Hierarchical Propagation of Geometric Constraints for Full-Custom Physical Design of ICs

Maximilian Mittag*, Andreas Krinke†, Göran Jerke*, Wolfgang Rosenstiel‡
*Robert Bosch GmbH, Automotive Electronics, Reutlingen, Germany
Email: {maximilian.mittag, goeran.jerke}@de.bosch.com
†Dresden University of Technology, Germany
Email: andreas.krinke@ifte.de
‡University of Tübingen, Department of Computer Engineering, Germany
Email: rosenstiel@informatik.uni-tuebingen.de

***Abstract*—In industrial environments, full-custom layout design of analog and mixed-signal ICs is done hierarchically. In order to increase design efficiency, cell layouts are reused in the design hierarchy. Constraints forming relations between instances in different hierarchical contexts are of critical importance. While implementing a cell layout, these constraints have to be available in the cell's context. In this paper, a general definition of hierarchical constraints for a constraint-driven design flow is given. Furthermore it is shown, how top-down declared constraints can be propagated into another hierarchical context. Only by propagation they become visible and verifiable for bottom-up cell design. The feasibility of our proposed methodology is shown by applying it to a modular Smart Power IC of the automotive industry.**

I. Introduction

The increasing complexity of IC layout together with shortening time-to-market schedules is tackled by layout synthesis in the digital domain. The fact that custom analog and mixed signal (AMS) IC layout design has not evolved as much as its digital counterpart is mainly attributed to the circumstance, that there are many more individual design requirements (*constraints*) to be considered in the AMS domain. The reuse of cell layouts is one of the approaches to meet tight time-to-market schedules.

For reusing a cell, it is instantiated multiple times in the design hierarchy. Each instantiation may pose constraints on the design objects used to implement the cell's layout. These hierarchical constraints declared on instances impose limitations in the solution space for the layout that are rooted outside the cell itself. Conventional AMS design flows rely on a bottom-up methodology for the implementation (see Fig. 1a). A cell layout that is implemented ignoring these hierarchical constraints may have to be modified later in the design process when it becomes obvious that the constraints were not met. The resulting design flow iteration across hierarchical boundaries is time consuming and must be avoided.

This work has been partly funded by the German Federal Ministry of Education and Research (BMBF) within the Research Project RESCAR 2.0 (project label 01 M 3195 B).

We propose to formalize these hierarchical constraints allowing their declaration during a top-down design step. This step happens before the layout implementation, as can be seen in Fig. 1b). The implementation of a cell's layout during the bottom-up phase triggers the propagation of constraints declared on instances of the cell in other hierarchy levels. Only then they are visible and verifiable in that cell and design iterations can be avoided.

A. Related Work in Constraint-Driven Physical Design

Constraint-driven design is a prerequisite on the path to analog layout automation [2]. In pursuit of this goal, we don't want to achieve automatic optimization such as [3], [4]. Besides the fact that hierarchy is not considered in these optimizations, we rather want to give the layout designer full control of how to achieve the specific optimization goals.

To declare constraints independently of the EDA tool, we use some formalism of [2], [5] but a different methodology to handle the constraints. Their proposed methodology can only handle flat designs as hierarchy is not considered. Besides, constraints are handled by design databases in a native way [6], [7].

The design flow proposed by [1] relies on both, a uniformly top-down schematic design phase and completely automated layout generators on device and block level. Top-down layout planning occurs only after finishing layouts on lower hierarchical levels. As opposed to device-level layout generators no

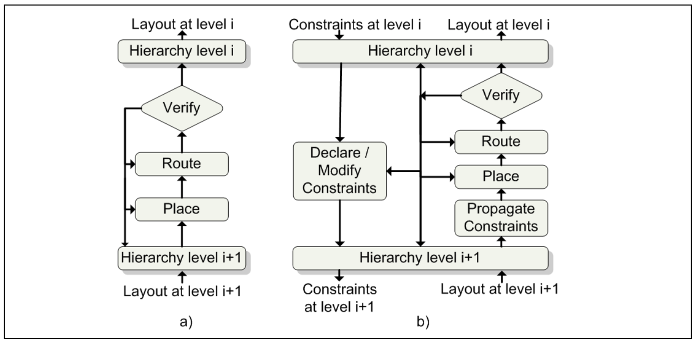

Fig. 1: *a)* Conventional bottom-up flow *b)* Enhanced flow incorporating hierarchical constraints (adapted from [1])

automated technologies are used on block level layouts for industrial AMS circuits yet. This makes iterations over hierarchical borders time consuming and error prone, as already finished layouts have to be modified manually accounting for layout constraints from higher hierarchical levels.

In [8] a constraint-driven design methodology is introduced. Assignment of constraint instances to design objects is discussed and three different types are identified: top-down, bottom-up and a combined assignment type. All assignments are done in a single design hierarchy tree and no solution is provided on how to make hierarchical constraints available in subtrees.

B. Contribution of this Work

The contribution is twofold: First, we give a formal definition of hierarchical and non-hierarchical design constraints in Section II. This definition is generic such that constraints resulting from other physical domains than geometry can be expressed. It is extensible as it allows the expression of arbitrary constraints between cell instances. Second, we define how these constraints are handled in a hierarchical layout in Section III. Constraints have to be available at the hierarchical level the designer is implementing, no matter where they were declared. The application of this methodology for ensuring bondability by constraining bond pad placement is shown in Section IV followed by a conclusion and outlook on further research.

II. Constraint Definition

Constraints used in AMS designs can be application and technology specific [9] and must be representable in the design system. Therefore, the constraints used in a design system have to be extensible to meet application specific needs of the IC to design. The definition proposed in this section is based on [10]. As only set theory is used, the definition of arbitrary new constraint types is possible.

Each of the constraint instances (subsequently constraint) relate a set of design objects. This set is denoted as *members* M of a constraint. The relation between the members is described by a set of *constraint parameters* P of arbitrary type (e.g. numbers, points, enumerations, etc.).

Every constraint can be assigned to a specific *constraint type* t, which serves as a classification property. The parameters P are the same for each instance of the constraint type while the parameter values may differ. The type also defines which kind of design objects the constraint relates. We denote the set of all constraint types of a design system T.

To describe a constraint instance c these properties are grouped into a tuple $c = (t, M, P)$ where $t \in T$. E.g.: a 'maximum distance' constraint c_1 between e_1 and e_2 of $5\mu m$ is described as $c_1 = (\text{maxdist}, \{e_1, e_2\}, \{5\mu m\})$.

The state (fulfilled or violated) of a constraint instance is determined by evaluating a constraint-type specific *verification function* verif_t. This function has to be developed for each constraint type $t \in T$. It takes a constraint instance as input and returns either true or false indicating the constraint's state:

```
Algorithm: $\text{verif}_{\text{maxdist}}(\text{constraint } c = (\text{maxdist}, M, P))$
Returns: true, if the distance between all elements in $M$ are smaller than or
    equal to $p \in P$, false otherwise
1: for all $(e_1, e_2) \in M \times M$ do
2:     if $\text{distance}(e_1, e_2) > p \in P$ then
3:         return false                                  //constraint violated
4: return true                                           //constraint fulfilled
```

Fig. 2: $\text{verif}_{\text{maxdist}}$ uses a geometric function $\text{distance}()$ to evaluate the state of the constraint instance

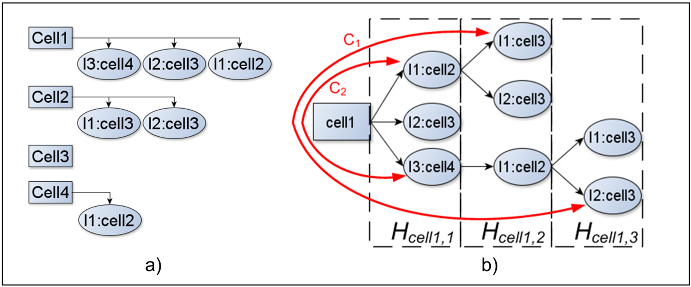

Fig. 3: *a)* Library (flat) view of a design and *b)* hierarchical context of cell1 with the hierarchical levels $H_{\text{cell1},1}$ to $H_{\text{cell1},3}$

$$\text{Constraint } c = (t, M, P) \text{ fulfilled} \Leftrightarrow \text{verif}_t(c).$$

The function verif_t can contain arbitrary verification code and ensures the extensibility to formulate application specific constraint types. A verification function $\text{verif}_{\text{maxdist}}$ for the aforementioned example is shown in Fig. 2.

When implementing the layout of a cell, other cells are instantiated (see Fig. 3a). This cell forms a design hierarchy i.e. a hierarchical context, that is often represented by a directed tree (see Fig. 3b). This tree can be represented in set theory by a partially ordered set of instances where the order is hierarchical instantiation. The leaf nodes of this tree are elements of the process development kit library and/or parametric layout generators (see cell3 in Fig. 3).

Constraints relating instances in the first hierarchy level of one cell are local constraints, as they do not affect instances further down the hierarchy (see C_2 in Fig. 3). But in general, constraints can form relationships between instances in different contexts of the design. Their members are identified using a list of instance names in the tree defined by the context of the cell. We denote this subtree, where all members of a constraint are instantiated the *context* of a constraint.

A constraint is called hierarchical, if one of the members is not on the first hierarchical level (e.g. C_1 as opposed to C_2 in Fig. 3). As these are declared where the member instances are addressable, the constraint exists hierarchically above the instances it affects. Hierarchical constraints have to be available, i.e. visible and verifiable, in the context the constrained instance will be implemented, which differs from the context they were declared in. The process of making a hierarchical constraint available in another context is referred to as propagation and is defined in detail in the next section.

III. Hierarchical Constraint Handling

In order to implement a cell, the constraints declared on any of its instances must be available to the designer. To make constraints visible and verifiable in a different hierarchical

```
Algorithm: propagate(context$H$, constraint $c = (t, M, P)$)
Returns: void, constraint $c$ is visible in all addressed contexts
1: for all $m \in M$ do
2:     for all $\bar{H}$ in the Design do
3:         if $m \in \bar{H}$ and $H \neq \bar{H}$ then
4:             create $\bar{c}$ with parent $c$ in $\bar{H}$
5:             set sibling property of $m$ to $\bar{c}$
6:             call propagate($\bar{H}$, $\bar{c}$)
```

Fig. 4: Algorithm *propagate* recursively creates hierarchically propagated sibling constraint in a different context $\bar{H}$

```
Algorithm: verification_delegate(context $\bar{H}$,
    constraint $\bar{c} = (t, \bar{M}, \bar{P})$)
Returns: void, verification function of the originating constraint is called
1: for all $m \in M$ do
2:     $H \leftarrow$ context of $m$'s parent
3:     if $H \neq \bar{H}$ then
4:         call verif$_t$ with $H$ and delegate$_t(H, \bar{H}, \bar{c})$
```

Fig. 5: Recursive *delegation* of verification of a propagated constraint $\bar{c}$

context, they are propagated through the design hierarchy. The propagation is defined in the next subsection. Following that the verification delegation of a propagated constraint is described.

A. Hierarchical Constraint Propagation

A constraint $c = (t, M, P)$ declared within a cell cell1 relates the design elements $M = m_1, m_2, ..., m_n$ in the context H_{cell1} of this cell. If one of the cells of M is also present in the context of another cell cell2, the constraint will have to be propagated to a constraint $\bar{c} = (t, \bar{M}, \bar{P})$ in H_{cell2}. After propagating the constraint, it is visible during the implementation of cell2.

The propagation algorithm that creates a sibling constraint $\bar{c}$ of the parental constraint c is shown in Fig. 4. Because the origin of a propagated constraint is relevant for the layout implementation, it has to be saved in the newly created constraint, which can be seen in line 4. As every member $m \in M$ can cause a propagation, there has to be a sibling property for each m, which is set in line 5. The two properties provide a navigable connection between the two contexts, which is needed for verification purposes.

B. Delegation: Verification of Propagated Constraints

To aid making a design decision, a constraint's state has to be visible in the current context. Therefore, the verification function for the constraint has to be called. As the members may not be visible and accessible in the propagated context, the verification has to be delegated to the parent constraint in the original context. The verification function is called in this context and the result is fed back to the propagated constraint. Only that way it is ensured that all necessary information for the verification function is accessible.

The delegation as shown in Fig. 5 is not constraint-type specific. This recursive algorithm will continue to delegate the verification until the constraint instance, from which the propagation was started, is reached.

The function delegate_t, which is called in line 4, takes a parental and sibling context as well as the sibling constraint as input and returns the constraint in the parental context. It is constraint-type specific as it changes the parameters P and/or the members M of the sibling constraint depending on the context of the parent. Similar to the verification function verif_t itself, the delegate_t function has to be defined when a hierarchical constraint type is defined.

IV. Experiments and Results

The introduced methodology of propagating constraints through a design hierarchy was applied to an industrial Smart Power IC. Due to high currents and voltages bond pads in these ICs are placed irregularly on the chip area near to or on top of the transistor arrays that form the power stages [11].

To build self-contained power stage blocks the pads were instantiated in these block contexts, making the use of hierarchical constraints an essential need. In Fig. 6 the simplified layout of the IC is shown. The constraints resulting from each block instance were propagated into the power stage cell, before detailed placement was done. This ensured that the placement of the bond pads was correct-by-construction regarding bondability for each instantiation of the power stage cell.

In the following, the implemented geometric constraint types restricting bond pad placement are introduced. They are implemented as Cadence Custom Constraints in the Virtuoso Design Environment [7]:

- Relative bond path length over chip
- Absolute bond path length
- No crossover of bond paths
- Bond path angle between wire and chip border

For each constraint type a verification function was developed, using the proprietary tool-specific functional language SKILL [12]. As all the implemented constraint types limit geometric placement, a single delegation function was sufficient as shown in Fig. 7.

The correctness of the propagation and delegation functions was validated by applying the methodology and comparing reported errors to those of a proprietary package assembly rule checker.

A. Propagation of Constraints to a Cell's Context

To validate the propagation mechanism we compared the number of propagated constraints in a hierarchical top-level layout to the one in a flattened top-level layout. The top-level layout contained the package, 82 instances of a bond pad and 8

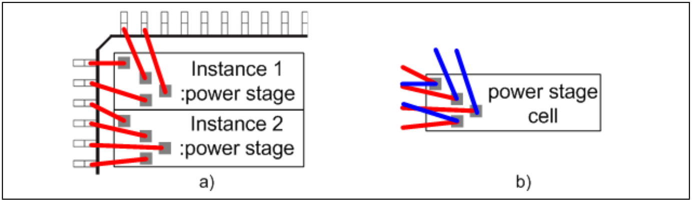

Fig. 6: *a)* Layout of power stage instances in top cell. *b)* Propagated constraints in power stage cell (from instance 1 = blue, from instance 2 = red)

```
Algorithm: delegate$_{geom}$(contexts $\bar{H}$ and $H$,
    constraint $\bar{c} = (t, \bar{M}, \bar{P})$)
Returns: constraint $c$, that is transfered from context $\bar{H}$ to $H$
1: $c = (t, M, \bar{P}) \leftarrow$ empty constraint with copy of $\bar{P}$
2: trans $\leftarrow$ transformation matrix from $\bar{H}$ to $H$
3: for all $\bar{m} \in \bar{M}$ do
4:     if $\bar{m} \in H$ then
5:         $\bar{m}$.position = trans $*$ $\bar{m}$.position
6:     $M \leftarrow M \cup \bar{m}$
7: return $c$
```

Fig. 7: Algorithm *delegate$_{geom}$* returns a constraint c in which the member coordinates of the propagated constraint $\bar{c}$ are transfered from context $\bar{H}$ to H by coordinate transformation

TABLE I
SUMMARY OF DECLARED CONSTRAINTS

Number of constraint types		4
Constraint instances (flat top-level layout)		584
Constraint instances (hierarchical layout)	local	328
	hierarchical	256
Propagated constraints in cell powerstage per instance of cell bondpad	4·8	=32
Sum of propagated constraints in cell powerstage	32·8	=256

instances of a power stage cell. The power stage cell contained 8 bond pads itself and was to be developed as one single layout that could be reused for every instance in the top level context. The final flat top layout contained 146 bond pads all together.

We used the flat layout to declare local constraints for the bond pad placement. As each bond pad was attached to each of the constraint types introduced in the previous subsection, we had 584 constraint instances in a flat top-level layout. Of these constraints 328 were declared on bond pad instantiations in the top layout, while 256 constraints were declared on bond pads resulting from power stage instantiations. This is shown in the upper part of Table I.

Then we used the hierarchical layout and declared the same total number of constraints on the bond pads. After applying the propagation algorithm introduced in Fig. 4, we inspected the power stage cell and discovered 256 constraints in that context (see lower part of Table I). All relevant constraints had been propagated from a hierarchical top level context to the power stage cell context.

B. Verification Delegation to Top Context

To show the feasibility of the propagated constraints' verification, we used a simple form of error injection. First we moved bond pads in the top cell context. Then we flattened the top layout and exported it to a proprietary package assembly rule checker. We compared the reported results of the assembly rule checker to the errors reported by the local constraints defined on bond pads in the top context.

Then we moved bond pads in the power stage cell context randomly. The constraint verification of the propagated constraints could be run in that context. Delegation was used to obtain the results immediately after bond pad movement. To compare them to the reported errors by the assembly rule checker the top layout had to be flattened after a context switch.

The reported errors were the same no matter if generated in the flattened top layout or the hierarchical one. This shows that delegation is functioning properly.

V. Conclusion and Outlook

We introduced a formal definition of design constraints, which form relations between cell instances in a hierarchical design. Top-down declared hierarchical constraints are made visible for bottom-up layout design by propagation: Creating constraint siblings in a hierarchically lower context. The verification within the context of a sibling is possible only by delegation to the originating constraint. We validated the proposed algorithms by implementing geometric placement constraints for bond pads and applying them to both, a hierarchical and a flat design to compare the reported errors by hierarchical constraints to those of a proprietary package assembly rule checker.

The benefits of the methodology were twofold: First, when implementing the layout of a cell, the constraints imposed on the bond pads (resulting from the multiple instantiations of that cell in other contexts) were visible and verifiable in that cell's context. The cell layouts were correct-by-construction regarding bondability. The time used for package assembly rule checking on a flattened layout shortly before tape out was reduced by about 20%.

Second, constraint violations became visible as soon as the placement of any instance containing bond pads was changed regardless of its hierarchical level. Normally these errors are discovered only at the end of layout design when final package assembly rule checks are done. The methods introduced in this paper greatly assisted in keeping track of design closure regarding bondability of the final chip layout.

Future research will focus on extending this methodology to allow for more complex constraint types, which are functional depending on other constraints.

References

[1] G. Gielen and R. Rutenbar, "Computer-aided design of analog and mixed-signal integrated circuits," *Proc. IEEE*, pp. 1825 –1854, 2000.

[2] G. Jerke and J. Lienig, "Constraint-driven Design - The Next Step Towards Analog Design Automation," in *Proc. of the Intl. Symp. on Physical Design*. ACM, 2009, pp. 75–82.

[3] A. Nassaj, J. Lienig, and G. Jerke, "A constraint-driven methodology for placement of analog and mixed-signal integrated circuits," in *15th IEEE Intl. Conf. on Electronics, Circuits and Systems*, 2008, pp. 770–773.

[4] A. Nassaj, J. Lienig, and G. Jerke, "A new methodology for constraint-driven layout design of analog circuits," in *16th IEEE Intl. Conf. on Electronics, Circuits and Systems*, 2009, pp. 996 –999.

[5] J. Freuer, G. Jerke, J. Gerlach, and W. Nebel, "On the verification of high-order constraint compliance in IC design," in *Design, Automation and Test in Europe, Proc.* ACM, 2008, pp. 26–31.

[6] D. Mallis, D. Cottrell, E. Leavitt, and B. Pfeil, *Si2 OpenAccess API Tutorial*, 9th ed. Silicon Integration Initiative, Inc., 2009, oA 2.2 DM4.

[7] Cadence Design Systems Inc., "Speeding design of custom silicon," http://w2.cadence.com/whitepapers/Virtuoso_WP.pdf.

[8] G. Jerke, J. Lienig, and J. B. Freuer, "Constraint-driven design methodology: A path to analog design automation," in *Analog Layout Synthesis*, H. E. Graeb, Ed. Springer US, 2011, pp. 269–297.

[9] J. Scheible, "Constraint-driven Design - Eine Wegskizze zum analogen Designflow der nächsten Generation," *MPC-Workshop*, pp. 1–9, 2010.

[10] A. Müller and B. Walliser, "Constraint Management im Full-Custom-Entwurfsablauf," *ITG FACHBERICHT*, pp. 181–184, 2003.

[11] B. Murari, F. Bertotti, and G. Vignola, *Smart Power ICs: Technologies and Applications*. Springer, 1995.

[12] T. Barnes, "Skill: a cad system extension language," in *Design Automation Conference, Proc.*, 1990, pp. 266 –271.