2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

A lightweight Python framework for analogue circuit design, optimisation, verification and reuse*

Wolfgang Scherr, Department of Integrated Systems and Circuit Design, Carinthia University of Applied Sciences, Villach, Austria (*w.scherr@cuas.at*)

Violeta Petrescu, Department of Integrated Systems and Circuit Design, Carinthia University of Applied Sciences, Villach, Austria (*v.petrescu@cuas.at*)

Johannes Sturm, Department of Integrated Systems and Circuit Design, Carinthia University of Applied Sciences, Villach, Austria (*j.sturm@cuas.at*)

Dirk Hammerschmidt, Department of Integrated Systems and Circuit Design, Carinthia University of Applied Sciences, Villach, Austria (*d.hammerschmidt@cuas.at*)

Santiago Sondon, Department of Integrated Systems and Circuit Design, Carinthia University of Applied Sciences, Villach, Austria (*s.sondon@cuas.at*)

Abstract— **Efficiency and reuse in analogue circuit design is still an important topic. Capturing design expertise in a procedural, descriptive way can simplify re-use of circuits even in new semiconductor technologies. It allows basic optimisation techniques a designer often just does "by hand" in a schematic. But such ways of automation are usually accompanied by multiple forms of programming, which is easy to do for a computer scientist, but not necessarily a key expertise of a circuit designer. This work presents a lightweight Python framework, which shall lower the entry barrier and simplify the use of automation in parallel to the classical design entry flow. It is explicitly developed with circuit design - and not programming - in mind. The idea is not to replace designers, but to increase their efficiency and improve re-use within and across technologies in a designer's community, by simple sharing single files. One sizing example is presented as well. Two implementation work flows are presented, an "agile approach" for developing new ideas interactively and a "waterfall approach" to simplify standard design tasks fully automatic. Both options implicitly include documentation of the sizing process, which would not be easily available with a classic schematic-only entry method. This also opens a path to allow generative AI to incorporate the design process and decisions a designer made more efficiently than just analysing a circuit from a netlist.**

Keywords— Analogue Design, Design Automation, Circuit Sizing, Python Framework, Agile Workflow, Waterfall Workflow, Design Reuse, Generative AI

I. INTRODUCTION

The semiconductor industry requires many experts. Analogue design is still ranked highly on lists of required talents [1], just among SoC design, data analysis and AI expertise. As such competences are not easy to come by, the existing workforce must become more efficient. With the retirement of experienced designers, there is also the risk that valuable know how gets lost. Furthermore, there is still the gap between design complexity and productivity [2]. The classical way of hand-made circuit synthesis in an analogue environment may be often by the following – or a similar - approach:

- When starting with a new technology, set up some simple single-device test benches to analyse the behaviour and collect some know-how about the technology.
- For a certain analogue function and specification, select a circuit architecture, often based on experience from existing circuits.
- Start with a basic circuit structure (using some hand calculations for the sizing), set up a basic PVT (process, voltage, temperature) test benches to check functionality and performance; debug the functionality, optimise device sizes in an iterative approach or using specific device changes based on experience. This is done quite often in the schematic, and immediately checked by simulations.
- One might also apply optimisation tools without layout in mind in this step or later including layout and thus parasitic effects.

* The financial support of this Josef Ressel Centre for "System-on-Chip Design Automation" (SODA) by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development and the Christian Doppler Research Association, is gratefully acknowledged.

- If performance limits are reached, try to re-structure or extend the circuit by improvement measures like cascodes, just to name one. In extreme cases, a different circuit architecture has to be selected, if the specification cannot be fulfilled.
- Extend the test bench for example by using Monte-Carlo methods to check for further parameters like matching. Document the design for layout specific instructions.
- When the circuit seems to look correct and required layout constraints are defined, start with the physical implementation.
- Afterwards, re-run all simulations including parasitic extractions where needed. If issues arise, loop back to previous steps. Finally, document the results for signoff/review.

Fig. 1 consolidates these steps and marks the most relevant ones for the proposed framework to be automated. The post-layout verification can re-use procedures implemented for the pre-layout simulation and debugging, and thus use the same framework. This is not explicitly marked, as it is not directly covered here in this paper.

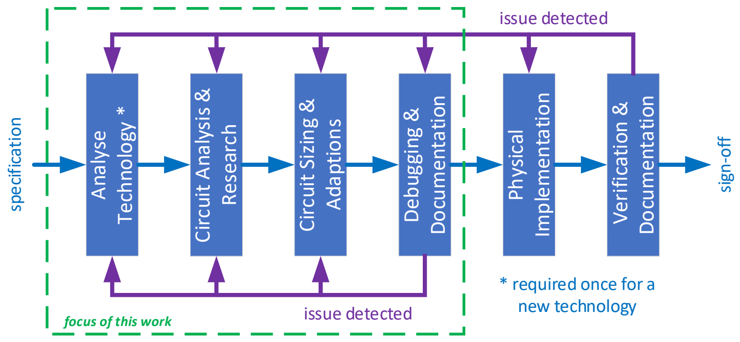

issue detected
specification
Analyse Technology *
Circuit Analysis & Research
Circuit Sizing & Adaptions
Debugging & Documentation
Physical Implementation
Verification & Documentation
sign-off
* required once for a new technology
focus of this work
issue detected

Figure 1. A classic analogue design flow.

The "natural" output of such a flow, a schematic and a layout, can be shared, but this process of creation it in a classical GUI tool, can hardly be captured in a useful way. And this does not cover the thought process required for e.g., the circuit sizing itself. Thus, such schematics have to be accompanied by some written notes, calculations from tools and similar, which must also be actualised each time the schematic is modified.

Analogue circuit design automation is not at all new. In the key speech of Austrochip 2022 a recent summary of existing approaches was presented [3]. Although they can speed up design work and help handle complexity, extending such solutions by designers with limited programming skills can become a showstopper. Additionally, the complexity of a designer's work can be re-considered as being a strenuous task, surrounded by many small but repetitive steps. Each of these steps is as important as the whole design block, like an ADC, which needs to be finally created.

For example, a task can be the basic circuit investigations in a technology not used before. One cannot just use an existing schematic from an existing product and just use it for a new technology. Usually, even the schematic has to be re-drawn, besides the complete circuit sizing and optimisation steps, assuming the basic circuit architecture can still cope with the new technology and requirements (in aspects of voltage ranges, intrinsic gains, parasitic effects and so on). And this is a task done over and over again, taking a lot of the valuable time of a designer, for current mirrors, OTAs, voltage/current references, comparators and so on. Not to mention the documentation of those steps, which is often skipped on those levels and described more general on a higher level of the design hierarchy.

Some very useful tools are focusing on the circuit sizing as well as layout task and require technology information in form of process design kits (PDK) within its framework, thus cannot be easily exchanged in an open-source approach due to legal restrictions, unfortunately [4][5]. Extending such frameworks might also not be trivial to do, as they may require significant skills in scripting languages and thus programming. There are well implemented tools which are easy to use (if one is aware of basic circuit sizing principles), but do not really foster efficient re-use like collection of sizing procedures or provide ways for improved automation because of their GUI-based nature [6].

This introduction can only mention a very small selection of automation tools. Listing them all is probably not even possible on a single page. Anyhow, none of them can gain a foothold in a larger scale in the industry. Especially experienced designers are often reluctant to try any programmatic approaches. The reason for this might be the lacking of one or more crucial aspects of existing solutions:

- A simple to use and easy to maintain script language with very low entry barrier.
- Focus especially on circuit designers' skills and needs to capture circuit sizing efficiently.
- Easy to lay out sizing and/or optimisation processes in the script.
- Support of any sizing methodology, based on square-law model or g_m/I_D techniques (to name two most common ones), including the use of look-up tables.
- PDK independent scripting of basic circuit generators, to simplify exchange.
- Generators independent of the design system, also to simplify exchange.
- Allow generation of test benches, simulations and standardise verification included in the generator.
- Easy to share and re-use scripts – ideally only a single text file shall be required to reproduce the whole sizing and verification steps of single building blocks.
- Allow properly commenting in scripts, as add-on to understand the sizing procedures and decision processes properly.

Programming and circuit design are fundamental different disciplines, experts usually have to focus on one of them to advance in their respective field. So, the idea is, in short, to set up a framework, where a designer learns quickly how to use it just by using a generator script from a colleague as starting point. The script contains easy to follow commented steps for the sizing and verification, very similar to hand calculations. This generator shall size circuits in any technology the framework was set up for. It should be immediately visible, that the effort is similar to the conventional, hand-made work - including proper documentation of the design procedure and verification – and that it provides an efficiency boost when re-using again for similar circuits.

Finally, such a script captures all relevant steps of circuit design, which can be used as input for further optimisation like [7] or generative machine-learning approaches, instead of relying only on the circuit representation alone [8]. Such (properly commented) scripts could be used to train generative AI tools like ChatGPT [9] more efficiently than just by using sized circuit netlists for training. This would be a similar way as it is already done for hardware description languages for digital design - to “ask” an AI for the design of a counter and get back ready to use HDL code.

II. FRAMEWORK STRUCTURE

The proposed CUAS Cell Control (CCC) framework [10][11], implemented in Python, allows separating any EDA tool, including its supporting data like a process design kit (PDK), from a circuit generator script. To achieve this, a two layered API approach is used, as illustrated in Fig. 2. This makes circuit sizing and generation in principle tool- and technology- agnostic.

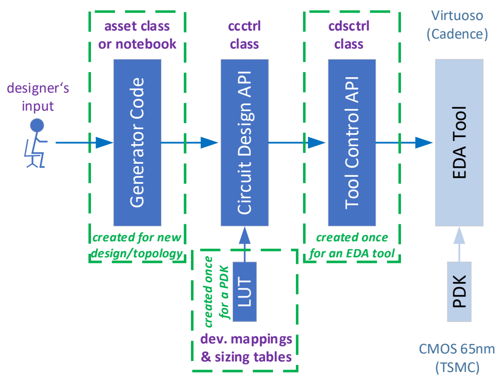


Figure 2. Framework structure (exemplary EDA tool and PDK).

On the first, lowest API layer, tool functionality like “creating design views like schematics, symbols and layouts”, “creating instances within design views” or “launching simulations and extracting/postprocessing results” is abstracted. Basically, it maps all the basic design tasks to an individual tool. For a tool like the Cadence Design System (CDS), this API is consolidated in a class called "cdsctrl". Nevertheless, methods of this class would stay the same for different tool vendors.

For the mentioned commercial tooling, which is widely used in the industry as well as in education, an existing Python package to control the design environment called "skillbridge" [12] was incorporated. This package was also a big inspiration for this framework. It minimised the effort to create this API layer. For other EDA tools, a similar Python package may already exist, at least as long as the tool provides an interface for any form of programmatic tool control. One such alternative API layer may be implemented using the PySpice package [13], which would offer a minimal, but completely open-source solution to run CCC circuit generators based on generated SPICE netlists.

Devices and its parameters are still dependent on the respective PDK. Therefore, on a second API layer, the PDK functionality is mapped to a generic set of devices. This allows to code technology-independent circuit generators and verification setups. This class is called "ccctrl", which means "cell creator control". Mapping of devices had to be standardised in a way, that a generator can use either a "default" device from a technology, or a designer selects a device for a certain voltage class or threshold level. The strategy for mapping can then be selected in one of the following ways:

- the generator stops, in case no explicit device can be chosen (a 1.8V device is requested, but only a 2.5V or 1.2V device is available). The user can check first before selecting the next option.
- the generator automatically selects a device close enough to the requirement (a 1.8V device is requested, but the next available voltage class is 2.5V and thus selected).
- the generator stops, as the PDK does not offer the same or higher voltage class for the requested device, hence a direct implementation of the particular circuit with this script is not possible anyhow.

Therefore, such a mapping also implies to provide basic information about the device back to a generator script, to be technology independent. This includes minimum and maximum allowed device sizes, its voltage class and other relevant data. Thus, sizing in a generator can be implemented in a generic way, allowing not only re-use between different technologies, but also for different devices in the same technology.

		PROCESS DESIGN KITS (PDKs)		
		TECH_A	**TECH_B**	**TECH_C**
DEVICE MAPPINGS	**MN**	mn	nmos	nch
	MP	mp	pmos	pch
	MNL		nmos_lvt	
	MPL		pmos_lvt	
	MNH		nmos_pvt	
	MPH		pmos_pvt	
	MN09	mn		
	MP09	mp		
	…			
	MN33		nmos_3v3	nch_3v
	MP33		pmos_3v3	pch_3v
	…			

Figure 3. Exemplary, hypothetical PDK mapping of devices.

Fig. 3 provides an example of device mappings of exemplary PDKs. The core n-channel and p-channel devices of a particular technology can be accessed as "MN" or "MP". The devices use the default voltage class and threshold level. But one may also explicitly request a low-threshold device with "MNL" or a 3.3V device with "MN33". Or, continuing this concept, a low-threshold 3.3V device would be "MNL33". This example only presents mapping of MOSFET devices, the similar concept is used for BJTs or different kinds of passive devices (like R, L, C).

The more generic (and less restrictive) device types are used by a generator, the easier it is to re-use a generator for a new PDK. This of course does not necessarily mean that the generator will immediately produce a correctly sized design for any new technology. Especially when considering the step from MOSFET to FINFET. However, it is a viable start for a designer to reach a feasible solution more quickly by adopting an existing sizing script. It should be also mentioned, that in some cases the result may still not be completely sufficient for all specification parameters. In such cases, the generator may at least produce a very good starting point for a shorter run of a following optimiser tool as well as a potentially lower risk to find a suboptimal optimum of the circuit.

As mapping can be generalised, there is no framework re-coding required on this level. All mapping information is kept in YAML files, which is set up once for each technology. Extracted device data for pre-computed look-up tables is kept in HDF5 files, a file format designed to store large quantities of structured data. Ready to use Python packages are used to handle those formats. Creating this look-up date can make use of the very same framework, which means those generators can be as well re-used for any new PDK, as long as the initial YAML file is set up to define the device mappings and basic technology parameters.

A fully generic circuit generator will retrieve basic parameters like allowed device dimensions, device voltage classes and similar. Furthermore, using pre-computed look-up tables (LUTs) does not only have advantages when using methodologies like g_m/I_D [14]. For a classic square-law model (SLM) approach [15][16], they also offer a good path to well sized devices even in modern technologies, as with sizing the device also the classical SLM parameters like $k = \mu Cox$ or V_{th} may vary. We will show later, how LUTs can be used for this. But first, we want to elaborate on two basic work flows this framework supports.

III. AGILE VERSUS WATERFALL WORK FLOW

For developing new circuit concepts, especially with challenging requirements to fulfil, it is always indicated to break down complexity into smaller steps, with smaller scale changes in shorter time periods to check. This way, also changes from e.g., a system engineer can be still incorporated with reasonable effort, while an analogue engineer is implementing circuits. One will notice, this approach shows similarities to agile software development [17], so the name is borrowed (or reused) for this form of using this design framework.

A lot of steps, although not too complex, may require manual, repetitive and time-consuming rework for such an approach and can therefore be outsourced to a simple programmable solution. For analogue designers it is easy to draw schematics. However, these cannot document well the decision finding process. Code on the other hand can do this quite well. Combination of both approaches would be preferable, to allow running commented code in parallel to a schematic, to modify device parameters, for example.

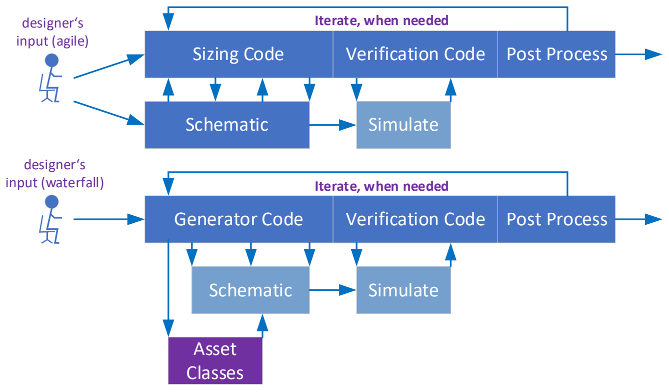


Figure 4. Agile versus waterfall design entry.

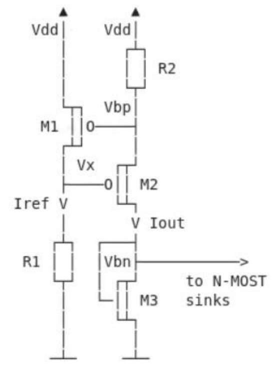
With a certain stock of analogue modules developed, more complex setups can be built as well. Such analogue modules can be categorised and handled as assets, which can be consolidated in classes, like band-gap references, op-amps, bias generators and so on. Asset classes consist of circuit generators and test bench generators. So, they can be seen as units, similar to software development, with accompanied unit tests. These elements would be best used in a classical waterfall approach. Here, engineers plan and orchestrate the use of the modules for the first time, parameterise them and then run the implementation in a single script, finally creating some documentation.

Fig. 4 provides a graphical comparison of the two approaches. With an agile approach, where decisions are made along the development, a notebook-like entry with code snippets, comments and intermediate plots is very useful. An analogue designer would also like to see the schematic, and work with it interactively. For a waterfall approach, a classic code editor would be sufficient, similar to a digital synthesis flow. Luckily, within the Python infrastructure, there are development environments, like Spyder, supporting both approaches: Jupyter notebooks as well as a classical Python integrated development environment (IDE) [18].

Using the notebook approach, designers can use the script along a design environment. One can modify an existing schematic using scripting to optimise or analyse the design in a transparent way, before updating the sizing procedure itself. On top, the designer can make use of Python for the calculus of sizing, similar to hand calculations. This provides a similar experience to a manual optimisation, but with the advantage that the method of optimisation is captured in a script. This does not require exploiting many features of Python, but just basic loops and conditional branches as one would do in a manual design optimisation work.

IV. AGILE AND WATERFALL FLOW BY EXAMPLE

Using the g_m/I_D method with look-up tables is well known and shown with CCC in another work [19]. Therefore, we demonstrate the look-up table approach for the square-law sizing method. It shall also demonstrate that such an approach can be used to iterate a sizing procedure to end up with a reasonable circuit, without manual "wiggling" of transistor sizes. Here, we show one way to implement a threshold-based, textbook bias generator using the agile work flow. The real PDK name is replaced by some generic name, like we did in the last chapter, and the code is condensed as good as possible, with a minimum of comments added. One should start with a short description of the generator code and sizing, which might look like shown in fig. 5. We will not dive into explaining the circuit as such as it can be found similarly in [20], as we focus on the scripting.



```
Iout = Vgs/R2 = ( Vth + sqrt(2Iref/(uCox(W/L))) )

Iref should be quite small.
If (W/L)1 is large enough, then: Iout ~ Vth,p/R2.
```

Figure 5. Circuit to be generated (for any PDK, ideally).

Next, we load the cell creator framework package and some utilities to simplify tasks like plotting waveforms. We also include the specification one might want to adapt prior to generating the circuit (Fig. 6, left). Then we open a session and extract the relevant data from the technology look-up tables. For the device sizes we need some starting point for the look-up approach (Fig. 6, right). We use a method, which does not require absolute device sizes in the specification itself. Designers might use different strategies to retrieve parameters for initial hand calculations, this script shows just one possible approach. It also shows the use of look-up functions, which should be self-explanatory in this context. Based on this, device sizes can be calculated like one would do in hand calculations, which we also don't focus on here.

Especially in modern technologies, device parameters cannot be assumed independent of device sizes anymore. Thus, such a simple approach will not work well for modern technologies. But one could use a simple loop around this calculation scheme. In this loop, parameters are extracted again from the calculated W of the devices (assuming L stays constant). This will provide an improved sizing of the circuit. The iteration can be stopped, if the changes of device sizes are not significant anymore or just by a loop counter. This is a swift as well as easy to understand and to control concept.

```
# setup cell creator control and utilities
import ccctrl as cc
from ccutilities import f2s, graph
# specification
Iref = 5e-7
Iout = 5e-6
WL1 = 20
tech = "TECH_A"   # "TECH_B" # "TECH_C"
# target library and cell name
libname = "genlib"
cellname = "vth_bias_pmos"
# we use the basic (core) elements of the PDK
nmos = "MN"
pmos = "MP"
# also passives can be handled like this
res = "RP"
```

```
# open a circuit sizing session and keep in 's'
s = cc.CccSession("",tech)
# MOST parameters look-up for sizing
Vdd = s.GetDevParam(nmos,"vnom")
Wmin = s.GetDevParams(nmos)['w'][0]
Lmin = s.GetDevParams(nmos)['l'][0]
# we use multiples of minimum MOST sizes (PDK agnostic)
Wtyp = Wmin*8
Ltyp = Lmin*4
# this gives us some reasonable values for circuit sizing:
Vth_n = s.LookUpPar(nmos,"vth",Wtyp,Ltyp)
Vth_p = s.LookUpPar(pmos,"vth",Wtyp,Ltyp)
k_n = s.LookUpPar(nmos,"ucox",Wtyp,Ltyp)
k_p = s.LookUpPar(pmos,"ucox",Wtyp,Ltyp)
# resistor parameters for sizing
rsq = s.GetDevParam(res,"rsq")
```

Figure 6. Setup framework (left) and retrieve initial data (right) for a PDK agnostic approach.

The advantage of this framework is, that all these calculations can be done without any time-consuming simulation runs. All data is retrieved from look-up tables. Finally, one would create the circuit, a test bench for the circuit, and run simulations. Here, we just show the circuit creation, as everything else would just be building a hierarchical setup of instantiating that design into a new schematic e.g., with some voltage sources as stimulus. If required, one could still use a simulation-based optimising step in the script (or directly an optimising tool), to improve the device sizes even further.

In fig. 7 (left), we show such a (simplified) script for circuit creation. Regarding post-processing of simulation results, one can use all kind of scientific packages available for Python. This makes the framework complete in respect of circuit sizing. Such a generator can now be easily passed on to colleagues, as only one file has to be handled. If a generator is well established and used more often, it can be "packed" in an asset class for the waterfall approach. In fig. 7 (right) an empty template of such a script is shown, which needs to be basically filled by the well-established procedures from the notebook file. This means instead of a notebook with step-by-step execution, a script file is generated. Of course, all steps from the agile approach requiring interactive work, have then to be automated as well (if any). Ideally, these steps are at least properly commented by the design engineer, so it should be easy for a Python expert to automate those parts as well. Assets can then be used in more complex generators to set up larger function blocks, or could even be reused in other agile generators, where suitable. Here the expertise of circuit design and programming comes together.

```
# open session with design/simulation tool
s = cc.CccSession("my_first_project",tech)
x=0
y=0
sch=s.CreateSchematic(libname, cellname)
# supply pins
s.CreatePin(sch, "Vdd", "input", x,y+1)
s.CreatePin(sch, "Vss", "input", x,y-1)
# first branch
s.CreateInstanceAndConnect(
  sch, "",pmos, "M1", x+1, y+1,
  {"D":"Vx","G":"Vbp","S":"Vdd"},
  {"W":f2s(W1), "L":f2s(L1), "NF":"1"})
# ... other instances removed ...
s.CreatePin(sch,"Vbn","output",x+3,y+-1)
# we are done, check, save and close
s.CheckSchematic(sch)
s.Save(sch)
s.Close(sch)
# we need a symbol for instantiation later
s.CreateSymbol(libname, cellname)
```

```
# asset class: bias generator with NMOS mirror
class biasgen():
  def __init__(self,tech,libname,
          cellname,env,debug):
    self.env = env
    self.tech = tech
    self.debug = debug
    self.libname = libname
    self.cellname = cellname

  def size_circuit(self,iout,gentype,mout,nmos,pmos,res):
    self.pmos = pmos
    self.nmos = nmos
    self.res = res
    # ... sizing as shown in the notebook...
    # we return MOST, R and C areas
    return [Amost, Ares, Acap]

  def generate_schematics(self):
    # ... circuit creation as shown in the notebook...
    self.created = True
```

Figure 7. Code snipped of schematic generation (left) and the structure of a generator class for an asset library (right)

In a similar way, a test class has to be created as well, containing the methods "generate_testbench", "simulate", "check_results" and "create_model". The first methods should be self-explanatory, the last methods allow to e.g. create a model of the design based on the simulation results - according to the design flow used in the team. There is no predefined automatism behind this function, the designer of the generator could implement a macro model similar to the design. The reason for separating the generator class from the test class is, that one might need to define the test class for a certain circuit function only once. But for generator classes there could be several circuit options for implementation.

V. Conclusion and Outlook

This work proposes a framework for analogue circuit design sizing and basic optimisation. The main intention was to introduce Python coding for experienced analogue designers, using a low-threshold approach which does not require extraordinary programming skills. It does not include layout generation, but it should fit well to existing layout automation tools. Due to the open nature of the framework and the use of Python is also true for many other tools one might additionally use in a design environment.

Designers who were shown the tool recognised its benefits immediately. Although there are no larger, reliable efficiency studies of using this framework yet, first indications promised, that it could be useful to save time and effort especially for basic circuits which need to be implemented relatively often. This includes also basic, reproducible and documented verification on a very low level, which may otherwise not exist for small, hand-made blocks. After a reasonable learning curve, one could design a generator as presented in this work in approximately the same time as one would require for a hand design, including proper verification and documentation (those efforts are sometimes neglected). But each re-use afterwards will already save time in respect to a manual entry, ideally it can be re-used without further modifications. Furthermore, notebooks from designers contain sizing procedures, which could be used as input to generative AI systems in a similar way as it is already done for digital hardware description languages.

As it is not feasible to include a whole example and API description in this publication, it is planned to release the framework into the open-source world after implementing some more examples. In the first phase a pre-release of the framework is shared for educational use to interested design experts in industry and academics, to get a close discussion on the approach and collect relevant feedback. This is done on request, to avoid overloading the small development team.

References

[1] S. Brugmans, O. Burkacky, K. Mayer-Haug, A. Pedroni, G. Poltronieri, T. Roundtree, B. Weddle, "How semiconductor companies can fill the expanding talent gap", McKinsey&Company, Web Article from www.mckinsey.com, Feb. 2024.

[2] R. Collett, D. Pyle, "What happens when chip-design complexity outpaces development productivity?", McKinsey&Company, Web Article from www.mckinsey.com, Number 3, Autumn 2013.

[3] B. Prautsch, "Automatic Analog Design and Layout Generation: Chances and Hurdles", Keynote, IEEE Austrochip Conference, Villach, Austria, Oct. 2022.

[4] J. Crossley, A. Puggelli, H.-P Le, B. Yang, R. Nancollas, K. Jung, L. Kong, N. Narevsky, Y. Lu, N. Sutardja, E. J. An, A. L. Sangiovanni-Vincentelli, E. Alon, "BAG: A designer-oriented integrated framework for the development of AMS circuit generators" IEEE/ACM International Conference on Computer Aided Design, Nov. 2013.

[5] M. Bio, W. Scherr, A. S. Agbemenu, S. M. Sondón, J. Sturm and V. Hande, "BAG2 Assisted Hierarchical Analog Layout Synthesis for Planar Technologies," Austrochip Workshop on Microelectronics, Graz, Austria, September 2023.

[6] O. Hesham, "The Analog Designer's Toolbox (ADT) Towards a New Paradigm for Analog IC Design," chapter 5 in SMART Integrated Circuit Design and Methodology, River Publishers, 2023.

[7] K. Antreich et al., "WiCkeD: analog circuit synthesis incorporating mismatch," Proceedings of the IEEE 2000 Custom Integrated Circuits Conference, 2000.

[8] R. Mina, Ch. Jabbour, G. E Sakr, "A Review of Machine Learning Techniques in Analog Integrated Circuit Design Automation", HAL Electronics, 2022.

[9] T. Wu *et al*., "A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development," in IEEE/CAA Journal of Automatica Sinica, vol. 10, no. 5, May 2023.

[10] W. Scherr, "Standardised Circuits by a Heuristics-Driven Automation System", Master Thesis, Carinthia University of Applied Sciences, 2023.

[11] W. Scherr, V. Petrescu, J. Sturm, D. Hammerschmidt, and S. Sondon, "An easy to use Python framework for circuit sizing from designers to designers", IEEE Austrochip Conference, Vienna, Austria, Sep. 2024.

[12] T. Markus, "Circuit Design Automation for High Speed Interconnects in Advanced Nodes", Dissertation, Ruprechts-Karls University Heidelberg, 2021.

[13] F. Salvaire, "Pyspice 1.5: Simulate electronic circuit using Python and the Ngspice / Xyce simulators", URL: https://pypi.org/project/PySpice/.

[14] P. G. A. Jespers, B. Murmann, "Systematic Design of Analog CMOS Circuits using Pre-Computed Lookup Tables", Cambridge University Press, 2017.

[15] B. Razavi, "Design of Analog CMOS Integrated Circuits", McGraw-Hill Press, 2015.

[16] R. J, Baker, "CMOS Circuit Design, Layout, and Simulation", 3rd Edition, IEEE Press Series on Microelectronic Systems, 2010.

[17] M. S. Murugaiyan, S. Balaji, "Succeeding with Agile software development," IEEE-International Conference On Advances In Engineering, Science And Management (ICAESM), Nagapattinam, India, 2012.

[18] "Spyder, the Scientific Python Development Environment", web page URL: https://www.spyder-ide.org/.

[19] R. Arasada, V. Petrescu, W. Scherr, G. Paoli, S. M. Sondon, J. Sturm, "Design Automation of a 2GHz Dynamic Comparator using the CCC Framework," IEEE Austrochip Conference, Vienna, Austria, Sep. 2024.

[20] M. Gray, P. Hurst, S.H. Lewis, R.G. Meyer, "Analysis and Design of Analog Integrated Circuits,", Wiley Press, 5th edition, 2009.