INVITED: mflowgen: A Modular Flow Generator and Ecosystem for Community-Driven Physical Design

Alex Carsello, James Thomas, Ankita Nayak, Po-Han Chen,
Mark Horowitz, Priyanka Raina, Christopher Torng
Stanford University, Stanford, CA

ABSTRACT

Achieving high code reuse in physical design flows is challenging but increasingly necessary to build complex systems. Unfortunately, existing approaches based on parameterized Tcl generators support very limited reuse as designers customize flows for specific designs and technologies, preventing their reuse in future flows. We present a vision and framework based on modular flow generators that encapsulates coarse-grained and fine-grained reusable code in modular nodes and assembles them into complete flows. The key feature is a flow consistency and instrumentation layer embedded in Python, which supports mechanisms for rapid and early feedback on inconsistent composition. We evaluate the design flows of successive generations of silicon prototypes built in TSMC16, TSMC28, TSMC40, SKY130, and IBM180 technologies, showing how our approach can enable significant code reuse in future flows.

Keywords: physical design, ASIC flows, VLSI, modularity, reuse

1 INTRODUCTION

Rising non-recurring engineering costs in advanced technology nodes are motivating the hardware community to adopt agile development principles and new methodologies to reduce design effort. Code reuse is particularly important to reduce the effort to build complex physical design flows. The physical design community has been slow to adopt agile principles for a few key reasons. First, physical design is culturally characterized by the "one big release" operating model with high stakes and strict annual schedules. Opportunities for code reuse disappear quickly as risk-averse teams customize scripts aggressively for their specific design and technology. Existing approaches offered by commercial EDA vendors typically exploit reuse by leveraging parameterized Tcl templates and generators to create initial design- and technology-agnostic flows [2, 4]. These flows enable efficient reuse until a need arises for which no parameter exists. As flows are inevitably customized, these frameworks do not support propagating reusable code to different designs and technologies. Second, the Tcl language continues to dominate commercial EDA toolflows, but it lacks language features that can help compose reusable code from different sources (e.g., introspection, gradual typing). Furthermore, modern machine learning CAD solutions are emerging that may not leverage Tcl at all but must still compose with existing flows [3, 5]. Future physical design flows seeking to reduce design effort must aggressively preserve reusable code as codebases are specialized while supporting a heterogeneous mixture of Tcl and non-Tcl code.

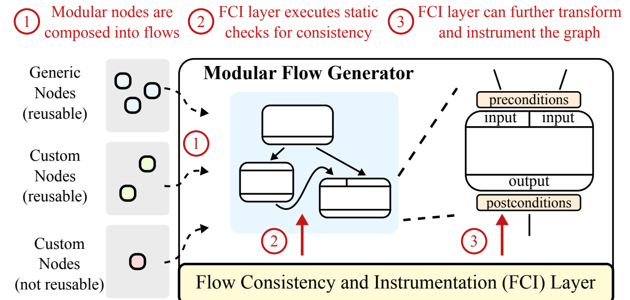


Figure 1: A Modular Approach to Physical Design Flows – Flow code can be difficult to reuse. Modularity enables reuse, and flow assembly in high-level languages (e.g., Python) enables checking for consistent composition.

We explore a vision and framework to enable reusable physical design flows based on *modular flow generators* coupled with a *flow consistency and instrumentation (FCI) layer* embedded in Python. Unlike existing parameterized Tcl generators, the goal of a modular flow generator is not to emit Tcl but to provide the necessary abstractions to compose and reuse code. Figure 1 shows how a modular flow generator composes modular nodes from both generic sources and custom sources (i.e., design- or technology-specific) into a graph representing the assembled flow. Since nodes from different projects can be inconsistent with each other, we introduce a Python-embedded FCI layer that allows designers to express and annotate the properties required for a node's composition and reuse, and the mechanisms to lift and statically check these properties across a distributed code base. The layer can also instrument each modular node to extend its reuse to different scenarios.

Modular flow generators provide the interfaces and tools to separate concerns and can enable an ecosystem of nodes for community-driven physical design. Our work contributes (1) *mflowgen*, an open-source (m)odular (flow) (gen)erator with a flow consistency and instrumentation layer and mechanisms to reuse code across different designs and technologies, as well as rapid and early feedback on inconsistent composition; (2) a common reusable library of over forty technology- and design-agnostic modular nodes for both commercial and open-source tool flows[1] ; and (3) a detailed evaluation of physical design flows for silicon prototypes in TSMC16, TSMC28, TSMC40, SKY130, and IBM180 technologies, demonstrating the potential for significant code reuse in future physical design flows.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530633

[1]https://github.com/mflowgen/mflowgen

This work is funded by the DARPA Domain-Specific System on Chip (DSSoC) program and Stanford's Agile Hardware Center and SystemX Alliance.

2 SYSTEM GOALS

In this section, we overview the overarching design goals and principles that motivate our approach and to maximize the potential for significantly reducing design effort in physical design.

Goal 1: Must achieve *significant* code reuse – Complex physical design flows require a tremendous effort to build. Building a *different but similarly complex design* will again require a similar effort. As a result, meaningfully reducing design effort will likely require that most of the physical design flow be reused (e.g., 90%+). Three key design principles follow from this requirement for significant code reuse. First, it is important to capture not only coarse-grained code reuse like most existing approaches (e.g., synthesis, place, route) [2, 4], but also fine-grained reuse (e.g., glue scripts, reporting and analysis, generator wrappers). Second, we must support a mechanism to *tweak* reusable code since small changes should not preclude reuse. Third, the friction to design for reuse must be low to encourage the widespread adoption necessary for success.

Goal 2: Composition must support code from different designs and technologies – Physical design flows are aggressively specialized, and there is no avoiding this fundamental need. However, design-specific flow code can feasibly be reused across technologies (e.g., a tile-based array floorplan). Technology-specific flow code (e.g., design-for-manufacture rules) can similarly be reused in neighboring blocks of the same design. Two key design principles follow from the requirement to support such reuse. First, our approach must support a mechanism for checking composability and consistency across nodes and a shared language for expressing requirements. Second, we must *require a static code analysis approach* because code fragments in a physical design flow are distributed across tools and files and *not in memory at the same time.*

Goal 3: Feedback on inconsistent composition must be both rapid and early – Physical design flows have long run times, with RTL-to-GDS iterations often consuming days of compute on powerful server farms. Run-time assertions late in the flow execute after long waits, and control flow must reach problematic code to trigger errors. This can result in flows that fail intermittently, breaking trust in a reusable approach. We make the key observation that it is possible and reasonable to separate two aspects of flows: (1) running the tools to physically construct the design, (2) running the tools to evaluate variables which turn out to be inconsistent in a composed flow. We hypothesize that generating feedback on inconsistent composition does not require the former, and we need not pay the run time penalty. The key takeaway reinforces a static code analysis approach and formal property checks, which enables rapid and early feedback without waiting for physical construction.

3 MODULAR FLOW GENERATORS

Our system goals motivate a flexible modular node abstraction capable of capturing both coarse- and fine-grained opportunities for reuse. Specific examples of captured code reuse may include a bump routing methodology for flip chip packages, an approach for design for manufacture (DFM) structures, adding power domains, ECO timing fixes, or a hierarchical power distribution strategy.

3.1 Modular Node Abstraction

We illustrate the schema for a sufficiently flexible modular node abstraction in Figure 2 which resembles a function signature with

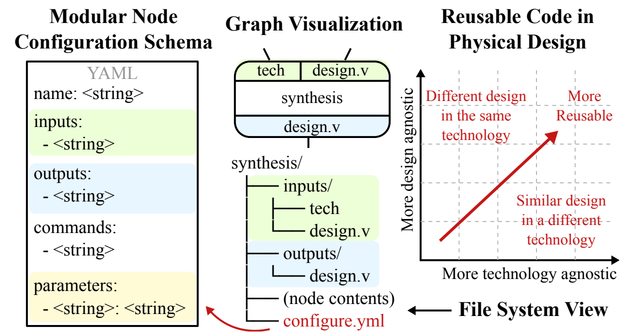


Figure 2: Modular Node Abstraction – Schema for the configuration of a modular node. Graph edges are file-based. The commands in each node consume inputs and optional parameters and produce the outputs.

file-based inputs and outputs. The schema differs from a traditional software function because physical design depends heavily on files (e.g., netlists, databases, cell libraries), but is also higher than a low-level build system (e.g., make) since nodes are self-contained with internal scripts that access parameters defined in the configuration. The example node synthesizes a gate-level netlist from a technology package and RTL design. There is no built-in support to ensure that nodes produce the expected results. Section 4 will explore mechanisms for stronger guarantees.

3.2 Categorization of Nodes that Capture Reuse

Figure 2 also shows two axes of reuse for technology- or design-agnostic code. Code agnostic to both is most reusable and often least performant, but many reusable code blocks have little impact on performance (e.g., converting libs-to-db). Code agnostic along only one axis can be challenging but still feasible to reuse (e.g., design-specific tile-based array floorplan, technology-specific DFM tasks). Code in the lower-left region has no opportunity for reuse.

Representing the upper-right region, we have built a common library of technology- and design-agnostic modular nodes[2] with a wide range of common functions (e.g., synthesis, floorplan, DRC) that can be assembled into basic flows that are functional out-of-the-box in many modern technologies (see Section 5). This capability is similar to existing work [1, 2, 4] but is designed for our system vision. Each common library node is parameterized (e.g., hierarchy flattening, clock gating, target slack), technology-independent (e.g., distances are multiples of metal track pitch), may be swappable between vendors (e.g., synthesis with Cadence Genus, Synopsys DC, or open-source yosys), and can be replaced entirely or decomposed into finer-grained nodes to more precisely capture reuse.

3.3 Flow Assembly

We provide a Python-based DSL for programmatically connecting modular nodes into graphs that represent assembled flows (see Figure 3). The DSL supports a basic graph structure and can add or modify each node's parameters. This approach satisfies Goal 1 from Section 2 by providing an environment to rapidly assemble coarse-grained and fine-grained code fragments *using the same modular abstraction* (in contrast to existing approaches built from coarse-grained steps and Tcl hooks that are more difficult to modify).

[2] https://mflowgen.readthedocs.io/en/latest

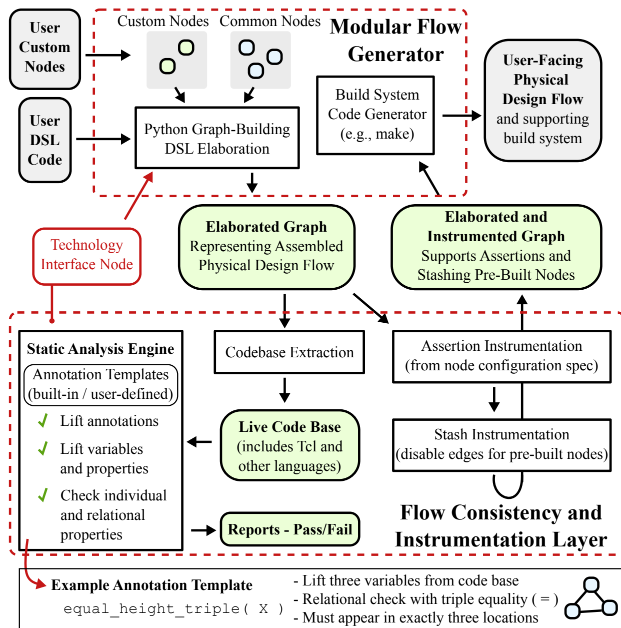


Figure 3: Complete Toolflow Block Diagram – mflowgen assembles the physical design flow from user DSL code. The flow is checked for consistency and instrumented before producing the final flow.

A modular flow generator enables physical design engineers to productively assemble flows of varying complexity including basic flows for initial prototyping and partial flows for test. In academia, simple teaching flows can be assembled from common library nodes and individual nodes can be incrementally swapped or added for educational purposes. The Python DSL also opens opportunities for graph transformations, for example unrolling a loop and sweeping a parameter (e.g., clock period) for design-space exploration.

4 FLOW CONSISTENCY & INSTRUMENTATION

The goal of the FCI layer is to rapidly detect inconsistencies and to provide stronger guarantees on node functionality. For example, an otherwise reusable node for adding power domains may use a specific power switch cell that is not composable in a different technology. Our approach pulls these checks forward using static code analysis to detect inconsistencies at graph elaboration time.

4.1 Consistency Checks

Figure 3 shows how the modular flow generator feeds the FCI layer by elaborating the user DSL code into a detailed in-memory graph model representing the assembled flow. The FCI layer introspects the graph model to assemble the live code base of modular nodes.

Our tool flow provides the infrastructure to add *property-check annotation templates* to the static analysis engine. The first role of these templates is to translate between the node's programming language (e.g., Tcl) and the host language (e.g., Python) to unify all property checks under the same language. Through this translation, templates identify the variables that are relationally connected and their properties across the distributed code base.

The simple example in Figure 3 shows a template that enforces equality across three annotated variables. This may be used to enforce height-matching of three floorplan layouts such that they can be tiled together side by side. Mismatched values, or having only two of three annotations, would fail the property check.

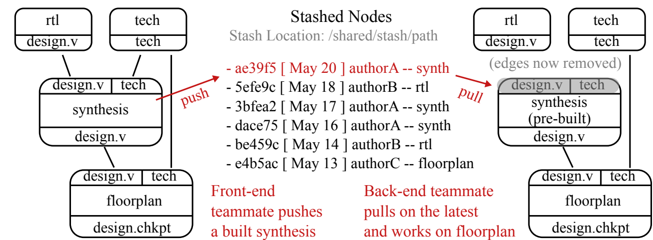


Figure 4: FCI Layer Transformations for Pre-Built Nodes – The FCI layer enables building and stashing modular nodes, and sharing them as vendor packages across a team (with input edges removed).

4.2 Flow Instrumentation

Figure 3 also illustrates how the FCI layer can instrument all modular nodes with additional functionality. For example, modular nodes are natural checkpoints to share pre-built collateral across a team. Figure 4 shows how built nodes can be *stashed* into a shared team space from which other team members can pull into their graphs. On a stash pull, the FCI layer transforms the graph in-place to break input dependency edges, resulting in a static vendor package that simply supplies outputs and is never re-built *regardless of the build state of prior nodes* (unlike Makefiles), enabling agile team workflows. The FCI layer can also extend the modular node configuration schema to insert run-time assertions before and after each node (e.g., in Figure 1), addressing scenarios where a desired check is data-dependent (e.g., parsing and flagging unexpectedly poor-quality results). Pre- and post-conditions are Python snippets and can be run with full-featured software testing tools (i.e., pytest).

5 EVALUATION

We have built silicon prototypes in multiple technologies to evaluate code reuse with a modular flow generator approach. Our primary indication of success will be (1) achieving significant code reuse for custom code reused to build 2nd+ generation designs, because we expect existing frameworks [1, 2, 4] to perform similarly to build 1st generation designs, and (2) the speed of static property checks running on large complex codebases. Table 1 lists the high-level specifications of six chips. At a glance, total code reuse for each chip was high (totals of 80%+ lines of code reused), with 2nd+ generation designs achieving good code reuse from previous designs. All chips were completed with very short timelines of six months or less.

5.1 Benefit of Modularity

We evaluate the benefit of modularity against existing approaches based on parameterized Tcl generators [2, 4], which encourage implementing custom features in Tcl hooks (e.g., injected pre- and post-each step), rather than directly composing both fine-grained and coarse-grained code fragments with the same modular abstraction.

We consider DenseAccel16 and its evolution from two previous iterations of the same design (less complex) in the same 16nm technology and metallization. We refer to the iterations as DenseAccel16-1 to DenseAccel16-3. All three iterations instantiated a 2D tile array of processing elements with a per-tile power domains feature. The first iteration integrated the feature monolithically into the Tcl

Table 1: Chips Built and Fabricated with mflowgen

	DenseAccel16	**MiniCGRA**	**CryptoAccel**	**DenseAccel40**	**RVMulticore**	**BaseSynch**
Application Domain	Image Processing/ML	Image Processing	Cryptography	ML	General Purpose	Wireless
Technology	TSMC 16nm	SKY 130nm	SKY 130nm	TSMC 40nm	TSMC 28nm	IBM 180nm
Area (mm^2) / Frequency (MHz)	25 / 750	10 / 60	10 / 325	29.2 / 200	1.25 / 500	2.31 / 20
Number of Cores	384 PE, 128 MEM	24 PE, 8 MEM	1	256 PE	4	2
# of Power/Clock Domains	513 / 3	1 / 1	1 / 1	1 / 4	1 / 1	1 / 4
% reused from common lib	30%	58%	94%	over 80%	86%	84%
% reused from prev. designs	50%	24%	First design	First design	First design	First design
Months to tapeout	6	2.5	2.5	6	2	1.5
Static check run time	2.2 sec	0.8 sec	0.2 sec	0.6 sec	<1 sec	<1 sec

floorplan code mixed with other features. The second/third were built as modular nodes. The **time to port power domains code from DenseAccel16-1 to DenseAccel16-2 was two months, while the time from DenseAccel16-2 to DenseAccel16-3 was two days, both for a single student of high expertise, representing a 30x improvement**. We attribute the difference to the tangling of features in monolithic Tcl scripts, requiring our designer to spend weeks understanding every line of code to decide which lines affect power domains and which lines also break other features (e.g., place blockages with multiple purposes). Effort went to gathering relevant code into one place and debugging the port over time with other features. In contrast, moving from DenseAccel16-2 to DenseAccel16-3 was far simpler. A single node captured all code related to power domains, and the node was designed as a vendor package supplying code fragments across the flow.

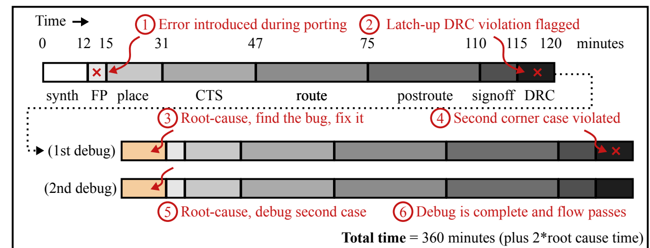
5.2 Evaluating Static Property Checks

We break down flow tool spin times for a task that involves porting the power domains feature from the DenseAccel16 processing element tile to a memory processing tile of similar complexity. Figure 5 shows timelines with and without static property checks. The entire synthesis-to-DRC flow completes in 120 minutes.

Impact on debug loop – In the baseline flow without static property checks in Figure 5(a), our physical design engineer attempted an initial port (with multiple undiscovered bugs interacting with code downstream) and ran the entire flow before finding a latch-up DRC violation two hours later. They root-caused the bug (orange bar split in first debug timeline, this could span minutes, hours, or days). After proposing a fix for this bug in isolation, which included understanding the DRC report, the purpose of all code statements, and filtering lines for blame, they attempted the full flow again, discovered the second bug, and began the next debug timeline. This debug loop repeated. In the second flow with static property checks in Figure 5(b), the initial designer not only implemented the feature but also captured properties expressing their *design intent* (e.g., align the placement of an always-on power region to the standard-cell row height, to avoid latch-up DRCs). A 2nd+ generation designer no longer needs to rediscover these properties, and the FCI layer executes these checks statically within a few seconds at graph elaboration time (before any physical design tools are run). **The entire debug loop reduces by about 3x**, simply passing the property checks.

Static property check run times – Table 1 quantifies the FCI layer run time to inspect the entire codebase for each chip. We sum

(a) No static property checks in use



(b) Includes static property checks

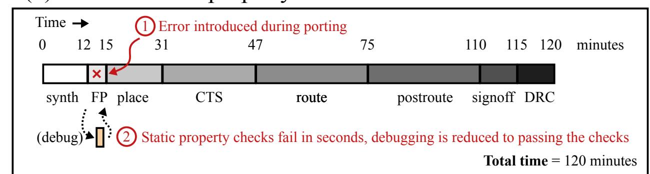


Figure 5: Debug Loop Timelines with Static Property Checks

the times for hierarchical sub-designs, with each number collected over five trials on a 2.4GHz Quad-Core Intel Core i5-8279U laptop-class CPU. The static check run times are quick, ranging from 0.2s to 2.2s for the largest codebase. We ran a study scaling up to 1000 property checks, which still completes quickly in under 20s.

6 CONCLUSION

Modern physical design approaches lead to aggressively tuned flows that prevent code reuse. We present a system vision and framework based on modular flow generators to maximize the potential for significantly reducing design effort. We use the same modular abstraction to compose both coarse-grained and fine-grained code fragments to capture greater reuse, and we provide infrastructure to annotate static property checks for flow consistency. Our evaluation demonstrates potential for modular flow generators to support a future of community-driven, reusable physical design.

REFERENCES

[1] SiliconCompiler. https://docs.siliconcompiler.com/en/latest.
[2] Synopsys Reference Methodologies. http://solvnet.synopsys.com/rmgen.
[3] A. Mirhoseini et al. A graph placement methodology for fast chip design. *Nature*, Jun 2021.
[4] E. Wang et al. A methodology for reusable physical design. *Int'l Symp. on Quality Electronic Design (ISQED)*, Mar 2020.
[5] Y. Zhang et al. GRANNITE: Graph neural network inference for transferable power estimation. *Design Automation Conf. (DAC)*, Jul 2020.