A Layout-to-Generator Conversion Framework With Graphical User Interface for Visual Programming of Analog Layout Generators

Sungyu Jeong, *Graduate Student Member, IEEE,* Chanhyong Lee, *Graduate Student Member, IEEE,* Minsu Kim, *Graduate Student Member, IEEE,* Iksu Jang, *Graduate Student Member, IEEE,* Myungguk Lee, *Graduate Student Member, IEEE,* Junung Choi, *Graduate Student Member, IEEE,* and Byungsub Kim, *Senior Member, IEEE*

***Abstract*—We propose a visual programming framework that helps a designer easily convert an existing analog layout into the layout generator. Using a graphical user interface (GUI), designers can easily load an existing layout, convert it into the layout generator, and visually verify the generated layout result. A GUI-supported visual programming method enables intuitive and straightforward programming to significantly reduce the required programming skills and coding workload. Through program blocks, designers can easily describe and compile a layout generator. Layout-code synchronization updates the program blocks automatically when layout elements are created, edited, or deleted via GUI. Expression assistance semi-automatically completes parameterized expressions of layout geometry through simple GUI manipulation. These methods greatly reduce the time and workload for development of the analog layout generator. Using the framework, a complex 20 Gb/s high-speed wireline receiver was converted to the generator in 20 hours, which is just 45% of the time required for manual coding of the layout generator. Other high-speed analog blocks including DCDL, and DCC circuits were also converted to generators. The generators created hundreds of different layout designs, all of them passed design rule verification. The generated layouts achieved almost the same performance in post-layout simulation as the reference layouts.**

***Index Terms*—analog layout generation, layout-to-generator conversion, code automation, graphical user interface**

This work was supported in part by the Commercializations Promotion Agency for Research and Development Outcomes (COMPA) grant funded by the Korea Government (MSIT) (No. 2023I100); in part by Institute of Information and Communications Technology Planning and Evaluation grant funded by the Korea Government (MSIT) (No. 2022-0-01171); in part by BK21 FOUR Project of NRF for the Department of Electrical Engineering, POSTECH; in part by National R&D Program through the National Research Foundation of Korea(NRF) funded by Ministry of Science and ICT(2020M3H2A107804514).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors. This includes a multimedia MPEG format movie clip, which shows layout-to-generator conversion demo with an inverter example.

Sungyu Jeong, Chanhyong Lee, Minsu Kim, Iksu Jang, Myungguk Lee, and Junung Choi are with the Department of Electrical Engineering, Pohang University of Science and Technology, Pohang-si 37673, South Korea.

Byungsub Kim is with the Department of Electrical Engineering, the Department of Convergence IT Engineering, and the Department of Semiconductor Engineering, and the Graduate School of Artificial Intelligence, Pohang University of Science and Technology, Pohang-si 37673, South Korea, and also with the Institute for Convergence Research and Education in Advanced Technology, Yonsei University, Seoul 03722, South Korea (e-mail: byungsub@postech.ac.kr).

I. Introduction

THE increased complexity of design rules has made the traditional manual design of analog layouts time-consuming and costly. For example, design rules for manufacturability enforce additional constraints on layout designs to prevent various defects such as opens of vias and shorts of interconnect in recent technology nodes. In addition, as the performance of an advanced analog circuit is seriously affected by its layout, the physical design requires careful consideration for parasitic components, skew matching, and power integrity, and so on. As a result, the long design time and large cost of the analog layouts have become one of the major problems of the semiconductor industry.

To shorten the design time, various analog layout generation techniques have been proposed [1]–[6].

In template-based layout generation [1]–[3], placement of layout elements and routing connections are carefully specified and parameterized for a particular type of analog circuits in the program code of the generator. Once the layout procedure is codified in the generator, design time for the specific circuit is significantly reduced. Because the designer has complete control over placement and routing, the generator can usually produce high-quality layouts if it is programmed with careful planning of the placement and routing by an expert analog circuit designer [7]. Therefore, once programmed well, the generator enables quick and easy creation of high-quality layouts of the target analog circuit type for various design parameters in various different technology nodes.

However, the difficulty of manual programming to automate layout design causes the following limitations of the template-based layout generation methods. First, designers have to spend too much time manually writing code for the generator templates. Because the quality of the generated layout completely depends on the planning of the placement and routing in the program, various design issues like matching, parasitic components, and design rules must be carefully considered during writing the code. Second, designers must have expertise in both analog circuit design and programming to write the program code for high-quality layout generation. Because such designers are rare and usually expensive, implementation of this approach is limited by human resources.

In optimization-based layout generation [4]–[6], layouts are

directly generated from circuit netlists to avoid the limitations of template-based layout generation. Using various algorithms for auto-annotation on netlists, recognition of layout constraints, placement and routing, etc., these methods could significantly reduce the programming efforts of human designers compared with the template-based methods.

However, they still partly rely on template-based generators in order to produce low-level building blocks of analog circuits. In this approach, template-based generators including PCells are utilized to generate high-quality low-level building blocks, and then high-level placing and routing for them are fully automated instead of developing high-level generator templates. In this way, the programming effort is no longer required for high-level layout design while the developed generators for low-level blocks can be reused to quickly generate many different layout instances in various technology nodes, improving the development efficiency. Therefore, the long development time and the programming difficulty of template-based generators are also important issues in optimization-based layout generation methods.

This paper presents a layout-to-generator conversion framework in order to address the long development time and the programming difficulty of the template-based layout generation methods. The proposed framework helps a designer quickly convert a reference layout to the corresponding template-based generator.

The layout-to-generator conversion technique can greatly reduce the development times of analog layouts. For decades, many handcrafted analog layouts have been stored in databases of semiconductor companies. If we convert these layouts to the generators, then the converted generators can be directly used in design modification and technology migration, greatly reducing the layout workloads. In product development, a large part of the workload of designing analog layouts is reproducing analog layouts that are similar to one of old designs with different design options. For example, with technology scaling, a layout design is frequently migrated into a new technology node without changing the circuit topology. Also, analog circuits of the same topology are often differently designed for various different target specifications set by diverse market needs. For example, device sizes or aspect ratios are differently designed. Furthermore, in many cases, different analog applications have similar analog sub-cell layouts of the same circuit topology but with different design options. For instance, typical two-stage op-amps are widely used as sub-cells in various different analog applications. If such widely-used sub-cells are converted to the generators, then large workloads of drawing such sub-cells can be reduced. In addition, if a designer manually makes a new analog layout using the generated sub-cells, then the newly designed layout can be also converted to a new layout generator, which is also a valuable design asset (Fig. 1). In such a way, a company can easily grow the design assets of layouts and layout generators through such generation cycles (Fig. 1) utilizing layout-to-generator conversion. The expanded design assets will further accelerate layout development.

Using the proposed framework, the professional expertise required for the development of the generators as well as the analog layouts can be greatly reduced too. If a well-made reference layout is converted to the generator, then the layout styles and the know-how applied in the reference layout are preserved in the converted layout generator code. Therefore, the designer does not need to re-create the plan for the layout generation procedure, and thus the knowledge of analog design is less required. Even, a novice designer can easily make high-quality layouts using the generator converted from a well-made reference layout. In this sense, the quality control of layouts during product development can be improved by this approach. Also, because the framework provides a convenient GUI-based visual programming method and various programming assistance techniques, the programming skills and workload required for code development are greatly reduced.

The major contributions of this work are listed as follows.

- The framework provides a convenient graphical user interface (GUI) for layout-to-generator conversion.
- Visual programming enables easy handling of a complex generator hierarchy, simple description for a generator, and simple compilation of the generator description.
- Programming assistance techniques including layout-code synchronization and expression assistance are proposed to reduce the amount of coding work and programming skills required of circuit designers.

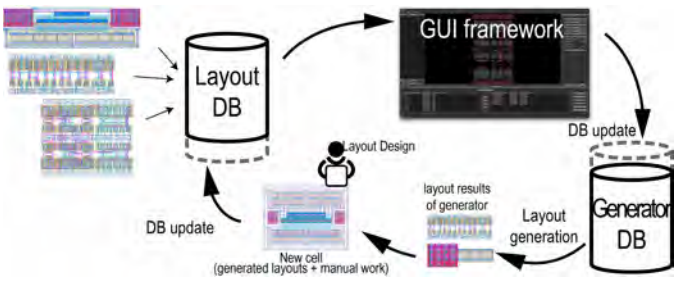


Fig. 1. The generation cycle of libraries of layouts and layout generators using the framework.

II. Proposed Framework

A. Overview

The framework provides a convenient visual programming environment for editing/handling a layout and a generator as well as for layout-to-generator conversion (Fig. 2). A designer can easily load a layout from a GDSII file or draw it using GUI. Creating, editing, loading, and saving layouts and generators can be done with GUI at any time for convenient

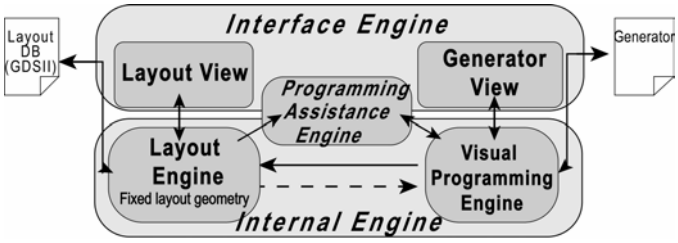


Fig. 2. The architecture of the proposed framework.

work management. In addition, drawing of layouts and layout-to-generator conversion can be done piece by piece in a mixed order, allowing step-by-step design, conversion, and verification.

In the proposed framework, instead of writing code to develop a layout generator, a designer can graphically manipulate a program tree to convert a layout into the generator. A program tree is a visual representation of code for a layout generator. A program tree (Fig. 7) is composed of program blocks. Each program block represents an operation and corresponds to a unit piece of generator code in the program tree. From the layout, its program tree can be easily constructed and can be later compiled into the generator code. The design hierarchy is visualized by the structure of the program tree and can be easily modified by changing its structure with GUI manipulation. For example, a designer can easily load an NMOS layout from a GDSII file or draw it using GUI. From the NMOS layout, its program tree can be easily constructed and can be later compiled into the generator code. The designer can easily parameterize the NMOS's geometric dimensions like the finger width by editing the program block on GUI. Once the parametric description of the NMOS is done, the program tree can be compiled into the generator code written in Python. The designer can run the converted generator and visually verify the generated NMOS layout through GUI. In this way, design modification is very easy and convenient: for example, the finger width can be easily modified by changing the parameter and running the generator via GUI.

The visual programming method reduces the required coding skills and the amount of workload for the generator development. First, a designer can easily handle the hierarchical structure of a complex layout generator with the visual programming method. Because the program tree has an advantage in manipulating hierarchical data due to its tree structure, the visual programming has an advantage in handling a hierarchically complex layout generator, too. Second, the visual programming helps a designer describe a layout generator easily. Program blocks of a program tree are graphical representations of abstraction methods like macros. Because visual programming provides a high-level abstraction method, a designer can easily describe and intuitively understand a generator program via GUI interface without knowing the complex programming syntax required by the conventional text-based programming method. Therefore, a designer does not need to type a long text of the program code as in the conventional text-based programming method. Third, visual programming enables the designer to intuitively change the generator flow by simply graphically manipulating the structure of the program tree. By selecting the appropriate program blocks and placing them in the appropriate places in the program tree, a designer can easily manipulate the generator flow. Therefore, the amount of workload for generator development can be greatly reduced.

The visual programming process is further enhanced by two program assistance techniques: 1) layout-code synchronization, and 2) an expression assistance tool.

With the first technique, when a visualized layout instance is modified, the corresponding program tree and blocks are automatically synchronized. For example, when a designer loads a reference layout, its temporary program tree is automatically initialized. The temporary program tree consists of initialized program blocks without appropriate configuration of input parameters. By appropriately configuring these input parameters, the program blocks are ready to be compiled into pieces of generator code. Therefore, this method greatly reduces the required coding skills and the workloads.

The second technique is a GUI-based tool that helps a designer describe parametric expressions. Complex parameterized geometric expressions of a layout element can be semi-automatically completed by simple GUI operation by using the expression assistance tool. The second method also greatly helps to reduce the workloads and required coding skills.

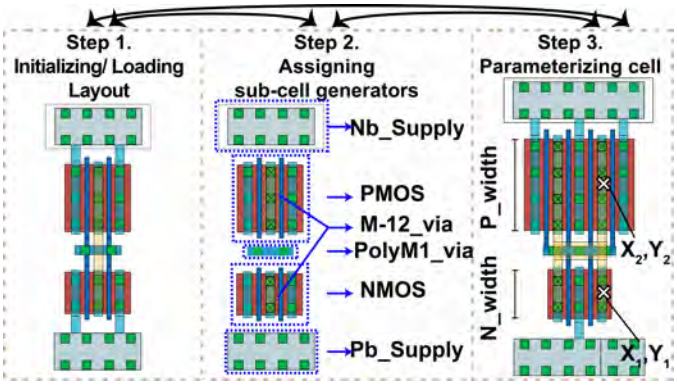


Fig. 3. An example design flow of inverter generator development using the proposed framework. We have included a supplementary MPEG format movie clip, which shows inverter generator development using the framework. This will be available at http://ieeexplore.ieee.org

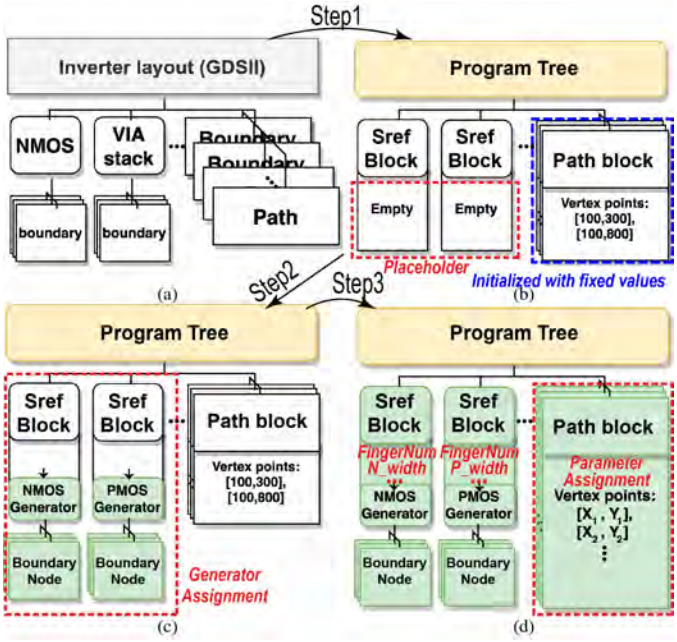


Fig. 4. An example design flow of visual programming of an inverter generator using the proposed framework.

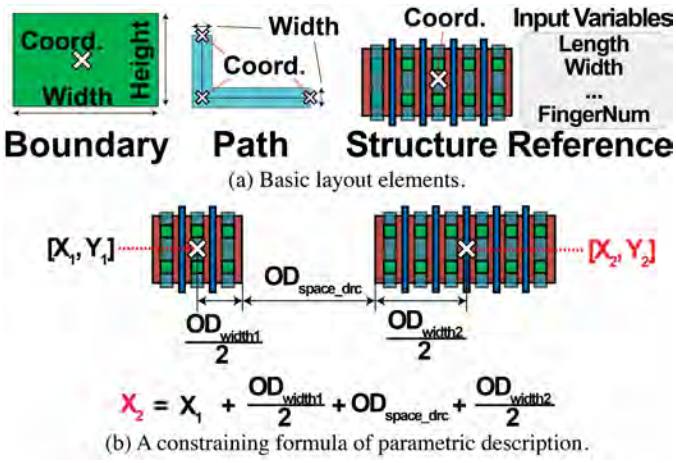


Fig. 5. Basic layout elements and parametric descriptions.

B. Proposed Design Flow

The framework has three design steps (Fig. 3, Fig. 4). A designer can freely move between steps at any time as needed.

In step 1, a reference layout is loaded (Fig. 4a), and then a temporary program tree is initialized (Fig. 4b). The initialized temporary program tree has the same design hierarchy as the reference layout. For the sub-cells (Sref in GDSII format), placeholders of the generators for the sub-cells (sub-generators) are automatically assigned in the initialized program tree. For basic layout elements such as boundaries and paths, the program blocks of their types (these can be compiled into the generators of boundaries and paths) are automatically assigned. The input arguments of the program blocks are initialized with the dimensions and positions of the loaded layout elements. Because the initially generated program blocks are not described with parametric expressions, they are not ready for compilation to the generator code yet.

In step 2, available generators are assigned for sub-cells (Fig. 3 and Fig. 4c). Because the program blocks of sub-cells are initially filled with sub-generator placeholders in step 1, the generator hierarchy is not completed in step 1. In step 2, if any generator available in the library can generate one of the sub-cells, then the generator can be assigned to the sub-cell by a designer. In Fig. 3 and Fig. 4c, for example, the appropriate generators are assigned to the sub-cells 'Nb_Supply, 'Pb_Supply, 'PMOS', 'NMOS', 'PolyM1_via', and 'M-12_via'. For this purpose, a designer can navigate through the layout hierarchy, select the target sub-cell, and assign the correct generator using GUI. If there is no appropriate generator for the sub-cell, a designer can flatten the sub-cell or can develop its generator first. By reusing available generators for sub-cells, the description time can be dramatically reduced.

In Step 3, geometric dimensions, coordinates, and design options of layout elements are parameterized and described with formulas and variables in program blocks (Fig.3, Fig. 4d, and Fig. 5). Parametrization can be done by filling in the corresponding argument fields of the program blocks through GUI with almost no coding skill. For example, the finger widths of the PMOS and the NMOS of the inverter in Fig. 3 are parameterized with the input variables P_width, and N_width, respectively. For this purpose, P_width and N_width are assigned as input variables of the sub-cell generators of PMOS and NMOS, respectively (Fig. 4d). Relative positions such as vertex coordinates of a path are also parameterized with variables [X1, Y1] and [X2, Y2] (Fig. 3 and Fig. 4d). By changing these variables, the generated inverter layout can be easily modified accordingly. Relative positions between elements or geometric dimensions of elements are described by formulas in terms of variables and design rules in order to impose geometric constraints [3] (Fig. 5b). If these formulas are appropriately described, then the generated layout can be in an appropriate shape even though its layout elements are resized and repositioned by change of input variables.

Once the generator conversion is completed, the generator can be used in various ways. A designer can generate GDSII files by simply entering input parameters into the generator using the framework. Alternatively, the generator can be converted to a layout generator source code written in Python to be used without the framework. It can also be used as a sub-cell generator of a new layout-to-generator conversion project.

C. Software Architecture

The framework consists of the internal engine, the interface engine, and the programming assistance engine (Fig. 2). The internal engine processes the key data like layout information and the generator description. The interface engine visualizes design information on windows and provides GUI for development. The programming assistance engine helps designers by partially automating generator development.

1) **Internal Engine:** The internal engine consists of the layout engine and the visual programming engine (Fig. 2). The layout engine handles data of layout instances. The data are updated when a reference layout is loaded from a GDSII file or when the generator under development is executed. From the data, the layout engine can create a GDSII file too. The visual programming engine handles the generator description that can be compiled into the generator source code. It can also execute the programmed generator in order to update the layout data through the layout engine. Because the compilation of the generator description and the execution of its compiled generator can be done at any time, the designer can conveniently verify the generator code during development.

2) **Interface Engine:** The interface engine has several windows: the layout view, the generator view, and the input variable window (Fig. 6).

The layout view provides an interface that 1) visualizes layout instances and 2) provides a convenient environment for generator development.

The layout view allows a designer to edit a layout through GUI like a regular layout editor. In the layout view, the layout element can be easily created, modified, or deleted by the designer. In addition, the layout view can selectively display layout elements of the selected layers. A new layout view can pop up to show the layout elements of a sub-cell.

The layout view also provides the user with a convenient interface for the development of a layout generator. There are

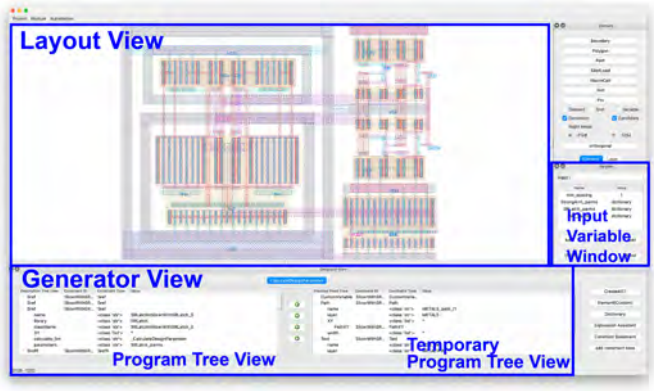


Fig. 6. A screenshot of the proposed framework.

three main interface functions: 1) referencing generator information using the layout element, 2) synchronizing generator code with the layout element, and 3) selectively displaying layout elements based on their development stage.

First, by selecting layout elements in the layout view, a designer can easily refer to the various types of information from the program block that generates the selected instance of the layout element. Those various types of information include the variables and the sub-cell generator classes used for the program block. If the top layout has very complex design hierarchies, then referring to a layout element in a very low level of design hierarchy is tiresome in the conventional text-based programming method. In such a case, the designer needs to dig into the code to find the path of the deeply nested objects. However, selecting layout elements and getting information in a GUI environment is much more intuitive. Therefore, this interface function provides a simple way to visually refer to the hierarchically nested objects.

Second, by modifying the layout element in the layout view, a designer can easily modify the generator code. For example, if a designer selects a path element that uses the metal-2 layer and changes its layer to the metal-3 layer in the layout view, then the corresponding program block is updated according to this layer change. Therefore, a simple modification of a generator can be easily done via GUI.

Third, the layout view can selectively show the fully converted instances of layout elements without showing the rest, and vice versa. For example, a designer may load an inverter cell as a reference layout and convert it into a generator. In this process, the designer may develop the generator by sequentially converting the sub-cells. If the NMOS has been fully converted into a generator and the PMOS has not been converted yet, then the layout view can graphically show the conversion progress. The layout view can selectively display the converted NMOS cell without showing the PMOS cell, and vice versa. This allows designers to progressively convert each instance of layout element to the generator while visually tracking the development progress.

These interface functions of the layout view make the generator development very convenient. This GUI-driven interfacing especially enhances productivity if the reference layout is hierarchically complex and the number of layout elements is large.

The generator view provides a convenient GUI environment that allows a designer to edit the generator by a visual programming method that barely requires coding workload and knowledge. To reduce coding workload and knowledge, the generator view provides a workspace to edit a program tree. By choosing the proper program block and passing its arguments, the designer can easily specify the operation without knowing the syntax. After editing individual blocks, the designer can easily modify the generator operation flow by moving the program blocks and modifying the program tree structure. Therefore, the generator view provides an interface to modify the program tree and blocks, reducing the required knowledge and the amount of coding for generator development.

The generator view is composed of two views to provide a convenient development environment for easy modification of the generator description flow: a program tree view and a temporary program tree view. The program tree view provides GUI for editing a program tree that can be directly compiled into the generator code. On the other hand, the temporary tree view displays the program tree in progress and provides an environment to modify it. Because the program tree in the temporary tree view is still incomplete, it cannot be compiled. A program block defined in each view can be moved to the tree in the other view at any time. A designer can modify incomplete program blocks in the temporary tree view, move complete program blocks to the program tree view, and then verify them without having interference from incomplete program blocks in the temporary program tree view. Therefore, with this separation, the designer can complete and verify program blocks one by one. For example, the temporary program tree view stores a temporary program tree which is initialized from the layout instance by layout-code synchronization when the reference layout is loaded. Its program blocks are not ready for generator compilation and must be edited by a designer. Once edited completely, each program block can be easily moved to any position of the program tree in the program tree view like playing with Lego blocks. Similarly, a subtree, which is a small tree whose root starts from a node of its program tree, can be also easily moved to any position of the program tree like a program block. In this way, the generator view provides an environment where the generator tree can be modified step by step, and the generator flow can be easily modified.

The Input variable window provides GUI to manage the input variables of the generator. With this window, a designer can see and edit input variables. These variables are categorized into required and optional input variables [3]. For example, the widths of PMOS (P_width) and NMOS (N_width) are the required input variables of an inverter generator while the numbers of contacts and the path width are the optional input variables. User can also set the default values of these variables. These variables are listed in the input variable window. The designer can specify the values for these variables or describe them with complex functional expressions.

3) **Programming Assistance Engine:** The programming assistance engine assists designers by partially automating the development of the program tree. Because the program

tree can be directly compiled into the generator source code, partially automating the development of the program tree corresponds to automating parts of generator programming. The programming assistance engine employs two main techniques: 1) layout-code synchronization and 2) expression assistance.

Layout-code synchronization enables a designer to develop or edit a generator without writing code from scratch by updating the corresponding program blocks accordingly when instances of layout elements are loaded, created, edited, or deleted. Another main technique is GUI-based expression assistance, through which a designer can easily get a complex parameterized expression describing the geometry of a layout element by simple GUI manipulation. By utilizing the automation techniques of the programming assistance engine, the engineer can greatly reduce the amount of coding and required coding skills.

III. Proposed Technique

For easy description of the generator and partially automating the development of the program tree, the framework is enhanced by various techniques, including the three key techniques: 1) visual programming, 2) layout-code synchronization, and 3) expression assistance.

A. Visual Programming

We adopted a visual programming method for layout generator development to reduce both coding workloads and required coding knowledge. This method provides a high-level representation of program operation, and thus designers can easily develop a generator with limited knowledge of coding. A circuit designer who is not familiar with the syntax can easily create a generator with little or no coding by just filling in the argument fields of the program blocks. For example, in order to add a piece of code generating boundary (a type of layout element) to a generator, a designer can create the program block of boundary class in the generator view. Or, the designer can draw a boundary in the layout view, and then the corresponding program block will be automatically generated by layout-code synchronization. To parameterize this program block, the designer just needs to pass the appropriate expressions to the arguments of the program block in the generator view. Or, the designer can do the same task more conveniently. The designer can open the dedicated pop-up window by clicking the instance of the layout element generated by the program block in order to fill the appropriate expressions in the argument fields through the pop-up window. In addition, visual programming allows easy manipulation of the layout generation flow. By arranging program blocks in a program tree via drag-and-drop, a designer can determine the execution order of the code. Likewise, the method of drag-and-drop for the program blocks in our visual programming enables easy handling of the complex design hierarchy of the layout generator. Therefore, our visual programming reduces the coding knowledge as well as the workloads required during the layout generator development.

This visual programming method requires much less amount of coding and programming knowledge than the method using

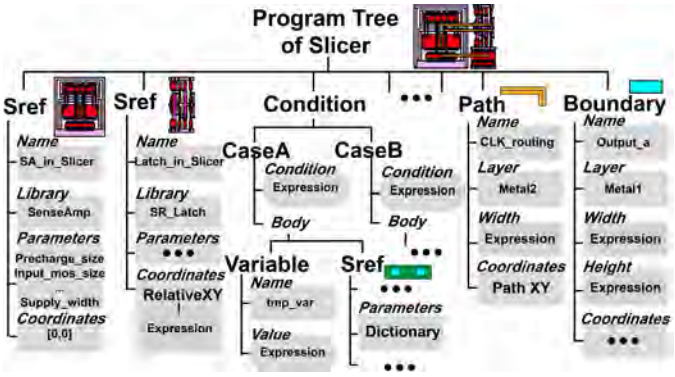


Fig. 7. Diagram of program tree for visual programming with Slicer example.

GUI-based templates in the prior art [3]. In [3], frequently used pieces of source code are provided as template options: the designer may select one of them and then write the rest of the source code. If the appropriate option is not provided, the designer must type the entire source code from scratch in [3]. Therefore, [3] requires coding knowledge to develop a layout generator. In contrast, the designer only needs to fill in the fields of program blocks without knowing the syntax of the program in our visual programming method.

We adopted a program tree to conveniently describe and edit a hierarchically complex layout generator via visual programming. The tree structure can store/access hierarchical data efficiently and is also efficient in inserting or removing sorted data. Therefore, the program tree is advantageous in hierarchically describing a layout generator and editing the flow of layout generation. The program tree consists of program blocks (Fig. 7). The program block is a basic unit for visual programming, which can perform a specific operation. For example, by creating a program block of boundary class and passing arguments, it can generate code that creates a boundary layout instance. The program tree can be compiled into a generator while each program block corresponds to a unit piece of generator code that can be easily added to or removed from the generator source code. Using the generator view, a designer can easily add a program block to or remove it from a program tree, like playing with Lego blocks. Therefore, a designer can easily describe a complex hierarchy of a layout generator by putting the appropriate program blocks in the appropriate locations in the program tree according to the generator's design hierarchy (Fig. 7).

There are two classes of program blocks: 1) the layout program block class and 2) the functional program block class.

A layout program block class can be compiled into a piece of source code that can generate a layout element. A boundary block, a path block, a sref block, an array block, and a hard macro cell block belong to this class. A boundary block has argument fields of layer, width, height as well as coordinates of the position to describe the boundary to be generated (Fig. 5a and Fig. 7). An sref block is used to describe a sub-cell, and therefore, it has argument fields of position coordinates as well as the name and input variables of the sub-cell's generator (Fig. 5a and Fig. 7). A hard macro cell block is used to refer to an existing sub-cell layout with fixed dimensions,

which can be seamlessly integrated into a layout generator. For example, a phantom cell that hides detailed layout information for security purpose belongs to this type. An array block provides a convenient way to describe an array of layout elements instead of using many other layout program blocks. Because a usual layout has many arrays, a designer can much more efficiently describe arrays with array blocks.

Functional program block class helps a designer easily describe parametric expressions and logical statements. Such expressions and statements include parameterized coordinates, vertex coordinates of a path, arithmetic operations, conditional statements, and repetitive generation of layout elements. For example, Fig. 8c shows an example usage of a logic condition block to place gate contacts of a MOSFET in a small area depending on its finger count. If the MOSFET has more than one finger, then the distance between the gate contact and diffusion layer is constrained by the minimum distance rule of metal 1 (M1 DRC in Fig. 8c). Otherwise, it is constrained by the minimum distance rule of poly layer (Poly DRC in Fig. 8c). By using the condition block, a designer can conditionally place the gate contact of the MOSFET. Fig. 8d shows an example generation of a resistor bank. Using loop blocks, a designer can easily describe the repetitive generation of resistor units depending on the number of rows and columns.

B. Layout-Code Synchronization

We proposed a layout-code synchronization to synchronize the instance of a layout element with the program block in order to reduce the generator development workload. When the instances of layout elements are loaded, created, edited, and deleted in the layout view, the corresponding program blocks are automatically updated. For example, when a designer draws a path element in the layout view, a program block that can generate the path element is automatically created. In another example, when a designer modifies attributes of a layout element, such as its position or layer types, the input values for arguments of the corresponding program blocks will also be updated. Similarly, when a layout element is deleted from the layout view, the corresponding program block is automatically deleted as well. By using a familiar interface of the layout view to manipulate layout elements, a designer can intuitively modify the generator without manipulating the program blocks directly. Therefore, the layout-code synchronization technique reduces the need of manual coding by automating the modification of the program block when a layout element is updated on the layout view.

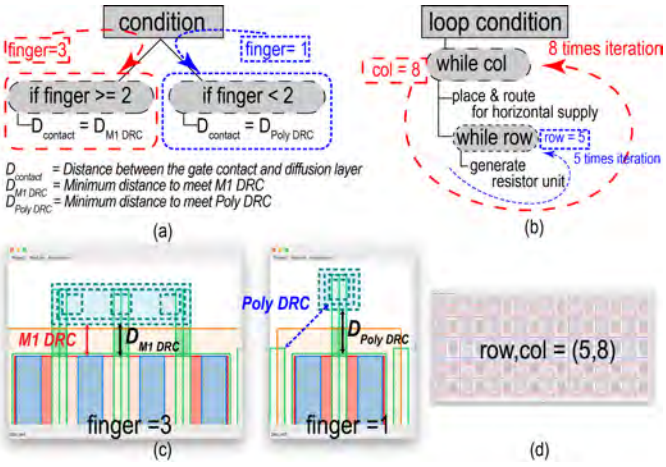


Fig. 8. The concepts of (a) condition block (b) and loop block. Generated (c) gate contacts examples using a condition block, (d) and a generated resistor bank example using a loop block.

By automatically creating a temporal program tree when a reference is loaded, the layout-code synchronization reduces the burden of creating the program tree from scratch. When a designer loads the layout, the structure of the loaded layout is analyzed, and the temporal program tree is automatically created. This temporal program tree is composed of program blocks that can generate the reference layout. Except the program blocks for the sub-cells, the input arguments of the program blocks in this temporal program tree are equal to the loaded layout's geometric dimensions. For the program block of the sub-cells, placeholders of the sub-generators are assigned by default. By replacing fixed dimensions with parametric expressions, the designer can easily complete the layout generator. Therefore, the layout-code synchronization can greatly reduce the workload of developing the skeleton of the program tree to create the generator of a reference layout.

For sub-cells, a sub-cell management feature of the layout-code synchronization enables a designer to manipulate the design hierarchy of the generator in the layout view. When a designer manipulates a sub-cell in the layout view, its modification is automatically updated to the program block by the layout-code synchronization. If a suitable generator is available for the sub-cell, then the designer can assign the sub-generator. Then, for the program block of the sub-cell, the initially assigned placeholder is automatically replaced with the suitable sub-generator. Therefore, much less amount of coding effort is required to describe the sub-cell. If the layout generator for the sub-cell is not available, then the sub-cell can be flattened or the sub-cell layout generator can be developed first. When a designer flattens a sub-cell, the program blocks for the flattened layout elements are automatically created. Then, the program tree is updated accordingly. When the detailed information of the sub-cell is hidden like a phantom cell, the designer can assign a program block of macro cell class for the sub-cell. By assigning a macro cell, the designer can use the sub-cell with a fixed geometry dimension in a generator. These sub-cell management features can be applied not only at loading a layout but also at any development stage of a generator. Therefore, a designer can easily build an appropriate design hierarchy of a generator by intuitively handling the sub-cells through the graphical interface of the layout view with the layout-code synchronization.

The layout-code synchronization helps a designer easily refer to the program block utilizing the interface of the layout view. In order to easily refer to the program block, a program block can be easily highlighted with layout-code synchronization when a designer selects the corresponding layout element in the layout view, and vice versa. This makes it easy to identify the program block which corresponds to the selected layout element, and vice versa. For instance, by selecting a

NAND gate cell in the layout view, the corresponding program block in the generator view can be highlighted. Compared to the conventional text-based programming, this highlighting feature offers significant efficiency in developing and editing a generator. In the text-based programming, to modify a code that generates a specific layout element, a designer must find the variable name of the generator object which generates the layout element. This process is especially time-consuming when a designer works with code written by others or does not remember the code details. Therefore, developing a layout generator requires time-consuming tiresome work such as finding the names of variables in the text-based programming method. In contrast, the highlighting feature of the layout-code synchronization significantly reduces the time spent searching for an object that corresponds to a specific layout element. By just selecting a layout element, the designer can identify the program block that generates the selected layout element without spending time searching for the variable names or the program block. More conveniently, by selecting a layout element, a designer can open the pop-up window which automatically refers to the corresponding program block and provides GUI for its edition. Therefore, the layout-code synchronization can save designers time by helping them quickly find the variable name of the program block of the selected layout element in developing and editing generators even if the generator was developed by others.

C. GUI-based Expression Assistance

We propose a GUI-based expression assistance technique which allows designers to describe geometric expressions easily through GUI. Referring to layout elements' geometric variables and formulas by manual coding is time-consuming. Especially, when referring to an element through a deep design hierarchy, an expression can be very long because of the long hierarchical path to the referred element (Fig. 9b). Memorizing such a long and complex expression is not easy, and thus a designer may easily make a mistake in writing code. The expression assistance reduces the burden of describing geometric expressions whereas writing of the geometric expressions was previously the most time-consuming task in the text-based programming method.

GUI-based expression assistance provides two interfaces to refer to the variables easily and to describe parametric expressions.

The referring interface of the expression assistance enables a designer to easily get a variable by several clicks on the GUI. Through the layout view, the designer can navigate through the layout hierarchy to refer to a layout element by selecting it. After selecting the layout element, by clicking the appropriate buttons on the expression assistance, the designer can get a variable of the referred element, without looking up the naming hierarchy and remembering the long name. For example, a designer can navigate through the design hierarchy of the layout to select the 'layer C' boundary in 'Cell B' in 'Cell A' (Fig. 9a). Just by clicking the 'position left' button, the designer can get a variable for the parameterized coordinate expression of the left edge of the

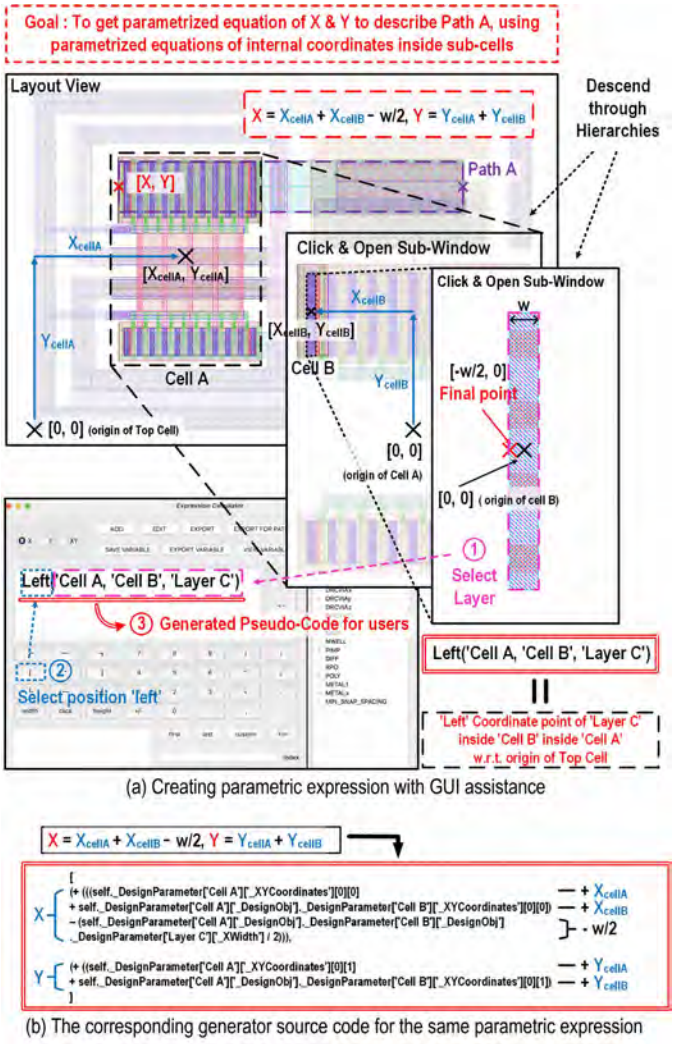


Fig. 9. (a) An example usage of expression assistance and (b) the corresponding generator source code.

selected boundary. The designer can also refer to a geometric dimension defined in design rules as a variable [3] through the expression assistance. Because such dimensions are handled as variables, the designer can easily describe a process-portable layout generator using the variables. Utilizing the expression assistance, a designer can easily refer to the variables of the layout elements and dimensions defined in design rules.

The describing interface of the expression assistance helps a designer easily describe a complex parametric expression. The expression assistance has an expression calculator to describe and edit a parametric expression. The expression calculator displays variables referenced by the user and allows them to be combined with other variables, functions, or expressions using arithmetic operations. The expression calculator provides buttons for addition, subtraction, and other functions, like an interface of a regular calculator. The expression written in the expression display is a pseudo-code that is much easier to understand and much shorter than the corresponding source code (Fig. 9b). For convenience, a designer can also directly edit, copy and paste the pseudo-code in the expression calculator. For highly complex parametric expression, the ex-

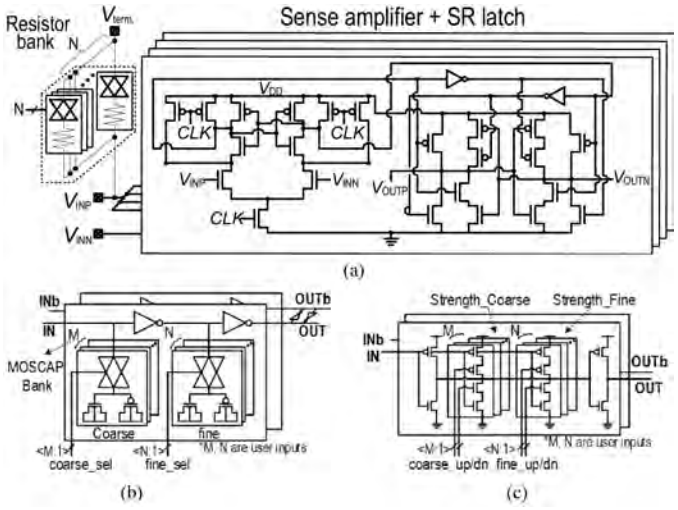


Fig. 10. Schematic diagrams of (a) the high-speed link receiver, (b) the DCDL, and (c) the DCC.

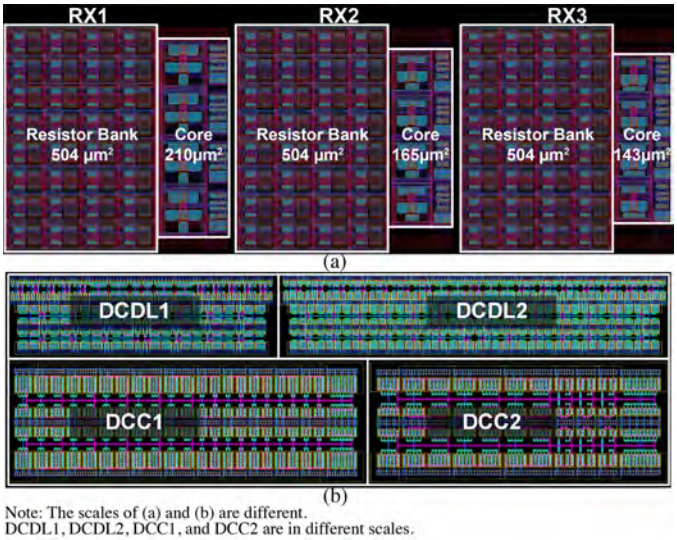
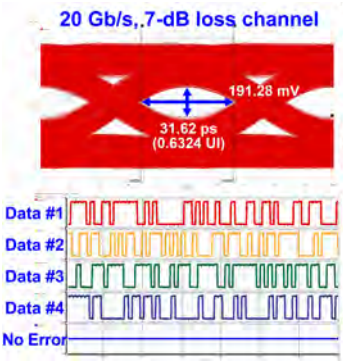


Fig. 11. Differently sized layout instances generated for different design parameters using the converted generator code.



		RX1	RX2	RX3
Maximum Data Rate (Gb/s)		20	16	12
Active Region	Power (mW)	1.611	0.912	0.626
	Area (μm^2)	210	165	143
Resistor Bank	Power (mW)	8.487	8.548	8.601
	Area (μm^2)	504	504	504
Eye height (mV)		191.28	208.03	195.23
Eye width (UI)		0.6324	0.6749	0.7513
Generation time (sec)		0.289	0.289	0.284

Fig. 12. Post-layout simulation of the generated RX layouts.

pression calculator supports temporary storage. A designer can create simple expressions and store them in temporary storage. Then, the designer can describe highly complex expressions by combining the stored simple expressions. Therefore, by utilizing the expression assistance, the designer can easily make a complex expression.

IV. Experimental Result

In this section, we examine the proposed layout-to-generator framework by converting several high-speed analog circuit layouts to the generators. In our experiments, we evaluated and compared the post-layout simulation results, development time, and the efficiency of coding. However, the direct comparison between the two methods is difficult because of the different design environments. Nonetheless, we did our best to fairly evaluate various metrics in various scenarios.

To evaluate the layout-to-generator framework, we develop several layout generators by both our proposed framework and the manual coding method. We evaluated the two methods with several high-speed analog layouts in 28 nm CMOS: a quarter-rate high-speed wireline receiver (RX) core, a digitally controlled delay line (DCDL), and a duty cycle correction (DCC) circuit.

The RX consists of a resistor bank, four slicers, and inverters (Fig. 10a). The resistor bank is an array of resistor units, each of which comprises a transmission gate and a poly resistor. A slicer consists of a sense amplifier and a high-speed set-reset (SR) latch. The DCDL consists of inverters and a bank of metal oxide semiconductor capacitors (MOSCAPs) and switches (Fig. 10b). The DCC circuit consists of arrays of inverters and tri-state inverters (Fig. 10c). Because these circuits operate at high speed between several and tens of Gb/s, high-quality layouts are required to meet the speed constraint. Therefore, automating their layouts is challenging.

We converted the layouts of the RX, the DCDL, and the DCC circuits into their generators using the framework. All of their sub-cells were also converted into the generators, which were used in the generators of their parent cells. In overall, 11 different layout generators were converted from the manually designed RX, the DCDL, and the DCC circuits as well as their sub-cells.

Various layout instances were generated in 28 nm CMOS technology by using the converted generators (Fig. 11). Hundreds of layouts were successfully generated and passed design rule verification using random parameter inputs.

We have included a demo video of layout-to-generator conversion for an inverter example using the proposed framework. This will be available at http://ieeexplore.ieee.org.

A. Post-layout Simulation

Three RX layout instances (RX1, RX2, and RX3) were generated with different size parameters in about 0.3 seconds on a PC with an Apple M1 Max CPU, targeting data rates of 20, 16, and 12 Gb/s (Fig. 12). The generated RX layouts were DRC/LVS-clean. The generated resistor banks are identical to provide the same 50-Ω matching regardless of the target data rate. RX1, RX2, and RX3 achieved 20, 16, and 12 Gb/s without any error in post-layout simulation, respectively. In each simulation, the RX recovered a 31-bit pseudo-random binary sequence (PRBS31) received through a 7-dB loss

TABLE I
PERFORMANCE COMPARISON BETWEEN THE GENERATED AND REFERENCE LAYOUTS

	Generated Layout	Reference Layout
Receiver		
Sensitivity	2.05 mV	1.9 mV
Offset	3σ =16.4 mV (1σ=5.48 mV)	3σ=16.74 mV (1σ=5.58 mV)
Input referred noise	0.317 mV_{rms}	0.385 mV_{rms}
DCDL		
Delay range	17.54-25.66 ps	17.35-25.79 ps
Resolution - (INL)	701 fs	788 fs
Resolution - (DNL)	658 fs	724 fs
DCC		
Correction range	38-60%	38-60%
Resolution - (INL)	1.093%	1.114%
Resolution - (DNL)	0.657%	0.659%

Reported numbers are post-layout simulation results.
RX1 is used for generated receiver simulation.
DCDL and DCC were simulated with 5 GHz clock.
The generated and reference layouts have the same design parameters.

TABLE II
DEVELOPMENT TIME COMPARISON

Cell	GUI (hour)	Script(hour)	Time Improvment
Receiver_top[1]	5	10.5	2.1x
├Sense amplifier	5.5	13.5	2.5x
├SR latch	4	11.5	2.9x
├Resistor bank	5	7	1.4x
└Inverter	0.5	2	4.0x
Receiver total	20	44.5	2.2x
DCDL_top[2]	7.5	17	2.3x
├MOSCAP bank unit	1.5	19.5	13.0x
└Inverter	0.5	10	20.0x
DCDL total	9.5	46.5	4.9x
DCC_top[2]	2.5	21	8.4x
├Inverter	0.5	12	24.0x
└Tri-state inverter	1.5	18	12.0x
DCC total	4.5	51	11.3x

Note
[1]:Expert designers each GUI, Script
[2]:Novice designers each GUI, Script
Reported time includes DRC+LVS runtime and code correction time.
Time measurement number is rounded by half hour.

channel. Fig. 12 shows the simulated eye diagram at the RX1 input, the waveforms of the recovered bits, and the error detector's output at 20 Gb/s. Table I compares post-layout simulation results of the generated RX1 and the reference RX. Because both RX1 and the reference RX targeted 20 Gb/s, they have the same design parameters. Their layouts are almost identical but just very slightly different in placing and spacing because the generator code is described by general formulas to satisfy design rules in various cases. Therefore, their performances are very similar and just a little different. Similarly, the performances of the generated DCDL and DCC layouts are very similar to the ones of their reference layouts in post-layout simulation because they are generated with the same design parameters as their reference layouts (Table I).

B. Comparison of Development Time

The development times by using the proposed framework and by manually writing code were reported in Table II. The development time includes 1) referring to a reference layout, 2) describing layout generator code, 3) testing generation of GDSII files including DRC/LVS-verification of the generated layouts, and 4) correcting the generator if there are errors. The development time was measured by a stopwatch for each generator development. This evaluation method was strictly compiled during multiple experiments in which several circuits were designed by several designers.

Two expert designers developed the receiver generator using the proposed framework and the manual coding method, respectively (Table II). Both have years of experience in developing layout designs and layout generators through manual coding methods. The total development time of the RX generator using the framework is 20 hours, which is only 45% of the manual coding time of 44.5 hours. Two novice designers designed the DCDL and DCC generators. Before the experience, they were trained just once for design of an inverter layout and its generator development. Before the training, they have skills of python coding, but neither of them has experience in developing analog circuit layouts nor layout generators. The total development time of DCDL and DCC were 9.5 hours and 4.5 hours, which are only 20% and 9% of the manual coding time of 46.5 hours and 51 hours. The framework improved the development time by more than half compared with the manual coding time in the case of expert designers.

V. CONCLUSION

To solve the problem of low productivity of analog layout design, a GUI-based analog layout-to-generator conversion framework employing visual programming was developed. The framework helps a designer convert a reference layout to a layout generator easily and conveniently by providing a convenient GUI environment for efficient development. Visual programming is adopted to reduce the required coding knowledge and workload to develop a layout generator. A program tree consisting of program blocks was also proposed easily to describe a hierarchically complex layout. Because a program tree can be directly compiled into the generator, the designer just needs to fill in the fields of visual program blocks through GUI to develop a generator. Therefore, visual programming greatly reduces the amount of programming knowledge and workload required to develop generator code. The proposed framework is further enhanced by programming assistance techniques: layout-code synchronization and expression assistance. The developed layout-code synchronization automatically updates the program block when the visualized layout elements are modified. Therefore, the layout-code synchronization allows a designer to intuitively edit a layout generator via GUI instead of manually coding every low-level programming detail. The developed GUI-based expression assistance provides a convenient GUI environment to get a complex parametric expression by simple GUI interaction. Using the programming assistance techniques, a designer can

easily develop a layout generator with less coding knowledge and coding effort.

For demonstration, layouts of a reference high-speed wireline RX, a DCDL, and a DCC circuits were converted to the generator code using the framework. Various different DRC/LVS-clean layout instances were generated. The generated and the reference layouts achieved very similar performances in post-layout simulation. The framework greatly reduced the overall development time of the RX generator by 55% compared to manual coding performed by an expert engineer. For a novice designer, who is not very familiar with the manual coding, the development time of the DCC generator was reduced by up to 91% in the experiment. Furthermore, by converting well-made reference layouts to generators with the framework, novice designers can easily leverage the design knowledge accumulated in the reference layouts to make new high-quality layouts. Therefore, the framework also relieves the human resource problem associated with a lack of expert analog engineers.

ACKNOWLEDGMENTS

The EDA tool was supported by the IC Design Education Center(IDEC), Korea.

REFERENCES

[1] E. Chang, N. Narevsky, K. Settaluri, and E. Alon, "Bag: A process-portable framework for generator-based ams circuit design," in *2019 IEEE Custom Integrated Circuits Conference (CICC)*, Conference Proceedings, pp. 1–20.

[2] J. Han, W. Bae, E. Chang, Z. Wang, B. Nikolić, and E. Alon, "Laygo: A template-and-grid-based layout generation engine for advanced cmos technologies," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 3, pp. 1012–1022, 2021.

[3] S. Han, S. Jeong, C. Kim, H. J. Park, and B. Kim, "Gui-enhanced layout generation of ffe sst txs for fast high-speed serial link design," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, Conference Proceedings, pp. 1–6.

[4] K. Kunal, M. Madhusudan, A. K. Sharma, W. Xu, S. M. Burns, R. Harjani, J. Hu, D. A. Kirkpatrick, and S. S. Sapatnekar, "Invited: Align – open-source analog layout automation from the ground up," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, Conference Proceedings, pp. 1–4.

[5] B. Xu, K. Zhu, M. Liu, Y. Lin, S. Li, X. Tang, N. Sun, and D. Z. Pan, "Magical: Toward fully automated analog ic layout leveraging human and machine intelligence: Invited paper," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Conference Proceedings, pp. 1–8.

[6] R. Martins, N. Lourenço, and N. Horta, "Laygen ii—automatic layout generation of analog integrated circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 11, pp. 1641–1654, 2013.

[7] Z. Wang, M. Choi, K. Lee, K. Park, Z. Liu, A. Biswas, J. Han, S. Du, and E. Alon, "An output bandwidth optimized 200-gb/s pam-4 100-gb/s nrz transmitter with 5-tap ffe in 28-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 1, pp. 21–31, 2022.

Sungyu Jeong (Graduate Student Member, IEEE) received the B.S degree in electrical engineering from Pohang University of Science and Technology (POSTECH), Pohang, Korea, in 2018, where he is currently pursuing the Ph.D. degree.

His current research interests include the layout automation of analog circuits, as well as design automation enhanced by deep learning.

Chanhyong Lee (Graduate Student Member, IEEE) received the B.S. degree in Electrical and Computer Engineering from University of California San Diego(UCSD), San Diego, California, USA, in 2019, and he is currently pursuing the Ph.D. degree in Electronic and Electrical Engineering from Pohang University of Science and Technology (POSTECH), Pohang, Korea. His research interests include high-speed link circuit and computer aided design.

Minsu Kim (Graduate Student Member, IEEE) received the B.S. degree in electronic and electrical engineering from Sungkyunkwan University (SKKU) ,Suwon, Korea, in 2020 and received the M.S. degree in the electronic and eleectrical engineering from Pohang University of Science and Technology (POSTECH), Pohang, Korea, in 2022, where he is currently pursuing the Ph.D. degree. His research interests include high-speed link circuit and computer aided design.

Iksu Jang (Graduate Student Member, IEEE) received the B.S. degree in electrical and computer engineering from Ajou University, Suwon, South Korea, in 2018, and the M.S. degree from the Pohang University of Science and Technology (POSTECH), Pohang, South Korea, in 2020, where he is currently pursuing the Ph.D. degree in electrical engineering. His current research interest is high-speed link circuit.

Myungguk Lee (Graduate Student Member, IEEE) received the B.S. degree in electronic engineering from Kumoh National Institute of Technology, Gumi, Korea, in 2015, and the M.S. degree in electrical engineering from the Pohang University of Science and Technology (POSTECH), Pohang, Korea, in 2017, where he is currently pursuing the Ph.D. degree. His research interests include high-speed link circuits, signal/power integrity, and agile hardware design.

Junung Choi (Graduate Student Member, IEEE) received the B.S. degree in electrical and computer engineering from the University of Seoul, Seoul, South Korea, in 2021, and the M.S. degree in electrical engineering from the Pohang University of Science and Technology (POSTECH), Pohang, South Korea in 2023, where he is currently pursuing Ph.D. degree in electrical engineering. His research interests include high-speed link circuits and analog layout automation.

Byungsub Kim (Senior Member, IEEE) received the B.S. degree in electrical engineering from the Pohang University of Science and Technology (POSTECH), Pohang, South Korea, in 2000, and the M.S. and Ph.D. degrees in electrical engineering and computer science from Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, in 2004 and 2010, respectively. He was an Analog Design Engineer with Intel Corporation, Hillsboro, OR, USA, from 2010 to 2011. In 2012, he joined the faculty of Department of Electrical Engineering, POSTECH, where he is currently a Professor. Dr. Kim received several honorable awards. He received the IEEE Journal of Solid-State Circuits Best Paper Award in 2009. In 2009, he was an also co-recipient of the Beatrice Winner Award for Editorial Excellence at the 2009 IEEE International Solid-State Circuits Conference. For several years, he served as a member of Technical Program Committee of the IEEE International Solid-State Circuits Conference and has been serving as the Chair of Wireline Sub-com of the IEEE Asian Solid-State Circuit Conference.