrRef" nil t)	
22	(CDPC PCell "CurrMirrRefRouter" "CurrMirrRoute" list("G:CurrMirrRef" @sPattern) list("bTrimWires" t "sVrtSpac...	
23	(CDPC PCell "CurrMirrRefPattern" "Pattern" list("G:CurrMirrRef" @sPattern) nil t "CurrMirrRefAgent")	
24		
25	(CDPC Group "DiffPair" 'Sch list("PM0" "PM1"))	
26	(CDPC SetVar 'sPattern "A B ; B A")	
27	(CDPC Modgen "G:DiffPair" @sPattern nil)	
28	(CDPC PCell "DiffPairAgent" "Quad" "G:DiffPair" list("aWiresVrtSpace" "0.2 0.07" "aWiresHztSpace" "0.21" "a...	
29	(CDPC PCell "DiffPairPattern" "Pattern" list("G:DiffPair" @sPattern) nil t "DiffPairAgent")	
30		

Status: `[Line 9/92, Step 6/6] Keeping pcell instance.`

Fig. 12: The execution cockpit – CAPABLE's user-interface.

execution cockpit (CAPABLE's graphical user-interface) is shown in Fig. 12. With the input field (a), the user can enter or select a schematic circuit name. The circuit's design constraints are then read from the schematic and listed in table (b). The constraint interpreter builds the command sequences for the read constraints and concatenates them into the layout script which is then displayed in table (c). Using the elements of (d), layout scripts can be saved as text files and loaded from the file system at a later time. Also, a layout script text file can be manually edited with a plain text editor.

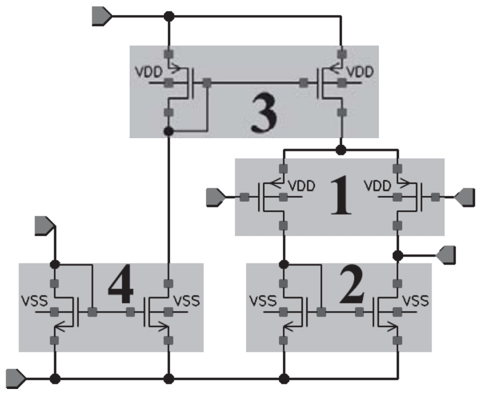
Each line of the layout script represents one call to a CAPABLE command. As already mentioned, the execution cockpit has to convert each command call into basic code of the design environment's native programming language (in our implementation: SKILL). Based on this conversion, the execution of a command is split into multiple fine-grain steps to let the user precisely track every single action that is being performed in the layout. The buttons (e) give control over the execution and allow the user to carry out

- a single step,
- one command,
- a section of commands,
- all commands up to the selected command,
- all commands till the end of the script is reached.

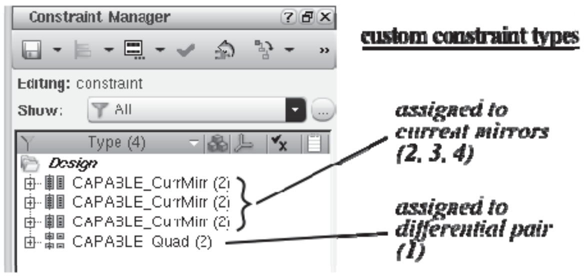
For every single step, the status field (f) displays textual messages that are generated during the execution.

V. Example And Results

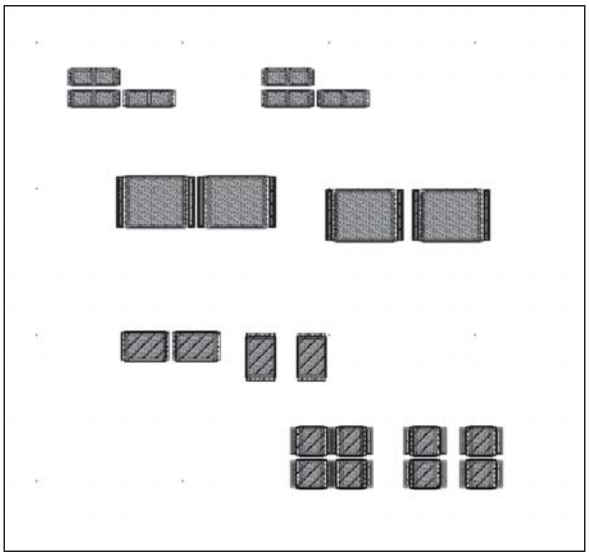
As an example for the application of the CAPABLE approach, Fig. 13 (a) shows the schematic diagram of a p-input OTA. The circuit consists of one differential pair (1) and three current mirrors (2, 3, 4). As displayed in subfigure (b), a custom *CurrMirr* constraint is assigned to each current mirror, and a custom *Quad* constraint is assigned to the differential pair. The CurrMirr constraints allow the designer to define specific interdigitation patterns, whereas the Quad con-



(a) Schematic diagram of the p-input OTA circuit example.



(b) Custom types of constraints, as assigned to the OTA circuit.



(c) Initially generated layout instances for the circuit transistors.

Fig. 13: An OTA circuit to exemplify the application of CAPABLE.

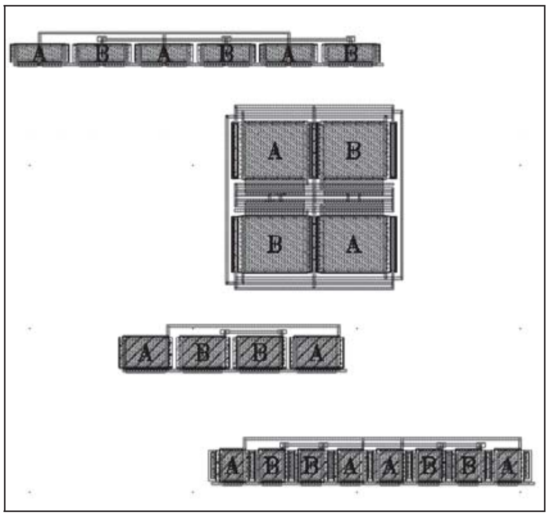
straint inherently denotes a common-centroid AB/BA placement. For the layout creation, CAPABLE initially instantiates the circuit transistors in the layout by calling *Generate from Source*. This is a native schematic-driven-layout functionality by which the dimensions of the layout devices are directly taken from the schematic circuit. The initial constellation is shown in subfigure (c) and represents the starting point for the subsequent constraint-administered imposition of context-enhanced module PCells (illustrated in Fig. 14).

First, the layout devices for each of the four circuit structures are interdigitated by CAPABLE. This is conveniently done using the native *Modgen* tool (but could have also been achieved with a PCell). Then, dedicated *quad* and *current mirror PCells* perform the detailed placement and module routing, which leads to the intermediate layout result shown in Fig. 14 (a).

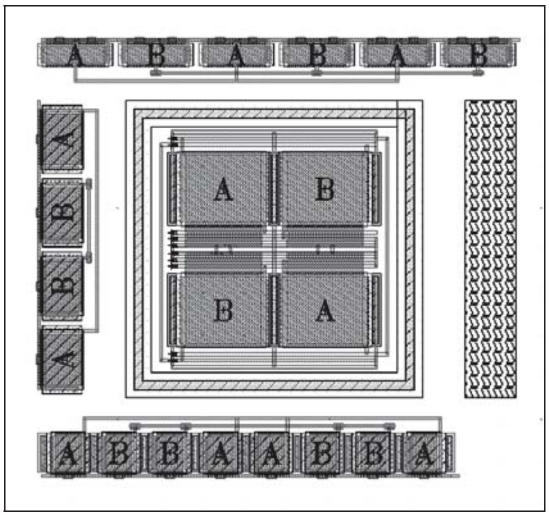
The layout script was manually extended to account for the further tasks of the layout creation. Thus, the differential pair is enveloped in a guard ring as created by a context-enhanced *isolation PCell* which smoothly clasps itself around the devices of the quad. Afterwards, a *placement PCell* moves the OTA modules to a feasible arrangement according to a pre-defined layout template. As shown in subfigure (b), the PCell also generates a blocking cap for reasons of symmetry and leaves sufficient space for the subsequent routing.

The routing between the modules is performed with a *wire PCell*, realizing the creation of multi-segment routing paths. While the paths can be explicitly specified via point lists, the PCell is (thanks to context-enhancement) also capable of snapping the routing wire to existing device terminals. The PCell furthermore supports transitions of the metal layer and automatically creates the required vias at the respective transition locations. In the given example, all inter-module connections are achieved with the above wire PCell, and finally another isolation PCell is put around the entire OTA. The resulting layout is presented in subfigure (c).

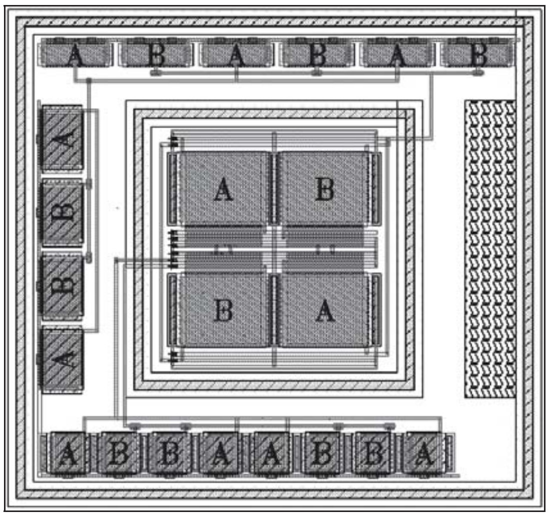
One might be skeptical about the efficiency benefit of CAPABLE's script approach compared to the mere manual creation of an analog block layout. However, a particular bottleneck are design *iterations* where even small modifications to the schematic circuit require laborious adjustments in the layout. In that regard, CAPABLE is supposed to be an especially profitable concept, allowing to easily re-generate the layout with just as little modifications. In general terms, every finished layout design represents one individual determined *solution*, whereas a CAPABLE script rather encapsulates a solution *strategy* that can be executed to generate finalized solutions. Conceptually, this approach is equivalent to the idea of PCells, but on a higher level of abstraction: a PCell performs native layout operations to create a design, while CAPABLE employs PCells to produce more complex results.



(a) Intermediate layout result with quad and current mirror modules.



(b) Intermediate placement with quad guard ring and blocking cap.



(c) Final layout result with full routing and enclosing guard ring.

Fig. 14: Layout results of applying CAPABLE to the OTA circuit.

VI. Summary And Outlook

This paper presents a Constraint-Administered PCell-Applying Blocklevel Layout Engine for analog layout automation (CAPABLE). It implements three central features: (A) an interface fabric which facilitates the development of context-enhanced module PCells, (B) a constraint interpreter for transforming custom design constraints into command sequences that apply dedicated context-enhanced module PCells, and (C) an execution cockpit to carry out such command sequences as a concatenated layout script.

By implementing appropriate module PCells and assigning respective constraints to a schematic circuit, designers can use CAPABLE to build and run a layout script that generates the block layout in a bottom-up fashion by successively imposing module PCells onto each other. This approach is closer to a layout expert’s manual design style and more intuitive than the traditional conception of complex module PCells covering multiple levels of hierarchy. In particular, CAPABLE eliminates the need to pass low-level device parameters across multiple hierarchy levels, it is able to consider intricate design requirements implicitly, and it allows the user to transparently track every single step that is performed during the layout creation.

In the long run, the CAPABLE approach should encourage design experts to abandon the common, manual style of layout creation in favor of rather capturing their invaluable solution strategies. For that purpose, future work on CAPABLE targets the automation of the script creation, e.g. by recording manual layout actions and converting them into script commands. Apart from that, the idea of context-awareness gives rise to an entirely new species of PCells that can not only *act* (on demand), but also *react* (to environmental changes) and even *interact* (with each other) – a vision that adds fundamentally new conceptions of flexibility and intelligence to the classical PCell concept.

Acknowledgement

We would like to thank Andreas Gerlach for providing the OTA example circuit, as well as Thomas Burdick and Peter Herth for their support on PCells.

References

[1] R. Rutenbar, “Analog CAD: Not done yet,” *presented at NSF Workshop*, Arlington, Virginia, Jul. 8-9, 2009.

[2] C. Lee, “An Algorithm for Path Connections and Its Applications,” *IRE Trans. on Electr. Comput.*, vol. EC-10, issue 3, pp. 346-365, 1961, doi:10.1109/TEC.1961.5219222.

[3] D. Hightower, “A Solution to Line-Routing Problems on the Continuous Plane,” *Proc. 6th Design Automation Conference*, pp. 1-24, 1969, doi:10.1145/800260.809014.

[4] M. Breuer, “A Class of Min-Cut Placement Algorithms,” *Proc. 14th Design Automation Conference*, pp. 284-290, 1977.

[5] N. Quinn Jr., “The Placement Problem as Viewed from the Physics of Classical Mechanics,” *Proc. 12th Design Automation Conference*, pp. 173-178, 1975.

[6] S. Kirkpatrick, C. Gelatt Jr. and M. Vecchi, “Optimization by Simulated Annealing,” *Science*, vol. 220, issue 4598, pp. 671-680, May 1983, doi:10.1126/science.220.4598.671.

[7] J. Rijmenants, J. Litsios, T. Schwarz and M. Degrauwe, “ILAC: An Automated Layout Tool for Analog CMOS Circuits,” *IEEE Journal of Solid-State Circuits*, vol. 24, issue 2, pp. 417-425, Apr. 1989, doi:10.1109/4.18603.

[8] M. Mogaki, N. Kato, Y. Chikami, N. Yamada and Y. Kobayashi, “LADIES: An Automatic Layout System for Analog LSI's,” *Int. Conference on Computer-Aided Design*, pp. 450-453, Nov. 1989, doi:10.1109/ICCAD.1989.76989.

[9] V. M. zu Bexten, C. Moraga, R. Klinke, W. Brockherde and K. Hess, “ALSYN: Flexible Rule-based Layout Synthesis for Analog IC's,” *IEEE Journal of Solid-State Circuits*, vol. 28, issue 3, pp. 261-268, Mar. 1993, doi:10.1109/4.209992.

[10] Y. Kim, H. Cho and K. Yoon, “INALSYS: A Layout Automation System Based on Analog Layout Constraints,” *Proc. of the 40th Midwest Sympos. on Circuits and Systems*, vol. 2, pp. 1209-1212, Aug. 1997, doi:10.1109/MWSCAS.1997.662297.

[11] J. Scheible and J. Lienig, “Automation of Analog IC Layout – Challenges and Solutions,” *Proc. of the ACM 2015 Int. Symposium on Physical Design (ISPD'15)*, pp. 33–40, Mar. 2015, doi:10.1145/2717764.2717781.

[12] T. Reich, U. Eichler, K. 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*, vol. 239, Mar. 2013, ISBN:978-3-8007-3467-2.

[13] IPGen 1Stone Developer, [online] *http://ipgenme.de/eda-and-ip-products/1stone-developer.html* (accessed 2015-06-23).

[14] Synopsys PyCell Studio, [online] *http://www.synopsys.com/cgi-bin/pycellstudio/req1.cgi* (accessed 2015-06-23).

[15] Anaglobe GOLF, [onl.] *http://www.anaglobe.com/web/wp-content/ uploads/2014/03/PCell_brochure2010.pdf* (accessed 2015-06-23).

[16] G. Jerke, T. Burdick, P. Herth, V. Marolt, C. Bürzele *et al.*, “Hierarchical Module Design with Cadence PCell Designer,” *pres. at CDNLive! EMEA*, Munich, Apr. 2015, session CUS02.

[17] S. Gohm, D. Marolt and J. Scheible, “Parametrisierte Lay-out-Module im analogen IC-Entwurf” (transl. “Parameterized Lay-out Modules in analog IC Design”), *MPC-Workshop*, vol. 48, pp. 57-63, Jul. 2012, ISSN:1868-9221.

[18] D. Gajski and R. Kuhn, “Guest Editor’s Introduction: New VLSI Tools,” *IEEE Computer*, Dec. 1983.

Daniel Marolt studied mechatronics at Reutlingen University, where he received the B.Eng. degree in 2008 and the M.Sc. degree in 2009. Since 2009 he works as an academic employee at Reutlingen University, where he pursues his Ph.D. degree in electrical engineering at the Robert Bosch Center for Power Electronics since 2011. His research interests focus on PCell-based automation of full-custom analog circuit and layout design.

Jürgen Scheible got his diploma in 1987 and the Ph.D. (Dr.-Ing.) degree (both in electrical engineering) in 1991 both from the University of Karlsruhe. From 1992 on, he was with the automotive electronics division of Robert Bosch GmbH. Since 2010, he is full EDA professor at the Robert Bosch Center for Power Electronics. His research interests include the automation of analog IC design with a special emphasis on physical design and methods for electro-thermal simulation.

Göran Jerke received his diploma degree in electrical engineering from the Dresden University of Technology. Since 1999, he is with the automotive electronics division of Robert Bosch GmbH, where he is responsible for constraint-driven design methodologies. His research interests include design flow concepts in general, as well as methods for physical design implementation, verification and robustness validation of IC designs.

Vinko Marolt accomplished a dual vocational training from 1973 to 1976 and worked in a developmental laboratory until 1979. Then he attended a technical school in Reutlingen and got the degree of certified technician in 1981. Since 1981 he works as a layout engineer and tool expert for the automotive electronics division of Robert Bosch GmbH. His motivation is the advancement of the layout design flow and of PCell development tools.