Design Synthesis by Sketching

A New Approach to Taming Exploding Design Complexity

Andrew Becker
LAP, I&C, EPFL

***Abstract*—This report surveys three disparate works from the domains of high-level synthesis, testing and verification, and programming languages. We show how the high-level synthesis process works, and why it is insufficient to tame the growing design complexity problem. We survey the basics of the Boolean satisfiability problem and its applications in functional verification, and tie them into recent work investigating partial programming as a design paradigm. Finally, we synthesize the knowledge gained from this corpus into a few promising directions for future research in the hardware design domain.**

***Index Terms*—SAT, QBF, high-level synthesis, digital design, sketching**

I. Introduction and Background

WHILE transistor densities have increased at an exponential rate over the last couple decades, designer productivity has stagnated. Hardware engineers still design in fundamentally the same way as a decade ago. Logic synthesis tools have certainly improved, but designers still write too much register transfer level (RTL) code. This paradigm suited hardware engineers well when VLSI referred to designs with tens of thousands of gates, but has not scaled well to designs with hundreds of millions of gates.

Proposal submitted to committee: September 3rd, 2012; Candidacy exam date: September 10th, 2012; Candidacy exam committee: Martin Odersky, Paolo Ienne, Viktor Kuncak

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(R. Urbanke) (signature)

Over the same period, researchers have focused heavily on high-level synthesis as a means to raise the design abstraction level, and thereby increase logic design productivity and accessibility. High-level synthesis tools take a behavioral description of a circuit, often written in a subset of ANSI C or a C-like language, and produce structural RTL ready for logic synthesis. After decades of work, these tools are finally reaching commercial maturity.

In section II, we examine the design of and techniques used in a representative high-level synthesis tool, and analyze its applicability to general digital design problems.

In section III, we review advances in SAT solving techniques, especially as they relate to model checking, and examine how the recent research focus on problems similar to SAT can expand the power of formal verification techniques.

We then survey how techniques used in formal verification can be applied to program synthesis from incomplete designs in section IV.

Finally, we discuss how the lessons learned from this previous work can be used to alleviate the growing design complexity problem and lay out a potential research direction.

II. Modern High-Level Synthesis Techniques

To understand why modern high-level synthesis is inadequate for digital design, we must examine how these tools operate. Typical of high-level synthesis tools, SPARK [1] accepts a behavioral description in a restricted subset of ANSI C and outputs synthesizable RTL describing the datapath, memories, and finite state machine necessary to control the data flow. We will examine the process SPARK uses to derive its output from the input description, as well as some particular techniques employed by SPARK to achieve better results.

A. Design Flow and Hierarchical Task Graphs

As shown in Fig. 1, the first step in this process is to perform an initial analysis of the behavioral description in a manner nearly identical to the lexing and parsing of a regular high-level language. The ANSI C description is first parsed into an abstract syntax tree in a special form called a *Hierarchical Task Graph* (HTG).

The resulting HTGs are directed acyclic graphs. Each node in an HTG is of one of three types: simple, compound, or loop. A simple node is a node without any hierarchy; it is used to represent a set of operations which may be scheduled concurrently. A compound node is a node with an acyclic hierarchy – it encapsulates a subgraph which is itself a HTG.

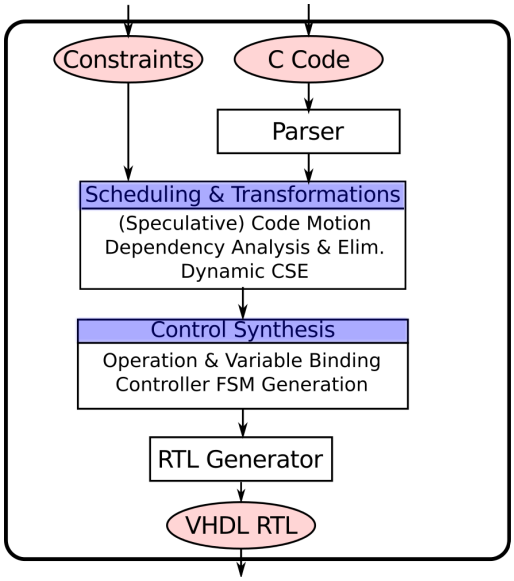

Fig. 1: Overview of the SPARK tool flow, as described by the authors.

For example, an if-then-else expression is a compound HTG node which has a structure detailed in Fig. 2. The compound node gives the hierarchical task graph its name.

A loop node is identical to a compound node, except that it requires exactly one cycle-inducing edge (the back-edge) between the iteration term sub-component and the condition check sub-component. As the name implies, this node exists solely to support the loop constructs of C.

The most important feature of these HTGs is that each node is a strongly connected component with exactly one incoming and exactly one outgoing edge. This regularity is important, because it simplifies the analysis required for code motion optimization techniques, which are explained later in this section.

This HTG, combined with data dependency information, composes the entire SPARK intermediate representation.

Then, SPARK enters the most crucial phase: scheduling. The core problem in scheduling is to assign every operation in the HTG, or another semantically equivalent HTG, to discrete time steps corresponding to individual states in the generated controller, while minimizing the required number of steps through the controller FSM (i.e. schedule length). Assuming resource constraints (the type and number of available functional units, memories, etc.) are given, the key to high performance is to minimize the length of the schedule.

After scheduling, the only phases left are binding and code generation. Binding is the phase in which operations in each scheduled time step are bound to specific functional units in the datapath – for example, if two adders are available and two additions are scheduled in the current time step, which addition is performed on which adder? Similarly, variables referred to in each scheduled operation must be bound to specific memories. Though binding is an essential step in any high-level synthesis flow, it is not discussed in the SPARK paper and the details have little bearing on the overall goals of this work.

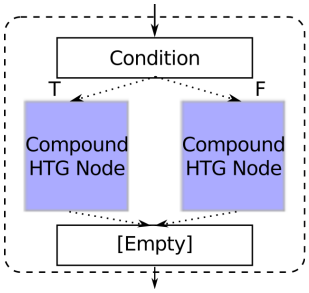

Fig. 2: An ”If-Then-Else” compound HTG node.

Code generation is the rote process of translating the scheduled HTG, complete with functional unit and memory assignments, into RTL describing the datapath and the controller FSM required to drive the datapath according to the schedule. Once again, code generation is essential to the high-level synthesis flow, but is not discussed in the paper.

B. Scheduling and Code Motion

One way to reduce the length of the schedule is to keep the functional units in the datapath as busy as feasible; a schedule which uses two available resources in one time step is better than a schedule which only uses one of two available resources in two time steps. Functional unit underutilization is, prima facie, caused by the inability of some operations to be scheduled at a particular time step.

These scheduling constraints are created by dependencies between operations. However, some dependencies can be eliminated. For example, antidependencies (write-after-read) can be resolved with the use of an extra variable: the write operation is redirected to the new variables, and the original expression causing the antidependency is replaced with a copy from the extra variable. Output dependencies (write-after-write) can be resolved in a similar manner. SPARK allows use of these extra variables with a technique called *dynamic renaming* – if the scheduling heuristic detects that it is beneficial to move an operation, and a dependency constraint preventing its move can be resolved with the use of another variable, the variable is introduced and the operation is free to move.

Flow dependencies (read-after-write) can be resolved by *combining*, an inline expansion of the variable being read with the value previously written to that variable. Obviously, this may cause duplication of operations, but will cause no overhead when the write operation causing the flow dependency is a simple copy operation (for example, when that write operation is caused by dynamic renaming). Fortunately, the authors present another technique to alleviate the extra operations: *dynamic common sub-expression elimination*, which will be discussed in sub-section C.

It is important to note that SPARK avoids some traditional problems with dependency detection and the resulting dependency overapproximations by disallowing the use of pointers.

To utilize functional units more effectively, SPARK can move some operations to different nodes in the HTG, provided

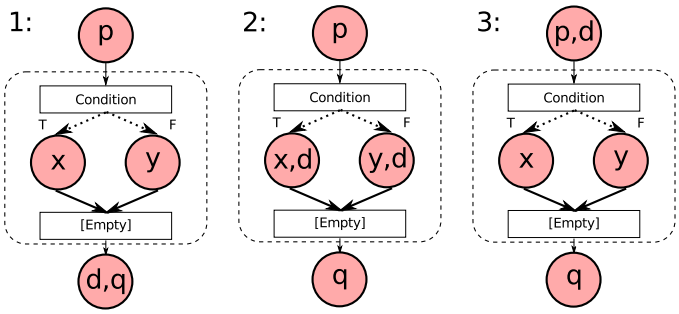

Fig. 3: An example showing how percolation scheduling can iteratively move ('percolate') an operation (d) past independent operations (x, y, p, and q) through the HTG.

that the original HTG and the new HTG are semantically equivalent. The techniques used to perform these *semantics-preserving transformations* are called *code motion* techniques. In short, the goal is to keep functional units as busy as possible by re-ordering, speculatively scheduling (scheduling an operation before evaluating whether it will be used), and (where necessary) duplicating operations in the HTG.

The authors of the SPARK system present two code motion techniques: percolation scheduling and trailblazing.

Percolation scheduling, demonstrated with conditional speculation in Fig. 3, is a set of semantics-preserving transformations which allow operations to be moved to adjacent nodes in the HTG. Repeated application of these incremental transformations allows operations to 'percolate' through the HTG, hence the name. The original author of percolation scheduling [2] defines four basic transformations, of which three are essential.

The "Move-OP" transformation moves an operation from a node n to a predecessor node m, so long as there is no data dependency between any operation in m and the operation being moved. If there is a path in the HTG through node n which does *not* pass through node m (i.e. node n has more than one predecessor), node n is first duplicated to n'. All other (i.e. not from m) predecessors of the original node n are redirected to pass through n', while n's sole predecessor becomes m (where the operation was moved).

The "Unification" transformation moves two separate but identical operations from m's successor nodes n_1 and n_2, provided that no data dependency exists between the operations in n_1 and n_2 and m, and unifies them into one operation in m. Once again, if there is a path in the HTG through either n_1 or n_2 which does not pass through m, the node(s) with other predecessors must be duplicated and the control flow edges redirected through the duplicate node along that path which does not pass through m.

Finally, the "Deletion" transformation allows a node n with no operations (i.e. because they've all been moved with the other transformations) to be elided, and each predecessor node m_i's outgoing edge (m_i, n) to be replaced with the edge $(m_i, successor(n))$.

Trailblazing is very similar to percolation scheduling, but exploits the hierarchy of the HTG to avoid the expensive incremental updates required for the former. For example, in Fig. 3, operation d may be moved across the compound if-then-else node represented by the dashed box without visiting every sub-node within the if-then-else. It's important to note that this is only possible because each HTG node is a strongly connected component with exactly one incoming and outgoing edge – the whole set of operations contained within such a compound block do not need to be analyzed, since we know control flow must first pass from that node's successor, and the compound node conveniently has all the dependency information of the contained sub-nodes.

It is often advantageous to speculatively schedule an operation before it is known whether the operation's result is useful. If scheduling an operation only performed in the 'then' branch of a conditional would require an extra time step, it makes sense to speculatively schedule that operation in the basic block before the if-then-else if doing so doesn't extend the length of the schedule.

Similarly, it can be useful to speculatively schedule an operation in the basic block after the if-then-else in *both* branches of the conditional (i.e. duplicate the operation), if doing so reduces the length of the schedule. Phase one in Fig. 3 demonstrates this *conditional speculation*.

Finally, while it is often the goal to expose as much parallelism as possible as early as possible, it can also be helpful to move operations to successor nodes. For example, if scheduling an operation in the basic block before an if-then-else would require an extra time step, scheduling the operation in *both* branches of the conditional would reduce overall schedule length if functional units aren't fully utilized in either branch. SPARK allows this 'reverse speculation' so long as dependency constraints are not violated.

C. Dynamic CSE

The other key to reducing schedule length is to avoid redundant computations wherever possible. Common sub-expression elimination (CSE) is a mature, well-known technique to trade memory space for computation time. The idea is simple: a sub-expression common to two or more statements can be evaluated once and then stored. The following statements then use the stored value instead of re-evaluating the sub-expression.

This is undoubtedly an effective strategy, but the code motion techniques described above, especially speculation which results in duplicate operations, can expose additional opportunities for common sub-expression elimination. The traditional technique of discovering common sub-expressions prior to scheduling will miss these additional opportunities. Discovering common sub-expressions *after* scheduling makes no sense at all, since the schedule has already been determined.

The solution the authors presented was to discover these common sub-expressions *dynamically* during scheduling, rather than making a static pass before scheduling begins. This is performed very simply: any time a code motion technique which may provide an opportunity for common sub-expression elimination is applied, check all other 'ready to schedule'

nodes for sub-expressions in common with the current node. Any common sub-expression found in any of those 'ready to schedule' nodes is then replaced with a reference to the sub-expression's computed value.

D. Discussion

This paper offers an overview of the typical high-level synthesis process and focuses heavily on the transformations used to produce higher-quality results. It also serves as an example of how *not* to approach the design complexity problem.

Starting with a purely behavioral description is not necessarily suitable to hardware. When a designer uses an operator like, say, a multiplier, he or she is by design not responsible for describing *how* to implement that operator. The high-level synthesis tool then functions as a glorified library. It may have ready-made designs for a 32x32- and 64x64-bit multiplier, but what if the designer knows each operand will only occupy 23 bits?

Similarly, the rigid basic strategy of building a full processor (FSM and datapath) from the input description is not always appropriate, especially for streaming applications. Hand-designed circuits are usually more natural, less congested, and more efficient.

Additionally, high-level synthesis suffers from a 'black box' problem. That is, a designer provides absolutely no detail about the structure of the circuit he or she wishes to implement, and is provided with a purely structural description of the circuit which can be exceedingly difficult to read manually. If the tool's output does not meet certain constraints, it can be very difficult to determine *why*, and more importantly, how to fix the problem in the behavioral description. The design process is reduced to trial and error, or the designer gives up and designs the circuit by hand anyway.

With that said, high-level synthesis has substantial merit. Because design errors could cost tens of millions of dollars and months of re-spin time, 'first time success' is crucial. This means testing at all design levels is essential. One key benefit of high-level synthesis is that if the tool is trusted, functional verification can be completed on the much simpler behavioral description.

Finally, note that the term, "high-level synthesis" is a bit of a misnomer: it should be called, "high-level transformation to RTL," or, "compilation to hardware". There is not much synthesis involved. The high-level synthesis process is mostly *transformational*. Some synthesis is involved, e.g. to implement operators, but this is more of a library lookup than the creation of a circuit from building blocks provided by the designer.

III. Recent Advances in SAT Solvers

This paper is a rather long survey of a great deal of work, some of which is not relevant to the proposed research direction, and so only the most important and relevant parts are reviewed here

A. Boolean Satisfiability

Over the past couple decades, tools for solving the Boolean satisfiability problem have witnessed tremendous improvements in performance and scalability. This has led directly to the increasing use of SAT in a number of domains, especially formal verification of both software and hardware systems. But to understand why this is, we must first define the problem itself.

Given its power, the Boolean satisfiability problem (SAT) is remarkably simple to state. The problem asks whether there is an assignment to the variable x for which some Boolean formula φ over x evaluates to `true`:

$$\exists x \ \varphi(x)$$

In other words, the Boolean SAT problem is to find a *satisfying* assignment to the variable x such that the Boolean formula φ evaluates to `true`, or to report that no such assignment exists (and, optionally, provide a *refutation proof* demonstrating so).

The SAT problem is proven to be NP-complete in the general case [3], but various conflict-based learning, conflict-driven backtracking, and efficient boolean constraint propagation (implication generation) techniques have proven quite effective in practice. As a result of this research, the Boolean satisfiability problem has morphed from the canonical example of an intractable problem to an attractive encoding for many seemingly-unrelated problems.

The CAD community has employed the recent developments in SAT solver technology to improve the typical logic synthesis and verification flow. ABC [4] is a perfect example of such a tool; it employs a bundled SAT solver ([5]) for Boolean reasoning during synthesis (e.g. for SAT sweeping) and verification (e.g. combinational equivalence checking and bounded model checking). First, we examine the basic SAT algorithm and the advances which have catapulted SAT to the forefront of the verification world.

B. DPLL SAT Algorithm and Heuristic Optimizations

The basic SAT algorithm, "Davis-Putnam-Logemann-Loveland" (DPLL) [6], [7], is paraphrased here from [3] for reference:

```
Solve()
   if preprocess() = CONFLICT then
      return UNSAT
   unbounded loop do
      if not decide-next-branch() then
         return SAT
      while deduce() = CONFLICT do
         level := analyze-conflict()
         if level = 0 then
            return UNSAT
         do-backtrack(level)
      done
   done
```

It is crucial to note here that the above algorithm is both sound and complete: a solution is reported if and only if a solution exists. In other words, if a solution exists which

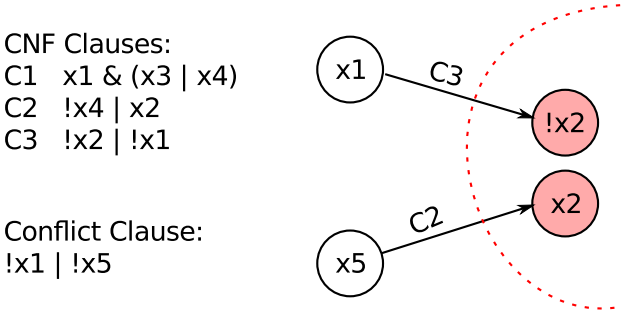

Fig. 4: An example set of clauses and an implication graph representing a possible state of the SAT solver while running. Note that the nodes x_2 and $!x_2$ both appear in the implication graph, representing a conflict.

satisfies the Boolean formula, it will be (eventually) found, and if no solution exists, that will (eventually) be reported. Therefore, all techniques presented here exist solely to reduce solution time, not to allow the SAT solver to solve more problems or improve its accuracy.

The function `preprocess()` is responsible for eliminating some inefficiencies in the SAT problem encoding, like equivalent literals, and it may discover trivial facts from which it can deduce the problem is unsatisfiable.

The core of the algorithm lies within the unbounded loop. First, the function `decide-next-branch()` chooses a free variable and a value to assign it. If there are no free variables, the current assignment to all variables is a valid solution.

Then, all variable assignments which can be deduced from the new assignment are made in `deduce()`, which also detects conflicting implications. This process is known as *Boolean constraint propagation*, as it propagates constraints on one variable in one clause to other variables in other clauses. If there are no conflicting implications, the program advances to the top of the unbounded loop, and chooses another free variable to assign a value to. If there are conflicting implications, then the current values assigned to non-free variables will not lead to a solution.

The original DPLL algorithm simply "backtracked" chronologically; that is, it freed the most recently assigned variable and tried the other value. Modern SAT solvers perform a detailed analysis of the *implication graph*, which represents the entire internal state of the SAT solver, to determine the cause of the conflicting implications. The above algorithm presents this analysis as the function `analyze-conflict()`. In the original DPLL algorithm, the call to `analyze-conflict()` would instead be a reference to the most recent decision variable.

Before turning to *how* the analysis of the implication graph is performed, or what is done with the analysis to improve performance, we will first examine the structure of the implication graph itself.

An implication graph is a simple structure where nodes represent non-free variables and edges represent the clauses causing implication. When each non-free variable appears at most once in an implication graph, no conflict exists. A conflict only exists when separate clauses imply two different assignments to the same variable; in other words, when both nodes x_i and x_i' appear in the implication graph.

An demonstrative set of clauses and an example implication graph are shown in Fig. 4.

C. Conflict Analysis

Conflict analysis is performed by identifying the conflicting nodes and creating a cut which includes both conflicting nodes and excludes any decision nodes (i.e. nodes representing values chosen by the solver). After this, the nodes which have at least one edge crossing the cut are recorded. Since the current assignment to variables (either decided by the solver or implied by those decided by the solver) creates a conflict, we can be sure that at least one of these nodes must have an incorrect assignment, or the clauses are unsatisfiable.

This information can be used in at least two ways: for *conflict-based learning* or *conflict-driven backtracking*.

Conflict-based learning simply entails constructing a new clause asserting the disjunction of the negation of each recorded node and adding it to the set of clauses considered. This improves performance on future iterations because any assignment to variables which produces the same assignment to these marked variables (i.e. which instantiates these nodes in the implication graph) will be detected and rejected more quickly. Instead of detecting the constraint violation after following many implications, the violation will be detected as soon as the implication of the conflict clause is determined.

On the other hand, conflict-driven backtracking is a complementary technique which tries to determine the most likely decision variable which caused the conflict. Though it is not mentioned directly in the paper, one possible heuristic is to backtrack to the nearest decision variable that has freedom in variable assignment, where 'nearest' means the fewest number of implications between that node and the decision variable node.

D. SAT for Verification

Verification of combinatorial circuits is easily encoded as a SAT problem. While there are a number of mechanisms for formal verification of a Boolean network, the most common and scalable is simply equivalence checking. In combinatorial equivalence checking, the primary outputs of both the unit under test (UUT) and a model of the specification for that unit's behavior are compared under all possible assignments to their primary inputs. If the primary outputs of both units (the UUT and the specification model) are equivalent for all possible inputs, then the UUT is said to be *functionally equivalent* to the specification.

To this end, both the UUT and the specification of that unit are combined into a single circuit, called a miter [8]. This miter, visualized in Fig. 5, has the same primary inputs as the UUT and the specification model, and only one primary output. Each of the miter's primary inputs has a fan-out of exactly two: each is connected to the primary inputs of both the UUT and the specification model. Thus, the UUT

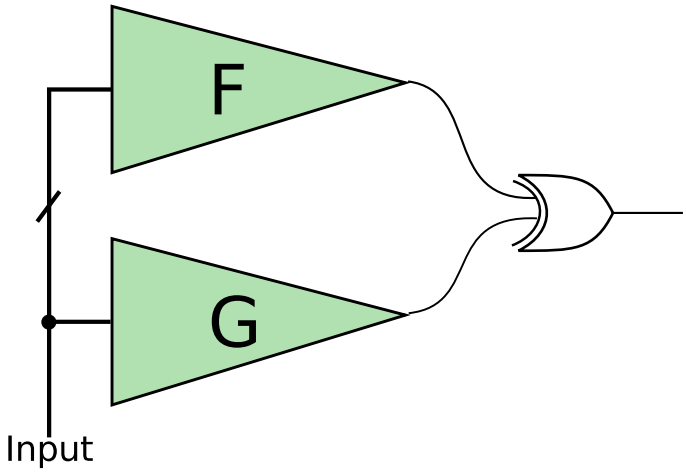

Fig. 5: An abstract visualization of a miter circuit. Boolean functions F and G are equivalent when the output of the XOR gate is stuck at 0 under all input vectors.

and specification model receive identical assignments to their primary inputs.

The corresponding primary outputs of the UUT and the specification model are compared (XOR-ed), and the results of these comparisons are OR-ed together. This OR gate is the primary output of the miter. Thus, if the primary outputs of the UUT differ from the specification model's under the same input assignment, the primary output of the miter will be assigned `true`.

After the miter circuit is constructed, it is usually encoded into conjunctive normal form and passed to a SAT solver, though some SAT solvers work directly on the Boolean network. As the name implies, the SAT solver tries to find a satisfying assignment to the miter's primary inputs – an assignment to the primary inputs of the miter that results in a primary output value of `true`. If such an assignment is found, it is reported, and the miter is said to be *satisfiable*. Otherwise, the miter is said to be *unsatisfiable*. A satisfiable miter represents a failing equivalence check: the primary outputs of the unit under test and the specification model differ for at least one input value. Conversely, an unsatisfiable miter represents a succeeding equivalence check; the unit under test and the specification are functionally equivalent.

E. Bounded Model Checking

Verification of sequential systems is a different story. The most common technique for verifying sequential systems, bounded model checking, is actually not a verification method but a 'buf-checking' method.

Bounded model checking can be viewed as a partial unrolling of the sequential logic. In essence, the model is checked for equivalence (or violation of a safety property) at a series of discrete time steps corresponding to the clocking of the sequential elements in the design. Just how many time steps are considered is the 'bound' in bounded model checking. Since all logic between registers must be duplicated for each time step, BMC suffers from a serious explosion in problem size as the bound grows to infinity.

There are some techniques to minimize the amount of unnecessary logic included in the model at each time step, but they are beyond the scope of this review.

F. Quantified Boolean Formulae

As SAT solvers have improved, academic interest in other related problems has grown. One such related problem is the quantified Boolean formula (QBF) satisfiability problem, which allows an arbitrary number of quantifiers (either $\exists$ or $\forall$) over the variables. Thus, it is a generalization of SAT which evaluates more than just propositional formulae.

A simple example from [3] is:

$$\exists y[\forall x[x \leftrightarrow y]]$$

Although the statement above is trivially found to be false, the QBF satisfiability problem (QBF-SAT) is a much more difficult problem: it is PSPACE-complete rather than NP-complete [3].

However, the increased difficulty comes with increased power. Symbolic reachability is also known to be PSPACE-complete, and so there exists a polynomial-complexity reduction from one to the other. This means the QBF-SAT problem is essentially equivalent to sequential property checking.

IV. Combinatorial Sketching for Finite Programs

One such domain is presented here [9], where the authors use a restricted form of QBF to search for possible completions of a partial program which are equivalent to a given specification. The observation that forms the motivation for this work is that for many algorithms, a high-performance implementation is much more time-consuming to create than a behavioral description of the functionality. Often the difficulty with hand-optimizing algorithms is the abundance of corner cases, magic numbers, lookup tables, and other rote details. The path from abstract optimization idea to a well-tuned implementation is usually long and tedious.

The authors propose a new paradigm for developing optimized programs: partial programming. In the SKETCH [9] framework, programmers provide most of a program's description and a completely un-optimized implementation which is nevertheless functionally equivalent. Crucially, programmers leave some variables (almost) unconstrained and let the automated tool discover an assignment to the free variables that satisfies the equivalence property over all possible inputs. Technically the free variables aren't truly free, since they *are* constrained to be less than some arbitrary number of bits in size.

The idea is this freedom allows programmers to focus on the "big picture" and leave the nasty details to the machine. The authors present impressive results which demonstrate how the SKETCH tool synthesizes a high-performance implementation of the AES round function from an intuitive C-like description on the order of only a dozen lines.

A. Implementation Details

To synthesize a correct implementation given only a partial program and a functional specification, the partial program and

specification are first lexically analyzed and parsed, just as in any high-level language. Free variables are called "holes", and are denoted by the programmer as **??**.

An example sketch and its associated functional specification is reproduced from the paper for clarity:

```
bit[W] pop (bit[W] x) {
   int count = 0;
   for (int i = 0; i < W; i++) {
      if(x[i]) count++
   }
   return count;
}

bit[W] popSketch(bit[W] x) implements pop{
   loop(??) {
      x = (x & ??) + ((x >> ??) & ??);
   }
   return x;
```

The partial program and functional specification are then translated into a Boolean network, with both program inputs and holes exposed as primary inputs, and program outputs exposed as primary outputs. Immediately, we hit upon a few key details:

- The translation process from description to Boolean network is similar to high-level synthesis, but does not consider any resource constraints and only generates combinatorial elements. Sequential processes must be fully unrolled before implementation, similar to a bounded model check unrolling sequential logic; this is why the tool *only* works for finite programs.
- The Boolean networks created for the partial program and the specification have different signatures. With m bits of program inputs, n bits of program outputs, and a total of k bits of hole variables, the partial program has the signature $\{0,1\}^{(m+k)} \rightarrow \{0,1\}^n$, while the specification has the signature $\{0,1\}^m \rightarrow \{0,1\}^n$.

We will ignore the restriction to finite programs for now; this will be discussed later.

The Boolean network derived from the partially-evaluated sketch has the form $P(x,c)$, where x is the input vector and c is the control vector (that is, the value assigned to the holes). The specification has the form $S(x)$.

To find a set of hole values which satisfy the equivalence relation, the tool asks its built-in QBF-SAT solver to evaluate:

$$\exists c \forall x P(x,c) \leftrightarrow S(x)$$

This is a restricted form of the more general QBF that allows a special strategy, which will be discussed shortly.

If the QBF-SAT solver reports that such a c does not exist, the tool informs the user that their sketch was buggy; no value assigned to the holes (within the constraints) satisfies the equivalence relation. In other words, the sketch and the specification cannot be made to be equivalent with any synthesized control vector.

On the other hand, if the solver reports a c which satisfies the above relation, the tool substitutes the solution bits into the AST nodes that represented each hole before the partial evaluation, and generates a complete program from the now-complete AST.

B. Counterexample-Driven Solver

This particular type of QBF is a member of a restricted class of quantified Boolean formulae known as 2QBF.

The authors use this fact to their advantage; their solver, which will not solve any general quantified Boolean satisfiability problem, consists of two SAT solvers working in tandem. One of the main motivations for doing this is that SAT solvers are much more mature than QBF-SAT solvers, and so they can leverage decades of previous work with very little effort.

The solver the authors present is more clever than a brute-force implementation. In the spirit of conflict-based learning, they separate the two SAT solving procedures into the *synthesizer* and the *verifier*. The synthesizer is responsible for (non-deterministically) choosing a vector x, and finding a control vector c which satisfies the 2QBF problem *for that input* x. If there is no feasible c, the problem is unsatisfiable and the sketch and the specification obviously cannot be equivalent with any hole values. Otherwise, this vector c is passed to the verifier, which will attempt to verify that c induces an unsatisfiable miter (i.e., the sketch and the specification are equivalent on all inputs). If the verifier returns UNSAT, then the solver is finished and the control vector c is reported. If not, the sythesizer takes the counterexample (the satisfying assignment to x) provided by the verifier and adds it to the set of input vectors to consider when looking for a new c, and the procedure is repeated.

C. Discussion

The simplified synthesis is effective for the purposes of this work, but unrolling and inlining creates an unfortunate explosion in the size of the resulting Boolean network, just as it does in bounded model checking. Furthermore, the benchmarks provided in the paper show serious scalability problems – many finite programs are not even near feasible to sketch. For example, when sketching the `pop` operator on four input bits, with 77 hole bits, total solution time is only 80 miliseconds. But when sketching the operator on sixteen input bits, with 173 hole bits, total solution time explodes to over 1000 seconds.

However, even with these severe limitations, it is easy to see how such a tool could be extremely useful as a "programming helper".

V. Lessons Learned and Research Proposal

While high-level synthesis tools have come a long way, they are not the panacea they were widely predicted to be. These tools are generally successful at raising the level of abstraction, but the RTL they produce does not usually describe particularly efficient circuits (in terms of delay, area, or power). Even worse, while behavioral specification is ideal for some circuits, design components with special area, timing, or power constraints usually require a more direct structural description in RTL. These components are some of the worst

parts of a circuit to design – the parts for which we'd most like avoid using RTL.

Part of the problem is that generating hardware and software is too different. One of the key differences between compiling high-level languages to assembly and synthesizing hardware from high-level procedural languages is that high-level languages were designed to ease the construction of known constructs (loops, function calls, etc.) from relatively few building blocks with a relatively high-level interface (the ISA). Obviously high-level languages have come a long way, and now provide useful type systems and generic patterns, but they still use the same high-level interface. Hardware designers, on the other hand, have a much higher degree of freedom, and high-level synthesis gives generally poor results because the abstract constructs designers write are too far away from the implementation.

High-level synthesis tools are also at a disadvantage compared with the first commercial compilers. Compiled software could afford to take a performance penalty not only because essential kernels could still be hand-written in assembly, but because software is generally malleable. Excepting mission-critical code, if performance needed improvement, a software engineering team made their adjustments and recompile, then redistribute their new code as an update. In an era of razor-thin profit margins and design re-spins that cost millions of dollars up front and untold millions more in time-to-market elongation, hardware designers do not have these luxuries.

In short, while high-level synthesis tools are valuable, designers need a new design paradigm and accompanying tools to alleviate the tedious and error-prone RTL design of these circuits. An alternative approach deserving of investigation is to allow designers to provide partial hardware designs and a functional specification; to do for hardware what Solar-Lezama et al. did for software.

The requirement for a specification is not at all an impediment to the adoption of such a technique – these specifications are *already* required in the typical design flow. As an added benefit, functional verification comes for free.

Any code generated by the design synthesis tool is guaranteed to be functionally equivalent to the provided specification. Scalability problems will surely pop up, but this is an opportunity to improve the state of the art for QBF solvers.

Another complementary research avenue is to explore the use of functional languages as a supplement to traditional RTL. It would be even better to combine a functional language with metaprogramming features (like Scala) with sketching to create an even more powerful design language.

At first this work would focus exclusively on combinatorial circuit sketches, but extending the work to sequential circuits is another potentially fruitful research direction.

REFERENCES

[1] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *VLSI Design, 2003. Proceedings. 16th International Conference on*, jan. 2003, pp. 461 – 466.

[2] A. Nicolau, "Percolation scheduling: A parallel compilation technique," Ithaca, NY, USA, Tech. Rep., 1985.

[3] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *STTT*, vol. 7, no. 2, pp. 156–173, 2005.

[4] B. L. Synthesis and V. Group. (2005, December) ABC: A system for sequential synthesis and verification. [Online]. Available: http://www-cad.eecs.berkeley.edu/ alanmi/abc

[5] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518. [Online]. Available: http://dblp.uni-trier.de/db/conf/sat/sat2003.html#EenS03

[6] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[7] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962. [Online]. Available: http://doi.acm.org/10.1145/368273.368557

[8] D. Brand, "Verification of large synthesized designs," in *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '93. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 534–537. [Online]. Available: http://dl.acm.org/citation.cfm?id=259794.259883

[9] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 404–415.