From Constraints to Tape-Out: Towards a Continuous AMS Design Flow

Andreas Krinke*, Tilman Horst*, Georg Gläser†, Martin Grabmann†,
Tobias Markus‡ Benjamin Prautsch§, Uwe Hatnik§, Jens Lienig*

*Technische Universität Dresden, Institute of Electromechanical and Electronic Design, Dresden, Germany
Email: {andreas.krinke, tilman.horst, jens.lienig}@tu-dresden.de
†IMMS Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH Ilmenau, Germany.
Email: {georg.glaeser, martin.grabmann}@imms.de
‡Heidelberg University, ZITI, Computer Architecture Group, Heidelberg, Germany
Email: tobias.markus@ziti.uni-heidelberg.de
§Fraunhofer IIS/EAS, Institute for Integrated Circuits, Division Engineering of Adaptive Systems, Dresden, Germany
Email: {benjamin.prautsch, uwe.hatnik}@eas.iis.fraunhofer.de

***Abstract*—The effort in designing analog/mixed-signal (AMS) integrated circuits is characterized by the largely manual work involved in the design of analog cells and their integration into the overall circuit. This inequality in effort between analog and digital cells increases with the use of modern, more complex technology nodes. To mitigate this problem, this paper presents four methods to improve existing mixed-signal design flows: (1) automatic schematic generation from a system-level model, (2) flexible automatic analog layout generation, (3) constraint propagation and budget calculation for dependency resolution, and (4) verification of nonfunctional effects. The implementation of these steps results in a novel AMS design flow with a significantly higher degree of automation.**

I. Introduction

Mixed-signal designs are the point of contact between the digital and analog world. Accordingly, they appear in the majority of todays microelectronic systems for sensing applications and mobile communication devices. However, because of miniaturization and migration to new technology nodes, complexity and cost of nanometer mixed-signal designs is increasing. To mitigate these problems, an integrated AMS design flow, which addresses digital as well as analog design with a high degree of automation, is required.

A. Challenges

As opposed to digital design, its analog counterpart is dominated by manual design work. Beyond that, while digital parts highly benefit from small technology nodes in terms of power, performance, and area, analog circuits suffer from increasing process variations going along with small technologies. These discrepancies complicate the integration of digital and analog design steps in advanced technologies.

B. Session Outline

In this paper, we propose a new AMS design flow. By combining different state of the art solutions for individual design steps to a consistent flow, we leverage their benefits and effectively mitigate the described problems in the field of AMS design. The foundation of our flow is the system-level description in SystemVerilog, from which we derive an architectural design hierarchy and identify general circuit classes, e.g, amplifiers or filters. To close the gap between this abstract hierarchical description and the actual layout, we collect constraints and propagate them throughout the design hierarchy. Simultaneous budget calculation facilitates the use of constraints in design decisions. Schematic and layout generation procedures will later consider these constraints to create results that comply with global requirements. Based on the identified circuit classes and propagated constraints, the automated generation of schematics and layouts is initiated. To ensure correct operation of the circuit even in case of advanced technologies, we propose a verification methodology, which integrates nonfunctional effects. Fig. 1 shows an overview of the contributions of this paper and their relationships.

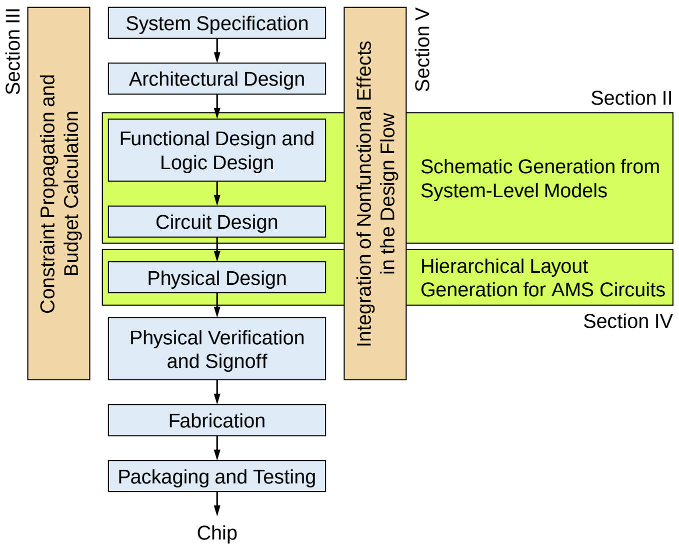

Section III
Constraint Propagation and Budget Calculation
System Specification
Architectural Design
Functional Design and Logic Design
Circuit Design
Physical Design
Physical Verification and Signoff
Fabrication
Packaging and Testing
Chip
Integration of Nonfunctional Effects in the Design Flow
Section V
Section II
Schematic Generation from System-Level Models
Hierarchical Layout Generation for AMS Circuits
Section IV

Fig. 1. Combined overview of our contributions presented in Sections II-V.

II. Schematic Generation Framework in a Mixed-Signal Top-Down Design Flow

This Section II presents an entry point for the mixed-signal top-down design flow which focuses on the framework for schematic generation from the system-level model.

The structural hierarchy and schematics for the full-custom part are generated from the system-level SystemVerilog real number model (RNM). Furthermore, the developed method offers a framework which automatically detects general models in the leaf cells and runs corresponding dimensioning scripts, which can range from simple scripts to abstract approaches.

The flow was used with great success in eliminating structural inconsistencies, related errors and avoiding additional rework. Additionally, it provides high automation, thereby reducing design time.

A. Concept

A digital-on-top top-down flow is chosen to start implementing large mixed-signal designs. The system-level design is implemented in SystemVerilog where structural descriptions and functional blocks are strictly separated. As a result, there are three different kinds of descriptions. First, there is the structural Verilog description which solely describes the structure of either digital or analog blocks. The other two descriptions are for leaf cells. For the digital part the leaf blocks are implemented in synthesisable Verilog. For analog blocks the leaf cells are modeled directly with RNM SystemVerilog or with generic RN models.

The presented tool framework uses the structural description and the leaf cells with generic RN models to automatically generate and build the needed Cadence Virtuoso libraries, schematic structures, black box schematic templates, sized schematics, and symbols.

The framework is implemented with Python 3 and uses Cadence Genus for structure elaboration. A TCP/IP socket-based communication to Cadence Virtuoso is implemented to control cell and schematic generation.

The main parts of a phase-locked loop (PLL) subsystem hierarchy in a multi-gigabit design in 28 nm is shown in Fig. 2. The different views in the system-level description are marked in the figure.

B. Implementation

The main flow of the schematic generation tool (SGT) is shown in Fig. 3. In the first step, a YAML configuration for the tool is parsed to setup where library generation is required and if building blocks exist and need to be kept untouched. Additionally, the tool is given the file list of the system level description and the cell from where to generate the schematics. In the PLL example the cell from which the full-custom generation starts would be PLL_CORE which contains all full-custom blocks. The complete toplevel file list is used to evaluate the subsystem with the system-level context and parameters.

The file list is then read, parsed and evaluated. At this point leaf cells are treated as black oxes. Only a structural elaboration

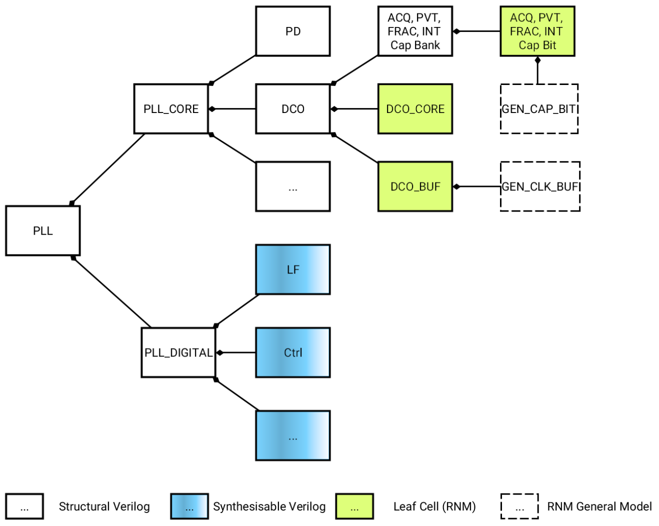


Fig. 2. Simplified phase-locked loop (PLL) design hierarchy and description types.

is needed. The parsed and evaluated structure is represented and accessible via the resulting abstract syntax tree (AST).

For each leaf cell, the corresponding RNM description and parameter configuration is identified and passed to the schematic and schematic template creation method.

As a first step in the generation process the leaf cells are parsed and parameters and connection are resolved. Then it is checked whether a generic model exists. A generic model is a RNM description from a model library which is often used in the design. It is not included in the physical design hierarchy. An example is the general model of a capacity bit that is used for each different capacity bit leaf cell (Fine, ACQ and PVT bit) in the DCO.

If no generic RN model exists, as for example for the DCO_Core, a symbol view and a schematic template for Virtuoso is generated. A schematic template is a black box schematic cellview with the corresponding pins from the leaf cell and additional information gathered from special comments and parameters in the RNM description.

If a generic RN model is found a physical description script is executed. This physical description script contains the relations between the high level parameters in the RNM descriptions.

In case of the capacitor bit the high level parameters are the off capacitance and switch capacitance from which the device primitive parameters (i.e. width, length, layers) are calculated. To achieve this, there is a module implemented to simply access device tables for a table based sizing approach, for example the g_m/i_d method [1] often used to size transistor topologies in nanometer designs. These calculated primitives are forwarded to an architecture script. Depending on high level parameters different architecture scripts and respectively schematic topologies can be chosen. At the moment an expert based approach were the physical description and architecture script are written by the designer, is chosen. This can be interchanged to other optimization strategies and script

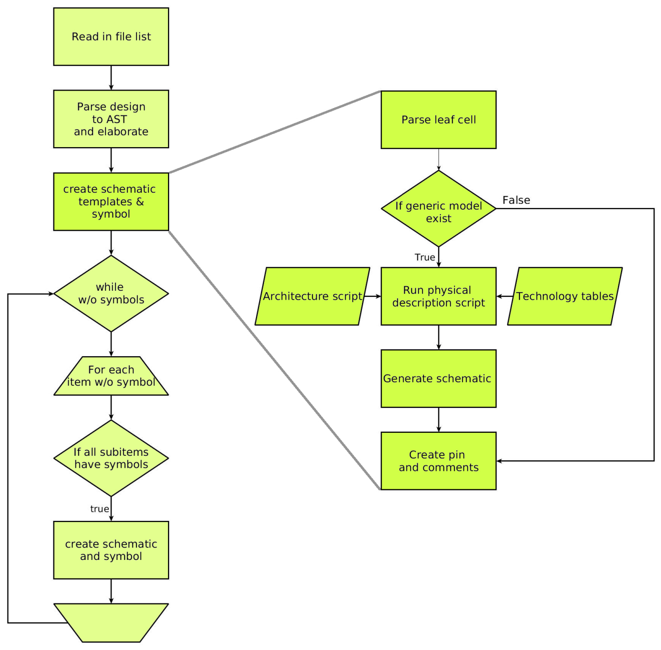


Fig. 3. Simplified schematic generation program flow.

languages. With this a sized schematic is generated.

In the following loop the rest of the hierarchy is generated from the deepest sub hierarchy in the hierarchy tree to the selected root cell. For each of the elements a schematic containing the created leaf cells and already created subsystems and corresponding symbol views are generated.

Furthermore, the framework does not only offer the schematic generation tool but also a Python package toolbox to simplify many tasks:

- Simple circuit sizing for table based sizing approaches
- An abstract interface for creation and manipulation of Virtuoso cellviews
- .f file parser and Verilog preprocessor
- Methods to work on the AST of the parsed design
- An object structure representing the created design

C. *Comparision and Results*

There are other tools for top down or schematic generation. Two prominent examples will be shortly described.

One example for a schematic/layout generation framework is the Berkley Analog Generator (BAG) [2]. The big difference between our tool and BAG is that it has no inherent connection to a system-level description and it does not solve the consistency problem between system-level and implementation.

The Cadence tool ”Verilog In” which exist within the Virtuoso environment [3] is able to convert Verilog descriptions to Virtuoso cellviews. This tool was used in early design stages of this framework but it was too limited in its functionality to further base the framework on it. ”Verilog In” does not support parameterized generate expressions and the possibility to further extend it. The possibility to extend it for the schematic generation on a leaf cell level, was not given.

Our SGT framework was used to implement a multi-gigabit transceiver in 28nm [4]. With this design several advantages could be observed. On the one hand several error sources could be eliminated by creating a consistent system level structure and implementation. Furthermore there is a significant speed up in the design flow since the structure is automatically generated. Another side effect of the structural consistency between the different design views was the possibility to back annotate block level SPICE simulation into the system level models. This enabled accurate system level simulations which could not been done in SPICE because of the runtime. One example for an accurate system level simulation was the PLL phase noise and its performance impact on the bit error rate.

Since then, the framework has been further developed to the schematic sizing functionality described in this Section. Parameterizable automatically sized schematics from leaf cells will result in a higher reusablility, more automation and less error sources.

Future work will include working upon the resulting object structure of the design and implementing more parameterizable leaf cells. Furthermore the framework will be improved in terms of logging and usability functions in the framework package.

III. Constraint-driven Cross-Hierarchical Design

The complexity of today’s microelectronic systems such as SoCs requires the overall design task to be divided into several less complex subtasks or modules. Typically, this results in a hierarchical system structure with different design teams working on these modules using different sets of tools. However, complex global dependencies, e.g. constraints on the layout position of elements, make it difficult to design the modules separately, because design decisions in one module may influence and ultimately violate constraints in other modules [5]. The cause of this problem is that dependencies are often not recognizable for the designers.

To cope with these issues, this Section III proposes a methodology to make constraints visible and verifiable in all relevant modules throughout the hierarchy. Starting with the constraints from the system specification, we propagate these and the constraints that designers add later during the design process within the design hierarchy. At the end of this process, we know the set of relevant constraints including, e.g., geometrical dimensions and pin positions, for every module. If the final layout fulfills all these constraints, the correct operation of the overall system according to the specification can be guaranteed.

A. *Constraint Definition*

Each AMS circuit contains many different *design elements*, e.g., cells, instances, nets, terminals, layout shapes, etc. These design elements have a variety of properties that can be changed during design. Examples of these *design parameters* are the power consumption of a cell and the layout position of an instance.

Constraints are requirements for the values of these design parameters, e.g., an upper limit for the power consumption of a cell or the horizontal alignment of two instances in the layout. All constraints must be met before tape-out for the circuit to meet the specification.

When defining a new constraint, the designer first chooses the cell to which the constraint applies, then selects target design elements in that cell, and formulates the requirement for associated design parameters. The cell to which a constraint belongs is called the *context cell* of the constraint. The constraint must be fulfilled for the context cell itself *and* for all its instances.

The value of a design parameter usually varies for each occurrence of the context cell in the design and depends on the current design state. Therefore, we model design parameters as functions ψ that associate a design state, an occurrence of the context cell and a tuple of target design elements to a value, e.g. a number or a physical quantity.

Based on these observations, we model the meaning of a constraint using a *constraint function* ϕ that is a mathematical expression of design parameters and constraint parameters. This expression calculates a Boolean value (true/false) that represents the state of the constraint (fulfilled or violated). To determine this value, the function requires information about the design state, an occurrence of the context cell, a tuple of target design elements as well as a tuple of constraint parameter values. The first three arguments are used to calculate the values of design parameters.

Consequently, when creating a concrete constraint, the designer chooses a context cell, selects or defines the constraint function, selects target design elements, and sets constraint parameter values. Therefore, a constraint is ultimately a tuple of these four parts.

B. Constraint Types

The concept of constraint functions allows the creation of an infinite number of different constraints. Reasons are the vast number of possible design parameters and the infinite number of possible structures of the constraint function. To make this variety manageable, their classification into different types is necessary. These types allow you to quickly grasp the meaning of a constraint by only looking at its type and target design elements without having to understand its constraint function. In addition, they simplify constraint management in EDA tools.

We use the mathematical structure of the constraint function to define constraint types. An example is the constraint function

$$\varphi\colon \big(m_1, m_2\big) \mapsto \mathit{distance}\big(\textsc{Position}(m_1), \textsc{Position}(m_2)\big) > s \cdot \big(w_1 \cdot \mathrm{PC}(m_1) + w_2 \cdot \mathrm{PC}(m_2)\big), \quad (1)$$

that defines a minimum distance of two instances depending on their respective power consumption (PC). The variables w_1, w_2 and s are constraint parameters. The mathematical structure of this function can be represented by an *expression tree* as shown in Fig. 4.

Constraints of the same type have similar expression trees. Therefore, we define constraint types as annotated expression trees, called *pattern trees*. Vertices are annotated with quantifiers that specify how often the corresponding subtree may occur. Comparable to a regular expression, the pattern tree of a constraint type describes a possibly infinite number of expression trees and thus constraint functions of this type.

TABLE I
Propagation Types for Design Parameters

Propagation Type	Example Design Parameters
Based on Instantiation	
Top-Down	Area, Power Consumption
Bottom-Up	Position/Orientation in the Top Cell
Based on Connectivity	
Based on Logical Connectivity	Load Capacity, ESD Protection
Based on Physical Connectivity	Parasitic Resistances & Capacitances
Based on Spatial Adjacency	Overlapping Instances, Metal Density

C. Constraint Propagation

For a constraint to be considered in design decisions, it has to be visible and verifiable in all relevant cells, i.e., in all cells where its state can be influenced. The process of *constraint propagation* determines this set of relevant cells [6].

Which cells are relevant for a constraint depends on the design parameters and mathematical structure of the constraint function, its target design elements, and its context cell. However, the *algorithm* to search for these cells only depends on the design parameters. In this regard, design parameters can be divided into small number of categories. Once the propagation category of a design parameter is known, its relevant cells can be determined according to a fixed set of rules.

1) Propagation Types: We identified five fundamental types of propagation. The assignment of a design parameter (DP) to a propagation type depends on how relevant cells are determined:

- Top-down propagation: DP depends on all cells that are further down in the design hierarchy,
- Bottom-up propagation: DP depends on all cells that are further up in the design hierarchy,
- Propagation based on logical connectivity: DP depends on properties of electrically connected instances,
- Propagation based on physical connectivity: DP depends on the physical implementation of a net, and
- Propagation based on spatial adjacency: DP depends on the layout in a specific region.

Tab. I lists example design parameters for all propagation types. For a single design parameter, propagation results in a *propagation tree* that contains a node for each relevant cell while edges represent the spreading of information from one cell to another.

2) Propagation Algorithm: Based on the propagation types of the individual design parameters, the propagation of a constraint determines the relevant cells for the complete constraint. For each global instance (occurrence) of the constraint

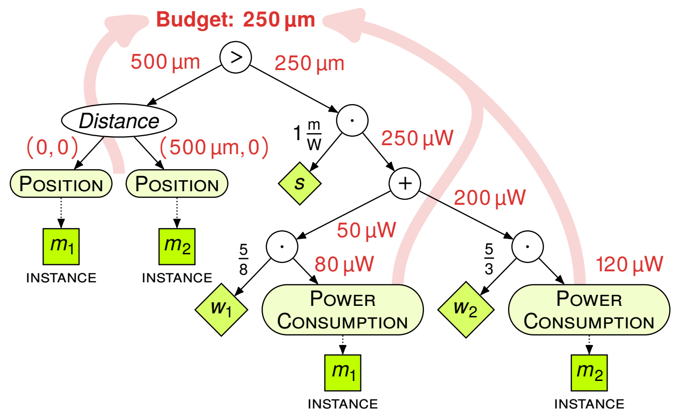


Fig. 4. Expression tree of (1) with current values of design parameters after bottom-up constraint verification.

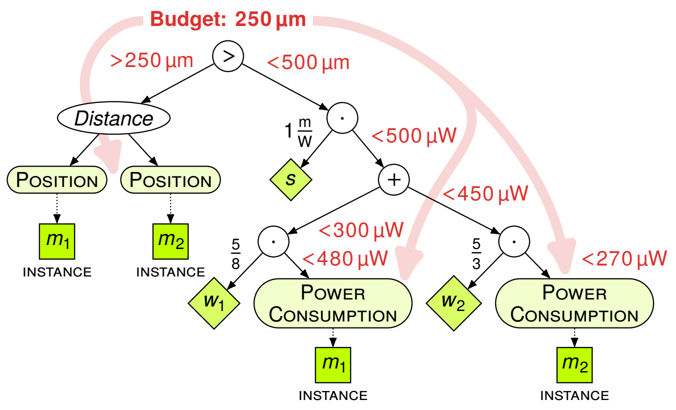


Fig. 5. Top-down calculation of design parameter budgets.

context cell in the design, there is an individual set of relevant cells. Therefore, for a specific global instance, propagation proceeds to process all design parameters and operations in the expression tree of the constraint in reverse topological ordering. For each design parameter, propagation is performed according to the corresponding type and (new) propagated constraints are created in all relevant cells.

Afterwards, the current value of the design parameter or operation is calculated and passed upwards in the tree. If the current value cannot be calculated, e.g., because of missing information in early stages of the design, the parameter can be modeled as a random variable based on design experience [7].

After propagation of all constraints, they are visible in all relevant cells and can be considered in design decisions in these cells.

D. Constraint Verification and Budget Calculation

Constraint verification requires the calculation or estimation of all relevant design parameters values. Since we have modeled design parameters as functions ψ, we can now evaluate these functions to calculate their value for the current design state. As stated above, yet unknown design parameters can be estimated using probability distributions.

At this point, we have all the necessary information to verify constraints by calculating their state. For this purpose, we evaluate constraint functions φ for the current values of the design parameters. The value of φ is either true or false, depending on the constraint's state. Constraint verification can be visualized using the expression tree of φ. As an example, Fig. 4 shows the expression tree for the constraint in (1). The current design parameter values flow upwards through the tree following the red arrows. The value calculated for the root vertex represents the constraint state.

Next, we determine the budgets of all relevant design parameters. This budget is the set of permissible values for one design parameter assuming that the values of all other parameters are unchanged. Therefore, we calculate these budgets by solving the constraint function φ for a single design parameter. Since constraints often use inequalities, budgets are also expressed as inequalities. For example, Fig. 5 shows the resulting budgets for individual design parameters.

Finally, calculated budgets are allocated to the individual propagated constraints according to the propagation trees of design parameters. As a result, propagated constraints not only signal an influence on constraints in other cells, but also define concrete requirements (budgets) for local design parameters.

E. Discussion

As part of the implementation of our methodology, we developed an adaptive data model for efficient constraint management and propagation [8]. It is built on top of Neo4j, an open source property graph database. We integrated both the graph database with the new data model and our methodology into Cadence DF II. The implementation supports the definition of new constraints, their propagation, and the visualization of propagation trees.

The presented methodology supports the propagation of arbitrarily complex constraints through the design hierarchy. It is complemented by methods for constraint verification and budget calculation. Not only do designers recognize the existence of dependencies between modules, but they can also actively take them into account when making decisions.

IV. Flexible Generation of Analog Integrated Layouts Using a Novel Floorplanning-Driven P&R Approach

A fault-free and verified layout is the final result of the analog design flow. One approach particularly aiding layout design is generator-based automation [9]–[11]. Generators create layouts and other views fast from a single-source description (generator code). They work in a parameterizable and structurally predefined bottom-up way. However, limited structural flexibility and limited access to formalized requirements such as constraints (see Section III) are so far disadvantages of generators.

This Section IV discusses a new generator concept that allows a mixture of both bottom-up layout description and abstract top-down layout description. Both constructive and iterative algorithms that use formal constraints as input aid the bottom-up and top-down styles, respectively. This way,

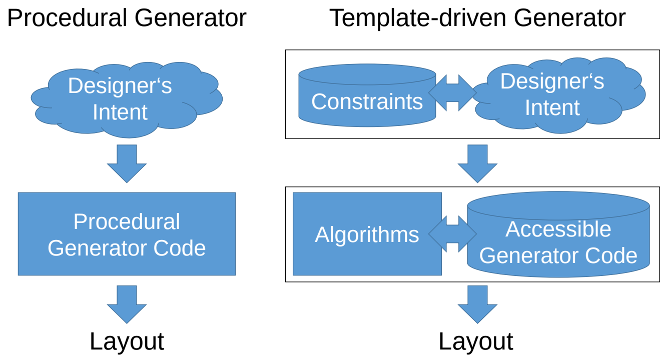


Fig. 6. Comparison between procedural generators without direct code accessibility by algorithms (left) and template-driven generators which can be accessed and adapted by algorithms (right).

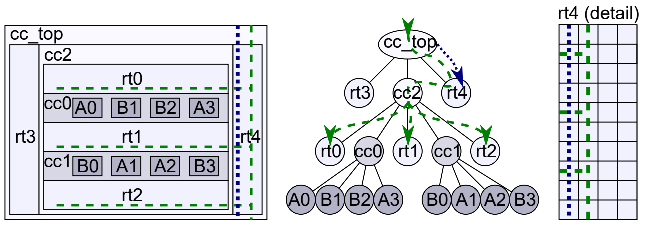


Fig. 7. Hierarchical composite structure of the layout representation. Left, the coarse floorplan is given which represents the global view (*Ax* and *Bx* are instances, *rtx* are routing channels, and *ccx* are helper compositions). In the middle, the related graph representation is shown and right, an exemplified partitioning of routing channel *rt4* into multiple bins is depicted. Both dashed green and dotted blue lines represent routing on either representation.

we overcome limits of procedural generators and generation gets standardized in a template-based way. The new approach bridges the gap between procedural bottom-up generators and optimization-based top-down approaches being a step towards a continuous analog design automation flow.

A. Procedural Generators vs. Template-driven Generators

Former entirely procedural generators have the drawback that the flow from the programmer's intent towards the realization is too implicit [5]. I.e. the idea which is to be implemented as generator code is first available as thoughts in the mind of the programmer. Second, these thoughts are transferred into procedural generator source code. Third, the code is run which results in the actual design data. Only after the code was run, the generator source code can be verified indirectly via DRC and LVS checks executed on the generated design data. This approach is both time-consuming and prone to errors as logical faults of the generator description itself cannot be discovered. Therefore, time-consuming and incomplete iteration of generation and DRC/LVS checking must be applied.

Recently, new template-driven generator approaches were presented [12], [13] which implement the template approach [14]–[16] into procedural generators. The basic idea is to overcome error-prone procedural descriptions which do not allow direct algorithmic access to the generator code. As the template-driven generators are programmed by well-defined constructive commands, these commands adapt a data structure which itself is accessible by algorithms. Thus, algorithmic access for both data extraction and data adaptation becomes possible even for generator code. The basic differences between procedural and template-driven generators are shown in Fig. 6.

B. Template-driven Approach for Placement and Routing

More specifically, we propose a template-driven generator implementation approach which organizes floorplannning, placement, and routing in a single and hierarchical abstract representation. This representation takes advantage of the composite design pattern which allows both edits and data aggregation on different hierarchical levels at any time. Compared to procedural generators, the order of the generator code is, thus, no more relevant and therefore dependencies in layout generation can be solved by algorithms (see [13]). Additionally, both high-level and low-level requirements can be described on this single layout object which propagates relevant information such as constraints [6] across the hierarchy. The partitioning approach of this layout representation as well as the related graph are depicted in Fig. 7.

In Fig. 8, the abstract layout decomposition of two instances and their local routing scheme are depicted. The two instances on top are connected by the horizontal routing channel at the bottom. The instances depicted by the gray areas are surrounded by additional routing channels. Both instances and routing channel areas are partitioned into routing bins which can be tagged with properties each. An instance including the surrounding local routing channels corresponds to a leaf node in the graph of Fig. 7. The horizontal routing channel is represented by another internal node of the graph. The green channels illustrate feasible wiring ways (provided by the device definition) which are automatically conform with the design rules. Depending on additional parameters like symmetry requirements or wire parasitics, the router will search the best routing solution. This routing solution will be influenced by the instance properties (e.g. available wire channels), but also by properties of the horizontal routing channel. For example, the used metal layer in the horizontal channel can influence the routing around the instances (path, metal layer, vias) and vice versa. The final routing is not hard coded in the generator anymore, instead it is controlled by properties of the graph nodes. Using this abstract layout representation, the actual layout generation is executed subsequently.

Based on this concept, an investigation was carried out in [17]. There, an abstract routing API was developed in order to represent routing based on a given placement in an abstract way. An example of this approach is given in Fig. 9 in which an automatically generated differential pair layout is depicted. The major advancement of this approach over former generators is that the abstract API is used according to the template-driven approach mentioned above. The layout, thus, is pre-compiled on an abstract level including automatic adaptation of PDK-dependent dimensions and spacings prior to generation. This first step both optimizes and checks the layout. Only afterwards,

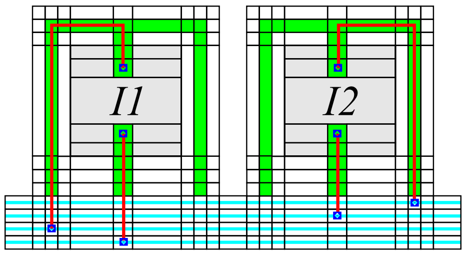


Fig. 8. Example of the partitioning of the abstract layout representation using a slicing approach. In grey, two instances *I1* and *I2* with surrounding bins are represented with two pins each. The green tracks are preserved for local device routing. Both red and blue lines are actual abstract wires on different layers and blue rectangles represent vias. The partitioning into bins depends on amounts of wires, pin and instance positions as well as sizes.

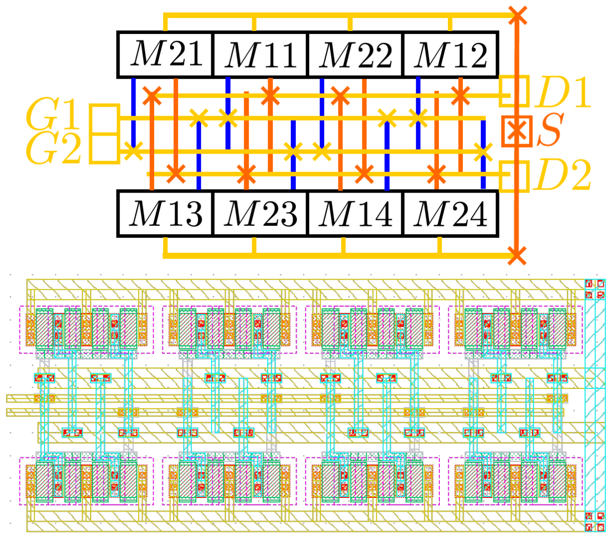


Fig. 9. Stick diagram representing floorplan, placement, and routing (top) and generated layout (bottom) [17].

the layout is generated based on the preliminary geometric considerations.

C. Outlook

For future layout generators, this approach will allow to include abstract constraints into the layout generation procedure and, therefore, the generated results will improve significantly. Both explicitly programmed layout structures and explicit constraints which are generated during other design steps will be joined in order to improve future analog layout automation as part of the entire analog/mixed-signal design flow.

V. Invasive Analysis Framework for Nonfunctional Effect Verification

Modern methods for system analysis and verification demand for either refining a given model or evaluating the impact of certain changes as for instance in design space exploration. In analog/mixed-signal systems, the hand-made models mostly show the purely functional behavior [18]. For including

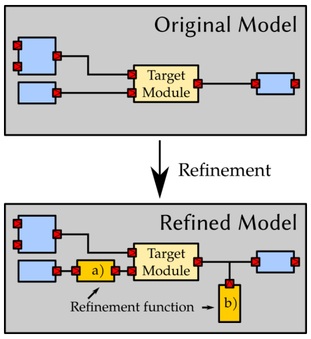


Fig. 10. Example for the refinement process: A system is refined by inserting new components (refinement functions) at the interconnections. These can be applied either a) along a *branch* connection or b) on a *node*.

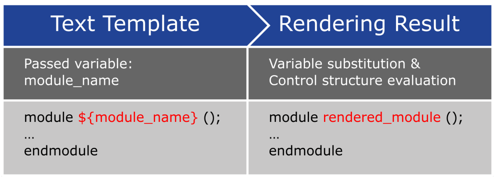
nonfunctional effects such as sensitivity for power supply non-idealities, the models have to be refined. In this Section V, we present a framework for processing Verilog-AMS models. This framework can be used to structurally modify a given system and interface to an industrial simulation environment for paving the way to novel analysis and refinement methods.

A. Concept

Invasive analysis refers to methods that extract information from a system by applying changes within the system. For example, the relevance of noise-coupling to certain nodes can only be observed by *applying* a noise source *within* the model to be simulated. Hence, conducting invasive analysis relies on refining a given model of a system in a certain way. Since most systems are a composition of several components, it is reasonable to apply these refinements to the interconnections and reference them to a *target module*. Consider the example shown in Fig. 10. In this illustration, an example system is refined by inserting new components (refinement functions). We call the insertion points due to their conceptual similarity with circuit engineering *nodes* and *branches*. Each refinement function may have several node and branch connections. For instance, a coupling network has to be connected to several branches while a very simple crosstalk model could be realized by a capacitance connected between two nodes. Consequently, our framework has to support modification of internal model connections of both types and is in addition able to insert instances of refinement functions. For further automation, functionalities that allow iterations over the model structure to perform for instance operations for all signals in a system have to be included.

B. Realization

For realizing this kind of model-rewriting framework, we created the structure shown in Fig. 12. The model parsing and rewriting infrastructure supports the processing of Verilog-AMS [18] and SystemC-AMS [19] models. The parser examines the structure of the given model and generates a *meta-model* datastructure that may be used for iterations or additional modifications. It is afterwards fed into the *Rewriter* together with the generated code of the refinement function. The code



Text Template	Rendering Result
Passed variable: module_name	Variable substitution & Control structure evaluation
module ${module_name} (); … endmodule	module rendered_module (); … endmodule

Fig. 11. Text templating provides our framework with a highly flexible multi-language code generator.

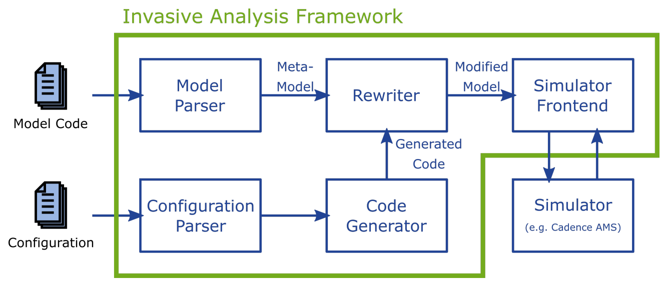


Fig. 12. The internal structure of our model *Invasive Analysis Framework* providing model parsing and rewriting facilities combined with simulator control.

generation scheme relies on text templating as shown in Fig. 11, which provides a highly flexible code generator for different modelling languages. Out of this data, the rewriter block re-assembles the model with the demanded modifications or refinements.

The resulting code can be directly passed to a simulator frontend that provides access to an industrial simulator such as for instance the *Cadence AMS Designer* [20]. By parsing the command-line output of the simulator, performance parameters of the conducted simulation are extracted.

Since this framework is implemented as a Python package, algorithms may be implemented on top of the shown functionality. Examples of these algorithms are shown in the following section.

C. Application

The invasive analysis algorithms shown in this section follow the implementation pattern shown in Fig. 13. Following, examples for analysis-specific algorithms will be sketched.

1) Parasitic Impact Rating: For the design of AMS circuitry, the parasitic couplings introduced in the layout phase are a major issue. Even for carefully verified circuits, the performance may be severely degraded by these elements. The commercially available frameworks provide the designer with an augmented netlist of the circuit including additional elements for modelling these couplings. Still, this netlist does not provide the designer

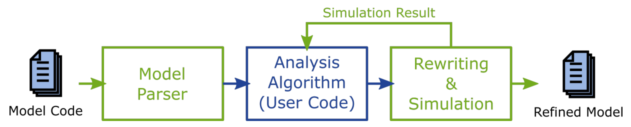


Fig. 13. Implementation pattern for the presented invasive analysis algorithms (Units realized by our framework shown in green).

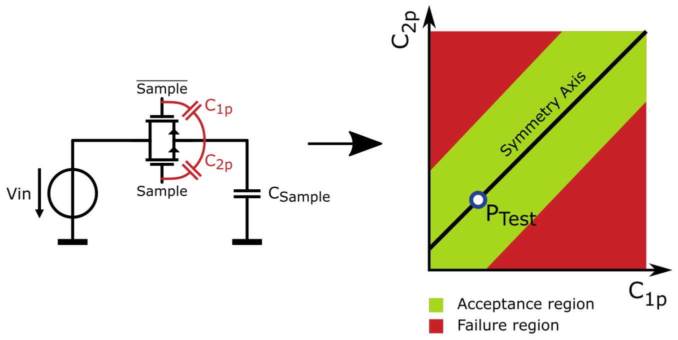


Fig. 14. The acceptance and failure regions of pairs of parasitics exhibit a certain shape in presence of symmetry constraints.

with information about the criticality of these elements – and therefore no starting point for optimization.

The proposed invasive analysis tries to answer this question: By successively removing subsets of these parasitic elements and re-simulating the circuit, the impact of individual parasitics can be extracted. Hence, the user algorithm selects the parasitic elements to be removed and creates a rating due to their impact on the simulated circuit performance. This rating provides hints for sensitive (or aggressive) signals in the circuit. It enables the designer to systematically enhance the circuit's robustness to the relevant couplings. In this way, the lengthy and costly process of post-layout circuit debugging can be significantly improved. It has been shown that this method is even applicable to EMI (electromagnetic interference) issues in industrial-size circuits [21].

2) Parasitic Symmetry Analysis: In analog layout, symmetry is a fundamental concept to ensure matching, i.e. the reduction of variations in size-relations. Traditionally, constraints for the symmetry of functional elements can be defined in modern design frameworks. For parasitic elements, the treatment of symmetry relations has not been in focus of research so far. Still, the knowledge about these relations may ease the overall layout generation process. Invasive analysis provides a novel approach here [22]. Each parasitic symmetry, as shown in Fig. 14, exhibits a certain (symmetrical) shape in the parameter space. In the context of our framework, the user algorithm *inserts* hypothetical parasitic elements into the netlist. By simulation, the parallelogram shape of the region is extracted using the maximum tolerable parasitic capacitor values. In the figure, these are given by the intersection of the red/green transition line with the axis. Following, a single point P_{Test} on the symmetry axis is evaluated. If the simulation passes, i.e. the circuit's performance target is reached, P_{Test} is part of the acceptance region and therefore, the symmetry constraint is present.

This method provides the designer with a powerful tool for extracting layout constraints from a given circuit without prior knowledge about the final parasitics. It has been proven that even in rather simple circuits, a high number of these constraints may be present. Therefore, the pre-knowledge is of significant value for the layout engineer reducing the effort for post-layout fixing. In the future, this algorithm could be of value for automated layout generation algorithms.

3) Monitor Generation and Insertion: The augmentation of existing functional models with additional checks is currently a mostly manual effort. Still, many of these checks could be generated automatically, for instance if the model is operating in a valid region [23], if glitches on digital control signals may distort the operation, or if the model is generating implausible output values outside of the supply-range. Note that most of these checks are in principle model agnostic, i.e. they can be realized in a similar way for different models. Using our framework methods for code generation, monitor modules can be conveniently derived. In combination with the model-rewriting step, an augmented model for system verification can be generated. For special cases, such as checking if the model is operating in a valid region, these checks can even be generated automatically as shown in [23].

This method can be used for instance in maintaining or enhancing existing model libraries. The code to be maintained can be reduced to the purely functional behavior while the checks and additions for nonfunctional properties can be added automatically.

D. Discussion

The presented Invasive Analysis Framework provides the functionality to examine the behavior of a system subject to nonfunctional effects. Methods for extracting properties such as criticality or symmetry constraints for parasitic elements can be implemented to ease the design process. From the modelling perspective, it enables the verification process by augmenting existing models with additional checks or properties. In perspective, the methods can be used for new kinds of sensitivity or stability analysis by insertion of saboteur modules in the design domain. In the verification domain, the augmentation of models by additional properties such as for instance transient power consumption can be targeted.

VI. Outlook

The methods presented here provide various ways to significantly improve the mixed-signal design flow. Further automation can only be achieved by combining existing methods – those presented and those published by other researchers – to create a novel design flow. This poses a great challenge on our way to an automated design framework that reduces risk and improves the quality of analog and mixed-signal design.

Acknowledgment

Our work on *constraint propagation and budget calculation* was supported by the German Ministry of Education and Research (BMBF) as part of the autoSWIFT project under grant 16ES03.

Achievements in *layout generation* result from the project Productive4.0 (grant no. 16ESE0183S) which is funded by EU, the German Ministry of Education and Research (BMBF), and Saxony.

The shown *invasive analysis framework for nonfunctional effect verification* is based on the results of the project ANCONA and is currently conducted by the research group IntelligEnt. The project ANCONA was funded by the German Federal Ministry of Education and Research (BMBF) in the ICT2020 program under grant no. 16ES021. The IntelligEnt research group is supported by the Free State of Thuringia, Germany, and the European Social Fund under the reference 2018 FGR 0089.

References

[1] P. Jespers, *The g_m/I_D Methodology, A Sizing Tool for Low-voltage Analog CMOS Circuits*. Springer, 2010.

[2] J. W. Crossley, “BAG: A designer-oriented framework for the development of AMS circuit generators,” Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec. 2014. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-195.html

[3] Cadence Design Systems, *Verilog In for Virtuoso Design Environment User Guide and Reference*.

[4] M. R. Müller, “Digital centric multi-gigabit SerDes design and verification,” Ph.D. dissertation, Combined Faculty of Natural Sciences and Mathematics, Heidelberg University, 2018.

[5] J. Scheible and J. Lienig, “Automation of analog IC layout: Challenges and solutions,” in *Proc. Int’l Symp. on Phys. Design (ISPD)*, 2015, pp. 33–40.

[6] A. Krinke, G. Jerke, and J. Lienig, “Constraint propagation methods for robust IC design,” in *Proc. 8th Symp. on Reliability by Design (ZuE)*, 2015, pp. 7–14.

[7] A. Krinke, L. Lei, and J. Lienig, “Predictive system-level constraint verification and optimization,” in *Proc. 9th Symp. on Reliability by Design (ZuE)*, 2017, pp. 40–45.

[8] 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 (ICECS)*, 2013, pp. 285–288.

[9] A. Graupner, R. Jancke, and R. Wittmann, “Generator based approach for analog circuit and layout design and optimization,” in *Proc. Design, Autom. and Test in Europe (DATE)*, 2011, pp. 1–6.

[10] B. Prautsch, U. Eichler, S. Rao, B. Zeugmann, A. Puppala, T. Reich, and J. Lienig, “IIP Framework: A tool for reuse-centric analog circuit design,” in *Proc. 13th Int’l Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2016, pp. 1–4.

[11] E. Chang, J. Han, W. Bae, Z. Wang, N. Narevsky, B. Nikolić, and E. Alon, “BAG2: A process-portable framework for generator-based AMS circuit design,” in *Proc. Custom Integr. Circuits Conf. (CICC)*, 2018, pp. 1–8.

[12] B. Prautsch, U. Eichler, T. Reich, and J. Lienig, “MESH: Explicit and flexible generation of analog arrays,” in *Proc. 14th Int’l Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2017, pp. 1–4.

[13] B. Prautsch, U. Hatnik, U. Eichler, and J. Lienig, “Template-driven analog layout generators for improved technology independence,” in *ANALOG 2018; 16th GMM/ITG Symp.*, 2018, pp. 1–6.

[14] R. Martins, A. Canelas, N. Lourenço, and N. Horta, “On-the-fly exploration of placement templates for analog IC layout-aware sizing methodologies,” in *Proc. 13th Int’l Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2016, pp. 1–4.

[15] A. Unutulmaz, G. Dündar, and F. V. Fernández, “LDS - a description script for layout templates,” in *Proc. 20th European Conf. on Circuit Theory and Design (ECCTD)*, 2011, pp. 857–860.

[16] S. Bhattacharya, N. Jangkrajarng, and C.-J. R. Shi, “Multilevel symmetry-constraint generation for retargeting large analog layouts,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 6, pp. 945–960, Jun. 2006.

[17] M. Schulze, “Entwicklung einer Programmierschnittstelle zur Verdrahtung von analogen integrierten Schaltungsblöcken,” Master’s thesis, Technische Universität Dresden, Germany, Sep. 2018. [Online]. Available: http://publica.fraunhofer.de/dokumente/N-519669.html

[18] K. S. Kundert and O. Zinke, *The Designer’s Guide to Verilog-AMS*. Kluwer Academic Publishers, Jun. 2004.

[19] M. Barnasconi, K. Einwich, C. Grimm, T. Maehne, A. Vachoux *et al.*, “Standard SystemC AMS extensions 2.0 language reference manual,” *Accellera Systems Initiative (ASI)*, 2013.

[20] Cadence Design Systems, *Cadence Virtuoso AMS Designer Environment User Guide Product Version 6.1.6*, Jun. 2015.

[21] G. Gläser, M. Grabmann, and D. Nuernbergk, “Impact rating of layout parasitics in mixed-signal circuits: Finding a needle in a haystack,” in *Proc. 15th Int’l Conf. on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2018.

[22] G. Gläser, B. Saft, and R. Sommer, “Parasitic symmetry at a glance: Uncovering mixed-signal layout constraints,” in *Proc. Frontiers in Analog CAD (FAC)*, 2017, pp. 1–6.

[23] G. Gläser, M. Grabmann, G. Kropp, and A. Fürtig, “There is a limit to everything: Automating AMS operating condition check generation on system-level,” *Integration*, Jun. 2018.