Constraint Propagation Methods for Robust IC Design

Andreas Krinke*, Goeran Jerke†, Jens Lienig*
*Dresden University of Technology, IFTE, Dresden, Germany, krinke@ifte.de, jens@ieee.org
†Robert Bosch GmbH, Automotive Electronics, Reutlingen, Germany, goeran.jerke@ieee.org

Abstract

Constraint engineering is one of the key enabling technologies to address robustness and reliability issues in today's IC designs. Design constraints are used to express and verify the customer's demands and the designers' intent. These constraints put limits on some design object's parameter values and, hereby, represent additional design information that is then used to enforce robustness, reliability and other design targets. In hierarchical IC designs, the states of constraints often depend on parameters of other design modules in the design hierarchy. Therefore, these constraints are propagated within the design hierarchy to be considered when taking design decisions and performing design verification. This propagation is an essential component of any robustness-, reliability- and constraint-driven design flow. To the best of our knowledge, this is the first work that presents a systematic classification and detailed discussion of the constraint propagation problem. Despite the vast number of conceivable constraint types, we found that there are only six different propagation categories. We derived a single but generic constraint propagation algorithm for all six propagation categories. Our work closes a critical automation gap in today's constraint engineering flows by proving the full automatability of the constraint propagation problem and by providing a comprehensive and consistent propagation solution. We also present experimental results from an industrial design that demonstrate the applicability for large design problems.

1 Introduction

Constraint engineering is one of the key enabling technologies to address robustness and reliability issues in today's IC designs [1, 2, 3]. It comprises the management of design constraints (constraints), the active consideration of constraints during design implementation (i.e., constraint-driven design), and constraint verification. Constraints hereby represent additional design information to express and verify the customer's demands and the designer's intent.
A constraint itself represents an information entity that is linked to design objects (e.g., cells, instances, nets, terminals etc.) and to referring design parameters (e.g., electrical resistance between terminals of a net, electrical potential of a net, terminal current etc.). It represents additional design information that is, among others, used to enforce functional, cost, robustness and reliability targets, such as max. chip area or guaranteed robust and reliable operation for a given application mission profile. Design parameters and corresponding constraints may occur in design domains, such as the electrical, thermal, mechanical or physical domain, depending on the design step and design context. Other constraint examples include the maximum timing delay between net terminals, the maximum IR-drop between net terminals, mandatory local and global device instance orientation, device parameter matching constraints, required net shielding, distance between layout cells etc.
Formally speaking, *constraints* restrict the available design space by applying limits to the set of possible design parameter values. Design parameters hereby belong to either a *local* or a *hierarchical* design context. For instance, a design parameter linked to the electrical resistance between two given net terminals belongs to the local design context if these net terminals are located in the same design cell. The design parameter belong to a hierarchical design context if the net terminals are located in different design cells that form a design hierarchy. *Constraint propagation* provides access to constraint-specific information, such as constraint parameters, type and status, by marking all affected design cells relevant to a particular constraint.
In case a design parameter's value can be influenced in more than one cell, then constraints on this parameter have to be *propagated* into these cells in order to be applicable for hierarchical design implementation and verification. The propagation of constraints within the design hierarchy is a key challenge for effective design implementation and verification. Constraint propagation is also essential to any automated and interactive reliability-, robustness- or other constraint-driven design flow.
Due to the vast number of conceivable constraints and constraint types one could expect a large number of different propagation methods, which then make a generic propagation solution highly improbable. Hence, finding a single but generic propagation approach that provides an applicable solution for the propagation of any arbitrary constraint type would provide a profound benefit for any constraint engineering tool.
Our work demonstrates that every constraint propagation path depends only on the constraint's own parameters and its referenced design objects. Leveraging this key insight allowed us to derive a categorization schema comprising six fundamental constraint propagation methods (five of which are shown in Fig. 1).
Our paper presents the first classification and detailed discussion of the constraint propagation problem. In addition, we derive a single but generic constraint propagation algorithm for all propagation categories. Any hierarchical design constraint can now be propagated automatically using

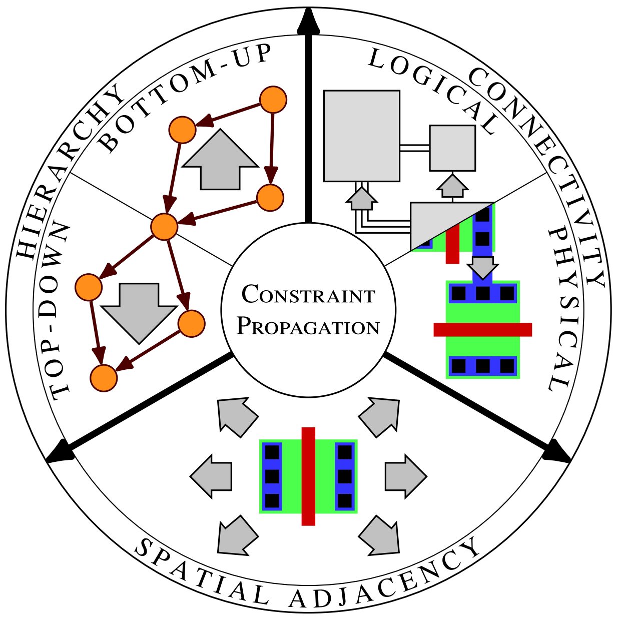

Figure 1 Overview of hierarchy-based, connectivity-based and spatial-adjacency-based constraint propagation methods. Every constraint can be propagated using either a combination of these methods, or global propagation.

a combination of these methods. Constraints thus become visible and addressable in all relevant cells and can be taken into account when making design decisions and verifying the constraint compliance. Our work hereby closes a critical automation gap in today's constraint engineering flows by proving the full automatability of the constraint propagation problem and by providing a comprehensive and consistent propagation solution.

After presenting related work in Section 2, a formal introduction to constraints, a generic propagation algorithm and constraint propagation classes are given in Section 3. Section 4 presents a novel definition of constraint types. We present experimental results that demonstrate the applicability for large design problems in Section 5. Section 6 provides a summary and conclusion.

2 Related Work

Our work is based on a long history of research into constraints and their propagation. Back in 1988, Ly and Girczyc [4] described constraint propagation as a method for automatically updating cell parameters upon instantiation. Then in 1992, Chang, Sangiovanni-Vincentelli, Balarin *et al.*[5] examined the generation of layout constraints from high-level electrical constraints and their subsequent top-down propagation into cells further down the design hierarchy. Three years later, Han, Stephanopoulos, and Douglas [6] approached the problem using top-down propagation (described as inducing new constraints) and constraint-value propagation between different abstraction levels. Later, Malavasi, Charbon, Felt, and Sangiovanni-Vincentelli [7] described top-down propagation and constraint transformation for creating low-level parasitic constraints from high-level performance specifications.

In 1998, Arsintescu [8] presented a hierarchical top-down design method. Constraint transformation was used to create low-level constraints from high-level ones in each hierarchy level. Afterwards, these constraints were top-down propagated to lower levels. His work also discussed classification criteria for constraints, including a constraint's level and type, and its transformation function type. In the same year, Malavasi, Charbon, Arsintescu, and Kao [9] outlined what later became the constraint manager for Cadence DF II. Besides top-down propagation for budgeting purposes, this work also explained bottom-up propagation for rebudgeting.

In 2011, Jerke, Lienig and Freuer [2] classified constraints in groups of technological, functional, design-methodical and commercial constraints. A second categorization distinguished complex constraints (affecting dependent design variables) and simple ones (affecting only independent variables). In addition to top-down and bottom-up propagation, they explained bottom-up top-down propagation, e.g., for connectivity-based propagation. In 2013, we presented a pragmatic approach to constraint propagation [10]. Without a classification of constraint propagation, we had to use type-specific transformation functions. Their formulation proved to be laborious given the vast number of constraint types. Katzschke *et al.* [11] extended our constraint model. Each constraint type is tied to a specific design tool in order to support cross-domain constraints. In addition, a constraint may comprise a set of child constraints—in other words, a set of constraints may be integrated into a single new constraint. Jerke and Kahng [12] discussed robustness aware design of automotive ICs using automated generation, transformation, propagation and usage of functional loads and environmental conditions, which may be considered as constraints as well. Recently, Crepaldi, Grosso, Sassone *et al.* [13] discussed a design methodology for smart systems that utilizes top-down constraint propagation.

To the best of our knowledge, no research has been carried out and published to date that presents a classification of constraint propagation methods or a unified and generic propagation approach.

3 Classification of Constraint Propagation

3.1 Introduction

Formally speaking, a constraint is a requirement on the values of one or more design parameters. These parameters form the constraint's *target parameters*. A constraint c can be formulated as a function of design parameters $c : X \rightarrow B$ that returns the constraint's state ("satisfied" or "violated") as a Boolean value. Therefore, its codomain is $B = \{\text{True}, \text{False}\}$. Arguments of c are tuples of design parameter values. Accordingly, the domain X contains all possible tuples of parameter values for c.

Each constraint belongs to a specific design cell that provides its local *context*. A constraint may reference only design parameters that belong to its context. As each parameter is associated with one or more design elements, these elements have to be part of the context cell as well. This requirement is met if the element is located in the context cell itself or

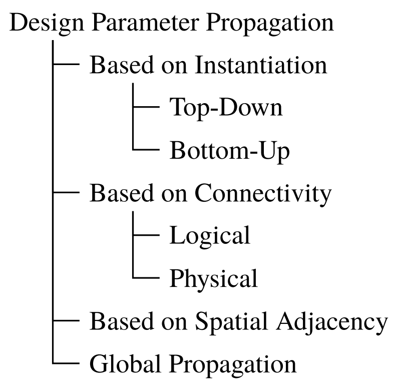

Figure 2 Classification of design parameter propagation. The propagation of a constraint may require different types depending on its target design parameters.

in any instance within this cell. Examples for such design elements are instances, cell and instance pins, nets, polygons and the context cell itself. The *members* of a constraint are those design elements that are associated with its parameters.
A constraint is denoted as *local* constraint if it references design parameters of its context cell only. A constraint is denoted as *hierarchical* constraint if it references at least one design parameter of an instance that is hierarchically below the context cell. Parameters of instances hierarchically above the context cell may not be referenced, because these upper hierarchy levels differ with each instance of the context cell. This is only possible using constraint generation methods. A constraint is denoted as *complex* constraint if it limits multiple design parameters at once.
A hierarchical IC design can be modelled as hierarchical design graph that contains a node for every cell of the design. If one cell C_1 contains an instance of another cell C_2, an edge $C_1 \rightarrow C_2$ is added to the graph. In contrast, a flat design graph contains no intermediate hierarchy levels. Instead, every global instance, i.e., every possible path from the topcell to a cell at the lowest hierarchy level in the hierarchical design graph, is modelled as a separate instance node. Edges then relate the topcell node with every instance node.
Each constraint has to be propagated into all cells, whose implementation may affect the constraint's status. Due to the vast number of conceivable constraints and constraint types one could easily assume that there is also a large number of different propagation methods. Through analysis of constraint types, we found that the set of target design parameters of a constraint or constraint type is here the only decisive classification criterion for its propagation within the design hierarchy. The modality used to construct this set of relevant cells hereby depends on the type of design parameter. As discussed in detail in Sections 3.3–3.6, we identified the following different constraint propagation categories (cf. Fig. 2):

- Propagation based on instantiation (top-down and bottom-up)
- Propagation based on connectivity (logical and physical)
- Propagation based on spatial adjacency
- Global propagation

```
1:  procedure PropagateConstraint($c, G$)
2:      $C \leftarrow$ Context($c$)
3:      for each parameter $p \in$ Parameters($c$) do
4:          $M \leftarrow$ Members($p$)
5:          $S \leftarrow$ PropagationSequence($p$)
6:          for each member $m \in M$ do
7:              $R \leftarrow \{G\}$        // Initially, all cells are relevant.
8:              for each propagation type $t \in S$ do
9:                  // Call type-specific propagation function.
10:                 $R \leftarrow$ Propagate$_t$($c, C, m, G, R$)
11:             end for
12:             for each subgraph $H \in R$ do
13:                 for each relevant cell $r \in H$ do
14:                     Mark($r, c, p, m, H$)   // Mark relevant cell.
15:                 end for
16:             end for
17:         end for
18:     end for
19: end procedure
```

Figure 3 Our generic constraint propagation algorithm. Input data are the constraint c and the hierarchical design graph G containing all relevant design elements. In lines 2, 4 and 5 the required information about the constraint and its target parameters are retrieved. The set R stores one or more subgraphs of G containing all cells with relevance for the constraint's status. Propagation type t is one of: top-down, bottom-up, based on logical connectivity, based on physical connectivity, based on spatial adjacency or global propagation (cf. Sections 3.3–3.6).

Each constraint can be propagated by implementing methods for each of these propagation categories. Complex constraints may require the use of multiple different propagation methods, depending on their set of target parameters.

3.2 Generic Propagation Algorithm

Based on the insight that propagation of a constraint depends only on its set of target parameters, we were able to derive a generic mechanism to propagate all constraints with a single high-level propagation algorithm (see Fig. 3). When propagating a constraint, this algorithm processes each target parameter independently. For each parameter, the set of associated design elements and its propagation sequence S are retrieved. This sequence depends only on the parameter type and defines what propagation methods have to be executed. Most design parameters can be propagated using just a single propagation method. In this case, S contains only this method. Only parameters that require combined propagation (cf. Section 3.7) result in S containing more than one propagation method.
For each type t of the six propagation types described in the remainder of this section, a separate propagation function Propagate$_t$ is required. The actual propagation is performed for each associated design element of the current parameter separately. The propagation methods in S are executed, each returning one or more subgraphs of the design graph. These subgraphs represent sets of relevant cells and can be refined by subsequent propagation methods

in S. Finally, all relevant cells with potential influence on the value of the current parameter are marked. Therefore, line 14 forms the result of constraint propagation.
As an example, the propagation of a constraint that limits the distance between two instances is described in the following. In this case, the distance is the only design parameter with the two instances as its associated design elements (members). For this particular parameter type, bottom-up propagation is the only required propagation method (as explained later). Therefore, bottom-up propagation is performed for each instance. Propagation finishes after marking the resulting relevant cells.
The constraint propagation categories are discussed in detail in the remainder of this section.

3.3 Propagation Based on Instantiation

The first propagation class applies to all design parameters whose value depends only on properties of parental cells higher up in the hierarchy, or of instantiated cells further down the hierarchy.

3.3.1 Top-Down Propagation

Each hierarchical cell in a design contains instances of other cells. All parameters of a cell whose value can be influenced by modifying these subordinate cells enforce top-down propagation of associated constraints. Examples of parameter types requiring top-down propagation include power consumption and area of a cell. Formally, this applies to every parameter p_C of some cell C, whose model depends only on the values of the exact same parameters of instances $I_1, I_2, \ldots, I_n$ within C:

$$p_C = f(p_{I_1}, p_{I_2}, \ldots, p_{I_n}) \qquad (1)$$

This causes recursive propagation, because the parameter value of such an instance depends, again, on the parameter values of containing instances. Therefore, the corresponding top-down propagation method PROPAGATE$_{\text{td}}$ returns a single subgraph rooted in C. This graph includes all cells that are instantiated in any hierarchy level below C. The subgraph and, therefore, the set of relevant cells is independent of the number of context cell instances in the design.
From a data modeling perspective, fast access to the cells that are instantiated in a given cell is required for efficient top-down propagation.

3.3.2 Bottom-Up Propagation

Layout parameters, e.g., global positions, alignment or orientations of instances in the circuit layout, are target parameters of the important group of layout constraints. The values of these parameters can be modified in cells higher up in the design hierarchy. Moving or rotating an instance in one of these parental cell layouts results in a change in these layout parameters for all instances further down in the hierarchy. Therefore, constraints on such parameters require bottom-up propagation. Formally, this applies to every instance parameter p_I, whose value can be expressed using only the value of the exact same parameter of its immediate parental instance I_P:

$$p_I = f(p_{I_P}) \qquad (2)$$

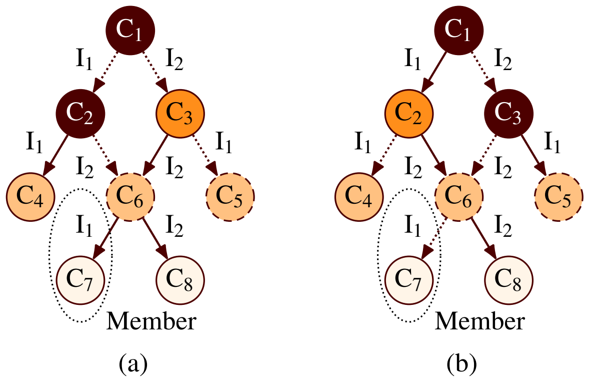

Figure 4 Exemplary design hierarchy including a constraint in cell C_6 on a bottom-up propagated parameter of instance I_1. Because the top cell C_1 contains two instances of C_6 (one in C_2 and the other in C_3), bottom-up propagation yields two sets of relevant cells (marked by dark nodes in (a) and (b), respectively). Therefore, after propagation, changes to C_2 and C_3 will have to consider one propagated constraint, while two propagated constraints limit modifications of C_1.

For example, the global position of an instance depends on the global position of its parental instance. Again, this causes recursive propagation which does not stop until a top-level cell is reached. A constraint's context cell and the number of instances of that cell have major implications for propagation (as opposed to top-down propagation). Fig. 4 illustrates that each instance of the context cell in the design has a different set of parental instances. Therefore, multiple sets of relevant instances exist for such a constraint, one for each global instance of its context cell. The propagation method PROPAGATE$_{\text{bu}}$ returns multiple subgraphs rooted in top-level cells.
From a data modeling perspective, fast access to the parental instances of a given global instance is required for efficient bottom-up propagation.

3.4 Propagation Based on Connectivity

Constraints on parameters whose values depend on the electrical connection of a given pin or net, have to be propagated based on connectivity. We distinguish two types, depending on the type of connectivity that influences a parameter's value: logical or physical connectivity.

3.4.1 Propagation Based on Logical Connectivity

Parameters such as the load capacitance of a net and ESD protection of a pin are influenced by instances that are logically connected. Therefore, constraints on such parameters are propagated based on logical connectivity. Formally, this applies to every pin or net parameter p_N, whose value depends only on the values of the exact same parameter of logically connected instances $I_{\text{LC}1}, I_{\text{LC}2}, \ldots, I_{\text{LC}n}$:

$$p_N = f(p_{I_{\text{LC}1}}, p_{I_{\text{LC}2}}, \ldots, p_{I_{\text{LC}n}}) \qquad (3)$$

Again, this definition results in recursive propagation that only stops at instances of basic cells at the lowest hierarchy level. The propagation method $\textsc{Propagate}_{\mathrm{lc}}$ returns multiple subgraphs, one for each global instance of the constraint's context and each containing all logically connected instances.

For efficient propagation based on logical connectivity, the data model should allow fast access to all instances of these basic cells that are connected to a net or pin. Therefore, a flat design graph without intermediate hierarchy levels is recommended.

3.4.2 Propagation Based on Physical Connectivity

It is important to differentiate between the set of instances that are connected to a given net or pin, and the set of instances whose layouts implement this connectivity. While the former is derived from logical connections defined in the schematic hierarchy, the latter is defined by the layout hierarchy. The set of instances that contribute to an electrical connection may differ between schematic and layout hierarchies.

Parameters, whose value depends on the physical implementation of an electrical connection have to be propagated based on this physical connectivity. They have to be visible in all instances that implement or may implement parts of this connectivity. Examples include parasitic resistance and capacitance of a net. Formally, this applies to every pin or net parameter p_N, whose value depends only on parameters $q_{\mathrm{PC}1}, q_{\mathrm{PC}2}, \ldots, q_{\mathrm{PC}n}$ of the layout shapes implementing the connectivity:

$$p_N = f(q_{\mathrm{PC}1}, q_{\mathrm{PC}2}, \ldots, q_{\mathrm{PC}n}) \tag{4}$$

Again, the propagation method $\textsc{Propagate}_{\mathrm{pc}}$ returns a subgraph for each global instance of the constraint's context. Each subgraph contains all instances that either implement or overlap parts of the connectivity's layout shapes.

Fast access to all layout shapes implementing an electrical connection and to the cell layouts they belong to are requirements for efficient propagation based on physical connectivity. As for logical connectivity, a flat design graph is the recommended starting point.

3.5 Propagation Based on Spatial Adjacency

Layout structures neighboring or overlapping an instance on the die may influence its parameters. In that case, constraints on these parameters have to be propagated based on spatial adjacency in the layout. This applies to, e.g., (matching) coverage, no overlap, and cluster constraints, where the bounding polygon of a set of instances must not contain other instances. Formally, this propagation type is used for constraints on parameters whose values depend only on layout elements within a certain distance.

For example, a coverage constraint might dictate that an instance is covered by a specific layer, e.g., the top metal layer. These metal shapes can be created in any instance whose bounding box overlaps with the one of the first instance. As a consequence, this constraint has to be propagated into all these cells. For each global instance of the constraint's context, the propagation method $\textsc{Propagate}_{\mathrm{sa}}$ computes a global bounding polygon for the set of parameter members. Then, for each bounding polygon, a subgraph containing all overlapping instances is returned.

A flat design graph with fast access to instances overlapping a given polygon is recommended for efficient propagation.

3.6 Global Propagation

There are design parameters whose values can potentially change when *any* cell in the design is modified. An example is the parameter “set of required layers”. This parameter is a property of cells and impacts both their fabrication cost and the set of devices that may be used for their implementation. For a cell, whose layout requires a special layer, a constraint for this parameter can represent this information. When this cell is subsequently instantiated, the constraint will be propagated globally into all other cells. Therefore, the information on availability of this special layer is available everywhere. The propagation method $\textsc{Propagate}_{\mathrm{g}}$ maintains a list of constraints with global relevance and returns no subgraph.

3.7 Combination of Propagation Methods

A very large number of design parameters triggers exactly one of the six propagation methods described earlier. However, there are parameters that need more than one of these methods to be executed. In this case, the first propagation method produces a subset of the design which is then further processed by all subsequent methods.

An example for this behavior is the class of net shielding constraints. When applied to a pin or net, such a constraint requires the layout implementation of all electrically connected shapes to be shielded. Its propagation requires two steps: First, propagation based on physical connectivity identifies all shapes that connect to the pin or net in the layout. Second, propagation based on spatial adjacency finds all instances, whose layout overlaps some of these shapes. The shielding shapes have to be created in a subset of these instantiated cells. Another example is a cell's internal resistance between two of its terminals. This parameter is of special interest for the design of power stages [10]. When processing constraints on such a parameter, the first step is finding all internal instances using top-down propagation. Thereby, all instances outside the cell are ignored. Finally, propagation based on logical connectivity identifies all internal instances that may influence this resistance.

The computation of relevant cells for a single member of one of a constraint's parameters is visualized in Fig. 5. After propagation, all relevant cells are marked in the database. If a user opens any marked cell, all constraints that marked the cell are displayed and can be considered during editing.

3.8 Influence of Constraint Context on Propagation

While the set of target parameters implies the propagation mechanisms of a constraint, the constraint context greatly impacts the reach of propagation. As explained earlier, the context of a constraint is defined by the cell in which it was created. Besides restricting the addressable design elements, the choice of context also has a critical bearing on how well

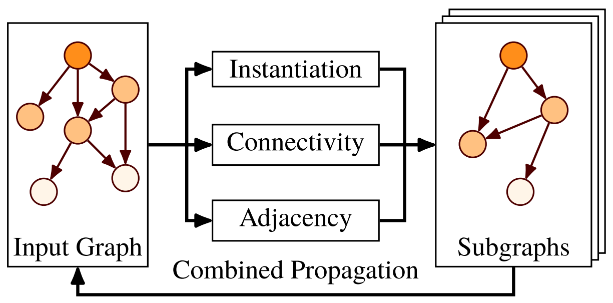

Figure 5 When applied to the design graph, all types of propagation create one or more subgraphs with all relevant cells. Combined propagation causes iterative processing of this set of subgraphs by the same propagation methods.

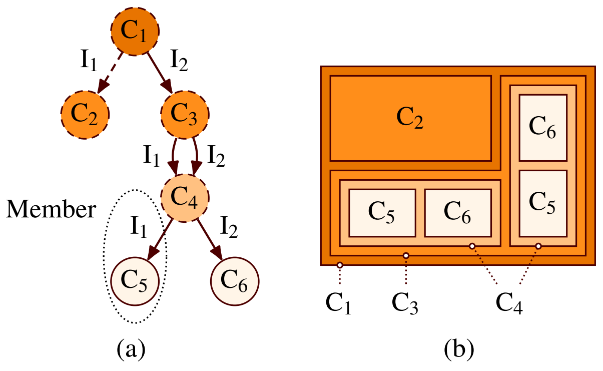

Figure 6 Exemplary design hierarchy (a) and corresponding hierarchical floorplan (b). Depending on the designer's intent, a parameter of instance I_1 in cell C_4 can be constrained in the context of either C_4, C_3 or C_1 (marked with dashed lines). A constraint in context C_4 applies to both of its instances in C_3. However, if the constraint is created in C_3 or C_1, only one of the two instances is affected. In this case, one instance of C_4 in C_3 has to be chosen when defining the constraint's target parameter.

the constraints reflect the designer's intent. The context defines which cell and, therefore, which instances the constraint targets. Fig. 6 shows an example for a constraint on a parameter of some instance. This constraint could be created in the context of any cell higher up in the design hierarchy. Because the restriction applies to all instances of that cell, the search for relevant cells during constraint propagation examines each of these instances individually (cf. instantiation-based propagation in Section 3.3.3). Therefore, the context has great influence on the set of relevant cells and, thus, on the result of constraint propagation. However, it is important to note that a constraint's context is not a criterion for determining applicable propagation mechanisms.

4 Constraint Types

4.1 Constraint Representation Using Ordered Trees

The functional definition of constraints allows the creation of an infinite number of different constraints. While there are no restrictions on the structure of constraint function c, current literature provides a large list of about 350 global

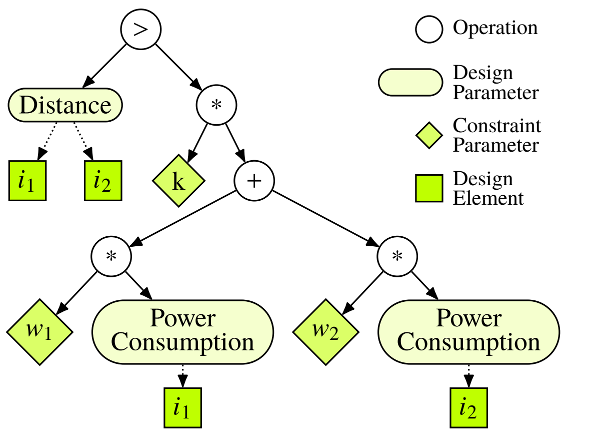

Figure 7 An ordered tree representing the constraint in Eq. (5). Dependencies of design parameters on design element properties are depicted with dotted arrows.

constraints that can be used as building blocks [14]. For practical purposes, it is essential to categorize constraints into different types in order to manage their diversity. This classification requires a more structured constraint representation to increase the scope of the general formulation. We propose to represent constraint functions as ordered trees. Each mathematical operation that is required to calculate the constraint function becomes a node in the tree. Arguments for these operations, i.e., values of design and constraint parameters or the result of nested operations, become children of the corresponding node. Thereby, the tree expresses the dependencies between operations and parameters and the order in which they are calculated. By starting at the leaf nodes and calculating the result of all operations while going up the tree, the value of the constraint function is evaluated once the root node is reached. An exemplary constraint on the minimal distance between two instances depending on their respective power consumption is shown in Eq. (5):

$$\begin{aligned}\text{Distance}(i_1, i_2) > k \cdot (&w_1 \cdot \text{PowerConsumption}(i_1) \\ &+ w_2 \cdot \text{PowerConsumption}(i_2))\end{aligned} \quad (5)$$

This specific constraint has three parameters k, w_1, w_2 and applies to two instances i_1, i_2 in the design. The values of the constraint parameters, as well as its members i_1 and i_2 need to be specified when the constraint is created. The corresponding ordered tree is shown in Fig. 7.

4.2 Constraint Type Definition

Constraints can be categorized according to the structure of the constraint function as described by its ordered tree. Therefore, we define a constraint type using a *pattern tree*—an ordered tree whose nodes are annotated with quantifiers that define how often each subtree may occur. Similar to quantifiers in regular expressions, one can specify that nodes have to occur zero or more times; one or more times; or at least m, and not more than n, times. A constraint type's pattern tree describes the (possibly infinite) set of all ordered trees, and, therefore, of all constraint functions that are of this type. Besides being a blueprint for creating constraints of a certain type, arbitrary constraint functions can be categorized with pattern trees.

The state of a constraint depends on the values of deployed (target) design parameters, which, in turn, depend on design element properties. These design elements become the members of the constraint. For example, the Distance parameter and, thereby, the constraint's state in Eq. (5) depend on the layout positions of two instances i_1, i_2. Therefore, if the Distance parameter is deployed, the constraint must have two members of type "instance".

5 Experimental Results

In order to demonstrate the feasibility of our presented constraint propagation approach, we applied all presented constraint propagation methods to an industrial automotive mixed-signal IC design. The used IC design comprises 413 analog and digital design cells in 11 levels of design hierarchy resulting in 972,539 instances of the flattened design.

Our presented constraint propagation approach is generally applicable to any representation of design data. However, we used a graph data model and a graph database to represent the design data incl. instances, nets, terminals etc. [15]. The usage of a graph database has the distinct advantage that propagation algorithms can be implemented and tested easily. The used graph database was accessible from the commercial design framework Cadence Virtuoso® through a SKILL++ based data interface.

At the beginning of our test, the IC design data (incl. constraint data) of the design framework was initially exported to the graph database and then kept in sync. The constraint propagation of all defined constraints was then initiated. The propagation results were first written to the graph database and then synced with the design framework's database.

Based on our findings, we implemented 9 constraint types, one for each of the following design parameters:

- Top-down propagation: cell area, power consumption
- Bottom-up propagation: instance position, instance orientation
- Connectivity-based propagation: net load capacitance
- Spatial-adjacency-based propagation: no polygon overlap, polygon coverage
- Combined propagation: pin-to-pin resistance, net shielding

For each constraint type, several constraints were created with different context cells and varying target design elements. After automatic propagation, we verified the correct marking of all relevant cells. This verification was carried out using manual graph database queries to inspect all cells marked as relevant and their relation to the original constraint's context cell. As a result, all relevant cells were correctly identified for all constraints. Afterwards, when opening a relevant cell, a complete list of those constraints, whose status can be influenced in this cell, was available.

The overall algorithmic complexity of our approach depends on the constraint's structure, i.e., the number of affected design parameters, their propagation sequence and their set of members. Almost all design parameters we examined did not require combined propagation and had only a small number of members, which is beneficial for runtime. Therefore, our method's complexity depends primarily on the efficiency of the four basic propagation methods. However, their efficiency in determining relevant cells is strongly tied to the modeling of design data [15]. For the used graph data model and the example circuit, for example, iterating over all 972,539 instances below the top-level cell—as required for top-down propagation—takes about 2.3 s using an 2.4 GHz Linux workstation. However, data modeling itself is beyond the scope of this paper, so please refer to [15] for more details.

It is important to note that the cost of constraint propagation only incurs on constraint creation and when relevant cells are modified. Later on, during the design process, listing all constraints with relevance for a cell is very fast and happens virtually instantaneously.

6 Summary and Conclusion

The constraint propagation problem is of tremendous practical importance for constraint engineering and its application to industrial IC designs. Propagation of constraint information throughout the design hierarchy allows well-grounded design decisions and design verification. It is an essential component of any robustness-, reliability- and constraint-driven design flow. To the best of our knowledge, this is the first work that presents a systematic classification and detailed discussion of the constraint propagation problem. We identified six generic propagation categories that cover all constraint types. As a result, we developed a single but generic constraint propagation algorithm for all propagation categories, which now allows for an easy and consistent integration into existing constraint-driven design tools and IC design frameworks. Our experimental results demonstrate the applicability for large industrial design problems.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments on this paper.

References

[1] R. A. Rutenbar, "Design Automation for Analog: The Next Generation of Tool Challenges," in *Proc. Int'l Conf. on CAD*, San Jose, CA, USA, 2006, pp. 458–460.

[2] G. Jerke and J. Lienig, "Constraint-driven Design – The Next Step Towards Analog Design Automation," in *Proc. Int'l Symp. on Phys. Design*, San Diego, CA, USA, 2009, pp. 75–82.

[3] J. Scheible and J. Lienig, "Automation of Analog IC Layout – Challenges and Solutions," in *Proc. Int'l Symp. on Phys. Design*, Monterey, CA, USA, 2015, pp. 33–40.

[4] T. Ly and E. Girczyc, "Constraint Propagation in an Object-Oriented IC Design Environment," in *Proc.*

25th Design Autom. Conf., Anaheim, CA, USA, 1988, pp. 628–633.

[5] H. Chang, A. Sangiovanni-Vincentelli, F. Balarin *et al.*, “A Top-Down, Constraint-Driven Design Methodology for Analog Integrated Circuits,” in *Proc. IEEE Custom Integr. Circuits Conf.*, Boston, MA, USA, 1992, pp. 841–846.

[6] C. Han, G. Stephanopoulos, and J. M. Douglas, “Automation in Design: The Conceptual Synthesis of Chemical Processing Schemes,” in *Intelligent Systems in Process Engineering, Part I: Paradigms from Product and Process Design*, ser. Advances in Chemical Engineering, G. Stephanopoulos and C. Han, Eds. Academic Press, 1995, vol. 21, pp. 93–146.

[7] E. Malavasi, E. Charbon, E. Felt, and A. Sangiovanni-Vincentelli, “Automation of IC Layout with Analog Constraints,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 15, no. 8, pp. 923–942, Aug. 1996.

[8] B. Arsintescu, E. Charbon, E. Malavasi, and W. Kao, “AC Constraint Transformation for Top-down Analog Design,” in *Proc. Int’l Symp. on Circuits and Systems*, vol. 6, Monterey, CA, USA, 1998, pp. 126–130.

[9] E. Malavasi, E. Charbon, B. Arsintescu, and W. Kao, “A Constraint Management System for IC Physical Design,” in *Proc. 11th Brazilian Symp. on Integr. Circuit Design*, Búzios, Rio de Janeiro, Brazil, 1998, pp. 240–243.

[10] A. Krinke, M. Mittag, G. Jerke, and J. Lienig, “Extended Constraint Management for Analog and Mixed-Signal IC Design,” in *Proc. 20th European Conf. on Circuit Theory and Design*, Dresden, Germany, 2013.

[11] C. Katzschke, M.-P. Sohn, M. Olbrich, V. Meyer zu Bexten, M. Tristl, and E. Barke, “Application of Mission Profiles to Enable Cross-Domain Constraint-Driven Design,” in *Proc. Design, Autom. and Test in Europe*, Dresden, Germany, 2014.

[12] G. Jerke and A. B. Kahng, “Mission Profile Aware IC Design – A Case Study,” in *Proc. Design, Autom. and Test in Europe*, Dresden, Germany, 2014.

[13] M. Crepaldi, M. Grosso, A. Sassone *et al.*, “A Top-Down Constraint-Driven Methodology for Smart System Design,” *IEEE Circuits Syst. Mag.*, vol. 14, no. 1, pp. 37–57, 2014.

[14] N. Beldiceanu, M. Carlsson, and J.-X. Rampon, “Global Constraint Catalog,” Swedish Institute of Computer Science (SICS), Kista, Sweden, Tech. Rep. T2010:07, Nov. 2010.

[15] A. Krinke, G. Jerke, and J. Lienig, “Adaptive Data Model for Efficient Constraint Handling in AMS IC Design,” in *Proc. 20th Int’l Conf. on Electronics, Circuits, and Systems*, Abu Dhabi, UAE, 2013, pp. 285–288.