Extended Constraint Management for Analog and Mixed-Signal IC Design

Andreas Krinke*  Maximilian Mittag†  Göran Jerke†  Jens Lienig*
krinke@ifte.de  maximilian.mittag@de.bosch.com  goeran.jerke@ieee.org  jens@ieee.org

*Dresden University of Technology, Dresden, Germany
†Robert Bosch GmbH, Reutlingen, Germany

Abstract—The consideration of a growing number of design constraints is becoming a bottleneck in the design of analog and mixed-signal integrated circuits and is blocking more, much-needed automation in this area. In this paper, we propose a solution to these issues with a new methodology for constraint propagation and transformation. This technique allows designers and software tools to consider all relevant constraints when modifying a design, regardless of where these constraints were originally created. We integrated our ideas in an industrial design flow. The implementation of an electrical constraint type demonstrates the practical relevance. With constraints of this type the ON resistance of power stages in smart power ICs can be limited for the first time.

I. INTRODUCTION

The design of analog and mixed-signal integrated circuits (AMS ICs) is dominated by manual, non-automated tasks, compared with the extensive automation of digital design. The lack of automation is essentially caused by the great number of constraints that need to be considered during analog design [1]. These constraints define mandatory requirements on design parameters. On the one hand, constraints can capture the engineer’s intent, e.g., the relative positions and projected cell sizes in a floorplan. On the other hand, they can express parameter requirements. An example is the maximum permissible voltage drop (IR drop) across a wire for module to work properly. Long-time, analog design constraints used to be implicit, existing only as so-called “expert knowledge” or, at best, as informal notes on a schematic sheet [2]. Recently, the concept of constraints in AMS design has been formalized and increasingly integrated into various design tools (e.g. [3], [4]). Using these tools, one can assign constraints to elements in the currently edited cell. These constraints become part of the design data and can be verified automatically.

The major drawback of this approach is the limited visibility and verifiability of constraints. Currently, constraints are generally only visible and verifiable in the original cell—where they were created in the first place. However, so-called hierarchical constraints (see next section) reference elements in other cells (e.g. nets and instances) and, therefore, are of relevance outside of the original cell. The same applies to constraints that do not use the design hierarchy for information transfer, such as the connectivity-based IR drop constraint mentioned earlier. In both cases, designers and software tools do not have the information at hand as to whether decision in the current cell influence constraints declared in some other cell. This problem is solved by propagating constraints throughout the design. Transformation as an extension of this concept allows the conversion of constraints and therefore, the derivation of additional information.

A. State of the Art

Modern software tools for the design of AMS circuits (e.g. Cadence Design Framework II [3]) support the common hierarchical design approach that breaks the circuit into cells. The design may contain multiple instances (copies) of each cell. The integrated constraint management systems allow users to create new constraints that belong to a specific cell. Each constraint refers to a set of design elements (e.g. instances and nets), called constraint members. Each of these members can be either local or hierarchical. While local members belong to the cell of the constraint itself, hierarchical members are located in cells further down in the design hierarchy (see Fig. 1). In addition, every constraint is of a specific type that defines the parameters of all derived constraints and the verification procedure used to check if those constraints are met (see Section II). An example is the resistance constraint type used to constrain the resistance of wire segments. All constraints of this type have to provide maximal and minimal resistance values and are verified using a type-specific resistance model.

Current constraint management systems, e.g. the one integrated in [3], support only connectivity-based propagation. Furthermore, it is not possible to step over specific devices that are connected by a net. Constraint transformation is not supported either. Today’s verification tools, e.g. Mentor Graphics’ Calibre PERC, support topological constraints. They check the existence of circuit structures (e.g. for ESD protection) that are required by design rules [5]. However, this verification step does not provide assistance during the process of designing a circuit.

B. Our Contributions

In this paper, we present two extensions to constraint management systems: (1) Propagation distributes constraints based on connectivity across the design hierarchy. This information is thus available in all relevant cells. (2) Transformation allows the automatic creation of new constraints from a single original
one. In combination, both methods enable the visibility and verifiability of constraints in all cells, where they potentially have influence on design decisions. Thus, all constraints can be considered continuously—regardless of where in the design hierarchy they were originally created. To demonstrate the practical relevance of our approach, we implemented an \( R_{ON} \) constraint type for the first time. Constraints of this type enable the \( ON \) resistance between package pins to be limited. Such constraints are transformed and propagated based on pin connectivity.

II. PROBLEM FORMULATION

A constraint \( c \in C(\text{cell}) \) is a type specification, represented by the set \( C \) of all its constraints. They are defined as tuple \( c = (t(c), M(c), P(c)) \) where \( t(c) \in T \) is the constraint type, \( M(c) = \{m_1, \ldots, m_j\} \) is a set of constraint members and \( P(c) \) is a tuple \( (p_1, \ldots, p_k) \) of specific values for all parameters of \( t(c) \). Members are design elements to which \( c \) has been assigned. Each member can be either local or hierarchical. Local members belong to the same cell as \( c \) itself, while hierarchical members are located in another cell further down in the hierarchy (see Fig. 1). If at least one constraint member is hierarchical, we call the constraint itself hierarchical as well. For each constraint type, there exists a Boolean-valued function that verifies all corresponding constraints:

\[
\text{verify} : \{c \in C : t(c) = t\} \rightarrow \{\text{TRUE}, \text{FALSE}\}.
\]

Given a constraint of type \( t \), \text{verify} returns \text{TRUE} only if the constraint is fulfilled. The constraint propagation problem is to guarantee visibility and verifiability of hierarchical constraints in all cells where they might influence design decisions.

III. CONSTRAINT MANAGEMENT

During the design of an integrated circuit, all cell modifications (e.g. adding a device) should obey all relevant constraints (e.g. some resistance limit) at any time. Aggravating this situation, these constraints may exist in all cells that are hierarchically connected to the current cell. Two cells are hierarchically connected if there exists a path between instances of the cells in the design hierarchy graph (cf. Fig. 1). Our approach propagates these constraints by deriving new ones in all relevant cells. Hence, they become visible and verifiable.

Besides top-down propagation, which distributes constraints further down in the hierarchy, Fig. 2 shows two additional types of propagation. As an example, bottom-up propagation supports constraints on the absolute position of cells in the layout. Cells higher up in the hierarchy influence this position and, therefore, these constraints have to be propagated. Finally, bottom-up top-down propagation allows arbitrary distribution of constraints in the hierarchy, e.g., along connectivity. Our approach implements constraint propagation in two stages: (a) top-down propagation and (b) general propagation. Their combination supports, among other things, all three types of constraint propagation.

A. Top-Down Propagation

For top-down propagation, we limit the search for relevant constraints to cells that are higher up in the hierarchy. For example, in Fig. 1 the cells corresponding to \( \text{TOP} \) and \( \text{I}_2 \) are considered during constraint propagation to \( \text{I}_{21} \)’s cell. Our algorithm in Fig. 3 stores all cells that might contain relevant constraints in a queue for later examination.

A constraint is propagated, if at least one of its members is reachable from the current cell by going further down in the hierarchy. Newly created constraints have the form \( c' = (t', M', P') \). The derived type \( t' \) defines additional parameters that help identify the original constraint while retaining the original values \( P(c) \). For \( M' \), all members’ paths are adjusted to start at the current cell.

B. General Propagation

Our approach to bottom-up and bottom-up top-down propagation utilizes a new kind of function, that is specific for a constraint type \( t \):

\[
\text{propagate} : \{c \in C : t(c) = t\} \rightarrow c'.
\]

The propagate function takes a constraint \( c \) of type \( t \) as argument and returns a single propagated constraint \( c' = (t', M', P') \) whose type \( t' \) extends the original type \( t \) (cf. previous section). These new constraints can be located in all cells higher up in the hierarchy. Therefore, bottom-up propagation is directly supported. Bottom-up top-down propagation is implemented by performing top-down propagation, as described in the previous section, after bottom-up propagation via \( \text{propagate} \). In order to support all three propagation types, we significantly extended the algorithm for top-down propagation presented in Fig. 3 as follows. Now, the search for relevant constraints not only inspects cells higher up in the hierarchy, but all cells that are reachable via hierarchical paths. For each constraint \( c \) with a corresponding function \( \text{propagate}(c) \), this function is called resulting in a propagated constraint \( c' \). Afterwards, top-down propagation creates the actual constraints in the currently edited cell.
Constraint transformation enables the derivation of constraints from existing ones while manipulating their types, members and parameters. As an example, if the maximum allowed voltage drop and maximum current for a wire are available as constraints, a constraint for the maximum allowed resistance is deducible. We integrated constraint transformation in the design flow by further generalizing the propagate function presented in the previous section. Ultimately, we specify a transform function for each transformable constraint type \( t \). This function is defined as:

\[
\text{transform}_{t} : \{ c \in C : t(c) = t \} \rightarrow \{ c_1', c_2', \ldots \}
\]

In contrast to propagate, multiple constraints of arbitrary type can be returned. As an example, one IR drop constraint can be transformed into multiple resistance constraints. Depending on the returned constraint(s), this function can replace propagate and perform both, propagation and transformation. It is important to note that while exact knowledge of the design hierarchy is required for finding all relevant constraints, their subsequent transformation may use arbitrary design information. One example is the transformation of connectivity-based constraints utilizing the netlist of the circuit.

Fig. 4 shows the final process of propagation and transformation. It starts in the currently edited cell by examining the constraints of all hierarchically connected cells. If no transformation function is associated with a constraint’s type, only top-down propagation is carried out. Otherwise, if transform exists, it is executed, thereby creating one or more new constraints. These constraints are only temporary and thus, invisible. Transformation is repeated for all temporary constraints until top-down propagation is the only remaining choice. Not until then does each influenceable constraint become visible and verifiable in the currently edited cell.

### IV. EXPERIMENTAL RESULTS

#### A. Implementation

The methodology we developed is generic and, therefore, independent of the specific design environment. For demonstration purposes, we implemented the process as an extension to the constraint management system of Cadence Design Framework II, using the built-in scripting language SKILL++ [6].

Constraint transformation and propagation, as shown in Fig. 4, is activated when a user opens a cell. The implementation is divided into the following steps: (1) Determination of design hierarchy. (2) Identification of all constraints in hierarchically connected cells. (3) Transformation and propagation of all these constraints.

#### B. Complexity

The run-time complexity of our approach depends on the size of the design hierarchy as well as the number and types of constraints. For hierarchies with \( n \) individual cells that contain \( m \) given constraints, the search for relevant constraints is \( O(n+m) \). The run-time of subsequent constraint transformation and/or propagation depends on the number of constraints of a certain type and the complexity of the corresponding transformation functions. In addition, these functions can create an arbitrary number of temporary constraints that might be transformable by themselves. Because these properties are specific to a project and the set of available constraint types, we are unable to characterize more comprehensively their impact on run-time. Nevertheless, it is self-evident that the implementation of transformation functions should be optimized for speed. For instance, caching mechanisms can be used to avoid complex recalculations of data. Additionally, such data should be persistent between multiple design tool invocations.

#### C. Demonstrator

Using our methodology, we created a custom constraint type for the specification of maximum resistances between package pins of smart power ICs [7]. Such integrated circuits contain large MOSFETs (power stages) whose drain and source terminals are connected to IC package pins, either directly or via measure or protection circuits. These transistors switch high powers and, therefore, occupy large areas on a chip (cf. Fig. 5). One of the main design goals is a low ON resistance, \( R_{\text{ON}} \), between package pins that belong to power stages. The specification defines an upper limit, \( R_{\text{ON,max}} \), which directly translates into a design constraint. The new constraint type transforms \( R_{\text{ON,max}} \) into maximum resistance constraints for all contributing circuit elements. For example, \( R_{\text{DS,ON}} \) constraints are assigned to all transistors with their values being calculated from \( R_{\text{ON,max}} \) and the current resistance values of all other elements between the two package pins.

Fig. 6 shows the outline of the transformation function. Because the transformation depends on the electrical connectivity...
In this paper, we presented a new methodology for constraint propagation and transformation. Regardless of where in the design hierarchy a constraint was created, these fundamental operations enable the constraint’s visibility and verifiability in all cells where it is relevant for design decisions. The methodology supports constraint types whose propagation and transformation depends on hierarchy and/or on connectivity. To the best of our knowledge, it is the first to include the ability to step over specific design elements while traversing nets. Beyond that, the integration of arbitrary application-specific constraint types is possible.

We believe that our approach is an important advance in the automation of analog and mixed-signal IC design. It provides a basis for the automatic consideration of all constraints which is a fundamental requirement for analog synthesis as aspired to in the long run. The methodology allows designs to be as correct as possible at all stages of the flow. Cell reuse becomes much more efficient because all constraints assigned to a cell’s instances are visible when the cell is being modified. The demonstrated creation of a completely new, and complex, constraint type emphasizes the practical relevance. Initial application to the design of AMS circuits in an industrial design flow has shown its usefulness by significantly easing constraint management, facilitating cell reuse and reducing the number of design iterations.

### REFERENCES


