Adaptive Data Model for Efficient Constraint Handling in AMS IC Design

Andreas Krinke* krinke@ifte.de
Goeran Jerke† goeran.jerke@ieee.org
Jens Lienig* jens@ieee.org

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

***Abstract*—Further automation of analog and mixed-signal integrated circuit design requires the consideration of a growing number of design constraints. The processing of these constraints needs specific design information and depends on their target parameters and the type of mathematical requirement. Both aspects allow the creation of numerous different constraint types with many demands on the available design data. This paper presents a data model for AMS IC designs that is based on a static model for common design data. In order to store all information necessary for utilized constraint types, this model dynamically adapts and extends its internal structure. Thereby, our data model does not limit the set of supported constraint types and allows uniform access to design and constraint data for constraint-driven algorithms. We preserve the graph nature of our data model by using a graph database for its implementation. Thus, constraint engineering becomes much faster compared to conventional static data models.**

I. Introduction

The design of analog and mixed-signal integrated circuits (AMS ICs) requires the consideration of a large number of design constraints. Many different types of constraints exist that vary in their properties and in the way they need to be handled. Examples include symmetry, alignment and orientation constraint types used to ensure device matching. Depending on an IC's application, e.g., for automotive, sensor, or smart power products, adapted sets of constraint types are used. Each design step should incorporate all relevant constraints to create valid results. Due to the difficult and distinct handling of these numerous constraint types, this problem is one of the main reasons for the lack of automation and thereby the dominance of manual tasks in AMS IC design [1], [2]. The comparison with sophisticated algorithms for digital design automation emphasizes this point. Hence, a data model that adapts to the requirements of constraint types eliminates one of the main obstacles towards the long awaited automation in AMS design.

Constraint types define classification criteria used to group constraints into classes. These criteria include the affected design parameters (e.g. symmetry, resistance or shielding) and the semantics of the constraints' restriction (e.g. matching or limiting the values of these parameters). Constraints of the same type share methods for their verification and propagation throughout the design hierarchy. A constraint type also specifies the information needed to complete these steps, such as the set of design element properties whose values influence verification. Potentially, the application of various design tools from different vendors is required to gather all required data [3]. One possibility would be to include this data collection in each algorithm. This approach has the disadvantage that information cannot be reused by different algorithms or for multiple executions of the same algorithm; it has to be reacquired in these cases. In our opinion, storing required information in a common database is a better approach and provides both data persistence and sharing. Furthermore, this data abstraction provides uniform data representation as well as identical access to the information of different tools, thereby greatly simplifying algorithm development. Finally, if the constraint data model is integrated with the model for (not constraint-related) design data, algorithms can access both types of data in the same way. Nevertheless, for such a shared database, the underlying data model has to support information with arbitrary structure, not necessarily known in advance. As a consequence, the data model needs to adapt to this kind of data.

This work was supported by the German Ministry of Education and Research (BMBF) under grant 01M3195B.

It is important to note that the approach presented in this paper applies equally well to multi-domain design data and related cross-domain constraints (e.g. for chip-package-board co-design). Due to space limitations, this paper focuses on the IC design domain.

A. State of the Art

Proprietary databases of common EDA software tools for AMS circuit design use static data models for design and constraint data. While the addition of new custom constraint types is typically supported, required information either has to be collected repeatedly during each algorithm execution or stored using a custom data model and separated from design data. In these cases, both databases need to be synchronized.

Over the past few years, the OpenAccess (OA) database was adopted as replacement for proprietary databases in many EDA software tools [4]. Its data model can be extended with new object types and properties (see [5]), but their integration is purely application-specific. For example, the system does not support queries based on extended data. In addition, OA's data model is confined to IC design data.

B. Our Contributions

In this paper, we present a novel adaptive data model for design and constraint data as well as for related information needed to handle the constraints. This model provides algorithms with uniform data access regardless of whether or not they consider constraints. IC design data is highly connected because cell instantiations and constraints may create nearly arbitrary links between database objects. Therefore, we found graph databases to be very well suited for data storage. They allow much shorter query times compared to other types of databases, for instance relational ones, that are not optimized for such highly connected data. We implemented our data model using such a graph database and interfaced it with a commercial EDA software tool. The database provides uniform data access using a built-in graph query language. Finally the whole system was applied to several large industrial smart power IC designs to show its practical benefits.

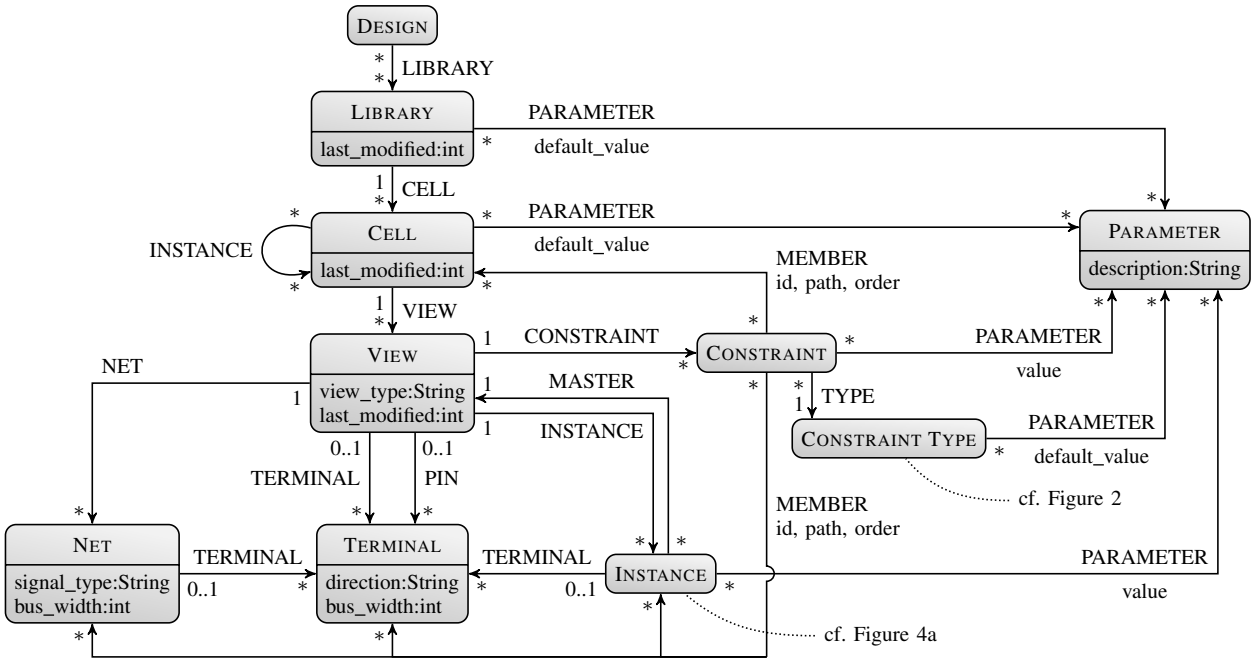

Figure 1. Graph describing valid nodes and relationships for the static part of our model for logical (schematic) data. It expands on traditional data models to include constraint types, constraints and their members, and a separate concept for parameters [6]. Most importantly, this model can adapt to information demands using mandatory and optional data model extensions (cf. Section III-D). Included are node and relationship parameters in lower case and names of relationships in upper case. The potential numbers of relationships starting or ending in a node are noted at the end or start of the respective arrow. Every node in the model has a string property *name* which was omitted for brevity.

II. Constraints and Constraint Types

The common hierarchical design approach breaks circuits into cells that may be reused multiple times in a design [6]. Modern software tools organize these cells in libraries and distinguish different views of each cell, e.g., symbol, schematic and layout views. Constraints belong to a specific schematic or layout view. They limit the allowed values of specific design element parameters and need to be fulfilled for design closure. The affected design elements (e.g. instances or nets) are the *members* of the constraint. Two types of members exist: (1) local members belonging to the same cell view as the constraint itself and (2) hierarchical members that belong to some instance further down in the design hierarchy. Unambiguous identification of such hierarchical members is based on the exact list of instances one has to traverse when going from the constraint's cell to the member. This path $i_1/i_2/\ldots/i_n/e$ starts in the view associated with the constraint, descends into instances according to the sequence i_1, i_2, …, i_n, and finally arrives at a design element e. For configuration purposes, constraints itself may have general and/or per-member parameters. Depending on the compatible design elements, affected parameters and the semantic of the restriction, different types of constraints can be identified. As a consequence, constraint types define how to process derived constraints. In addition to constraint verification, this includes propagation, i.e., their spreading throughout the design hierarchy into all cells with relevance to the constraint [7].

The diversity of possible constraint types is caused by the large number of parameters in AMS designs on the one hand, and the set of potential restrictions on the other. To illustrate this point, current literature lists about 354 such restrictions [8]. Algorithmic consideration of these numerous constraint types often requires additional information about the design. For example, algorithms that obey floorplanning constraints on the position, orientation and alignment (relative position) of instances need to have access to the respective parameters of all relevant instances.

We propose a single data model for design and constraint data that is dynamically extended in order to store this required information. Thereby, algorithms have a uniform interface for accessing information which not only simplifies their implementation but also increases their runtime efficiency.

III. Data Model

At its core, our data model uses a static model that is independent from the constraint types used in the design. Figure 1 shows this static model anchored in the node DESIGN which exists exactly once for each circuit. We store this AMS IC design and constraint data as property graph. This graph model consists of nodes and relationships, where latter are named and directed with exactly one start and one end node. Both nodes and relationships contain properties, described as key-value pairs [9].

A. Library Organization

Our data model complies with the common hierarchical design approach described in Section II. As shown in Figure 1, a DESIGN node references zero or more LIBRARY nodes (written as "$*$" near the arrow's head) using a relationship of type LIBRARY. A library may belong to several designs (noted as "$*$" near the same arrow's tail). Similarly, each library may contain an arbitrary number of cells. The same holds for cells referencing a set of views. Besides the name of the element, each library, cell and view stores a time stamp of its last modification used for database synchronization (cf. Section IV-B). In addition, each view has a property defining its type (e.g. "symbol", "schematic" or "layout"). Mapping views that assign logical and physical views to instances are not included in our data model.

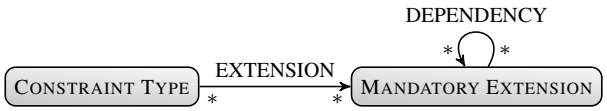

Figure 2. Graph data model for mandatory extensions required by custom constraint types. These extensions may depend on each other, thereby forming a dependency graph.

B. View Model

According to Figure 1, views contain INSTANCE, TERMINAL and NET nodes. Instances represent references of other cells and, therefore, relate to one of the symbol views of this cell using a MASTER relationship. At the same time, an INSTANCE relationship connects the corresponding cells. Later on, this helps to extract the design hierarchy and to answer the question which cell instantiates which other cells more easily. Each parameter that applies to some instance is modeled as a separate PARAMETER node. This includes, for instance, the width and length of a MOS transistor and its number of fingers. The current value of a parameter is defined using the *value* property of the PARAMETER relationship connecting instance and parameter. Default values can be set by relating libraries and/or cells with a parameter using a likewise named relationship while specifying its *default_value* property.

C. Constraint Model

As shown in Figure 1, a constraint instance belongs to a particular schematic or layout view. While local constraint members, i.e. nets, terminals and instances, are directly referenced using MEMBER relationships, hierarchical members require multiple such relationships. This set of edges is sorted using the "order" property and according to the element position in the hierarchical member's path as described in Section II. Thereby, hierarchically design elements are correctly addressable. Each constraint is associated with its constraint type, which defines all available constraint parameters and their default values. They can be overwritten using PARAMETER relationships to connect constraint and parameter nodes.

D. Adaptive Model Extension

The major innovation of the proposed data model is its adaptability. Conventional static data models for AMS IC design decrease the efficiency of procedures requiring data that is not available and yet has to be collected. In our case, the data model can be extended in order to maintain fast access to required information. On the one hand, such data model extensions can be *mandatory*, thus causing global database modifications. On the other hand, if database changes are local, the extensions of the model are *optional*. When choosing the kind of extension used for an algorithm, one has to balance between two opposing interests: The information retrieval speed on the one side, and the size and update time of the database on the other.

1) Mandatory Data Model Extensions: Mandatory extensions to the static data model have global effects on the structure of the database. During every data update, all these extensions have to be respected and corresponding data has to be created or updated accordingly. Therefore, superior information retrieval speed later on is traded for a potentially much larger database and slower database updates. For our application to custom constraint types, this balance depends on how many constraints of this type exist, how much information needs to be added to the database and how long the information retrieval takes.

As shown in Figure 2, custom constraint types may require multiple such extensions which might itself depend on other extensions. An example is a constraint type whose verification is based on the area of affected layout views. The corresponding extension requires information on the view's bounding box and, therefore, on another extension for calculating and storing the area in the database. Thus, all mandatory extensions form a dependency graph. After topological sorting, this graph dictates the extensions' execution order upon data export and synchronization. It is advisable to integrate the data model shown in Figure 2 into the main data model of Figure 1.

2) Optional Data Model Extensions: For rarely used constraint types, it is not advisable to use mandatory extensions for storing information related to constraint handling. The size of the database would increase disproportional in relation to the benefits. In this case, optional model extensions allow storing this data only for local, relevant parts of the graph. These optional model extensions are unique for each constraint type. Every type specifies constraint handling procedures which implicitly define these extensions simply by storing resulting information in the database accordingly. We recommend to perform the database update upon constraint creation. At this time, algorithms (e.g. for constraint propagation) calculate required information which is stored in the database together with the operation's result. Because such an extension is optional, the non-existence of matching structures in parts of the graph does not mean that corresponding design properties are missing. The information was just not collected. Therefore, optional model extensions must uniquely mark adjoined parts of the graph for later identification.

Constraints add essential information to the design, and thus, become an integral part of the design data. Therefore, the removal of constraints is a rare occurrence but should be supported nevertheless. Upon constraint deletion, the associated data according to the optional data model has to be removed from the database. If all constraints of a type are removed, the data according to the type's mandatory data model can be deleted during future database synchronization tasks. For both operations to work, all data in the database has to be uniquely identifiable as design data, or as belonging to a particular (mandatory or optional) data model extension.

IV. IMPLEMENTATION

The data model is one of the most critical parts of each EDA tool. Therefore, it should be tightly integrated with the tool for maximum performance. Cadence DF II, the testbed of our choice, comes with its own data model and database. Hence, we did not have the option of tight integration, and instead extended this IC design framework using a second database based on our model. Of course, this "shadow database" creates overhead caused by synchronization tasks, such as data gathering, cleanup and updating. As an example, the initial data export using the static data model takes about 15 minutes for an industrial design on a workstation. To mitigate this synchronization effort, partial database updates were implemented (see Section IV-B). It must be pointed out that this overhead is caused by our decision to extend a commercial design system and is no inherent problem of the novel data model.

The data model's implementation consists of two major parts: (1) a graph database, and (2) software components that allow database interaction using DF II.

A. Design Tool Integration

We implemented our data model using Neo4j [10], an open-source property graph database. It is schema-free, i.e., it does not restrict the structure of the graph to be stored. As a consequence, this means that any schema is implicitly defined by the application accessing the database.

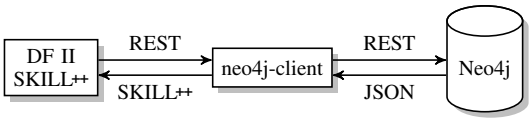

Figure 3. Overview of the data model integration into Cadence DF II using SKILL++ and the graph database Neo4j.

Figure 3 shows the integration of the novel data model into Cadence DF II using the built-in scripting language SKILL++ [11]. The goal is to access the graph database Neo4j using SKILL++. Neo4j runs as a server providing a REST API for database access. Because SKILL++ does not allow direct use of this API, we created the C program neo4j-client which translates between the two. Using standard input/output streams, one can use SKILL++ to send REST commands to neo4j-client. The commands are then forwarded to Neo4j which performs the requested database operation and responds using the JSON format [12]. This response is later converted to SKILL++ format by neo4j-client and can be evaluated and imported into DF II.

B. Data Export and Synchronization

During the data export, all libraries, cells and views of the IC design are processed and the corresponding nodes and relationships are created in the database in compliance with the static data model. For each library, cell and view, the time of its last modification is stored in the database. Therefore, the next data export can skip all unmodified elements. Depending on the number of changes, this technique allows a much faster data export. In addition, the very same method enables the update of modified views when a designer saves the recent changes. Overall, the overhead of a shadow database is thereby greatly mitigated.

V. Examples

We applied our new data model to several industrial IC designs. The largest of those included 175 libraries, 3407 cells, and 7396 views (among them 1348 schematic views and 1984 layouts views). The initial export to the database took about 15 minutes resulting in a graph with 167 548 nodes, 308 383 relationships, and 365 154 properties. The database has a size of 103 MB and easily fits into RAM, thus allowing fast access and query times. As an example, querying all cells that have at least one schematic view and are not instantiated anywhere in the design takes about 800 ms and finds 510 cells. Using SKILL++ in Cadence DF II, the same query takes about 44 s (best of 5 runs). We could provide many more examples with a benefit comparable to the 55-fold speed increase seen above. However, it should be noted that these isolated examples fail to show the key advantage of our data model. Its value results from the increase in efficiency for *all* aspects of constraint engineering including the runtime of constraint-driven design algorithms. For each constraint, propagation taken alone performs a large number of data queries similar to the example above, thus multiplying the saved time.

Figure 4 shows two data model extensions that demonstrate the adaptivity of our data model to custom constraint type requirements. The mandatory extension in Figure 4a is used for constraints generated during floorplanning, which include layout alignment, and maximum area constraints. This extension adds information about the position and bounding box of instances to the data model. Thereby, verification and propagation of these constraint types have direct access to required information. For constraint types that target current mirrors, the second extension shown in Figure 4b adds the ability to flag instances accordingly. As a result, procedures that handle such constraints are able to identify transistors that belong to current mirrors. This is an optional extension, because the large number of newly created CURRENT MIRROR nodes might otherwise lead to significant overhead in database size and synchronization time.

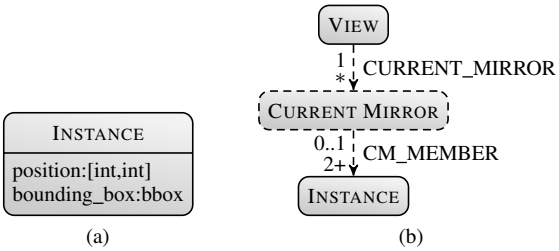

Figure 4. Two examples for data model extensions. (a) Mandatory extension for floorplanning constraint types that adds position and bounding box information to INSTANCE nodes (cf. Figure 1). (b) Optional extension (marked with dashed lines) that allows to store information about current mirrors and associated devices.

VI. Conclusion

In this paper, we presented an adaptive data model for AMS IC designs. This model can be applied either by integrating its principles into database implementations of EDA tools, or by using a shadow database as shown in this paper. It is built around a static data model for common design data that can adapt to emerging information requirements using data model extensions. This is in particular beneficial for custom constraint types, whose verification and propagation may require information not available in static data models. Our graph-based model was integrated into Cadence DF II using the Neo4j graph database. Thereby, constraint-driven algorithms get uniform access to design and constraint data potentially produced by many different design tools of various companies. This allows effective constraint handling and eliminates one of the main obstacles towards the long awaited automation in AMS design.

References

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

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

[3] G. Jerke, J. Lienig, and J. B. Freuer, "Constraint-Driven Design Methodology: A Path to Analog Design Automation," in *Analog Layout Synthesis – A Survey of Topological Approaches*, H. E. Graeb, Ed. New York: Springer, 2011, pp. 271–299.

[4] J. T. Santos, "Overview of OpenAccess: The Next-Generation Database for IC Design," in *OpenAccess 2003 Conference*, 2003.

[5] Z. Xiu, D. A. Papa, P. Chong, C. Albrecht, A. Kuehlmann, R. A. Rutenbar, and I. L. Markov, "Early Research Experience With OpenAccess Gear: An Open Source Development Environment For Physical Design," in *Proc. Int'l Symp. on Phys. Design*, ISPD, 2005, pp. 94–100.

[6] M. Bales, "Design Databases," in *EDA for IC Implementation, Circuit Design, and Process Technology*, Electronic Design Automation for Integrated Circuits Handbook, L. Scheffer, L. Lavagno, and G. Martin, Eds. Taylor & Francis, 2006, vol. 2, ch. 12.

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

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

[9] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*, 1st ed. Sebastopol, CA, USA: O'Reilly, 2013.

[10] (2013) The Neo4j Website. [Online]. Available: http://www.neo4j.org

[11] T. J. Barnes, "SKILL: A CAD System Extension Language," in *Proc. 27th Design Autom. Conf.*, DAC, 1990, pp. 266–271.

[12] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627, Jul. 2006.