Automated Circuit Elaboration from Incomplete Architectural Descriptions

Andrew Becker, David Novo and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH–1015 Lausanne, Switzerland
Email: {first.last}@epfl.ch

***Abstract*—Arithmetic circuits are some of the most common circuits, yet building generators for these circuits is usually both ad-hoc and error-prone. Often, generator designers do not directly use Register Transfer Languages, but instead use scripting languages (e.g., Perl) to generate RTL and overcome the limited expressivity of typical RTL languages. We present a new approach to generator construction, where the design language is natural and expressive, and designers are provided with special facilities to alleviate typical sources of errors. This builds on previous work, where designers write an incomplete design (a *sketch*) and provide a functional reference; a complete, correct design is then synthesized. Notably, we address scalability problems in the general case with an approach tailored specifically for arithmetic generators: satisfying values for the uncertainties inserted in the designs are discovered incrementally as the generator parameters grow the size of the generated circuits. This approach results in significantly reduced solution times, sometimes up to 25 times faster than the naı"ve strategy.**

I. Introduction

Because arithmetic circuits are such common components in many digital designs, it is essential to minimize the effort required to build them. The traditional approach to minimizing design effort has been to raise the level of abstraction, e.g., from transistor-level schematics to gate-level schematics, or from gate-level schematics to behavioral RTL, or most recently, from RTL to *High-Level Synthesis (HLS)*. Unfortunately, while HLS was originally thought to be quickly developed and improved, reality has turned out differently. HLS is still nowhere near as capable of producing quality high-performance arithmetic circuits as a skilled RTL designer.

Since HLS has not been as broadly useful as hoped, yet arithmetic circuits are just as pervasive as ever, designers have largely turned to Intellectual Property (IP) libraries, either commercial, proprietary ("in-house"), or self-built by the end user designers themselves, focused on circuit generators. These generators, or RTL metaprograms, are typically ad-hoc scripts (Perl is very common) which literally print RTL for specific circuit architectures based on various parameters, e.g., operand bit-width.

These RTL metaprograms are notoriously tricky to build correctly: they require the author not only to generalize their design to account for various parameters, but also to accomodate all possible edge cases and error conditions.

We show how to extend a modern incarnation of an RTL language, Chisel [1], to provide facilities for abstracting away details and automatically generalizing parts of the RTL design. Specifically, we extend our previous work on SketchiLog [2], which is responsible for concretizing these abstractions and providing complete RTL. This previous work provides mechanisms for designers to explicitly specify undetermined parts of their circuits as *holes*, and automatically finds satisfying completions to those holes to achieve functional correctness while maintaining the designer's intended architecture. We address one key limitation of this approach by allowing designers to explicitly mark the parts of what are usually regular structures created by their generators. This allows our automated "hole filler" tool to re-use the correct hole values across instances with different parameters, to reduce the size of the problem it must solve, and therefore to operate on instances that would otherwise be too large.

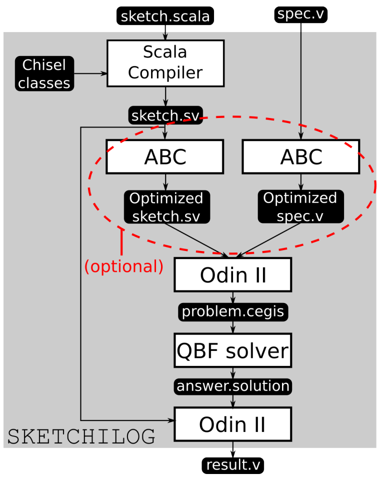


Fig. 1: The original SketchiLog tool flow: from sketch to completed Verilog file.

II. Sketching Circuits with SketchiLog

A. Code Generation and Solution Recovery

The process begins with a designer writing a generator in Chisel and including some holes. After the generator is written, the first step is the static elaboration phase: the user chooses some parameter values and runs the generator, resulting in pseudo-RTL code if the generator code is error-free. This pseudo-RTL, described with the extension *.sv* in Fig. 1 above, is netlist-like Verilog with an additional construct to abstract away signal values—signals can be assigned unconstrained values:

```
assign one_bit_hole = 1'b??;
```

It is then up to a modified version of Odin II [4] to parse the extended Verilog input, along with the user-supplied reference design. Odin II acts as a source-to-source translator; it gives each parsed hole signal a unique ID, and produces a text file containing the circuit produced by the Scala compiler, with hole signals renamed with their unique IDs, and the reference circuit in a flat HDL-like format acceptable to the CEGIS QBF solver [6].

In essence, both the sketch and the specification are translated to flat Boolean functions $\mathcal{P}$ and $\mathcal{S}$, respectively. Both boolean functions must take the same *k* bits of input parameters *x*, but the sketched function also takes an additional *m* bits as a parameter *c*. This number *m* corresponds to the total number bits for all holes in the sketched function. The sketching problem then reduces to solving the quantified boolean formula (QBF) satisfiability problem, which can be defined as:

$$\exists c \in \{0,1\}^m, \forall x \in \{0,1\}^k : \mathcal{P}(x) \Leftrightarrow \mathcal{S}(x,c)$$

Next, the CEGIS QBF solver is invoked on this file, and runs until either an assignment to all hole signals (*c*) is found which implies $\mathcal{P} \Leftrightarrow \mathcal{S}$, or it can prove that no such assignment exists. If there is such an assignment *c*, the solver produces a text file containing the value of *c*. If there is no such assignment *c*, there is no way to complete the sketch to be functionally equivalent to the specification, the failure is reported to the user, and the tool flow ends.

Finally, if a solution was produced, the modified Odin II is executed again in *recovery* mode. In this mode, when Odin II parses a hole generator signal assigned an indeterminate value from the sketch Verilog file, it replaces that indeterminate signal value with the corresponding value from the solution file.

B. SKETCHILOG*'s Sketching Constructs*

To expose the solver's ability to handle uncertainty in designs in a way that is easy for designers to understand and requires a minimal amount of extra glue logic, SKETCHILOG provides a few key facilities beyond what Chisel provides on its own. These facilities can only be used to express uncertainty in the structure generated; they cannot be used to express uncertainty in the generator code itself. For example, they can express an uncertain select line value for a multiplexer, but cannot be used to express an uncertain loop bound inside the generator.

While the extra glue logic required to implement these features expands the size of the QBF-SAT model solved, the resulting circuit can trivially optimize away most of that logic. This makes these constructs nearly free in terms of the solution's quality.

The first feature provides a naked *uncertain value* macro:

```
x := ??(n);
```

This is as close to the solver's raw abilities as possible: it is a placeholder for an unconstrained n-bit value which the solver must find under the sketch equivalence condition. This essentially means that the designer transparently inserts an n-bit existentially-quantified variable into the QBF-SAT instance constructed by the solver core. All subsequent macros can be constructed from this first feature; the following macros are simply there to handle common use cases which would be tedious and error-prone to re-implement.

The first of these convenience macros is an explicit multiplexer generator:

```
x := either choose v or w or y or z;
```

One of the most common uses for the SKETCHILOG constructs is to express an uncertain selection; this construct allows exactly that. The macro above would be used when a designer knows that `x` is actually one of `v`, `w`, `y`, or `z`, but is unsure which. SKETCHILOG's static elaboration phase will create a new *raw hole* (a 2-bit uncertain value) and a multiplexer in front of `x` with `v`, `w`, `y`, and `z` as inputs and the raw hole connected to the select line. The solver is responsible for determining which option is actually the correct choice.

In a similar vein, SKETCHILOG also provides the ability to select among signals in an array or bitvector, filtered by a partially-constrained expression:

```
x := an_array(2 * ??(n) - 1);
```

This is functionally similar to the selection operator above, but provides a concise way to specify which elements of the array or vector should be considered by the solver as possible values for x. In this case, indices 1, 3, 5, etc. would be considered—any index within the bounds of *an_array* that satisfies the above expression with an n-bit raw hole. If a possible satisfying index is outside the bounds of *an_array* (e.g., if the raw hole had a value 0 and therefore the index value was -1), the solver does not consider it. This provides a convenient way for designers to clearly express the intended relationship without cluttering the generator with bounds checks.

The final macro is perhaps the most powerful, but also the most dangerous (in terms of scalability). SKETCHILOG allows a designer to specify nothing more than a dependence relationship, and have the solver synthesize a look-up table implementing the correct functionality:

```
x = BB(dependent_on, n);
```

This could be accomplished by a designer with $2^{dependent\_on.width}$ n-bit raw holes and a block of glue logic, but this macro makes it easy for a designer to specify only the existence of a relationship between x and *dependent_on* and the bitwidth of x. This works best when the *dependent_on* signal is not too large.

C. *The Limitations of* SKETCHILOG

The main fundamental problem preventing SKETCHILOG from being widely used is scalability. SKETCHILOG's scalability is limited in three ways: first, in the size of the circuit being sketched; second, in the structure of the sketch and specification; and third, in the number of quantified variables (i.e., the number and size of holes).

To formulate a QBF-SAT problem from a given sketch and its specification, SKETCHILOG constructs a *miter* from the sketch and its specification. This essentially combines the two circuits into one—they share primary inputs, and their primary outputs are checked for equivalence. This means their corresponding primary outputs are XORed, and the outputs from those XOR gates are ORed together, giving the miter a single primary output whose value is 0 when the sketch and the specification are functionally equivalent for a given input vector, and 1 otherwise. The QBF-SAT solver's job is then to find a *control vector*, or assignments to the holes, such that the resulting miter is unsatisfiable; in other words, it must try different control vectors until either the sketch and its specification are provably equivalent or it can prove no such control vector exists.

In practice, this usually means launching multiple instances of a regular Boolean satisfiability (hereon, just SAT) solver on that miter with different control vectors (and certainly means so with the CEGIS [6] solver used in SKETCHILOG). The first two scalability limitations are well-known for SAT solvers.

SKETCHILOG already attempts to address the size limitations with an optional model re-synthesis stage, where the miter components are first imported to a logic synthesis tool for circuit minimization. The second scalability limitation, the structure of the miter, is inherent in the NP-complete SAT problem: some circuits likely simply cannot be solved efficiently. Other circuits might be efficiently solvable, but are not efficiently solved with current SAT solvers; this situation will improve with advances in SAT solving technology.

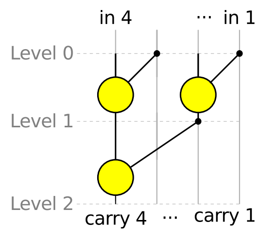
This work attempts to address the third scalability constraint: the number of bits in the control vector, or intuitively, the number of holes.

III. TAILORING SKETCHILOG FOR ARITHMETIC CIRCUIT GENERATORS

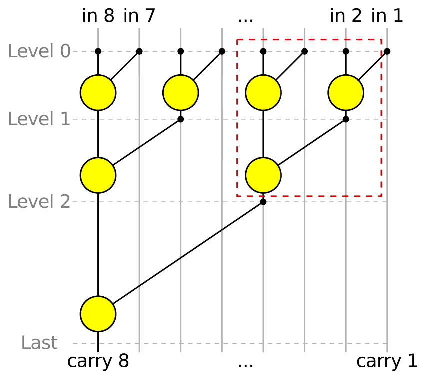
We address SKETCHILOG's limited scalability with increasing numbers of holes for arithmetic circuit generators by exploiting the regularity and incremental nature of many of these circuits' common architectures. If sketches of a 16-bit adder and a 32-bit adder share parts of their architectures in common, and the associated holes have the same satisfying values, it makes sense to first find the satisfying values for those holes the two share in common, then to find satisfying values only for the holes in the "new" parts of the architecture.

A. *Advantages and Limitations of an Incremental Approach*

For example, suppose a designer sketched a Kogge-Stone adder generator. Specifically, suppose the designer left the choices of cell connections partially unconstrained. If the designer tagged each of those choices with a `HoleTag`, our tool will detect hole values which remain unchanged from one adder size to the next.



(a) Part of a 4-bit Kogge-Stone carry cell graph.



(b) The structure of part of an 8-bit Kogge-Stone carry cell graph. Note how the connections in the 4-bit graph of (a) are subsumed in this graph.

Fig. 2: Two partial Kogge-Stone carry cell graphs. Many arithmetic circuit generators have the interesting property that the core of the circuit remains unchanged as certain parameters, like operand size, increase.

Figure 2(a) shows part of the carry cell graph which would be present in a solved sketch with a operand size parameter of 4 bits. Fig. 2(b) shows the same for an operand size parameter of 8 bits. Note the section of the graph highlighted in red: the connections, and thus the hole values, in this instance are identical to those of the smaller instance.

The sketch generated for the 8-bit instance can now re-use the information discovered by solving the smaller instance, and does not need as many existentially quantified variables in the QBF-SAT model used to solve the larger instance. Crucially, this enables an incremental approach to generator sketch solving, which significantly reduces the time required to concretize the hole values and synthesize a complete RTL description for the larger instance. This can be repeated multiple times, enabling the system to scale to sketches far larger than would be possible otherwise.

Until now, we have only considered cases where there is a unique solution to a sketch. In practice, unless the designer is very judicious in their use of holes, multiple solutions may exist. By removing some freedom from the larger sketch instances, we not only simplify the problem, but also remove from consideration other possible solutions which have different assignments to those tagged hole values.

Taken to the extreme, this could possibly result in a very sub-optimal solution, effectively forcing the solver into a sort of local minimum which is nowhere near the global minimum. This requires the designer to be mindful of the implications of each hole she inserts. It also means that using an incremental strategy is less effective if sketching to explore a design space—multiple solutions must be found for both the smaller and larger phases. Still, the incremental approach enables targeting sketch sizes that would be unfeasible with a nonincremental strategy.

B. The Key New Design Construct

Because there is no clear way to automatically detect which parts of an architecture are common between different operand sizes, we rely on the designer to explicitly tag each hole which will have a common solution across different operand sizes. While it may sound like a difficult task, we need only a simple and intuitive construct to enable the solver to correlate hole values across instances.

```
    t = HoleTag("main loop", i, j);
```

The value t can then be passed to the above hole interfaces to mark the hole instantiated with the information that it is associated with the specific values i and j in the generator's main loop kernel. For example, a designer might use the hole tag above like:

```
    x := ??(4, t);
    y := either choose w or x tag t;
```

Each such hole tag must have a unique tag string—the Chisel static elaboration phase will report an error if it detects more than one hole references the same tag, or if multiple tags are created with the same tag string. Any hole not explicitly tagged will be treated as unique to each instance.

When our tool solves a sketch, it determines for each contained hole whether there exists a hole in the provided previously-solved sketch with an identical tag string. If so, no existentially-quantified variables are added to the model being solved. Instead, those holes are assumed to have the same values as the previously-solved sketch.

It is important that the generator designer have this facility to explicitly specify the holes which will have identical values across instances. Not all holes will have the same values between instances, and holes may appear in a different order. Allowing the designer to specify the exact mapping between holes and the environment which created that hole gives the designer maximum flexibility while minimizing sources of confusion.

C. Incremental Design Expansion and Solving

The modifications needed to enable incremental sketch solving for generators are relatively minimal. First, the generator designer must annotate their designs with the tags described above. As shown in Fig. 3, the static elaboration phase (shown as the "Scala Compiler" unit) remains unchanged. However, it must now be invoked twice: once to elaborate the smaller

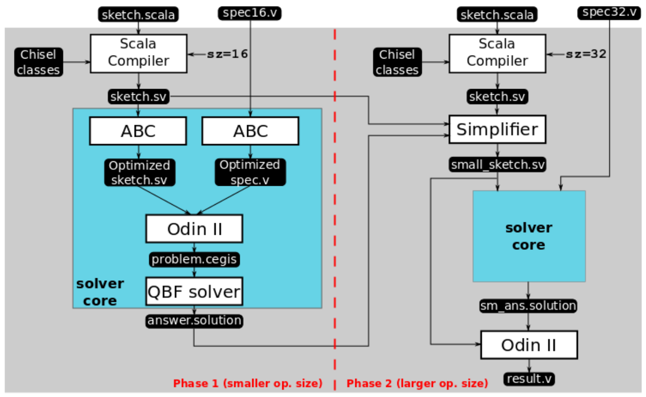


Fig. 3: The new tool flow: designs are elaborated at increasing operand sizes, and a solution from a previous stage is used to reduce the QBF-SAT problem complexity for larger operand sizes.

instance, and once more to elaborate the larger instance. The pseudo-RTL code emitted is unchanged from the original SKETCHILOG output, except for the tag string.

The smaller instance must be solved by the solver core, but it need not be concretized into a regular Verilog file (i.e., without holes). Instead, the solution generated is exported to the larger instance's simplification phase.

The simplification phase has two objectives: to find hole tags shared in common between instances, and to replace those holes in the larger instance whose tags match tags found in the smaller instance. When such a match is found, the simplifier replaces the hole assignment in the pseudo-RTL with a concrete assignment to the value discovered by the smaller instance's solver core. After all tag matches have been found and their corresponding assignments concretized, the simplifier emits a new pseudo-RTL file otherwise identical to the original larger instance pseudo-RTL but without those common hole assignments.

The solver core then proceeds normally, and solves the simplified larger instance without being aware it is solving an incremental sketch. The rest of the SKETCHILOG tool flow proceeds as usual, and the result is just the same as in SKETCHILOG: a complete Verilog file with all signals concretized.

IV. Experimental Results

We demonstrate the advantages of our incremental solver for arithmetic generators on a few different prefix adder design instances. Each of the experiments in Table I below was constructed with a generator and solved "from scratch" (with solution times shown as *Original Time*), then as an incremental sketch. Instances with 23- and 32-bit operand sizes were incrementally solved with the 16-bit operand size instances as the first phase. Instances with 48-bit operand sizes were incrementally solved with the 32-bit operand size instances as the first phase.

`KoggeStone` refers to a classic Kogge-Stone prefix adder structure [5]—it includes a parallel carry computation tree with the maximum number of cells. Each cell's rightmost (i.e., less significant bit position) connection was sketched as a multiplexer choosing between an unknown constant and, if it exists, the previous level's cell at the $i - 2^{level}$th position. Additionally, the final layer, which combines the computed carries with the partial sums, sketched the choice of each result cell (e.g., the choice of which cell connects to each final layer bit position was left undetermined).

`BrentKung` similarly refers to a classic Brent-Kung [3] architecture. Power-of-two position carry cells were sketched in the same way as in the `KoggeStone` examples. Non-power-of-two indexed cells have both connections sketched.

`Hybrid` designs refer to a hybrid between the two architectures. `max1` and `max2` refer to the limit for how many cells could lie between an input bit position and the bit position's output from the carry cell graph. For example, a given bit position might have 4 cells along the path through the `KoggeStone` carry cell graph; `Hybrid_max1` would re-use previous cell outputs (like in `BrentKung` unless that resulted in the bit position requiring more than 5 cells along the path. Similarly, `Hybrid_max2` would re-use cells unless that resulted in the bit position requiring more than 6 cells along the path. If the path constraint would be violated, the `Hybrid` generators instead instantiate a `KoggeStone`-style complete tree for that bit position.

These results show two main points: first, that SketchiLog is in some cases simply not scalable on its own, and second, that our incremental solving strategy can improve scalability by more than 25 times.

All examples use the very-simple corresponding `RippleCarry` architecture as a functional specification; statistics on those specifications are provided for reference. All experiments ran on Linux 3.2 with a Intel Core® i7-3770S and 32GiB of DDR3-1600 RAM.

Experiment	Op. Size	Hole Bits	Original Time (s)	Incremental Time(s)	AIG Depth	AIG Nodes
KoggeStone	16	427	3.732	n/a	12	229
Hybrid_max1	16	509	1.681	n/a	12	217
Hybrid_max2	16	368	0.697	n/a	13	196
BrentKung	16	334	0.842	n/a	16	160
RippleCarry	16	0	n/a	n/a	32	131
KoggeStone	23	713	22.414	5.980	12	379
Hybrid_max1	23	902	6.544	5.588	13	345
Hybrid_max2	23	570	3.434	3.224	15	296
BrentKung	23	522	2.884	2.350	18	237
RippleCarry	23	0	n/a	n/a	46	187
KoggeStone	32	992	93.620	29.245	14	545
Hybrid_max1	32	1754	27.984	25.787	14	529
Hybrid_max2	32	1166	19.772	16.767	15	488
BrentKung	32	720	10.558	7.808	20	333
RippleCarry	32	0	n/a	n/a	64	259
KoggeStone	48	1701	1697.3	67.036	15	936
Hybrid_max1	48	3634	250.52	93.201	15	884
Hybrid_max2	48	2168	75.701	45.893	17	763
BrentKung	48	1167	56.674	25.662	39	479
RippleCarry	48	0	n/a	n/a	96	387

TABLE I: Experimental results detailing instance operand size, original solver runtime, incremental solver runtime, total number of hole bits, and critical path delay (AIG depth) and area (AIG size) for the resulting completed design. The reader's attention should be drawn to the comparison between the original solver runtime, where each sketch is solved with no hints, and the incremental runtime, where certain holes are assumed to remain unchanged as operand size increases (in this case, 23-bit and 32-bit incremental solutions are seeded with values from 16-bit operand size instances, and 48-bit incremental solutions are seeded with values from 32-bit operand size instances).

V. Conclusions

The failure of HLS to develop as originally hoped has left designers "holding the bag" as design complexity continues to increase and performance requirements show no sign of relaxing. Designers have consequently turned increasingly to IP libraries and manually-tuned RTL generators. Unfortunately, while these generators help reduce redundant design effort and are capable of producing very efficient RTL, they can be very difficult to develop.

Yet, generators do not need to be so difficult to develop—if designers can relax the requirement to design perfect generators with perfect precision, they can increase their productivity without needing to raise the level of abstraction. SketchiLog presents a promising new way to achieve this goal, but its scalability problems restrict its applicability. Our work shows that for some classes of generators, there are indeed ways to make these techniques viable.

By exploiting regularity in the generated structures and commonality of solutions to the accompanying problems, we can improve the scalability of these techniques—in the most challenging case, by more than an order of magnitude. While these improvements are not applicable to all designs, they provide hope that further research efforts will bear fruit and sketching will become practical for complex circuits.

References

[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing hardware in a scala embedded language." in *Proceedings of the Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[2] A. Becker, D. Novo, and P. Ienne, "Sketchilog: Sketching combinational circuits," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium 2014 (*To Appear*).

[3] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *Computers, IEEE Transactions on*, vol. C-31, no. 3, pp. 260–264, 1982.

[4] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II - An Open-source Verilog HDL Synthesis tool for CAD Research," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 149–156.

[5] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *Computers, IEEE Transactions on*, vol. C-22, no. 8, pp. 786–793, 1973.

[6] A. Solar Lezama, "Program synthesis by sketching," Ph.D. dissertation, UC Berkeley, Dec 2008. [Online]. Available: http://eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html