Cocoon: An Open-Source Infrastructure for Integrated EDA with Interoperability and Interactivity

Jiaxi Zhang[1,*], Tuo Dai[1], Zhengzheng Ma[1,2], Yibo Lin[1] and Guojie Luo[1,2,†]
[1]Center for Energy-Efficient Computing and Applications, Peking University
[2]Peng Cheng Laboratory
Email: *zhangjiaxi@pku.edu.cn, †gluo@pku.edu.cn

***Abstract*—The increasing size and complexity of integrated circuit (IC) design introduce huge design cost and put forward higher requirements for EDA tools. Improving the quality and efficiency of chip design requires the efforts of both EDA workers and IC designers. In this paper, we first put forward the concept of Integrated EDA, a system composed of EDA points tools, designs and interfaces. And we point out the key features of integrated EDA and the possible solutions. Then we propose Cocoon, an open-source infrastructure for integrated EDA with interoperability and interactivity. It contains a set of cross-tool interfaces and plays the role of EDA agent that can help IC designers choose EDA point tools to assemble a legal design flow and to produce ICs with a higher quality of results (QoR). At last, we implement two demos using Cocoon to prove that such infrastructure is feasible and flexible for integrated EDA.**

I. Introduction

The increasing size and complexity of integrated circuit (IC) design introduce huge design cost and increase time to market. Higher requirements have been put forward to electronic design automation (EDA) tools. EDA researchers and vendors have proposed many methods to improve the quality and reduce the running time of EDA tools, thereby further helping chips designers to decrease the design cost. However, as the complexity of the chip architecture increases, it is not enough to rely solely on EDA tool developers. This requires the joint efforts of EDA tool developers, EDA system integrators and the IC designers.

The concept of electronic CAD framework has been proposed to define all of the underlying facilities provided to the CAD tool developers, the CAD system integrators, and the IC designers [1]. The primary goal of a CAD framework is to reduce the time and cost required to develop or modify a CAD system such that it meets the needs of its users. But it is not trivial to implement such a framework. In recent years, the open-source EDA community still have made attempts and great efforts to it. OpenRoad flow [2] developed an open-source toolchain covering from RTL to GDSII phase of system-on-chip design. DATC Robust Design Flow [3] constructed a database for design benchmarks and point tool libraries and provided an open-source design flow from logic synthesis to detailed routing. Such open-source toolchains provide IC designers with the possibility to modify the customized EDA flows to get higher design quality of results (QoR).

Due to a tool can not handle all designs to produce the best QoR, IC designers can customize the design flow by choosing different tools to get higher QoR for their design. Thus, IC designers need to be very familiar with every step in EDA toolchain and detailed scripts of each tool before they want to replace one step with another point tools to customize a design flow. Poor interoperability and interactivity hinder the integration and customization of EDA tools. Interoperability is the premise of CAD-IP reuse [4], and interactivity supports the toolchain customization for IC designers without expertise in EDA algorithms. Some EDA frameworks have been proposed to solve such problems. Edalize [5] provides a python library for interacting with EDA tools. It can create projects for supported tools and run them in batch or GUI mode. Hammer [6] means to enable re-use and portability of EDA tools between technologies. It provides a Python backend and exposes a set of APIs that are typical of modern VLSI flows.

In this paper, we expand the concept of electronic CAD framework to **integrated EDA**. The goal of integrated EDA is that users can easily call EDA point tools to assemble customized EDA flows as well as generate design layout without a deep understanding of all EDA tools. Interoperability and interactivity are two key features to achieve the above objectives. The major technical contributions of our work are threefold:

- We review the concept of Electronic CAD framework and expand it to integrated EDA. We point out the key features of integrated EDA and the possible solutions. (Section II).
- We propose Cocoon[1], an infrastructure enhancing the interoperability and interactivity for integrated EDA. With the help of interface definition, Cocoon can represent customizable and tunable EDA flows. (Section III).
- We implement two demos using Cocoon to prove that such infrastructure is feasible and flexible for integrated EDA. (Section IV).

[1]https://github.com/pku-dasys/cocoon

TABLE I: Interoperability and interactivity approaches in inter-connected cloud and their counterparts in integrated EDA.

Approaches in Inter-Connected Cloud	**Corresponding approaches in Integrated EDA**	**Available in existing flows?**
Open Standards/Protocols	Open Standards	YES
Cross-platform APIs	Cross-tool APIs	NO
Layers Abstraction	Steps Abstraction	YES
Cloud Brokers/Agents	EDA Brokers/Agents	NO
Model-based	Model-based	NO

II. Integrated EDA

In this section, we first define integrated EDA and point out the key features. Then we show some possible solutions borrowed from other fields that might enhance the interoperability and interactivity of integrated EDA.

A. What is Integrated EDA?

Integrated EDA is a system composed of EDA point tools, designs, and interfaces. It is more than a library of EDA point tools. The users of integrated EDA include three groups of people, EDA researchers, tool developers, and IC designers, each of whom have their own needs and particular emphasis. For EDA researchers and tool developers, they can quickly verify whether their algorithm or tool is useful in the end-to-end EDA flow. They can also create new design methodology and do cross-stage optimization. For IC designers, they can easily customize EDA flow without knowing details of EDA algorithms to get higher design QoRs.

Interoperability and interactivity are two key features of integrated EDA. In EDA context, **Interoperability** can be defined as *the ability of two or more toolchains to exchange point tool and to use the design file that has been exchanged*. **Interactivity** can be defined as *the flexibility that user can replace and deploy point tool and without knowing the implementation details*. The former one guarantees the interchange of point tools, while the latter feature enables cross-step optimization and flow tuning for users.

B. Possible Solutions

The problem of interoperability and interactivity will also appear in other fields. In the inter-connected cloud, the interoperability and interactivity can be defined as *the ability to seamlessly deploy, migrate, and manage application workloads across heterogeneous hardware and software resources provided by multiple datacenter cloud providers*. Different researchers have suggested various approaches to implement interoperability in inter-connected cloud [7], and inter-connected EDA can draw on the experience of these approaches. Table I summarizes the interoperability approaches in inter-connected cloud and corresponding approaches in integrated EDA. The last column shows whether these approaches currently exist in the EDA design flow.

Open standards are design exchange formats in chip design, and there have been several open standards in EDA tools. Liberty format defines a standard cell library. LEF format contains physical and technology information of standard cells. DEF format describes initial floorplan. SDC format includes design constraints such as clock period, driver information of each input port and load capacitance of each output.

Cross-tool APIs are the interfaces to easily call different point tools without knowing how to write the running scripts. The scripts of most EDA tools are based on Tcl. But Tcl does not support interoperability of different tools very well because it is tightly coupled with the tool. We need a higher-level abstraction to implement cross-tool APIs.

Steps abstraction are the process steps in the EDA design flow. The abstractions now are stable and adopted by industry and academia. The main abstractions include logic synthesis, floorplan, power plan, global placement, detailed placement, clock tree synthesis, globe routing, detail routing, design rule check, LVS, static timing analysis and parasitic extraction, et.al. But it's not enough if we only have these point tools. For those who are not familiar with EDA stages like IC designers, we still need some mechanism to check whether the order in which the tools are used is legal.

EDA brokers are individuals or enterprises that act as intermediaries between users and providers of EDA tools. They recommend the most suitable EDA point tools for designers. And they provide users with a simple API and UI, allowing users to work seamlessly with different point tools as if using a single tool. An EDA agent can be a software that helps choose point tools from different EDA vendors and even the open-source EDA community.

Model-based approaches request developers to model the functionality of the system. A model is usually specific with some model notations or a domain-specific language. This kind of method has not been applied to EDA.

In this paper, we focus on the cross-tool APIs and EDA agents to enhance the interoperability and interactivity of inter-connected EDA.

III. Cocoon

In this section, we first give the architecture and abstraction layers of Cocoon. Then we show how Cocoon supports customized flow definition and flow tuning.

A. Overall Architecture

Fig. 1 summarizes the Cocoon architecture. Cocoon has three layers of abstraction, applications (tools) layer, data layer and interfaces layer.

Applications layer contains all EDA stages in the entire flow, including but not limited to high-level synthesis, logic synthesis, floorplan, power plan, placement, routing, et.al. With the development of chip design and EDA, more applications may appear. But it does not affect the abstraction of the application layer; we can add or delete applications to this layer. For example, placement application can be divided into global placement, legalization and detailed placement. We can add these three sub-stages to the application layer. Each application is a set of EDA tools from the open-source community or EDA vendors. Tools in Cocoon will provide options and parameters,

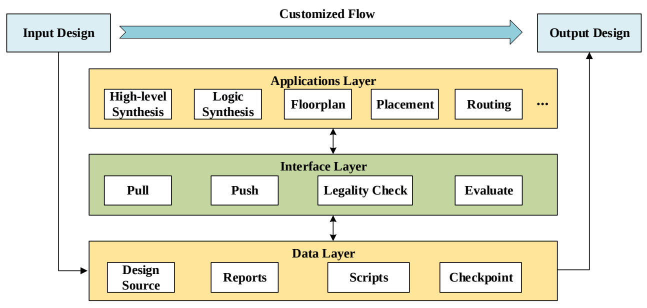
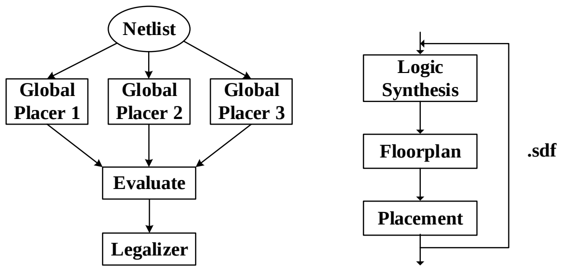


Fig. 1: Cocoon Architecture.



(a) Branching flow. (b) Iterating flow.

Fig. 2: Customized flow definition.

and these options or parameters can drastically impact design quality.

Data layer contains all information about the design, including design sources, reports, running scripts and checkpoints. Design sources include hardware design languages, standard cell library $*$.lib, physical and technology information of standard cells $*$.lef and design constraints $*$.sdc. Reports are the output files of EDA tools, including report files such as timing, power and running logs. Scripts are the files to control the execution of EDA tools, including Tcl scripts, makefiles or scripts with other formats.

The checkpoint is critical in the data layer and is different from other frameworks. In addition to the information included in traditional checkpoints like netlists, physical data, timing information, design constraints, etc., checkpoints in Cocoon also contains executed tools and tool parameters. Such checkpoint design makes it easier for applications to check whether the current input checkpoint is legal. And it can memorize the used tool for results reproduction without spending time choosing tools. The checkpoint can also be easily migrated to other design flow. For example, if a IC designer has tried tool A, B, C for global placement and he figures out that tool B produced layout with higher design quality for module X after finishing the latter design stages, he can use tool B for global placement in the selection of subsequent EDA tools.

B. Interface Layer

Interface layer is a bridge connecting data layer and applications layer. It is also a key point for supporting interoperability and interactivity. Pull and push are basic methods for applications to read or write data.

Evaluation operator pulls the checkpoints, compare the results and can push the checkpoint with higher QoR to other applications. Users can prune the bad design in early stage and choose the tools with better design quality via evaluation method. Evaluation operator can be implemented by pulling the reports and estimating the design quality by reports. Users can define their own evaluation method to realize such functionality. E.Ustun [8] use machine learning to accelerate the design tuning phase by pruning the bad design in early stage.

Legality check is defined to check whether the input checkpoint of the application is legal. This mechanism can help avoid the problem that a customized flow fails due to the incorrect application calls. For example, if in a customized flow, user calls a routing tool after logic synthesis and omits the placement, then this flow is illegal. Cocoon can save the applications that correctly executed, and give an alternative application.

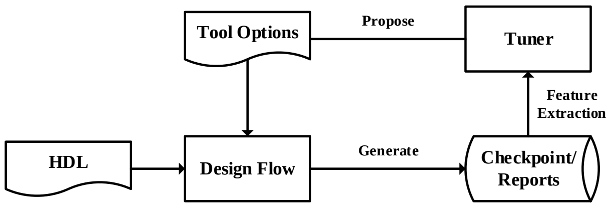


Fig. 3: A typical tuning process.

C. Customized Flow Definition

With the above layers and functionalities, Cocoon can flexibly support the definition of various forms of design flows, such as branching flow and iterating flow. These forms of design flows can help users choose better quality tools and research on new design methodologies without writing too many tool scripts.

1) Branching flow: Suppose the user wants to use yosys with different parameters or yosys and genus for logic synthesis at the same time, and choose a better result to perform the subsequent steps. This flow has branches. Fig. 2a gives the schematic of the branching example. Assuming that we can have 3 global placers to choose from, Cocoon can evaluate the checkpoint after finishing the global placement, and choose a better design for subsequent legalization and detail placement. The evaluated function can be implemented by reading the reports or developing learning-based method.

2) Iterating flow: The most common scenario of iteration flow is that if the timing is not satisfied and users need to rerun the entire flow with different tool or tool parameters. Back annotation is another example. Given another example, if a user want to use physical-aware logic synthesis [9] to improve the design quality, he needs to run the placement and routing applications first, and the feed the result back to the logic synthesis application. Fig. 2b gives the schematic of back annotation. Logic synthesis tool read the .sdf generated by placer to guide the re-synthesis to get higher QoR.

D. Flow Tuning

EDA tools provide numerous options and parameters that can drastically impact design quality. These parameters create an enormous and complex design space, and it's impossible to try all the design point to find the optimal solution. Several design space exploration methods like intelligent search strategies, parallel computing and learning-based method [10] can help solve such problem. Due to interactive layer abstraction, design space exploration methods can be easily imported to Cocoon. Cocoon can not only tune the tool parameters of flow, but also tune which tool to use in the customized flow. Figure 3 shows how cocoon support the flow tuning. The tuner pulls the features extracted from the checkpoint or reports and propose tool options to the design flow. Design flow adopts the tool options, executes and generates new design checkpoint and reports. Such process finishes until the design meets the performance requirements or reaches the maximum number of iterations.

IV. Implementation and Evaluation

This section presents a preliminary implementation of Cocoon and gives two demos to show that Cocoon is feasible and flexible for integrated EDA.

A. Implementation

We use Python 3 to implement Cocoon. The application layer in Cocoon is implemented as a class. Each class instance will wrap the implementation of a tool, including the member functions like parameters setting and scripts generation. The data layer in Cocoon is implemented as a database with a fixed directory. Interface layer are implemented as global functions. Users need to modify design class to specify the design source, and then write their customized flow class, with specifying the tool name and parameters as well as execution order. Cocoon will analyze the customized design flow automatically, and then assert legality check and run the flow.

B. Evaluation

We gives two demos to demonstrate that Cocoon can support customized flow definition and flow tuning.

1) Branching flow: Figure 4 shows the code of a branching flow. Cocoon first push the HDL to Genus and Yosys and then pull the checkpoint after the synthesis. Then cocoon call evaluate method to compare the results and push the checkpoint with better estimated results to Innovus to do the floorplan and placement. Also parameters in every application step can be define in the flow in the same time as shown in line 10. The tag after application tool name is used to mark checkpoint status for now.

2) Flow tuning: The flow tuning problem can be classified as a black-box optimization problem. We treat the flow is a black-box by supplying input design and parameter settings and measuring the output response in terms of design quality. In this demo, we use hyperopt [11], an open source hyperparameter optimization tool to solve the flow tuning problem. Demo is in our github repository.

```
1   def flow(self):
2
3       app_synth = []
4       app_synth.append(("YosysSynth", "to_synth", "Timing"))
5       app_synth.append(("GenusSynth", "to_synth", "Timing"))
6       self.apps.append(app_synth)
7
8       app_floorplan = [("InnovusFloorplan", "to_floorplan")]
9       self.apps.append(app_floorplan)
10      self.params_fp.append(("r","1.0 0.7 0.0 0.0 0.0 0.0"))
11
12      app_pdn = [("InnovusPDN", "to_pdn")]
13      self.apps.append(app_pdn)
14
15      app_place = [("InnovusPlace", "to_place")]
16      self.apps.append(app_place)
17
18      app_cts = [("InnovusCTS", "to_cts")]
19      self.apps.append(app_cts)
20
21      app_route = [("InnovusRoute", "to_route")]
22      self.apps.append(app_route)
```

Fig. 4: Demo code for branching flow.

V. Conclusion and Future Extensions

In this paper, we first expand the concept of electronic CAD framework to integrated EDA. Then we propose Cocoon, an infrastructure enhancing the interoperability and interactivity for integrated EDA. With the help of interface definition, Cocoon can represent customizable and tunable EDA flows. We implement two demos to demonstrate that Cocoon is feasible and flexible for integrated EDA.

Cocoon is under-development now about legality check, embedded flow implementation. In the future, Cocoon will consider about cloud deployment, including distributed checkpoint design and computation graph scheduling of applications.

References

[1] T. J. Barnes, D. Harrison, A. R. Newton, and R. L. Spickelmier, *Electronic CAD frameworks*. Springer Science & Business Media, 2012, vol. 185.

[2] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, "Toward an open-source digital flow: First learnings from the OpenROAD project," in *Design Automation Conf. (DAC)*, 2019.

[3] J. Jung, P.-Y. Lee, Y.-S. Wu, N. K. Darav, I. H.-R. Jiang, V. N. Kravets, L. Behjat, Y.-L. Li, and G.-J. Nam, "DATC RDF: Robust design flow database," in *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2017.

[4] A. B. Kahng and I. L. Markov, "Impact of interoperability on CAD-IP reuse: an academic viewpoint," in *Int'l Symp. on Quality Electronic Design (ISQED)*, 2003.

[5] "Edalize," https://github.com/olofk/edalize, accessed: 2020-10-15.

[6] "Hammer," https://github.com/ucb-bar/hammer, accessed: 2020-10-15.

[7] K. Kaur, D. S. Sharma, and D. K. S. Kahlon, "Interoperability and portability approaches in inter-connected clouds: A review," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–40, 2017.

[8] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang, "LAMDA: Learning-assisted multi-stage autotuning for FPGA design closure," in *Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[9] S. Chatterjee and R. Brayton, "A new incremental placement algorithm and its application to congestion-aware divisor extraction," in *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2004.

[10] M. M. Ziegler, H.-Y. Liu, and L. P. Carloni, "Scalable auto-tuning of synthesis parameters for optimizing high-performance processors," in *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, 2016.

[11] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Int'l Conf. on Machine Learning (ICML)*, 2013.