Chapter 7
Constraint-Driven Design Methodology – A Path to Analog Design Automation

Göran Jerke[1], **Jens Lienig**[2], **Jan B. Freuer**[1]
[1] Robert Bosch GmbH, AE/EIM, 72762 Reutlingen, Germany
[2] Dresden University of Technology, IFTE, 01062 Dresden, Germany

Abstract Physical design for analog ICs has not been automated to the same degree as digital IC design, but such automation can significantly improve the productivity of circuit engineers. Analog design remains difficult to formalize due to a large amount of expert knowledge involved, such as sophisticated constraints that are specified manually and satisfied through manual layout. We therefore propose a constraint-driven design methodology – a suite of algorithms and methodologies to capture key rules governing analog layouts and to produce layouts that satisfy these rules. In this chapter, we identify major challenges in analog physical design, and relate them to constraints. We introduce techniques for constraint representation and highlight the essential components of a constraint-driven design methodology. Finally, we explain how constraint-driven design impacts a typical analog design flow, layout algorithms and the overall physical design methodology.

7.1 Introduction

While physical design automation of analog IC design has seen significant improvement in the past decade, it has not advanced at nearly the rate of its digital counterpart. This shortfall is primarily rooted in the analog IC design problem itself which is very complex even for small problem sizes [7, 16, 23, 29].

The quality of a design result is generally determined by the degree to which compliance constraints have been met and pre-defined design objectives achieved. Due to the lack of uniform representation and interpretation of design constraints in the analog design flow context, most of the constraints in today's analog designs are still specified and considered manually by expert designers (expert knowledge). Furthermore, analog constraints are often used implicitly (i. e., based on a designer's experience) rather than being explicitly defined, which prevents their effective use in design automation. However, progress in physical design automation for analog ICs is urgently needed as design sizes increase, along with significant challenges

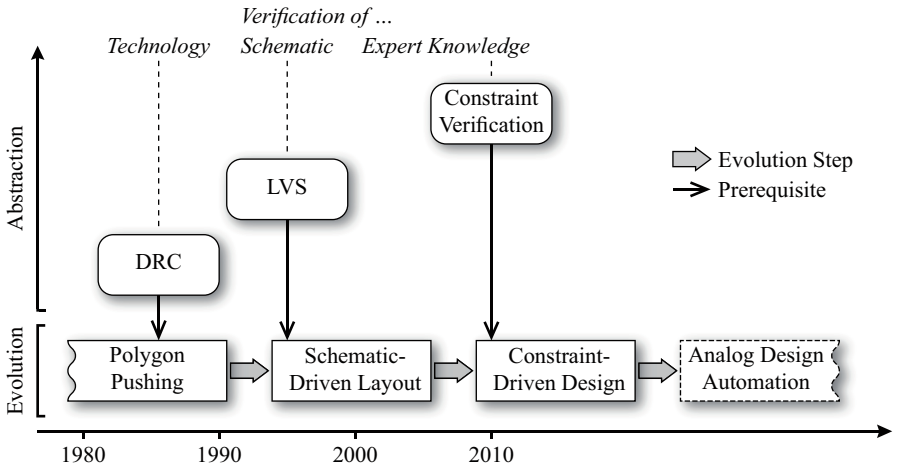

Fig. 7.1 The evolution of analog physical design methodologies, such as schematic-driven layout and constraint-driven design, towards the goal of a fully automated analog design flow.

such as increasingly stringent reliability and robustness requirements, and a rapidly widening verification gap.

Analog circuits are currently designed interactively, in terms of schematics, which are subsequently verified. Most researchers agree that this so-called schematic-driven layout (SDL) methodology will be replaced by analog design automation in the future, more in line with current practices for digital circuits. As we will show, constraint-driven IC design is both a necessary step toward full automation and also a precondition for it (Fig. 7.1).

The ultimate goal of fully automated analog design (analog design automation) can only be achieved if the schematic-driven design paradigm evolves into a constraint-driven design paradigm. This is based on the belief that we first need a methodology that allows for automatic inclusion of expert knowledge in the form of constraints, which also must be verified automatically. Only then one is able to tackle the task of analog layout synthesis in a comprehensive and consistent manner. In other words, the abilities of "analyzing" and "verifying" are a precondition for "synthesizing" [30].

This chapter provides an introduction to the concept of a constraint-driven physical design approach for arbitrary ICs in general, and for analog ICs in particular. First, we identify key similarities and differences between the physical design of analog and digital circuits, and the corresponding challenges, which we show are primarily constraint-related (Section 7.2). We discuss the constraint representation and classification in Section 7.3 and give an overview of the constraint-driven design flow and its essential components in Section 7.4. Here, we introduce fundamental components required in this flow, such as constraint representation, management, transformation and verification. The application and resolution of constraints, through constraint engineering, is discussed in Section 7.5. In Section 7.6, we then present the impact this methodology has on the overall IC design flow,

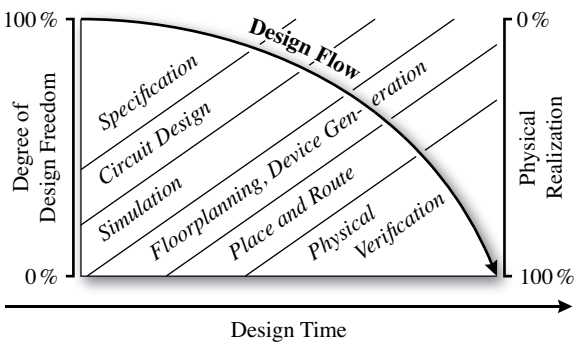

Fig. 7.2 Simplified design flow for analog IC design where design steps are typically overlapping. Multiple design steps are active at the same point of time [30].

the core design of design automation algorithms, and the required paradigm adjustments needed for analog physical design approaches. The chapter concludes with an anticipatory look at open problems (Section 7.7).

7.2 Problem Description

7.2.1 The Design Problem

In general, any (IC) design problem represents a complex and constrained optimization problem. The degrees of design freedom linked to the optimization problem span a multi-dimensional solution space which is at least partially constrained by the given global design constraints. A feasible solution for a specific design problem is obtained by sequentially removing all degrees of design freedom while traversing and reducing the solution space and considering all context-relevant constraints and application profiles.

This reduction is done by sequentially transforming functional representations with many degrees of design freedom into equivalent ones with fewer degrees of design freedom. For example, using suitable methods one may transform a given functional specification into a netlist[1], which is subsequently transformed into a floorplan, a placement order, a wired layout and finally into a physical mask layout[2] which contains no further degree of design freedom.

Several functional transformations (design steps) can be active at the same time during analog IC design (Fig. 7.2). The strategy of how and when to remove a degree of design freedom during the design phase depends on several specific factors in the design context. Among others, these factors may include the type of IC application, its usage profiles, reliability and robustness requirements as well as the current problem situation in a design phase with its linked constraints (design context).

[1] Functional representation of the given specification.

[2] Functional representation of the given netlist.

In general, design constraints *must* be fulfilled whereas design objectives *may* be fulfilled. A design objective that must be fulfilled hence represents a constraint, and must be treated as such. Similarly, any given design constraint that may be fulfilled should be considered as an design objective. The design goal is to achieve design results that fulfill all given constraints and which offer the highest level of achievement toward pre-defined design objectives.

7.2.2 Analog Versus Digital Design Automation

Analog IC designs often contain only a small number of devices as compared to digital IC designs. Nevertheless, the effort required to design analog function modules often matches or even exceeds the effort for digital modules. This is mainly due to a much richer set of constraints that must be considered simultaneously (Section 7.3).

On average, each design object (instance, net, path, etc.) in an analog IC design must comply with a larger and more extensive set of constraints to fulfill its intended function (compared to digital design). The primary reason for this observation is the higher level of functional abstraction offered in digital designs. This allows digital designs to use fewer top-level constraints to guarantee a robust function.

Furthermore, the majority of constraints may yet be unknown when the analog design process begins. This renders automatic top-level design planning for analog IC designs nearly impossible. It is one of the reasons that highly skilled design engineers are still required to perform top-level design planning manually.

This constraint-related problem also makes algorithm and tool development for analog IC design much more difficult because the number of specific design algorithms may increase with each new type of constraint. Considering today's conventional design-algorithm development approach (one type of constraint and one algorithm to handle it), this approach falls short when it comes to linked constraints (Section 7.3). This represents one of the primary reasons why analog design automation is lagging behind its digital counterpart and why this gap is currently still growing.

Another important reason for the design gap is rooted in the level of completeness and consistency that can be applied to the consideration of constraints during IC design. Today's digital design tools already offer consistent and seamless design solutions. This is mainly due to their focus on a small set of various types of constraints, such as delay and clock skew. A unified description of constraints is not used in today's analog design tools and algorithms[3]. A common understanding of design implications due to constraints is not guaranteed with existing approaches. Hence, many analog constraints must still be considered manually or semi-automatically leading to their often inconsistent and non-comprehensive consideration.

Any inconsistent or non-comprehensive consideration of constraints widens the existing constraint verification gap. This gap exists because the design rule check

[3] If not stated otherwise, the term "design algorithm" is subsequently used for both, design tools and their built-in algorithms due to their close relationship.

(DRC) and the layout versus schematic check (LVS) do not include the verification of all constraints. A tremendous amount of research effort has already been expended for the tailored consideration and verification of special types of constraint, such as signal delay, device matching and IR-drop. Nevertheless, a unified approach capable of dealing with all constraints during the entire design and verification phase is still missing.

Another difference between analog and digital IC designs is found in the way the functional transformations, i. e., the design steps, are linked and carried out. While most steps in digital IC design are separated from each other, the design steps of analog ICs are typically overlapping, and hence, tightly linked due to the impact of analog constraints (Fig. 7.2). For example, device generation, pre-placement and global routing usually occur simultaneously during the floorplanning phase of analog ICs. Analog design algorithms must thus consider various types of constraints simultaneously. This greatly reduces the impact of specialized design algorithms that handle only a small set of types of constraints.

In order to address the current shortcomings discussed in this section, a constraint-driven design approach is required that considers constraints in a comprehensive and consistent manner. Its cornerstones will be introduced in Sections 7.4 – 7.6.

7.3 Constraint Classification and Representation

Constraints for IC design (hereafter, constraints) are classified by their complexity, category, form and type. The classification criteria are discussed in this section.

From a formal point of view, constraints define relations between values of design variables (hereafter, variables). A relation between independent variables represents a simple constraint. Relations between dependent variables are denoted as complex constraints (Fig. 7.3). Constraints for IC design are linked to design objects which represent data objects in the database of a design tool, such as cell, cellview, instance, net, terminal.

In general, constraints belong to one of the following four categories:

- *Technology* constraints enable manufacturing for a specific technology node (e. g., wire width, spacing, layer thickness).
- *Functional (electrical)* constraints ensure the intended IC functionality (e. g., maximum IR-drop between two net terminals, minimum gain, maximum offset voltage).
- *Design methodology (geometry)* constraints reduce the overall complexity of the design process. They also guide transformations, enforce a specific design pattern or describe a context to which other constraints are associated to (e. g., maximum design hierarchy depth, maximum number of devices in a cluster, pre-defined layer for power-routing, bus width).
- *Commercial* constraints (e. g., maximum die area, number of layers).

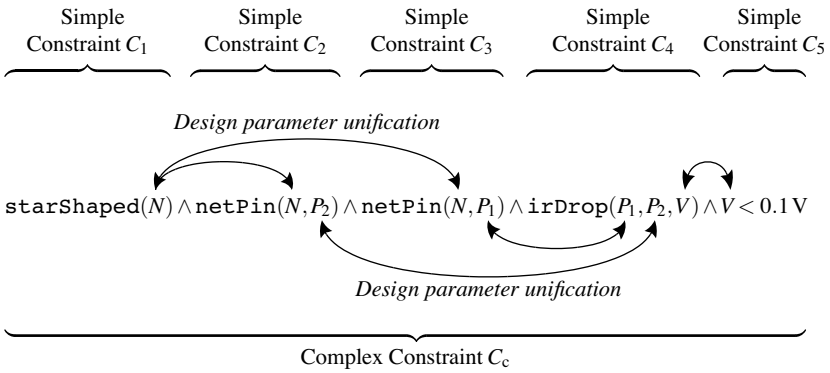

Fig. 7.3 Four simple constraints (`starShaped`(N), `netPin`(N,P_2), `netPin`(N,P_1), `irDrop`(P_1,P_2,V), and $V < 0.1\mathrm{V}$) form a complex constraint C_c through a conjunction defined in constraint logic programming (CLP) notation. The complex constraint C_c is satisfied if all four simple constraints are satisfied. The simple constraints are tightly coupled through the design parameters N, P_1, P_2 and V that must be substituted (unified) to resolve C_c.

A constraint is given in either an implicit or explicit form. An implicit constraint is not clearly expressed and may be given as plain textual note or may arise from assumptions intrinsically built into circuit descriptions or design algorithms. Implicit constraints represent non-formalized design knowledge. Contrary to implicit constraints, explicit constraints are clearly expressed and represent formalized design knowledge. Examples of implicitly defined constraints are the placement requirements of differential pair transistors – they must be placed symmetrically in order to maximize device matching. While this is obvious to any layout designer, the inclusion of such complex rules into both layout and verification tools is often not possible for applications that contain additional requirements, such as parasitic interconnect matching. Hence, due to its non-formal nature, implicit constraints cannot be utilized for any type of controlled and automated constraint-driven design. On the other hand, explicitly defined constraints are accessible to design algorithms and thus are a primary requirement for any constraint-driven design flow.

Each constraint belongs to a specific constraint type that represents a classification property for the same class of constraints. The type of a constraint always corresponds to the type of the corresponding design variables. Constraint types have a clearly defined physical, electrical, mechanical, mathematical or geometrical unit (e. g., the constraint type “IR-drop” has the unit Volt, the type “signal delay” the unit Seconds). The relevance and impact of a constraint type strongly depend on the specific design context.

In order to formalize design constraints, all constraints and all related design variables must be uniformly represented in an abstract form. The conversion of constraints into a uniform representation must be complete and unambiguous. An uniform representation enforces a common understanding of constraints among all involved design algorithms. Hence, it is a primary requirement for addressing the analog (constraint) design problem [11, 26]. Constraint Logic Programming

(CLP) [8, 19] embodies a feasible approach for uniform constraint representations. In CLP, constraints are defined in the body of conditions (clauses) (Fig. 7.3). All constraint examples discussed in this chapter are based on the CLP notation.

Assume an IR-drop constraint $V_{\mathrm{IR}}(P_1, P_2) < 0.1\,\mathrm{V}$ stating that the IR-drop between two layout pins P_1 and P_2 must be less than 0.1 V. This functional constraint is simple since it is completely independent from any other constraint. If this example is transferred to a more formal representation such as CLP, the IR-drop constraint must be written as a relation between design parameters. A possible representation is the relation $\texttt{irDropLessThan}(P_1, P_2, 0.1)$. However, this approach is very restrictive. For example, neither equality nor any other inequality can be expressed. In order to obtain a more general representation, it is advisable to split this constraint into a conjunction of a functional and an arithmetic constraint $\texttt{irDrop}(P_1, P_2, V) \wedge V < 0.1$ with V representing the actual IR-drop between pins P_1 and P_2.

The IR-drop between two net pins P_1 and P_2 is usually considered within a specific design context, in our case the net N which owns both pins. This introduces two structural constraints $\texttt{netPin}(N, P_1)$ and $\texttt{netPin}(N, P_2)$. Additionally, if the IR-drop needs to be considered only for nets with, for instance, a star-shaped layout topology, another structural constraint $\texttt{starShaped}(N)$ must be added. Figure 7.3 depicts the conjunction of these constraints that form the complex constraint C_{c}. The coupling of the simple constraints is obtained via substitution (unification) of the design variables N, P_1, P_2 and V (Section 7.5.1).

7.4 Components of a Constraint-Driven Design Flow

A design flow that considers all relevant constraints in a consistent and comprehensive manner is subsequently denoted as constraint-driven design flow. This flow requires several complementary design flow components that are shown in Fig. 7.4.

Constraint management provides the management of constraint data and the assignment of constraints to design objects (Section 7.4.1). In order to obtain design results meeting their specification, constraints are derived from design objectives (*constraint derivation*, Section 7.4.2). Constraints are transformed between the physical, electrical or geometrical domain to be suitable for design algorithms in a particular design context (*constraint transformation*, Section 7.4.3). The *constraint sensitivity analysis (CSA)* determines the sensitivity of a design parameter with respect to related constraints. The CSA finds the most constraint-sensitive design parameters in a particular design context. Constraint sensitivity information can then be used to guide the design generation (Section 7.4.4). Finally, despite the use of a constraint-driven layout generation, the compliance of a design result with its given constraints must be verified using *constraint verification* (Section 7.4.5).

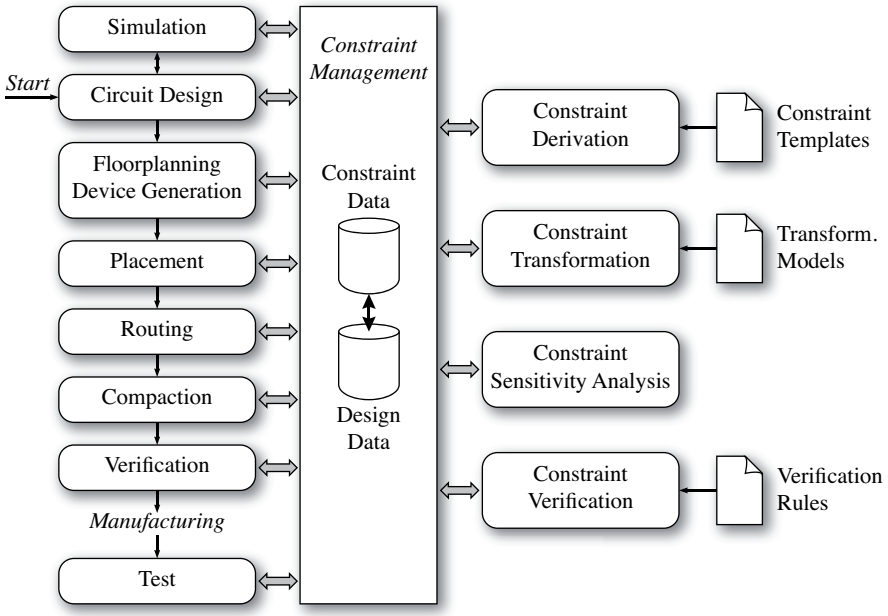

Fig. 7.4 Essential components of a constraint-driven design flow.

7.4.1 Constraint Management

The task of constraint management is to administer the storage of constraint data while synchronizing the link between constraints and design objects. The management system must also guarantee the semantic integrity of the constraints across different levels of abstraction, and support hierarchical relations between design objects and dependencies [6, 22]. Additionally, it is responsible for keeping constraints consistent and valid, which requires close interaction with design databases as well as with constraint and design data manipulating design algorithms. Furthermore, constraint-driven design algorithms require fast access to constraint information through (standardized) application programming interfaces.

The detection of over-constraints is an important sub-component of a constraint management system. It is made available by the constraint verification (Section 7.4.5). Over-constraints represent a condition in which not all given constraints can be fulfilled simultaneously. The related formal mathematical problem is denoted as constraint satisfaction problem (CSP). Over-constraints must be resolved by constraint satisfaction methods, such as constraint propagation, constraint relaxation or backtracking, in order to obtain feasible design results [20]. The use of constraint weights as a decision criterion to resolve over-constraint conflicts is a common approach. However, this method is likely to become unsuitable if the number of constraints increases since many constraints may have equal or similar weights, thus making them unusable as decision criteria.

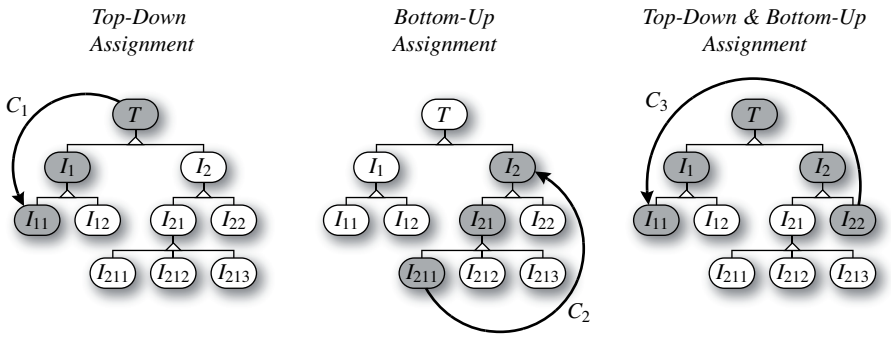

Fig. 7.5 Assignment of constraints to design objects in a design hierarchy tree. In this example, T represents a top cell incorporating several cellview instances I_1–I_{213}.

Constraint management also incorporates the assignment of constraints to design objects (i) in the existing design hierarchy, (ii) across the extent of design objects and design steps, as well as (iii) within a design hierarchy that is defined by design objects and linked constraints (virtual design hierarchy). The use of these assignment options strongly depends on the specific constraint. Furthermore, constraint assignment is either permanent or temporary depending on the particular design context.

The assignment of constraints within a hierarchical design can be either performed top-down, bottom-up or combined top-down-bottom-up (Fig. 7.5). For instance, a net shielding constraint may be assigned from the I/O pad in the top cell down to a specific instance terminal in a sub-cell (top-down assignment). The shielding constraint is then assigned to all connected net objects in the design hierarchy.

The assignment can also be performed across the extent of design objects, such as instances. Using the previous example, cellview instances (e. g., metal resistors) must be skipped if the net shielding constraint is to be assigned to all nets that are physically connected on the chip mask. The net shielding constraint is then also hierarchically assigned to all sub-nets that would connect to the main net if the metal resistors were shorted.

In case the I/O pad is located in a sub-cell, then the shielding constraint must be assigned to all connected lower level nets as well as to all higher level nets that are connected to the I/O pad cell's instances (top-down and bottom-up assignment). Here, net shielding constraints are assigned within a virtual design hierarchy that is defined by the I/O pads' location in the design hierarchy tree and the hierarchical connectivity of the nets to be shielded.

During top-down assignment of a single constraint, only one constraint is assigned to each related design object in the cellviews that are traversed in the design hierarchy tree. In contrast, the bottom-up assignment allocates as many constraints in the design hierarchy tree as instances of that cellview exist in the flattened design hierarchy, making it computationally more expensive.

7.4.2 Constraint Derivation

The process of deriving constraints from design objectives is denoted as constraint derivation or constraint generation. Design objectives are given as specification goals or requirements that must be met, but they can also arise from a local design context.

Design objectives are translated into constraints using (i) derivation rules, (ii) deduction processes based on logic calculus or (iii) the designer's expert knowledge. The first two derivation methods can be applied with a high degree of automation in case the IC specification is given in a computer-processable form, such as an executable specification. The derivation process creates constraints belonging to the technology, functional, design methodology, or commercial constraint category (Section 7.3).

The rule-based derivation of constraints utilizes a fixed rule to transform a design objective into a set of constraints while considering the particular design context. The constraint transformation discussed in Section 7.4.3 is a form of indirect constraint derivation since it creates lower level constraints that depend on higher level constraints.

Deduction-based constraint derivation can be seen as a high-level extension of rule-based derivation methods. Here, a logic reasoning system draws conclusions from design and constraint data and then applies a set of constraint derivation rules to relevant design objects. For example, based on a logical conclusion that MOS and bipolar transistors both belong to the same category of devices "transistor", a specific constraint rule may be applied to both MOS and bipolar transistors, even in the case where the derivation rule was only defined for one transistor type. This functionality permits the development of high-level constraint derivation methods and offers an important level of abstraction required for the reuse of analog blocks.

Expert knowledge is still often required to translate critical design objectives into constraints. This is especially the case for global design objectives that would result in various sets of complex constraints and cannot be easily resolved by automatic rule-based approaches. Unfortunately, the expert knowledge only exists in an unstructured and non-formalized form. Nevertheless, making expert knowledge more accessible represents a good starting point for further analog design automation.

7.4.3 Constraint Transformation

Constraint transformation translates higher level constraints into a set of equivalent lower level constraints and vice versa (inverse constraint transformation) using transformation rules [21]. Multiple transformation rules may apply for a specific higher level constraint resulting in different sets of lower level constraints. The choice of an appropriate transformation rule inherently constrains the solution space, thus reducing the number of global degrees of design freedom.

The choice of a transformation rule depends on the particular design problem and design context. Any transformation process must ensure a complete and unambiguous transformation result. The same applies to the inverse constraint transformation which must be defined for constraint verification purposes (Section 7.4.5).

The transformation of constraints is based on a particular transformation model which is translated into a set of transformation rules. Transformation rules for simple constraints are represented by independent equations. They contain the involved design variables in the higher transformation level and the variables in the lower level. Transformation rules for complex constraints are represented by a set of coupled equations containing all coupled design variables.

The relation of sub-constraints specific to each complex constraint type is not affected by the transformation since the transformation of simple constraints only focuses on their specific context. This statement is made here since it is assumed that any transformation will only produce lower level constraints that do not affect higher level constraints. In the case where lower level constraints affect higher level constraints, design iterations are very likely to occur (i. e., the design steps must be reversed and re-done with another design strategy).

In general, more than one transformation rule may exist for a particular type of constraint. The decision which transformation rule to use is specific to the design context, the design algorithm and the applied design strategy. For example, suppose the functional specification of a circuit results in a specific maximum IR-drop between an I/O pad and a specific instance terminal in a sub-cell. Assuming that the current flow in the interconnect is known, the transformation of the IR-drop constraint may result in constraints for I/O pad and sub-cell placement and a corresponding set of routing constraints. A constraint-driven design algorithm can then decide whether the placement in this context is more critical to deal with than the routing and act accordingly (see also Section 7.4.4). For instance, in case the placement is fixed, the final transformation of the given IR-drop constraint would then yield a set of routing constraints and local degrees of design freedom (i. e., routing design parameters such as wire length, layer, wire width). These can then be used by a routing algorithm to find a suitable interconnect layout.

7.4.4 Constraint Sensitivity Analysis

Constraint sensitivity analysis (CSA) determines the context-specific sensitivity of numerical design parameters with respect to related constraints. The CSA consists of two modules: a module that determines sensitivity of design parameters with respect to output parameters and a module that determines the relative distance of a design parameter value to its related constraints. Both modules provide valuable information that can be utilized by designers and by design algorithms.

The sensitivity analysis is based on a mathematical model which describes the physical, electrical or geometrical nature of a particular design sub-problem. The model represents an equation system that contains all relevant design parameters

and output parameters. Several approaches are reported to determine the sensitivity of design parameters. Among these approaches, local methods based on the partial derivatives of the model output parameter and statistical methods based on sampling, Bayesian and Monte Carlo methods are the most important ones [4, 18].

Considering a set of constraints $x_\mathrm{l} \le x \le x_\mathrm{u}$, the relative distance d of design parameter value x to a lower constraint boundary x_l and an upper constraint boundary x_u is determined as follows:

$$d_\mathrm{l}(x) = \exp(x_\mathrm{l} - x) - 1 \quad \text{and} \quad d_\mathrm{u}(x) = \exp(x - x_\mathrm{u}) - 1 \; . \tag{7.1}$$

The parameter value x matches with the lower bound constraint value if the relative distance $d_\mathrm{l} = 0$. A constraint violation is detected in (7.1) if $d_\mathrm{l} > 0$ while no violation occurs if $d_\mathrm{l} < 0$. The same applies to d_u while considering the upper bound constraint value.

Design decisions can be made by design algorithms based on the sensitivity information of parameters, the relative distance of parameter values to related constraints and a given design strategy. Design algorithms may use that information in several ways. Depending on the design strategy, a design algorithm may point its focus to the fixation of design parameters with a high sensitivity towards an important output parameter or it may focus on low sensitivity parameters. The information about the parameter distance lets the design algorithm recognize the severity of constraint violations. For example, design parameters violating related constraints may then be considered with a higher priority.

It is also of interest for a design algorithm to know which design sub-problems are independent from each other. A low sensitivity of design parameters towards a common output parameter means that they are weakly coupled with respect to that output parameter. The sensitivity analysis can be used as a method to identify local design task parallelism by searching for groups of design parameters and constraints that are either not or only weakly coupled. They can be dynamically partitioned into independent groups for which the next design step can then be performed independently from each other.

An example of a CSA application is given in Fig. 7.6. Here, a constraint sensitivity analysis is applied while routing a wire closely located to a heat source (e. g., a power transistor). Given an IR-drop constraint $V_\mathrm{IR} \le V_\mathrm{IR-max}$, a design decision has to be made whether to move the wire away from the heat source, thus varying the interconnect temperature T, or to fix the wire width w. The design parameters and the constraint in Fig. 7.6 are denoted as follows: wire width w, length l, thickness d, reference temperature T_ref, IR-drop constraint $V_\mathrm{IR} \le V_\mathrm{IR-max}$, $V_\mathrm{IR} = i \cdot \rho \frac{l}{w \cdot d} \cdot (1 + \mathrm{TK}_1 \cdot (T - T_\mathrm{ref}))$, DC current i. A constraint violation is likely in case T is varied while $w \approx w_1$, whereas it becomes less likely in case $w > w_1$. In order to avoid an IR-drop constraint violation, the modification of the design parameter w is the primary choice if $w \approx w_1$ due to its high local sensitivity related to the output parameter V_IR while w looses its impact for $w \gg w_1$. If CSA is used as a filter to find all sensitive design parameters, then w is only required to be considered if $w \ll w_2$.

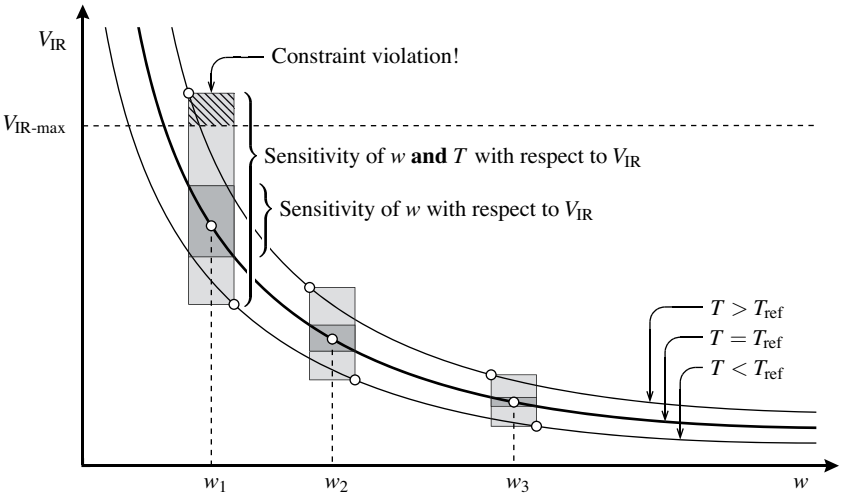

Fig. 7.6 A constraint sensitivity analysis is applied to parameters of a wire segment that is closely located to an on-chip heat source. A constraint violation is likely in case the interconnect temperature T is varied (by moving the wire's location) and a wire width w with $w \approx w_1$, whereas it becomes less likely in case $w > w_1$. In order to avoid an IR-drop constraint violation, the modification of w is the primary choice if $w \approx w_1$ due to its high sensitivity towards V_{IR} while w looses its sensitivity for $w \gg w_1$. (See text for parameter denominations and further explanation.)

The CSA allows designers to study the impact of local design decisions and to trace root causes in case compliance requirements cannot be met by the given set of constraints. Sensitivity analysis is the key to the power of decision analysis in situations where the influence of design parameters is not known precisely, since it considers the design context in which constraints apply. As is obvious from this explanation, the availability and application of the CSA allows new approaches for algorithm development and analog design automation.

7.4.5 Constraint Verification

Constraint verification comprises the verification (i) whether a set of constraints is fulfilled for a particular design result and (ii) if a given set of constraints raises mutual conflicts (over-constraint, Fig. 7.7). Constraint verification represents a key component of the constraint-driven design flow. This is due to its formidable contribution to reduce the verification gap discussed in Section 7.2.2. Constraint verification ensures correct application functionality, and it is essential to improve design quality, reliability and robustness. Commercially available constraint verification tools with yet limited verification capabilities currently comprise Mentor Graphics

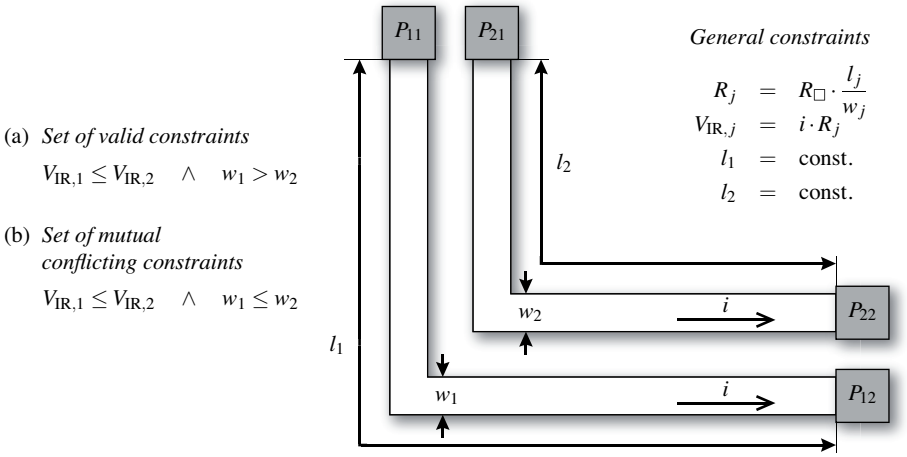

Fig. 7.7 Constraints illustrated in a two-net topology. A DC current i is present in both wires leading to a static IR-drop voltage V_{IR}. While the set of constraints in (a) is feasible, the two constraints in (b) are mutual conflicting (over-constraint). Here, a smaller IR-drop within one of two wires cannot be achieved if this wire is not allowed to be wider than the other one.

Calibre® PERC [24] and the constraint verification engine integrated into Cadence Virtuoso® IC 6.1 [5].

As mentioned earlier, a rich set of constraints must be considered during the design of analog ICs. A significant fraction of these constraints are complex constraints whose fulfillment cannot be verified with conventional verification approaches. This is due to the fact that all of today's verification approaches require one specific verification algorithm for each type of constraint. Clearly, conventional constraint one-to-one verification approaches (one verification algorithm for one type of constraint) are not feasible for the complete verification of analog IC designs. Making matters worse, many constraints (and constraint types) are still unknown at the beginning of the design process.

An approach to address the verification problem for complex constraints, the "meta-verification approach", was introduced in [11] and is discussed in more detail in Section 7.5.2. The core idea of meta-verification is that each complex verification problem can be subsequently resolved into smaller and usually independent verification sub-problems. These sub-problems can then be addressed using existing verification algorithms. The meta-verification references functionality accessible from external tools (e. g., design data access or specialized verification functions offered by a particular tool) to perform verification tasks. The meta-verification framework creates an abstraction layer around multiple design and verification tools, and it manages correct execution of the defined meta-verification tasks.

The CLP-based verification approach in [11] is capable to address independent as well as coupled, i. e. dependent verification problems. It also allows the detection of mutual constraint conflicts (over-constraints) by drawing logical conclusions from

the given constraint and design data information. The approach is described in more detail in Section 7.5.

The definition of verification tasks for a meta-verification system to check constraint compliance is generally done as follows. First, the constraint verification task is defined and formalized. The formal description of a verification problem is then translated into a set of constraint verification rules. Finally, the verification rules are used by circuit and layout designers to perform constraint verification tasks. The application of these rules may depend on the design context of the particular constraint verification problem.

Significant effort must be spent by PDK developers and designers to develop, optimize and verify the set of rules for constraint derivation, transformation and verification. The sequence in which sub-verification tasks are processed has a significant impact on the required overall time for constraint verification. For example, suppose there are short-running and long-running sub-verification tasks defined in a specific CLP-based meta-verification rule. If feasible for a particular verification task, it is beneficial to shift all long-running sub-verification tasks to the end of that rule in order to execute them later than the short-running sub-verification tasks. Sub-verification tasks are not executed if a previous sub-verification task of a rule already revealed constraint violations. This approach will effectively prevent unnecessary and potentially long-running sub-verification tasks from being executed. As obvious, verification rule development and optimization requires a deep understanding of the underlying verification task.

Practical application of the meta-verification approach has revealed that the required initial effort is comparable to the effort needed for the development of DRC and LVS rule sets [11]. The reuse of rules for constraint derivation and meta-verification is simple and efficient since, in general, data and rule abstraction can be used for technology, design and constraint data (Section 7.5).

Constraint verification is divided into static and dynamic constraint verification, based on the constancy of the constraint and design data. The corresponding constraint satisfaction problems (CSP) which are to be solved are denoted as static CSP and dynamic CSP [15]. For example, any sign-off verification of an IC design must be based on constant design and constraint data, hence, static constraint verification is applied in this case. Nevertheless, constraint-driven design algorithms can also use constraint verification for specific "what-if" analyses. Since these algorithms can change design and constraint data during their analyses and during the design step, the related constraint verification is based on dynamic data. Hence, the latter case represents dynamic constraint verification. Both, static and dynamic constraint verification can be applied to either the full set of constraint and design data, or to a design-context specific sub-set.

The required overhead for static constraint verification is typically significantly smaller compared to dynamic constraint verification. The additional overhead in the latter case is primarily caused by a cumulative data latency effect that occurs if design and constraint data are frequently accessed by design algorithms and/or the verification framework. Hence, low-latency access to design and constraint data will significantly speed-up dynamic constraint verification. For static constraint verifica-

tion, design and constraint databases are usually accessed only once during initialization, thus mostly avoiding data access latency issues.

7.5 Constraint Engineering

The application and handling of constraints during the IC design process is denoted as constraint engineering. In this section, we first provide a brief overview of computational approaches to address the constraint resolution problem (constraint programming). We then introduce the constraint engineering system (CES) which represents a framework that combines several constraint programming approaches in a single software framework. This framework integrates the previously discussed design flow components into a unified design environment which facilitates the interoperability between these components. It also increases the ability to perform design tasks on a higher level of abstraction, thus enabling new possibilities for analog design automation.

7.5.1 Constraint Programming

Constraint programming represents a programming paradigm where relations between (design) variables are stated in the form of constraints. These relations form a constraint satisfaction problem which is resolved by constraint solvers.

The resolution of constraints usually occurs when multiple constraints can be simplified or when the existence of one or more constraints leads to new (lower level) constraints. The constraint engineering uses specialized constraint solvers to handle all aspects of the constraint handling. The specialization is required since the handling strongly depends on the domain (or type) of the constraints. For example, a boolean constraint must be handled differently than a constraint that is defined over real numbers. The solving of arithmetic constraints, for instance, highly depends on the constraint complexity, such as linear or polynomial. Different constraint-solving approaches exist that are tailored to address various constraint satisfaction problems [3, 31].

As mentioned before, the formal constraint representation is a key requirement for a constraint-driven design flow. A formalism is required in order to describe the interaction of constraints which are mainly the constraint derivation (Section 7.4.2) and constraint transformation (Section 7.4.3).

There are many approaches where constraints have been integrated into traditional programming languages [1, 10, 17, 27]. Due to the stateless character of constraints, the family of constraint logic programming (CLP) languages [8, 9, 32] is the natural choice not only for the formalization of constraints. The declarative logic calculus approach in CLP has also the advantage that only the problem has to be formalized but not its solution. Compared to other constraint programming approaches,

the application of CLP significantly reduces the effort needed to provide the required rules for constraint derivation, transformation and verification. Therefore, the core of the constraint engineering system discussed in the next section is based on a logic calculus engine (Fig. 7.8).

With the introduction of constraint handling rules (CHR) in 1991 [12], it became very easy to define new constraint solvers that are perfectly tailored to a specific constraint problem. Via CHR, new constraint solvers can be defined through two different kinds of handling rules. The propagation rule creates one or more constraints from a given set of constraints. Assume for instance the less-equal constraint "≤". If there already exist two constraints $A \leq B$ and $B \leq C$, a suitable propagation rule would derive the constraint $A \leq C$. The second rule represents a simplification rule that removes one or more constraints from a given constraint set. Regarding the previous example, assume that there are two constraints $A \leq 5$ and $A \leq 7$. The simplification would remove $A \leq 7$ since it is overridden by $A \leq 5$. Due to efficiency reasons, there is usually also a third rule in CHR, the "simpagation". It combines simplification and propagation within a single rule.

7.5.2 The Constraint Engineering System

The application of constraint engineering is an important step towards a constraint-driven design flow. A flexible software architecture is required to integrate the new design flow components introduced in Section 7.4. Our approach of such an architecture will be subsequently denoted as constraint engineering system (CES) whose structure is depicted in Fig. 7.8 [11].

The CES is designed to act as a middleware between various design tools that offer an accessible application programming interface. The CES core engine is capable of making logical decisions based on multiple knowledge bases, which are provided from various external sources.

As shown in Fig. 7.8, the CES is based on a plug-in architecture that allows the flexible extension of its functionality. An extension point of the CES regards the access to all design tools that are accessible within the existing design flow. A translation layer, denoted as *Tool Integration Kit* (TIK), transforms the tool-specific data into logic calculus knowledge using CLP language so that it can be accessed by meta-verification. Vice versa, the TIK also provides the functionality of transferring data from the meta-layer back to the connected design tool. This allows the back-annotation of constraints that were processed in the CES to an external design tool. A TIK also enables a high-level access to the functionality of a design tool which can then be utilized by particular design algorithms or the constraint verification. For example, a schematic entry editor provides access to netlist (design) data, and a DRC tool provides the functionalities to merge polygons and to measure the distance between the edges of two layout polygons. Since every external tool is very unique in its functionality and the design data it processes, a specific TIK is required for each connected design tool.

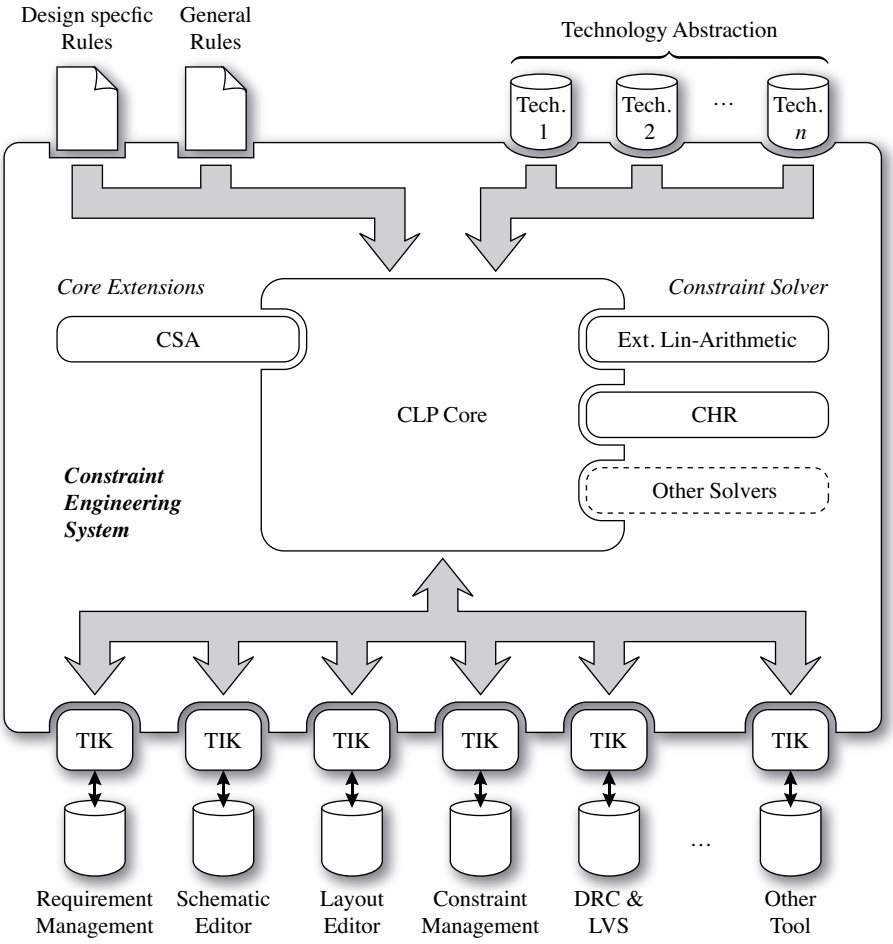

Fig. 7.8 Architecture and data flow of the constraint engineering system (CES). The tool integration kits (TIK) transform tool-specific data into the CLP language and vice versa.

Another extension point of the CES regards its internal handling of constraints. The CES enables the integration of arbitrary constraint solvers that are directly connected to its CLP core. The standard solver currently considers linear arithmetic constraints and non-linear constraints that can be subsequently reduced to linear constraints. This solver is very efficient due to the use of the simplex algorithm.

Additionally, the meta-verification rule developer can define new constraint solvers via CHR. The flexibility of CHR allows the definition of reusable solvers that are highly tailored to a specific constraint satisfaction problem. If neither the extended linear constraint solver nor the definition of new solvers via CHR lead to a suitable solution, new constraint solvers can be added via this extension point to the CES core. It is for instance expected that the resolution of polynomial and statistical

constraints within CHR would not lead to constraint solvers that are efficient enough to handle complex constraint problems of that domains. Hence, specific solver could be added that resolve these constraint problems more efficiently.

Constraint compliance is the main matter of interest in a CES application. As previously mentioned, meta-verification ensures that all complex constraints are fulfilled by the design result. The definition of meta-verification rules within the CES is simple. Figure 7.3, where several simple constraints form a complex constraint, can be used as an example.

It is advisable for the demonstration to slightly modify the complex constraint C_c in Fig. 7.3, so that all star-shaped nets within an IC design can be reported whose IR-drop between two pins is greater than a maximal allowed IR-drop $V_{\text{IR-max}}$. The following CES meta-verification rule depicts the definition of such a deduction using CLP:

```
starShapedIRDrop(P1, P2, V, Virmax) :-
   starShaped(N), netPin(N, P1), netPin(N, P2),
   irDrop(P1, P2, V), V > Virmax.
```

The predicate `starShapedIRDrop(P1,P2,V,Vmax)` encapsulates C_c so that it can be reused for other verification purposes. In order to obtain all pins of star-shaped nets that do not meet the criterion $V_{\text{IR}} \leq 0.1\,\text{V}$, the following query is to be submitted to the CES:

```
starShapedIRDrop(P1, P2, V, 0.1).
```

With that query, the CLP core tries to find suitable bindings for the unbound variables `P1`, `P2`, and `V`. If a solution is found, the CES reports a tuple consisting of two pins and the actual IR-drop between these pins. The search can be continued until all solutions, i. e. star-shaped nets violating the IR-drop constraint, are found.

The example of the complex constraint C_c in Fig. 7.3 demonstrates the application of constraints that originate from different external sources. The simple constraint C_4 instruments an external tool that is capable of computing the IR-drop between two given pins in a net layout. From the verification point of view, C_4 is a standard relation like the other constraints of this example. The CES then forwards the calculation of the IR-drop to an external IR-drop calculation tool. The transformation of parameters and result of the evaluation are performed by the TIK of this tool. The same applies to all other constraints with the difference that C_1, C_2 and C_3 originate from a layout editor tool. Finally, the constraint C_5 is evaluated by the build-in arithmetic constraint solver.

Regarding the constraint sensitivity analysis example illustrated in Fig. 7.6, the sensitivity of the wire width w and the temperature T can be determined with the CLP example below. In order to enable the CSA, the sensitivity variables need to be limited. The temperature T in this example should range from 218 K to 448 K and the wire width w from $0.18\,\mu$m to $2.0\,\mu$m. These ranges are added as additional constraints to the temporary constraint list.

```
{T>=218, T<=448, W>=0.18e-6, W<=2e-6}
  @ csa(V,[W,T],[SW,ST]).
```

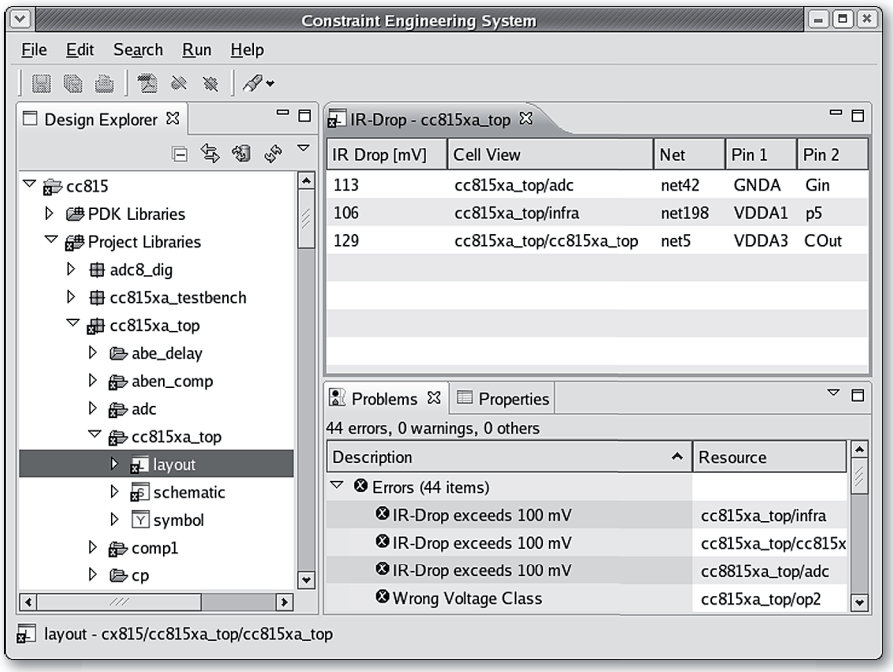

Fig. 7.9 The tabular result of the IR-drop constraint verification applied to an IC design. The shown star-shaped nets contain pin-to-pin connections having an IR-drop $V_{\mathrm{IR}} > 0.1\,\mathrm{V}$.

The `csa` predicate performs the actual sensitivity analysis. The first argument denotes the target function represented by the variable `V` ($= V_{\mathrm{IR}}$), the second a list of variables for which the sensitivity has to be determined (`W` $= w$ and `T` $= T$), and the last argument the resulting list of normalized sensitivity coefficients.

The CES provides a graphical user interface that simplifies the practical work with meta-verifications. The graphical user interface provides a uniform access to all meta-verification runsets that are associated with an IC design. Queries can be task-centric chosen and executed by a designer from the user interface. This releases the designer from the burden to manually specify verification queries. Figure 7.9 depicts the result of the previously described `starShapedIRDrop` query that has been applied to an IC design.

Constraint transformation, assignment and derivation can be directly obtained by providing assignment and transformation rules using CLP and CHR. The CES regards the reuse aspect such that these rules can be applied to multiple IC designs. The CES also supports multiple process technologies by providing specific technology properties as well as an abstraction of technology properties.

7.6 Impact Analysis

In this section, we discuss the impact of an automated constraint-driven approach on the overall IC design flow, the core design of algorithms used for design automation and the required paradigm adjustments for analog physical design.

7.6.1 Impact on Design Flow

A holistic approach to analog design automation requires several new design flow components to enable an automated constraint-driven IC design. The components of the constraint-driven design flow, such as constraint management, derivation, transformation, constraint sensitivity analysis and verification have been introduced in Section 7.4. Hereafter, they will be denoted as "new design flow components" whose impact on the analog IC design flow will be discussed in this section.

The new design flow components complement the existing analog IC design flow. They must be perpetually available during all design stages to allow a comprehensive derivation, application and verification of constraints throughout the entire design process (Fig. 7.4). Any breach of the constraint application can lead to inconsistent design and constraint data, and hence, to a reduction of constraint verification coverage and an inconclusive verification result. The persistent use of automatic constraint verification offers greater verification coverage and reproducibility than manual verification.

All utilized design tools must fully understand the syntax and semantics of the used constraint representation. If different constraint representations exist within the design flow, then constraints must be converted between design tools that are mutually linked by a particular design task (e. g., conversion of device placement constraints within a layout editor to be used by a connected external third-party layout compaction tool). Furthermore, linked tools must support all constraint types that are relevant within a particular design context.

Constraint verification complements existing verification methods (e. g., DRC and LVS) required for sign-off in order to guarantee the intended circuit functionality. The achievable verification coverage depends on the traits and capabilities of the constraint verification framework as well as on the set of verification rules (Section 7.4.1). The chance of design iterations may increase if constraint verification is applied consistently due to better verification coverage. A back-annotation of constraints and verification results is required in order to minimize these iterations by addressing only relevant violations.

The constraint management system must guarantee low-level constraint data consistency by keeping each constraint and its referencing design object synchronized. Additionally, the high-level constraint data consistency, i. e. the maintenance of design data and constraint data as single data entity on file and cellview level, must be guaranteed by design guidelines and design data management systems.

7.6.2 Impact on Design Methods

Several challenges have to be addressed for a successful practical application of constraint-driven design. Among others, these challenges comprise new responsibilities for designers and the way how designers communicate with each other. The impact of these challenges is strongly dependent on the structure of the design team and the IC applications to be designed.

Several challenges arise from the change of design responsibilities since designers now must provide *all* necessary constraint information in a formalized fashion. This may lead to additional and possibly error-prone design work whose effort must be considered in the project schedule.

As demonstrated in Fig. 7.2, the analog IC design flow exhibits overlapping design steps to account for concurrent design problems. This is partially addressed by assigning constraints and using them in subsequent design steps. Here, the key question is to clarify which constraints are to be defined at which design step. This question can be answered with good confidence for constraints having an immediate impact in the next design step. Unfortunately, it cannot be easily answered for constraints either having a continuous impact or only having an impact on remote design steps. Here, designers must currently rely on their expert knowledge while future research should address this problem.

The assignment of constraints also has an impact on the partitioning of now separated design tasks with many positive but also negative effects. While the availability of complete constraint information may now allow the use of fully constraint-driven design tools there is also an increasing chance of over-constraining. An over-constraining done in a previous design step may aggravate or even prevent an optimization in a later design step. After performing a root cause analysis to identify the causing over-constraints designers may consider two options: (i) return to a previous design step while avoiding the causing over-constraints (design iteration), (ii) override or elimination of the causing over-constraints and continuation. If root causes cannot be found then unwanted design iterations are very likely. The elegant consideration of over-constraints is a critical issue which strongly influences the acceptance and practical success of a constraint-driven design flow. This consideration is also subject to further research.

Simultaneous semi-automatic and manual design styles must complement each other as long as the relevant constraint types cannot be considered at all or in case their consideration is limited to a specific design context only. For example, in the latter case a constraint would only be considered by a design algorithm within a cellview instead of considering it within the design hierarchy (e. g., hierarchical IR-drop constraint).

In order to address the tight interaction between these design steps and to consider the concurrent nature of the analog design problem, all artificially introduced boundaries between existing design steps should be gradually dissolved in the future. The removal of degrees of design freedom should occur gradually rather than abruptly in order to keep them available for design optimization as long as possible. While the automatic approach to achieve this goal is still subject to further research,

this issue is also of relevance for semi-automatic and manual design. In current analog design approaches, the strategy by which the degrees of design freedom are removed strongly depends on the designer's expert knowledge and the design task partitioning in a design team.

The reuse of analog IP often fails because small differences may prevent a direct IP reuse. A direct reuse is often not feasible if all degrees of design freedom were already removed from an IP block. However, the consistent definition of constraints between design objects allows design reuse of structural information based on IP templates, such as circuit and layout templates, that already include constraints. The structural information represents the most valuable part of the design knowledge, and hence, it enables a more flexible reuse since relevant degrees of design freedom are not fixed yet. In that respect, analog design automation should address low-level layout generation and high-level design planning as discussed in the next subsection.

7.6.3 Impact on Design Algorithms

In this section we discuss the impact constraint-driven design has on design algorithms and design planning. Furthermore, we briefly discuss new concepts and ideas for constraint-driven IC design. While some of these design approaches are new, others, such as the application of the constraint sensitivity analysis or the introduction of standardized algorithm interfaces, have already matured and thus have led to new insights into the analog design problem [14, 23, 25].

Present design algorithms are special-built for a particular purpose (e. g., focusing on placement, global or detailed routing). While this provides several benefits, such as an optimized execution time and memory footprint, it also introduces several significant limitations to "conventional design algorithms", such as incompatible interfaces for design and constraint data and a lack of functional abstraction. These limitations aggravate further advances in analog design automation.

A primary limitation in conventional algorithm design is the narrow focus on fast, but low-level execution without an implementation of standardized data interfaces. A standardized data interface creates a layer around a core algorithm to enable a common understanding of the syntax and semantic of the design and constraint data representation. This layer connects a design algorithm to the design and constraint databases as well as to other concurrently executed design algorithms. Thus, all design algorithms share a common understanding of the syntax and semantic of the design and constraint data representation.

Standardized algorithm interfaces enable the modularization and abstraction of design algorithms. The abstraction of their algorithmic work greatly improves algorithm reuse and flexibility because a single algorithm can be used to solve similar design tasks (this concept is similar to algorithm abstraction available in various programming languages). In turn, this flexibility enables the construction of high-level

design algorithms that utilize modularized low-level design algorithms in order to perform specific design tasks on a higher level of abstraction.

The strategy in which the degrees of design freedom are removed must be carefully chosen as mentioned earlier. A removal strategy can be applied to actuate high-level design algorithms. The actuation greatly benefits from the constraint sensitivity analysis (CSA, see Section 7.4.4).

First, CSA can be used to identify design task parallelism by searching for temporary groups of design variables and constraints that are either not or only weakly coupled (dynamic design task partitioning). For these groups, the next design step can then be performed independently of each other. Note that the independency of design variables and constraints in these groups may only be temporary, and hence, may not exist anymore after the design step is completed.

Second, CSA can also be used to determine the most sensitive design parameters in a particular design context that will more likely violate a constraint than non-sensitive parameters. Sensitive design parameters could then be considered with a higher priority within the specific design context.

A dynamic hierarchy of concurrent design tasks can be established in which design algorithms perform functional transformations (instead of conventional distinct design tasks) (Section 7.2.1). These transformations could be governed by either a fixed execution regime or by more flexible approaches such as high-level design planning algorithms that are guided by a design strategy.

Another major advantage for the development of high-level design algorithms is the possible dynamic consideration of new constraint types without the need to introduce major low-level algorithm changes. High-level design strategies can be used to solve low-level design problems by eliminating degrees of design freedom in a top-down methodology. This approach typically leads to better design results because low-level constraints are now less likely to break high-level constraints (Sections 7.4.2, 7.4.3).

Most of these introduced approaches promise great potential, namely the dynamic design task partitioning, the actuation of high-level design algorithms and the replacement of conventional algorithms by a sequence of continuous functional transformations. Nevertheless, all of them are still subject to further research.

7.7 Outlook

Despite the recent advances in constraint-driven design for analog IC design, there are several problems that need to be addressed in the near future to further broaden the applicability of analog design automation approaches. Methods to check the completeness of a set of constraints and constraint (meta-)verification rules, as well as the achieved verification coverage, must be developed to guarantee IC functionality, reliability, robustness, etc. The set of meta-verification rules must be optimized to allow time-efficient constraint verification. Today, such optimization is done man-

ually but automatic rule-optimization methods should be developed to reduce this burden.

As mentioned earlier, constraint sensitivity analysis is a powerful tool to drive and support high- and low-level design decisions, and to develop high-level design algorithms that allow more gradual IC design. The scalability of existing constraint-sensitivity analysis approaches is still limited to a few thousand design variables. This is sufficient for mid-sized analog blocks with typically several hundreds of analog devices. Application to top-level design problems requires the development of new complexity reduction methods, as well as fast constraint sensitivity calculation methods to improve scalability.

Key factors for next generation analog design automation are design techniques that reduce the degree of design freedom gradually rather than abruptly while performing several conventional design steps concurrently. This will require that the current artificial boundaries between conventional design steps be (gradually) dissolved. While breaking with conventional design approaches, this paradigm change could lead to a new class of (higher level) design algorithms that brings us one step nearer to the goal of full-scale analog design automation.

Acknowledgment

We would like to thank Jürgen Scheible of Robert Bosch GmbH and Ammar Nassaj of IFTE at Dresden University of Technology for the many fruitful discussions related to the topic of this chapter.

References

1. S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK: A Java constraint kit. In *Proc. Int. Workshop Functional and (Constraint) Logic Programming*, volume 64, pages 1–17. Elsevier B.V., 2001.
2. F. Baader and W. Snyder. *Handbook of Automated Reasoning*, volume 1, chapter Unification Theory, pages 445–533. Elsevier Science B.V., Amsterdam, 2001.
3. R. Barták. Theory and practice of constraint propagation. In *Proc. 3rd Workshop Constraint Programming for Decision and Control (CPDC)*, pages 7–14, 2001.
4. D. G. Cacuci, M. Ionescu-Bujor, and I. M. Navon. *Sensitivity & Uncertainty Analysis: Applications to Large-Scale Systems*, volume 2. Chapman & Hall / CRC, 2005.
5. Cadence Design Systems, Inc. http://www.cadence.com.
6. J.A. Carballo and S.W. Director. Constraint management for collaborative electronic design. In *Proc. IEEE/ACM 36th Design Automation Conference (DAC)*, pages 529–534, 1999.
7. H. Chang, E. Charbon, U. Choudhury, A. Demir, E. Felt, E. Liu, E. Malavasi, A. Sangiovanni-Vincentelli, and I. Vassiliou. *A Top-Down, Constraint-Driven Design Methodology for Analog Integrated Circuits*. Springer, 1999.
8. J. Cohen. Constraint logic programming languages. *Commun. ACM*, 33(7):52–68, 1990.
9. M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proc. Int. Conf. 5th Generation Computer Systems*, pages 693–702, 1988.

10. B. M. Freeman-Benson. *Constraint Imperative Programming*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1991.
11. J. Freuer, G. Jerke, J. Gerlach, and W. Nebel. On the verification of high-order constraint compliance in IC design. In *Proc. IEEE/ACM Int. Conf. Design Automation and Test in Europe (DATE)*, pages 26–31, 2008.
12. Th. Frühwirth. Introducing simplification rules. Technical Report ECRC-LP-63, European Computer-Industry Research Centre, Munich, Germany, 1991.
13. Th. Frühwirth, A. Herold, V. Küchenhoff, Th. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming – an informal introduction. *Lecture Notes In Computer Science*, 636:3–35, 1992.
14. G.J. Gad El-Karim, R.S. Gyurcsik, and G.L. Bilbro. Sensitivity-driven placement of analog modules. In *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, pages 363–366, 1994.
15. V. Gerard and Th. Schiex. Solution reuse in dynamic constraint satisfaction problems. *Proc. Association for the Advancement of Artificial Intelligence (AAAI)*, pages 307–312, 1994.
16. H. Gräb, F. Balasa, R. Castro-Lopez, Y.-W. Chang, F.V. Fernandez, P.-H. Lin, and M. Strasser. Analog layout synthesis – recent advances in topological approaches. In *Proc. IEEE Int. Conf. Design Automation and Test in Europe (DATE)*, pages 274–279, 2009.
17. M. Grabmüller and P. Hofstedt. Turtle: A constraint imperative programming language. In *Proc. 23rd SGAI Int. Conf. Innovative Techniques and Applications of Artificial Intelligence*, 2003.
18. D. R. Insua. *Sensitivity Analysis in Multi-Objective Decision Making*. Springer, 1990.
19. J. Jaffar, S. Michaylov, P.J. Stuckey, and R. H. C. Yap. The CLP($\Re$) language and system. *ACM Trans. Programming Languages and Systems*, 14(3):339–395, July 1992.
20. V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.
21. E. Malavasi and E. Charbon. Constraint transformation for IC physical design. *IEEE Trans. Semiconductor Manufacturing*, 12(4):386–395, 1999.
22. E. Malavasi, E. Charbon, B. Arsintescu, and W. Kao. A constraint management system for IC physical design. In *Proc. 11th Brazilian Symp. Integr. Circuit Design*, pages 240–243, 1998.
23. E. Malavasi, E. Charbon, E. Felt, and A. Sangiovanni-Vincentelli. Automation of IC layout with analog constraints. *IEEE Trans. CAD of Integr. Circuits and Systems*, 15(8):923–941, 1996.
24. Mentor Graphics Inc. http://www.mentor.com.
25. P. Miliozzi, I. Vassiliou, E. Charbon, E. Malavasi, and A. Sangiovanni-Vincentelli. Use of sensitivities and generalized substrate models in mixed-signal IC design. In *Proc. IEEE/ACM 33rd Design Automation Conference (DAC)*, pages 227–232, 1996.
26. A. Nassaj, J. Lienig, and G. Jerke. A new methodology for constraint-driven layout design of analog circuits. In *Proc. IEEE Int. Conf. Electronic, Circuits and Systems (ICECS)*, pages 996–999, 2009.
27. J.-F. Puget. A C++ implementation of CLP. Tech. rep. 94-01, ILOG SA, Gentilly Cedex, France, 1994.
28. J. A. Robinson. A machine-oriented logic based on the resolution principle. *ACM*, 12(1):23—41, 1965.
29. R. A. Rutenbar and J. M. Cohn. Layout tools for analog ICs and mixed-signal SoCs: A survey. In *Proc. IEEE/ACM Int. Symp. Physical Design (ISPD)*, pages 76–83, 2000.
30. J. Scheible. Constraint-driven Design – Eine Wegskizze zum Designflow der nächsten Generation (in German). In *Proc. VDE ANALOG'08*, 2008.
31. G. Tack. *Constraint Propagation – Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.
32. M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, London, U.K., 1997.

Glossary

Constraint Constraints define relations between values of design variables. Constraints defining a single relation are denoted as simple constraints. Constraints defining a set of relations are denoted as complex constraints.

Constraint Assignment Process of linking constraints to design objects. In case the corresponding design objects are located in different design hierarchy levels, the linking is done by traversing the hierarchy tree either strictly top-down, strictly bottom-up or in a mixed top-down and bottom-up manner. The linking can be permanent or temporary.

Constraint Derivation Process of deriving constraints from design objectives. Constraint derivation is also known as constraint generation.

Constraint-Driven Design Design paradigm that considers all constraints in a consistent and comprehensive manner.

Constraint Engineering Design paradigm that comprises the use of several design flow components, such as constraint assignment, derivation, propagation, transformation and verification.

Constraint Engineering System (CES) Software architecture that implements the constraint engineering concept so that all components of the constraint-driven design flow are available during the design process [11].

Constraint Handling Rules (CHR) Programming language that, among others, allows the definition of problem-specific constraint solvers [12].

Constraint Logic Programming (CLP) Form of constraint programming, in which logic programming is extended to include concepts from constraint satisfaction. The unification process in CLP is extended by constraint handling in the boolean, real or integer constraint domain [13]. CLP is often implemented as an enhancement of Prolog-like computer languages with additional constraint solving mechanisms.

Constraint Management Software architecture to enable the storage, management, access and synchronization of constraint data. Features of the constraint management are used by all components of the constraint-driven design flow.

Constraint Programming Programming paradigm where relations between variables are stated in the form of constraints.

Constraint Satisfaction Problem (CSP) Mathematical problem defined as a set of objects whose state must satisfy a number of constraints. These problems represent the entities in a problem as a homogeneous collection of finite constraints over variables.

Constraint Sensitivity Analysis (CSA) Method to determine the sensitivity of a design parameter in relation to an objective function and related constraints.

Constraint Solver Mechanism to solve a given constraint satisfaction problem.

Constraint Transformation Process of transforming a higher level constraint into a set of lower level constraints of the same or a different domain and vice versa (inverse constraint transformation).

Constraint Type Type of a constraint that corresponds to the type of design variables which share a relation defined by that constraint.

Constraint Verification Verification process to ensure that no over-constraints exist and that all constraints are fulfilled by the design result [11].

Design Context Local context in which a particular design task is performed.

Design Object Data object represented in the database of a design tool, such as cell, cellview, instance, net, terminal etc.

Design Objective Design goal to be achieved or specification requirement to be met by either a final or a partial design result.

Design Rule Check (DRC) Verification process to ensure that all manufacturing-related constraints are fulfilled by the design result.

Design Tool Software tool for IC design generation and verification.

Expert Knowledge Entity of a designer's problem-specific design knowledge in formalized and non-formalized form.

Layout Versus Schematic (LVS) Verification process to ensure that a given device netlist matches a netlist extracted from the layout representation.

Logic Programming (LP) Software language paradigm based on logic, more specifically on resolution theorem proving in the predicate calculus [28].

Meta-Verification Verification process to ensure that all complex constraints are fulfilled by the design result [11].

Over-Constraint Condition in which not all given constraints can be fulfilled simultaneously.

Predicate (→ LP, CLP) Mathematical sentence that describes a common property by which a subset of objects can be identified within a global set of objects.

Propagation (→ CHR) The propagation within CHR is the derivation of one or more new constraints from a given set of constraints. It is triggered by the existence of one or more constraints that are already part of the constraint set. After the propagation took place, the new constraints are part of the constraint set [12].

Root Cause Analysis Class of problem solving methods aimed at identifying the root causes of problems or events. One approach of solving an existing design problem is to eliminate its root causes. Root cause analysis is often used iteratively (continuous improvement).

Schematic-Driven Layout (SDL) Design paradigm in which the layout generation is driven by the schematic representation of the circuit.

Simpagation (→ CHR) The simpagation is a combined application of propagation and simplification. While it can be expressed by solely using propagation and simplification rules, it can be handled more efficiently [12].

Simplification (→ CHR) The simplification within CHR removes one or more constraints from a given set of constraints. It is triggered by the existence of one or more constraints that are already part of the constraint set [12].

Tool Integration Kit (TIK) Data interface of the constraint engineering system that translates tool-specific data into the CLP language and vice versa.

Unification (→ CLP) Process that tries to match symbolic expressions by assigning sub-expressions to variables that are part of two expressions [2]. Unification is a core concept of logic programming.