Satisfiability-Based Methods for Digital Circuit Design, Debug, and Optimization

THÈSE N^{O} 8850 (2018)

PRÉSENTÉE LE 28 SEPTEMBRE 2018

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DES PROCESSEURS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Andrew James BECKER

acceptée sur proposition du jury:

Prof. V. Kuncak, président du jury
Prof. P. Ienne, directeur de thèse
Prof. Ph. Brisk, rapporteur
Prof. W. Hu, rapporteur
Dr B. Jobstmann, rapporteuse

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2018

To my family, whose guidance, support, and encouragement
are the earth from which all my achievements grow.

Résumé

Il est notoirement difficile de bien concevoir des circuits numériques. Cette difficulté découle en partie des nombreux degrés de liberté inhérents à la conception de circuits, et est généralement associée à la nécessité de satisfaire diverses contraintes.
Dans cette thèse, nous démontrons comment les formulations de problèmes de satisfaction peuvent être utilisées pour compléter une conception, ou pour trouver une architecture spécifique qui satisfait à certaines contraintes ; comment celles-ci peuvent être utilisées pour créer, déboguer et optimiser des conceptions ; et introduire un langage spécifique au domaine, bien adapté à la conception, au débogage et à l'optimisation assistées par la satisfaction.
Dans la première application, nous montrons comment des incertitudes explicites appelées "holes" peuvent à la fois être utilisées naturellement et favoriser la création de problèmes de satisfaction formels utiles à la conception de circuits. Nous développons également un langage DSL approprié pour rendre la conception avec des holes facile et efficace.
Nous montrons ensuite comment, en utilisant le même type de formulation de satisfaction, nous pouvons automatiquement instrumenter une conception buggy donnée pour remplacer les fragments de syntaxe suspects par des alternatives potentiellement correctes. Le solveur de satisfaction détermine alors s'il existe un ensemble possible de fragments alternatifs qui corrigent le bogue. Nous démontrons également que cette approche est raisonnablement évolutive, en partie parce qu'il y a moins besoin d'une spécification entièrement précise dans la formulation du problème de satisfaction.
Nous avançons ensuite au-delà du "hole-filling" et montrons comment une intégration étroite de l'élaboration du design avec des solveurs de satisfaction permet des approches totalement nouvelles. Nous utilisons cette intégration étroite pour créer les premières méthodes connues d'optimisation des circuits du modèle GLIFT (Gate-Level Information Flow Tracking) et pour faire des compromis de principe dans leur précision.
Enfin, en intégrant tous les travaux précédents, nous proposons un DSL plus puissant, spécifiquement conçu pour combler les lacunes du premier langage de "hole-filling". Ce langage, que nous appelons Nasadiya, permet des intégrations de satisfaction plus générales dans la conception et l'optimisation des circuits, et fournit une fonctionnalité de modélisation intégrée utile pour optimiser les propriétés extra-fonctionnelles comme le retard de chemin critique. Nous démontrons l'utilité de ces fonctions en implémentant un optimiseur automatique de puissance pour un type populaire d'additionneurs de préfixes parallèles.

Mots clefs : Satisfiabilité, SAT, QBF, débogage, optimisation, propriétés extra-fonctionnelles.

Abstract

Designing digital circuits well is notoriously difficult. This difficulty stems in part from the very many degrees of freedom inherent in circuit design, typically coupled with the need to satisfy various constraints. In this thesis, we demonstrate how formulations of satisfiability problems can be used automatically to complete a design, or to find a specific design architecture that satisfies certain constraints; how these can be used to create, debug, and optimize designs; and introduce a domain-specific language particularly well-suited for satisfiability-assisted design, debug, and optimization.

In the first application, we show how explicit uncertainties called "holes" can both be natural to use and conducive to the creation of formal satisfiability problems useful for designing circuits. We further develop a Scala-hosted Domain Specific Language (DSL) with appropriate syntactic sugar to make design with holes easy and effective.

We then show how, utilizing the same kind of satisfiability formulation, we can automatically instrument a given buggy design to replace suspicious syntax fragments with potentially-correct alternatives. The satisfiability solver then determines if there is any possible set of alternative fragments which fix the bug. We also demonstrate that this approach is reasonably scalable, in part because there is less need for a fully-precise specification in the formulation of the satisfiability problem.

We then advance beyond mere hole-filling and show how a tight integration of design elaboration with satisfiability solvers allows totally new approaches. To point, we use this tight integration to create the first known methods to optimize Gate-Level Information Flow Tracking (GLIFT) model circuits and to make principled trade-offs in their precision.

Finally, integrating all the previous work, we propose a more powerful DSL specifically designed to address the shortcomings of the first "hole-filling" language. This language, which we call Nasadiya, affords more general integrations of satisfiability into circuit design and optimization, and provides built-in modeling functionality useful for optimizing extra-functional properties like critical path delay and circuit area. We demonstrate the utility of these features by implementing an automatic power optimizer for a popular type of parallel prefix adders.

Key words: Satisfiability, SAT, QBF, Debug, Optimization, Extra-Functional Properties.

Contents

List of Figures

List of Tables

1 Introduction

In the span of only a few decades, computers and digital communications have exploded in both utility and complexity, and are now ubiquitous in nearly every aspect of the modern world. All these computers and digital communications systems are made possible by automated processes to etch transistors, wires, and other physical electronic components onto small pieces of silicon called "chips". As demand for these chips has grown, engineers have succeeded in making progressively smaller and more efficient features on those chips, driving further demand growth. Gordon Moore famously observed [Moore, 1965] that the apparent result is an approximate doubling of integrated circuit complexity approximately every eighteen months.

Although this "law" has been losing steam and appears to be nearing its end [Mack, 2011], it has held true for decades, and the results are truly fantastic: For a few hundred US dollars (or less), nearly any person can buy a smart phone—a hand-held computer—able to do far more computation far more quickly than is needed to control a lunar spacecraft. [Hall, 1996] This power has been unlocked by extreme miniaturization: Modern chips often boast *billions* of transistors.

Billions of transistors are great for software designers. Those transistors are used to make more cunning, faster implementations of the devices that execute software programs. Software programmers effectively have been getting a free lunch: Their programs have been getting faster without any need to modify their program code.

Unfortunately, those billions of transistors are a practical nightmare for hardware designers. In contrast to software design, where convenient abstractions of the executing machine make software cost-free to expand, update, and run faster on better computers, the relative paucity of convenient and scalable abstractions for hardware design makes every modification a major event. In the end, hardware design is entirely concerned with the placement, layout, routing, and timing of and between billions of actual physical components. Further, fabrication of a silicon chip is a time-consuming and expensive process [Maly, 1994], although the marginal cost of additional chip production is generally minimal. Combined, these factors mean that

increasing design complexity presents a serious financial and computational challenge to physical implementation. Making bigger circuits is not as simple as making bigger programs, and if a silicon chip has an error and needs to be updated, it can spell doom for the company's product.

Traditional approaches to managing this complexity revolve around raising the design's level of abstraction from the *Register Transfer Level* (RTL), where every logical component, memory element, and their connections is specified manually. So-called *High Level Synthesis* (HLS) approaches allow designers to write limited software programs and have them automatically transformed into RTL. While this has achieved some success, the best results are almost always obtained by hand; RTL design is here to stay.

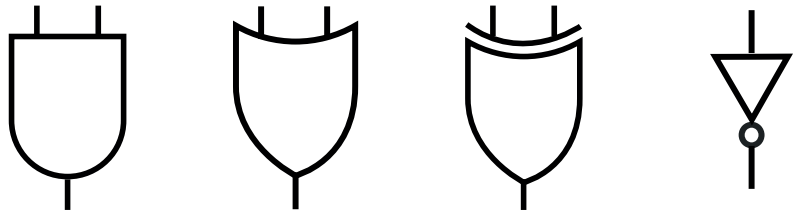
In the face of multi-million dollar costs for design errors, increasing design complexity also makes debugging, or the process of reasoning about circuits to find and correct their errors, increasingly difficult and critical to project success [Foster, 2015]. Hardware designers are also becoming increasingly aware that their design errors can manifest in fiendishly subtle features of even functionally-correct implementations, and that so-called "side channels" can undermine the security of their designs [Lipp et al., 2018, Kocher et al., 2018, Becker et al., 2013]. For example, designs that require protection of a secret cryptographic key must be painstakingly designed and analyzed to ensure that non-functional properties of the design that may be visible to potential attackers (e.g., the power consumed by the chip during cryptographic operations, or the time spent computing those cryptographic operations) cannot reveal information that could be used to reconstruct that key material—even if the cryptographic operations are functionally correct and do not directly leak information about the key. Millions of dollars of corporate revenue relies on such side channel resistance [Markantonakis et al., 2009].

Further, because of the immutable and physically-constrained nature of fabricated circuit designs, hardware designers always want to find better, faster, smaller circuits that meet their requirements: Not only could better performance or lower power consumption differentiate their product from competitors', but a clever idea to reduce a circuit's area can directly increase profit margins by allowing more chips to be fabricated on a single wafer.

Satisfiability is a technique that can be used to automatically reason about surprisingly complex logical formulae—for example, formulae that define the behavior of some digital circuit. This thesis presents novel techniques based on satisfiability to help designers deal with each of these problems posed by increasing design complexity by enabling new forms of automated reasoning about those designs and their components. As many of these problems manifest in the level of the language used to design the circuits, we believe that applications and integrations of satisfiability into the language level is the right place to provide automated reasoning. However, before describing in detail how new integrations of satisfiability fit neatly into solving these problems, it is essential for the reader to understand some background knowledge.

a	b	z
0	0	1
0	1	1
1	0	1
1	1	0

Figure 1.1 – Truth table for the universal Boolean function `NAND`.



(a) An `AND` gate. (b) An `OR` gate. (c) An `XOR` gate. (d) A `NOT` gate.

Figure 1.2 – Common Boolean logic gates. All gates except `NOT` have two inputs and one output; `NOT` has only one input.

1.1 Boolean Logic

While ubiquitous computing is only decades old, the foundations of electronic computation were laid nearly a century before the first stored-program electronic computer [Copeland, 2017]. George Boole first formalized the system of logic that bears his name in the early 19th century in his book "The Mathematical Analysis of Logic" and continued its development over the course of the following few years [Boole, 1847, 1854]. The logic system Boole developed, known as Boolean logic, underlies the implementation of every silicon chip of every computer that might be found in everything from so-called supercomputers to toaster ovens.

In Boolean algebra, values are either 1 or 0, also sometimes called `true` and `false`. Boolean functions, just like functions in conventional continuous real-valued algebra taught in grade school, compute a value by some operations on some input value(s). Unlike conventional algebra, those values are either `true` or `false`, and these functions can be completely specified in a *truth table* listing the truth value of the function for every possible input value.

For example, Fig. 1.1 shows the truth table for the so-called universal Boolean function `NAND`, which gets its name from the possibility of implementing any Boolean function using only the `NAND` function [Sheffer, 1913].

Boolean logic is not only the underpinning for the formalization of digital computation [Shannon, 1938], but also is very much at the core of the *implementation* of the digital circuits that power all modern computers. All silicon computer chips use etched transistors to implement so-called Boolean logic gates, or simple fixed-size Boolean logic functions, to realize the logical

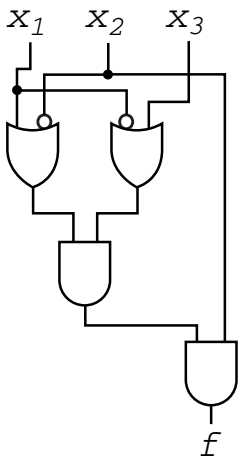


Figure 1.3 – An example schematic of Boolean logic gates for the function $f(x_1, x_2, x_3) = (x_1 + \neg x_2) \cdot (\neg x_1 + x_3) \cdot x_2$.

functionality the designers desire.

Fig. 1.2 shows some of the most common logic gates: from left to right, `AND`, `OR`, `XOR`, and `NOT`, which implement logical conjunction, disjunction, exclusive disjunction, and inversion, respectively. In plain terms, an `AND` gate computes 1 when both of its inputs are 1; an `OR` gate computes 1 when either or both of its inputs are 1; an `XOR` gate computes 1 when exactly one of its inputs is 1; and a `NOT` gate computes 1 when its single input is 0.

While logic gates are extensively used in this thesis in circuit schematics and design, Boolean logic functions are sometimes more naturally written with a more mathematical notation, where $a \cdot b$ represents the conjunction of a and b (`AND`), $a + b$ represents the disjunction of a and b (`OR`), $a \oplus b$ represents the exclusive disjunction of a and b (`XOR`), and $\neg a$, represents inversion (`NOT`).

For example, the logic function $f(x_1, x_2, x_3) = (x_1 + \neg x_2) \cdot (\neg x_1 + x_3) \cdot x_2$ describes the same function as the logic gate diagram in Fig. 1.3.

Unfortunately, the logic functions that fully describe most modern circuit designs are vastly more complicated than the one in Fig. 1.3. Modern circuit designs often have thousands of input variables and tens or hundreds of millions of logic gates. In the face of this kind of complexity, designers need help to reason about the behavior of their designs.

1.2 Boolean Satisfiability

The Boolean satisfiability (SAT) problem is a well-known problem in computer science famous for being the first problem proven [Cook, 1971] to lie in a class of problems that are (widely believed to be) inherently difficult to solve, called NP-complete problems. It has also emerged in the past two decades as the premier encoding for many problems requiring reasoning about

Boolean logic.

Formally stated, Boolean SAT solvers determine if it is possible for a propositional formula $\mathscr{P}$ to be true, as in the formula:

$$\exists \vec{x} : \mathscr{P}(\vec{x}). \tag{1.1}$$

In other words, given a logic function $\mathscr{P}$ described as a set of clauses of Boolean logic statements, the problem is to determine if there is some input value that *satisfies* the function (i.e., when provided this input, the function $\mathscr{P}$ computed 1). Typically, once a formula has been determined to be satisfiable, the *witness*, or the specific input value that satisfies $\mathscr{P}$, is trivial to report in addition to the confirmation of satisfiability.

For example, consider the formula for $\mathscr{P}$ below.

$$\mathscr{P} = \prod_i P_i \tag{1.2}$$

If each P_i contains only literals, inversions, and disjunctions of the same, this kind of formula is said to be in *Conjunctive Normal Form* (CNF), because it is the conjunction (Boolean `AND`) of a set of clauses P_i. The formula is true if all of its clauses P_i are simultaneously true under some assignment to the variables in those clauses. This form of problem is notable because any logic function can be transformed into CNF by the Tseitin transformation [Tseitin, 1983], and because Boolean SAT solves problems provided in CNF.

For a concrete example, consider the set of clauses below.

$$\begin{aligned} P_1 &= (x_1 + \neg x_2) \\ P_2 &= (\neg x_1 + x_3) \\ P_3 &= x_2 \end{aligned} \tag{1.3}$$

These clauses together represent following function.

$$\begin{aligned} \mathscr{P} &= P_1 \cdot P_2 \cdot P_3 \\ \mathscr{P} &= (x_1 + \neg x_2) \cdot (\neg x_1 + x_3) \cdot x_2 \end{aligned} \tag{1.4}$$

This formula—which happens to be the same formula described in the previous section—is satisfiable under the assignment $(x_1, x_2, x_3) = 111$, so a Boolean SAT solver given this problem would return an answer of `SAT` (i.e., satisfiable) and the witness 111.

Despite the inherent difficulty of solving NP-complete problems, certain algorithms for solving the Boolean SAT problem have been developed [Marques-Silva and Sakallah, 1999, Moskewicz et al., 2001] that often work quite well on modern hardware to find solutions even on large problem instances with millions of clauses. Thanks to techniques like conflict analysis and

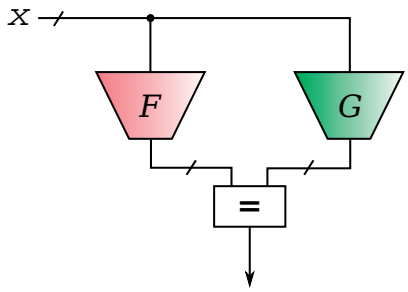


Figure 1.4 – An example of a miter circuit used to check the functional equivalence of a circuit F and a "golden reference" circuit G. Once this circuit is translated to CNF, a SAT solver can determine if F and G are equivalent under all input conditions.

conflict-driven clause learning and backtracking, modern SAT solvers like MiniSAT [Eén and Sörensson, 2003] have made Boolean SAT an attractive encoding for problems that arise in a number of domains, and the premier tool for reasoning about Boolean logic circuits.

Most relevant to this thesis, computer-aided design tools that help designers create digital circuits mostly moved from *Binary Decision Diagram* (BDD) based algorithms [Y., 1959, Akers, 1978, Lai et al., 1992] to Boolean SAT based algorithms [Goldberg et al., 2001, Sapra et al., 2003] soon after the turn of the century in order to cope with increasing design complexity.

This is a prime indicator that Boolean SAT is an effective and scalable way to reason about circuits. For example, SAT-based combinational equivalence checking is a prime application of SAT widely used in CAD. This is done by constructing a special circuit, called a *miter*, that represents the SAT problem (this circuit will be transformed into CNF like in $\mathscr{P}$ above). A miter is a circuit designed to produce exactly one output value. This value represents the satisfiability of the logical formula described by the miter function.

For combinational equivalence checking, the structure of this miter is shown in Fig. 1.4: it is composed of the circuit under test and a functional reference circuit, and computes 1 when an input assignment induces the circuit under test to compute a value that is not equal to that computed by the functional reference circuit. In other words, a buggy circuit (i.e., one that is not logically equivalent to the functional reference circuit) will result in a satisfiable SAT problem instance and any witness returned by the SAT solver is a counterexample, or error trace, that shows the input conditions for which equivalence fails.

The widespread application of Boolean SAT to many problems in CAD and its success has renewed interest in other applications of Boolean SAT to other problems in digital circuit design and also in other forms of satisfiability problems and their potential applications.

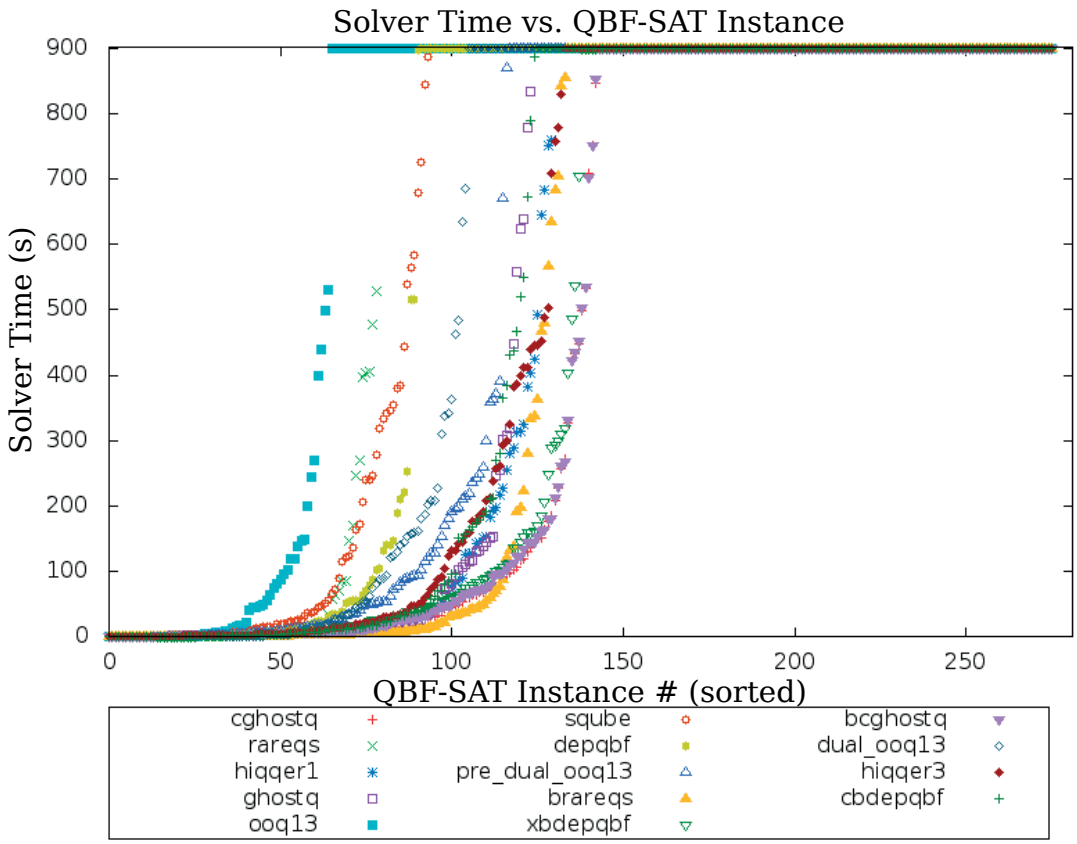


Figure 1.5 – A chart from the 2014 QBFEVAL gallery shows the sorted performance (i.e., solver time, on the vertical axis) of various QBF-SAT solvers for hundreds of instances (horizontal axis). The existence of so many solvers and the ingenuity that results from possible glory in a regular competition symbolizes the innovation in QBF-SAT solver heuristics over the past two decades.

1.3 2QBF-SAT

One of these other satisfiability problems is known as a QBF-SAT problem: A quantified Boolean formula satisfiability problem. Similar to Boolean SAT, these problems ask to find the truth of the following propositional formula $\mathcal{Q}$, with the use of additional quantifiers allowed, as represented by the formula below.

$$\exists \vec{h}\, \forall \vec{x} : \mathcal{Q}(\vec{h}, \vec{x}). \tag{1.5}$$

In other words, the problem is to find a concrete value for $\vec{h}$ such that $\mathcal{Q}$ is satisfied *for any value of* $\vec{x}$. In fact, this is a special sub-type of QBF-SAT problem, known as an exists-forall 2QBF-SAT problem, so named due to the sequence and number of quantifiers in its logical statement.

Just as with Boolean SAT, QBF-SAT solvers have seen dramatic practical performance improve-

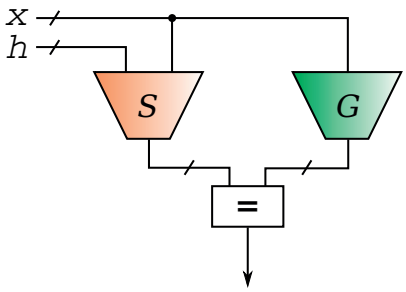


Figure 1.6 – An example of a 2QBF-SAT miter circuit. It is similar to the Boolean SAT miter used in equivalence checking. Here, some sketch S, which functionally depends on some existentially-quantified "hole" values h, is constrained to be equivalent to the "golden reference" circuit G.

ments in the span of only a few years [Lonsing et al., 2016]. Figure 1.5 shows a chart borrowed from the QBFEVAL'14 QBF Gallery [Janota et al., 2014 (published 2016, Jordan and Seidl, 2014]. This chart, which shows the performance of various solvers on all the instances they were able to solve in a 900-second timeout for various benchmarks, symbolizes the innovation in QBF-SAT solver heuristics over the past two decades. Although this chart only provides a snapshot of the contemporary state of the art, the annual competitions (continuing to this day) trace their roots back to the beginning of the century, and continue to promote innovation in QBF-SAT solver technology.

These kinds of satisfiability problems can also be represented as miters, like in Fig. 1.6. Note that this figure shows two distinct input vectors, $\vec{h}$ and $\vec{x}$; these represent the same existentially- and universally-quantified variables in Eqn. 1.5 above. The miter circuit shown in Fig. 1.6 visualizes a simple 2QBF-SAT miter. Here we again have two circuits S and G, but now the functionality of S also depends on the existentially-quantified value $\vec{h}$. We call this kind of meta-circuit, where the exact functionality of the circuit is determined by some solver-determined variables in $\vec{h}$, a "sketch". Once the 2QB-SAT solver finds a solution, the witness (or assignment to the $\vec{h}$ variable) describes a circuit: the combinational logic circuit that results from the specialization of the problem instance $\mathcal{Q}$ under a specific value for $\vec{h}$.

In this thesis, the existentially-quantified variable $\vec{h}$ is frequently referred to as a "hole". This terminology is not unique [Solar-Lezama et al., 2006], though it may seem odd upon first consideration. When a designer explicitly creates a circuit whose functionality depends on the value of a solver-determined variable, that "magic" variable is like a hole in the circuit: some vital but missing piece of information that is provided by the solver.

1.4 Overview

This thesis explores how increased integration of satisfiability, particularly 2QBF-SAT, into the circuit design, debug, and optimization process can be useful to help designers better design,

analyze, and optimize complex circuits. After demonstrating a number of novel, useful, and reasonably scalable applications, we introduce a domain specific language well-suited not only to implementing those applications, but also to facilitating nearly arbitrary applications of 2QBF-SAT to circuit design or analysis.

Chapter 2 describes SKETCHILOG a language for circuit design with an integrated and transparent satisfiability solver. In SKETCHILOG designers provide two implementations of a desired circuit: a simple, but known-correct circuit (called the "golden reference" circuit), and a circuit with certain parts (e.g., values, logical functions) unspecified. These latter circuits, called "sketches", might be more complex but potentially better implementations of the same function in the golden reference circuit. To help the designer make a correct design, he or she is allowed to leave certain values or functions explicitly unspecified, and the satisfiability solver determines if there is a way to complete the sketch so that it is functionally identical to the golden reference circuit under all input conditions.

Chapter 3 describes FUDGEFACTOR a circuit debugging aid that uses a syntax-guided synthesis (SyGuS) approach and a satisfiability solver to localize trivial circuit design errors and provide semantically meaningful corrections. In FUDGEFACTOR the user provides a known-buggy design, at least one failing input vector that exposes the buggy functionality, and the correct response, along with some number of other correct responses to other input vectors. With this information, FUDGEFACTOR first finds suspect error locations using an existing error localization tool, then attempts to permute the design source code around those suspect locations, in order to provide potential alternative source code that *might* represent what the user actually wants and correct the error. A satisfiability problem is then constructed and solved to determine if the substitution of some or any of those alternatives corrects the bug in the known failing vector, and maintains correct functionality for other vectors. In this way, designers can automatically determine if a circuit bug is due to certain kinds of simple errors, without wasting time with a manual root-cause analysis. Further, a provided solution isn't just an opaque jumble of logic gates, but a meaningful, simple alteration directly to the source code.

Chapter 4 describes two satisfiability-based techniques to optimize GLIFT (gate-level information flow tracking) models of digital circuits. Circuits that need to keep certain information secret at all costs, like HSMs (Hardware Security Modules), or circuits that need to guarantee separation of data domains, like in HA (High Assurance) systems, are unfortunately very complex. Interactions between various components or sequences of actions can allow information to flow in undesirable ways need to be detected and avoided. Information flow tracking is a class of techniques that are used to analyze how information propagates throughout a system. Gate-level information flow tracking is especially relevant to the security of digital circuits, as it provides a precise way to model the flow of information through an actual implementation of a digital circuit. Unfortunately, these GLIFT models are very complex, and are often composed of many more gates than are contained in the circuit being modeled. By recognizing that the traditional mapping procedure used to produce GLIFT models ignores internal don't-care con-

ditions and there is often an opportunity for simplification, we first describe a 2QBF-SAT miter formulation and a solving procedure that can be used to find as many such simplifications as are possible without reducing the GLIFT model precision. We then describe another 2QBF-SAT miter formulation that can be used to go even further and make disciplined trade-offs in the precision and complexity of GLIFT models, allowing some imprecision at the expense of false positives under certain input conditions.

Finally, Chapter 5 introduces Nasadiya, a language for solver-aided hardware design and optimization. Nasadiya, like SKETCHILOG is based on Chisel, and embeds a 2QBF-SAT solver. However, Nasadiya provides much more flexible constraint specification, allowing design-based specification of constraints, and explicit solver access, which means the user specifies exactly how to apply the satisfiability solver to achieve his or her specific aims. The explicit solver access makes user-defined iterative solving procedures possible. Nasadiya also provides two simple but powerful and extensible modeling facilities, allowing models of extra-functional circuit properties like delay and area to be used inside constraints. Combined, these features make Nasadiya a powerful tool for automated reasoning about and design and generation of digital circuits. We finish with a case study, showing how Nasadiya can replicate automatically the kind of analysis and reasoning about adder circuits that would otherwise take man-months of engineer time.

2 Satisfiability for Circuit Design Assistance

For decades, digital circuit design has been done at the *Register Transfer Level (RTL)*, and this has been one of the key bottlenecks to productivity. One of the most glaring problems is that RTL design requires the designer to suffer through a tedium of minutae. Thus a number of researchers have repeatedly attempted to raise the design abstraction level [Camposano, 1990]. Progress in the area of *High-Level Synthesis (HLS)* has been less steady than originally anticipated, with various generations of tools reaching the market [Martin and Smith, 2009, Cong et al., 2011, Wang et al., 2014] and perhaps only in the last decade achieving some concrete commercial successes. Yet, RTL still offers a designer the most control, and skilled designers' analytical intuitions about structural circuit optimizations and trade-offs are usually superior to those achieved by high-level compilers.

We have extended a modern RTL design language, Chisel [Bachrach et al., 2012], and found inspiration from the software world [Solar-Lezama et al., 2006], to take a new approach: instead of abstracting away fundamental features of the architecture—as in High Level Synthesis—abstract only those details for which the designer has uncertainty, and let a satisfiability solver reason about the circuit to figure them out. We propose to allow designers construct their circuits in RTL as usual but leave *holes*, or explicit indeterminacies, in their designs, and accept the help of a satisfiability solver to complete their designs.

SketchiLog, the tool implemented by this author and introduced in the paper, "SketchiLog: Sketching Combinational Circuits" [Becker et al., 2014] by Andrew Becker, David Novo, and Paolo Ienne, reads a regular RTL "golden reference" specification of a desired functionality (typically a trivial un-optimized implementation) and an *incomplete* optimized implementation of the same functionality (a *sketch*). SketchiLog determines whether the holes can be filled (i.e., assigned specific, concrete values) so that the functionality of the sketch matches that of the specification under all inputs. If such a substitution exists, SketchiLog outputs fully functional Verilog of the completed and fully-verified solved sketch.

Although the domain of applicability is limited by a restriction to combinational circuits, this effectively relieves designers from responsibility for some of the most annoying details of

an architecture and entirely avoids a common source of maddening and time-consuming bugs. This represents a novel step forward toward using automation to increase designer productivity: rather than trying to hide details from the user and automatically implement the best guess at the user's desired architecture, leave the designer to control all the details of the architecture, but allow a satisfiability solver to reason about the circuit to fill in whatever the designer did not specify.

2.1 Motivational Example

Any digital designer knows how to make an efficient two's-complement ADD/SUB unit. However, suppose for the sake of example that a designer does not remember *how exactly* to build the unit, but remembers that some voodoo with an adder's operands can implement a subtracter. Our designer might describe Fig. 2.1b as a reference and sketch Fig. 2.1c from fuzzy intuition—an adder, with inputs somehow permuted, can also implement a subtracter.

The core of this sketch can be expressed in SKETCHILOG as shown in Fig. 2.1a: a simple ripple-carry adder whose inputs are some undetermined function (a *black box* implemented with a look-up table of holes) of `d` and of the corresponding bits of the operands `a` and `b`. These holes correspond to the existentially-bound `h` bit vector in the QBF-SAT formula in Eqn. 1.5. SKETCHILOG solves the sketch and finds that the values shown in Fig. 2.1d for the holes in the look-up tables force the circuit to match to the reference design. Figure 2.1e shows what the solved sketch in Fig. 2.1d might look like after simple logic synthesis. Note that this logic synthesis is a very small and simple problem, where it can be sure the result will be high-quality; unlike large global logic restructuring, this is an area where traditional logic synthesis tools excel.

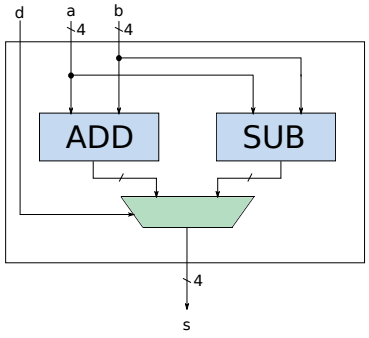
When a solution exists, correct hole values are always found and the resulting design is guaranteed to be functionally correct. If holes are not abused to give excessive architectural freedom, a given solution will usually be *very nearly as small and fast* as if the designer had no uncertainty at all. Our goal with SKETCHILOG is to provide useful and intuitive RTL language constructs which help designers focus on architectural intuition instead of nitty-gritty details, and yet can be encoded as a vector of unknown Boolean variables (holes).

2.2 Related Work

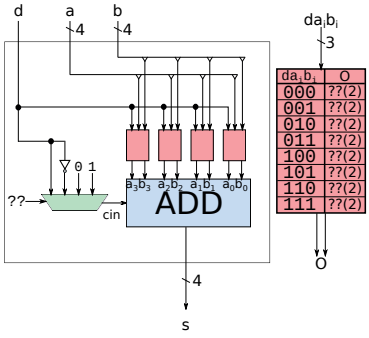
The idea of leaving the specific implementation of a program to a computer and, given a specification of the desired behavior, synthesizing an executable program has a long history [Manna and Waldinger, 1971, Pnueli and Rosner, 1989]. Full synthesis systems, like NuPRL [Constable et al., 1986] are based on deductive synthesis, where a designer specifies theorems about his or her desired application and guides an interactive process to create an executable proof.

```
1  carries(0) := BB(io.d, 1);
2  for(i <- 0 until 4){
3    val fadd = new full_adder;
4    val unkn = io.d##io.a(i)##io.b(i);
5    val bb = BB(unkn, 2);
6
7    fadd.io.a := bb(0);
8    fadd.io.b := bb(1);
9    fadd.io.cin := carries(i);
10   sum_sigs(i) := fadd.io.s;
11   carries(i+1) := fadd.io.cout;
12 }
```

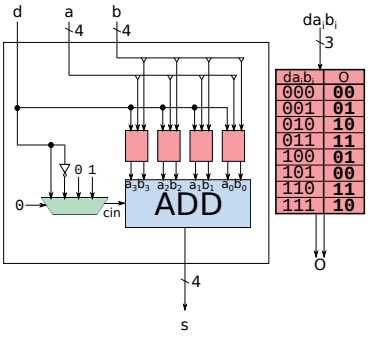
(a) SKETCHILOG code for a naïve sketch of an optimized ADD/SUB unit.



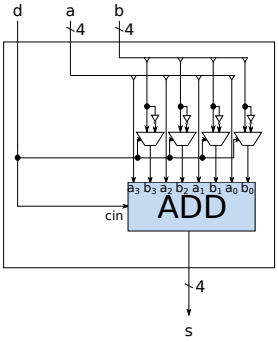
(b) A simple reference ADD/SUB unit.



(c) A visualization of the sketch in (a).



(d) A visualization of the solved sketch from (a).



(e) A more understandable representation of (d) that results from logic synthesis.

Figure 2.1 – A naïve sketch of an ADD/SUB unit. The solution (e) immediately reminds an inexperienced designer that the adder should be fed with signal `a` unmodified and with carry-in and `b` signals conditionally inverted upon the value of `d`.

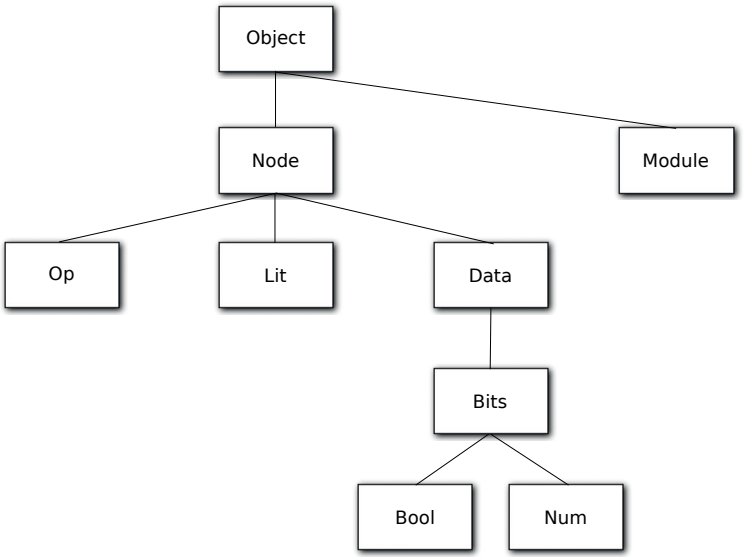


Figure 2.2 – A visualization of part of the Chisel type hierarchy.

More frequently, though, such general pure synthesis techniques are not used for a variety of reasons. Instead, domain-specific tools use accessible high-level specifications to synthesize programs [George et al., 2013]. Recent work has also improved the utility of high-level synthesis tools, which use software-like input specifications to synthesize increasingly decent hardware implementations [Gupta et al., 2003, Nane et al., 2015]. However, these tools often produce sub-par implementations, and are not capable of discovering any fundamentally new architecture.

The mechanics behind SKETCHILOG lie along the same vein, and are very similar to those behind Sketch [Solar-Lezama et al., 2006]. Sketch is a software compiler that allows programmers to embed holes in software programs and with the aid of an additional functional reference program, constructs a 2QBF-SAT problem to find concrete values for those holes such that the solved program is functionally identical to the provided reference program. This allows the programmer to focus on the more general aspects of the architecture, while automating away most of the tedious task of handling corner cases or specifying exact constant values. All this is accomplished by constructing exactly the kind of 2QBF-SAT miter demonstrated in Fig. 1.6, and is conceptually broadly similar to the SKETCHILOG system we describe here, but developed specifically for software, and is unsuitable for circuit design.

2.3 Implementation

Conceptually, SKETCHILOG translates both the sketch and the specification to pure Boolean functions $\mathscr{S}$ and $\mathscr{R}$, respectively. Both functions take the same `k`-bit input vector `x`, but the sketched function also takes an additional `m`-bits parameter `c`, representing the `m` hole bits in the sketch. The problem reduces to a 2QBF-SAT satisfiability problem that can be solved by building the appropriate miter, as described in Sec. 1.3.

```
class Multiplexer extends Module {
  var io = new Bundle {
    val sel = Bits(INPUT, 1)
    val in0 = Bits(INPUT, 1)
    val in1 = Bits(INPUT, 1)
    val o = Bits(OUTPUT, 1)
  }

  io.o := (~io.sel & io.in0) | (io.sel & io.in1)
}

// ... later instantiated:
val m = new Multiplexer;
```

Figure 2.3 – A simple example Chisel module.

In this case, the 2QBF-SAT problem instance is constructed to solve the following problem:

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

In other words, the object is to find an assignment for $\vec{c}$ such that $\mathscr{R}(x)$ and $\mathscr{S}(x,c)$ are equivalent for all possible assignments to $\vec{x}$.

We chose the Scala-hosted Domain Specific Language (DSL) Chisel [Bachrach et al., 2012] as the base language upon which to implement our language features. Its use of Scala [Odersky et al., 2010] lends it easy extension and customization, and its scripting-like functionality makes sketching more intuitive and a better fit for circuit generators, which are very commonly used to describe the combinational components SKETCHILOG targets.

Chisel generates regular Verilog code and (solved) sketched designs can be used in standard EDA design flows. The language features added to Chisel could also be added directly to a VHDL or Verilog compiler, though this would likely require a much less intuitive syntax and deny the designer the very useful facilities that Chisel provides. For example, a designer uncertain about a logic function *can* but likely would *prefer not* to manually describe a look-up table filled with holes to be used in place of that function. Using Chisel allows us to provide the designer with simple syntactic constructs that takes care of the implementation and leaves the designer free to focus on the bigger picture.

In Chisel, designs are represented as instances of the `Module` super-type, with a member `io` that points to all circuit primary inputs and outputs. Figure 2.2 shows part of the Chisel class hierarchy.

These I/O signals are objects of the `Node` class sub-types, and are linked with objects of the `Op` class type. In Chisel, all designs are design generators: Chisel designs are Scala programs that instantiate `Module` objects with `Node` objects linked together by various `Op` objects.

Figure 2.4 shows a visualization of the example Chisel module in Figure 2.3. After the object graph is constructed in this *elaborate* phase, the Chisel library provides facilities to emit Verilog RTL from it. Figure 2.5 shows this two-phase process for compiling a Chisel design to Verilog.

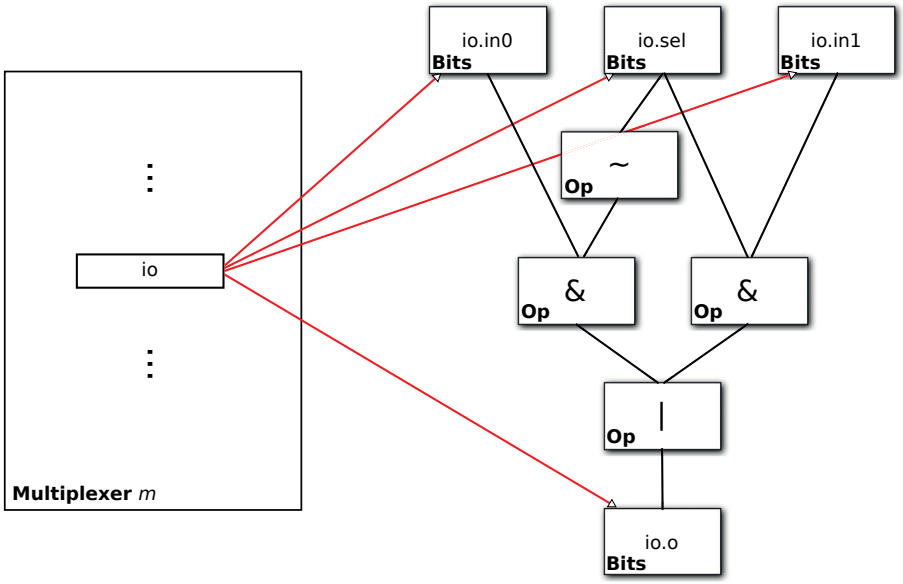


Figure 2.4 – A visualization of an example Chisel object graph created with the instantiation of the `Multiplexer` module in Fig. 2.3.

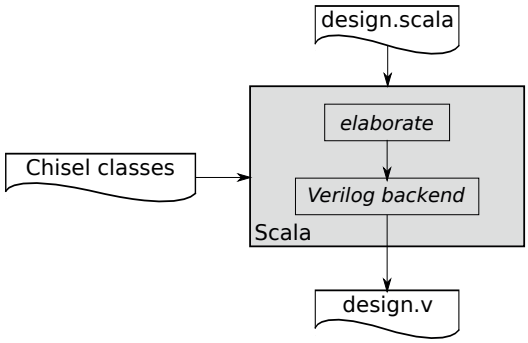


Figure 2.5 – The complete tool flow required to turn a Chisel design into a Verilog design ready to be used by any standard EDA tool.

2.3.1 SketchiLog

SketchiLog uses the same basic Chisel tool flow, but also adds a few new steps. The entire SketchiLog flow is described in Fig. 2.6. First, the designer creates his or her sketch in SketchiLog and names it `sketch.scala`, and creates a golden reference module whose functionality the sketch is designed to replicate exactly. Scala is invoked on the sketch, and a resulting "sketched Verilog" file `sketch.sv` is produced. This "sketched Verilog" file is identical to a regular Verilog file produced by Chisel, but crucially has special language support for specifying hole signals. All SketchiLog language constructs described below are compiled to Boolean logic referencing these special hole signals.

Next, optionally, the resulting `sketch.sv` and the designer-supplied golden reference module

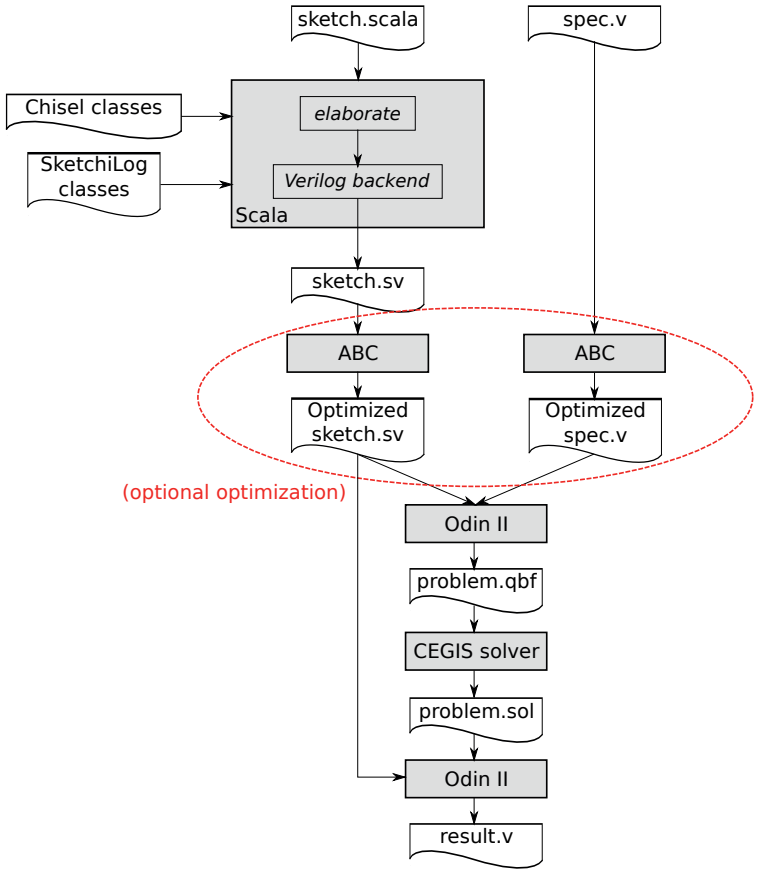


Figure 2.6 – The complete tool flow required to turn a sketch (`sketch.scala`) and a golden reference module (`spec.v`) into a complete, solved result `result.v` ready to be used by any standard EDA tool.

`spec.v` are provided to the ABC [Brayton and Mishchenko, 2010] logic synthesis tool for simplification.

Once any desired simplifications are performed, both the sketched Verilog and the golden reference are provided to a custom-modified version of Odin II [Jamieson et al., 2010], which reads both the sketch and the golden reference, combines them into a miter, and writes the resulting circuit as `problem.qbf`, in a format understandable to the CEGIS 2QBF satisfiability solver [Solar Lezama, 2008].

The CEGIS solver is then invoked on this problem file. If a solution is found, it is written to `problem.sol`; if not, an error is produced and reported to the designer: there is no assignment to the holes that induces functional equivalence between the sketch and the golden reference. If a solution is successfully found, Odin II is again invoked, this time in a special mode that replaces hole signals in the sketched Verilog with the corresponding concrete values the solver found and wrote to `problem.sol` and writes the final resulting Verilog file to `result.v`.

2.3.2 The Rules of the Code

On top of the standard Chisel features, we provide four intuitive constructs to support uncertainty in designs. Each construct can only be used to provide a value to Chisel data types (representing circuit elements) and never any regular Scala `Int` type (which are only useful to aid the construction of Chisel data types): the left-hand side of each expression below must be a Chisel data type.

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

This first construct, an uncertain constant (or *raw hole*) generator, serves as a substitute for a concrete signal value, and represents an `n`-bit constant signal whose value is undetermined. The value returned is a subclass of the Chisel `Bits` class, so it integrates seamlessly with regular Chisel code. This construct is the simplest both to understand and implement: SKETCHILOG infers an additional `n`-bits in the constructed satisfiability problem's existentially-quantified vector and will leave it to the satisfiability solver to find a concrete `n`-bit value that leads to a functionally correct design (if such a value exists).

All following constructs build upon this fundamental part of SKETCHILOG.

```
x := either choose signal1 or
     signal2 or signal3;
```

This second construct, a selection operator, allows a designer to express an uncertain choice of signals in a design. SKETCHILOG automatically creates raw holes which represent constant values for the select inputs for newly-created multiplexers that choose one of the specified signals.

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

This third construct, an undetermined index operator, allows a designer to express a partially-constrained index or bit in any indexed sequence data structure or Chisel signal type, respectively. It is more or less a further specialization of the second construct, selecting among the signals identified through every feasible index into `my_array` (e.g., 1, 3, 5, etc.). Any out-of-bounds index is silently dropped from consideration—helping designers not to worry about edge cases. A *feasible set* associated with each hole is computed by static analysis of the index expression similar to classic bit-width analysis [Mahlke et al., 2001].

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

This powerful construct, an arbitrary logic function generator, constrains a signal `x`'s value very loosely: only its dependencies and width are provided. Determination of exactly what

logic function to implement is left to SKETCHILOG. This adds $2^{depends.width} * n$ bits to the existentially-quantified vector in the satisfiability problem. It must be used cautiously, however, as the number of hole bits grows exponentially with `depends.width`. Its misapplication with unreasonably large widths or number of dependencies dramatically affects scalability.

2.3.3 Hardware Sketching vs. Software Sketching

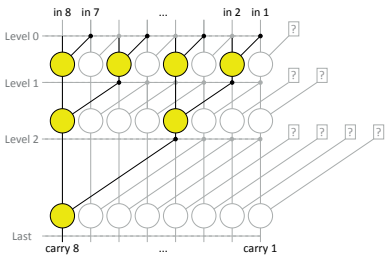
Solar-Lezama et al. pioneered the sketching concept in a software context with a language called SKETCH [Solar-Lezama et al., 2006]. The same group toyed with the idea of sketching hardware [Raabe and Bodík, 2009], but to the best of this author's knowledge never moved beyond the drawing board. We build our hardware flow upon the CEGIS 2QBF-SAT satisfiability solver originally designed for software sketching. All other parts of our system are carefully tailored to the hardware design process and are either built from scratch, borrowed from other work with minor modifications (ABC), embraced and extended from other work (Chisel), or heavily modified from their original form (Odin II [Jamieson et al., 2010]).

The main difference (other than the domain of application) between software sketching as presented by Solar-Lezama et al. [Solar-Lezama et al., 2006] and our SKETCHILOG hardware design framework lays in the generation of the satisfiability problem. First, the software SKETCH framework needs to build the Boolean circuit models used to solve the satisfiability problem from an imperative C-like language by a sort of high-level synthesis. This inherits many difficulties from HLS; the generated models are often more complex than required, leading to increased solution times. In contrast, in our framework the Boolean circuit model is the actual sketch, which is directly constructed by the designer. As part of the model itself, our hole bit-widths are always known precisely while, in SKETCH, assumptions must be made to constrain the size of potential hole assignments. Second, software SKETCH allows the end user to reference a raw hole nearly anywhere in the code. Instead, we provide the set of constructs detailed in Section 2.3.2 to encapsulate holes and thus prevent the user from misusing them in ways that are possible in SKETCH. For example, software SKETCH code can contain a hole in place of a loop bound, resulting in potentially enormous models as the loop is unrolled. Such uncertainties in circuit structure cannot happen with SKETCHILOG.

2.3.4 The Limitations of SKETCHILOG

SKETCHILOG is limited in scope, however, in two key regards. First, only combinational circuits are supported. This considerably restricts the domain of applicability, mostly to arithmetic or simple control structures (e.g., arbiters). Second, the empirical difficulty of solving 2QBF-SAT problems limits the feasible problem size, and solver performance is highly instance-specific. Minor changes in a designer's sketch might have a dramatic effect on solution time.

We believe these limitations do not fatally detract from the value of SKETCHILOG. While a limitation to combinational circuits seems severe, it still covers many use cases (especially for



(a) The adder structure.

```
1  for (k <- 0 until levels) {
2    for (i <- 0 to width) {
3      val GP = new carryOp;
4      GP.io.l := c(k)(i);
5      GP.io.r := either choose c(k)(i-pow2(k)) or ??(2);
6      c(k+1)(i) := GP.io.out;
7    }
8  }
```

(b) The simple code to build it in SKETCHILOG.

Figure 2.7 – The core of a generator for a Kogge-Stone Adder. With SKETCHILOG, the designer focuses on the intuition of creating a binary tree of `carryOp` cells for each output and essentially ignores trivial but annoying boundary conditions.

datapath components), and simple pipelined circuits are functionally isomorphic to combinational models. This makes SKETCHILOG applicable to many arithmetic circuit generators, which are often some of the most tricky circuits to do well and get right.

Further, QBF-SAT solvers are an established and active area of research [Lonsing et al., 2016, Giunchiglia et al., 2001] and significant performance improvements are likely to follow in the near future. While many real-world sketches can already be solved, solver scalability will only improve.

2.4 Experiments

This section demonstrates our tool through simple but conceptually representative use-cases. For clarity, we have selected simple architectures which are described in any basic course in computer arithmetic, even if they are readily available in synthesis libraries—the purpose is to illustrate the simplicity of the approach and how SKETCHILOG could even benefit library writers themselves.

2.4.1 Prefix Adders

The problem of adding two binary numbers as quickly as possible reduces to the problem of computing the carry signals C_i (represented in the form of a *generate* and *propagate* signal pair) for all bit positions `i` [Ercegovac and Lang, 2004, Parhami, 2010]. The computation of the carry signals can be posed in the form of a series of associative but non-commutative operations:

$$C_i = GP_i \star GP_{i-1} \star \ldots \star GP_1 \star C_0 \tag{2.1}$$

The ripple-carry implementation is an easy reference for SKETCHILOG but it is faster to compute all carry signals independently: they can be computed fully in parallel as binary trees of $\star$ operators, resulting in a *Kogge-Stone Adder* represented in Fig. 2.7. Even such a simple

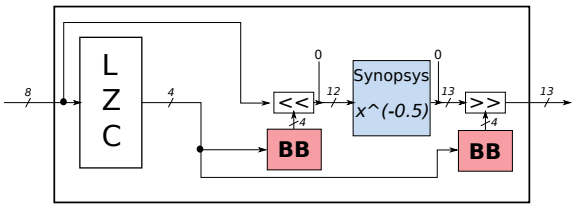


Figure 2.8 – Complex adaptation of an IP component. The intuition is that the shifters and leading zero counter (LZC) will help to scale the input into the component's domain and to correctly re-scale the output. The exact control logic is left to our tool.

structure requires careful attention to detail in the code: instantiating a complete binary tree is not possible for many `i` and if `width` is not a power of two, the largest tree is itself incomplete. Fig. 2.7b shows the actual code needed in SketchiLog to generate the correct hardware, using two of our SketchiLog-specific Chisel syntax extensions. Note the design is not obfuscated by clumsy boundary tests: the designer simply says "connect regularly if you can, or else find a suitable constant".

2.4.2 Sketching to Enable Design Re-Use

Suppose a designer would like to use a library component like a Synopsys DesignWare [Synopsys, 2018b] inverse square-root unit. Unfortunately, that IP component requires the input to be in the range $\left[\frac{1}{4}, 1\right)$, a restriction not adapted to the domain required by the designer. The designer would rather create an adaptation interface than re-implement the unit from scratch. Elementary algebra suggests a variable shift at the input and output of the unit. Intuitively, there must be a correlation between the magnitude of the input and the scaling factors. Unfortunately, finding the exact relations is tricky and error prone. Instead, the designer can construct a general architecture with just his or her intuition (see Fig. 2.8) and these lines:

```
val pre_shift_amt = BB(zero_count, 4);
val post_shift_amt = BB(zero_count, 4);
```

These lines specify that the shift amounts depend *somehow* on the signal `zero_count` and are 4 bits wide. When run with an extra sketched adjustment for the border cases against a trivial infinite-precision look-up table reference, SketchiLog finds the correct implementation—and automatically infers essential but trivial details, like that the input shift amount must be even to re-scale the output without loss of precision.

2.4.3 Strength Reduction of a Constant Divider

Our final example shows the case, common in arithmetic circuits, of finite precision operations implemented by simpler operators with so-called *magic numbers*. One well-known example

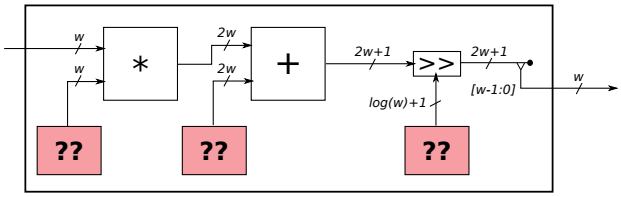


Figure 2.9 – A sketch of a possible strength reduction for constant division with a near-power-of-two divisor. By using a full-precision multiplier and discarding some bits of the result, in many cases the solver can find assignments to the holes such that the imprecision of this approximation is not visible on the outputs.

is the inverse square-root approximation found in, among other places, the *Quake III* video game source code [Lomont, 2003].

In our example, a designer wants to devise an efficient implementation of a fixed-point constant division unit with a near-power-of-two divisor (e.g., 65,535). Figure 2.9 shows how this might be sketched. A simple right shift is a passable approximation, but is not exact. The designer's intuition is again simple: perhaps there are some integers `x`, `y`, and `z` (represented in the figure as holes), such that $(i \cdot x + y) \gg z \equiv \frac{i}{65535}$. In other words, maybe some unknown numbers define a simple affine approximation that, in the face of limited precision, is exact. Such values do exist in this case, and SKETCHILOG finds a correct design significantly smaller than a naïve DesignWare divider with a constant operand.

2.5 Experimental Results

We sketched the circuits described in Section 2.4. We also sketched a few other adders (a Brent-Kung and *hybrid* prefix adders, which lie between the Kogge-Stone and the Brent-Kung). `RippleCarry` is a non-sketched ripple-carry adder. `Expanded_InvSqrt` is the Section 2.4.2 example; `Raw_InvSqrt` is the IP used inside. `DW_Const_Divider` is a DesignWare divider with a constant divisor. We run SKETCHILOG both with and without a sketch pre-optimization pass, where the sketch circuit model is first run through simplifying logic transformations in ABC. Some statistics on these sketches and CEGIS solver runtimes for these problems are reported in Table 2.1.

In our experience, this pass often tends to *hurt* solver performance as often as it helps, but it's possible improved heuristics and different transformations that are exclusively beneficial to solver performance. In any case, the resultant "optimized" circuit's AIG depth and size are shown after ABC simplifies it with structural hashing and SAT sweeping. We did not re-synthesize these circuits in, for example, Synopsys DesignCompiler [Synopsys, 2018a], because we are not concerned with super accurate results accounting for cell libraries, etc. Instead, we are more interested in the structural changes at the gate level, and ABC's integrated AIG depth reporting is more than adequate to give a sense of how the structural delay changes.

Experiment	Width	Hole Bits	Unopt. Time (*s*)	Opt. Time (*s*)	AIG Depth	AIG Nodes
KoggeStone	16	427	3.732	3.799	12	229
Hybrid_max1	16	509	1.681	1.450	12	217
Hybrid_max2	16	368	0.697	0.901	13	196
BrentKung	16	334	0.842	1.048	16	160
RippleCarry	16	0	n/a	n/a	32	131
KoggeStone	23	713	22.414	24.383	12	379
Hybrid_max1	23	902	6.544	8.374	13	345
Hybrid_max2	23	570	3.434	3.112	15	296
BrentKung	23	522	2.884	3.339	18	237
RippleCarry	23	0	n/a	n/a	46	187
KoggeStone	32	992	93.620	85.064	14	545
Hybrid_max1	32	1754	27.984	40.008	14	529
Hybrid_max2	32	1166	19.772	19.871	15	488
BrentKung	32	720	10.558	12.256	20	333
RippleCarry	32	0	n/a	n/a	64	259
Expanded_InvSqrt	8/13	96	1.131	0.928	370	4093
Raw_InvSqrt	8/12	0	n/a	n/a	371	4002
ConstDivision	32	40	26.867	7.960	84	1152
DW_Const_Divider	32	0	n/a	n/a	255	2007
ConstDivision	64	73	3373.790	333.083	164	4400
DW_Const_Divider	64	0	n/a	n/a	529	6291

Table 2.1 – Experimental results detailing instance bit-width, CEGIS solver runtime both with and without an optimization pass, total number of hole bits, and critical path delay (AIG depth) and area (AIG size) for the resulting completed design.

Our data show that for most experiments, the solver runtime is low enough to enable SKETCHILOG's use as a real design aid. The adder experiments in particular show that our framework is scalable enough to be used as part of a standard design flow, at least for some important circuits. The inverse square-root example demonstrates that the described sketching constructs require very little overhead in the final solved circuit.

2.6 Conclusions

RTL design is here to stay—it may be complemented by higher level abstractions, but likely will not be supplanted.

We demonstrate here some first attempts at a new way to improve RTL design by allowing designers some explicit indeterminacy in designs. Despite the simplicity of our examples, the potential benefits of sketching circuits are clear: SKETCHILOG removes the burden of those small details which often cause errors, and are most annoying to get exactly right. It is this kind of precise reasoning at which satisfiability solvers truly excel.

Since a golden reference circuit is assumed to be available (of any quality—hence naturally simple to write and debug) and the 2QBF-SAT formulation ensures functional equivalence to this reference design, SKETCHILOG not only takes the dirty work from the designer but also

guarantees that the resulting design is correct. On the other hand, if *filling in the holes* and obtain a working circuit is impossible, SKETCHILOG immediately reports so.

Although in some domains, like digital arithmetic, the tool is already able to produce practical results, it remains ripe for further exploration, extension, and improvement. One example can be found in Chapter 5 of this thesis. Other future work might switch to other, potentially more powerful satisfiability solvers; increase the number of syntactic constructs to express design uncertainty without resorting to raw holes; attempt to use a sort of bounded model checking to handle sequential circuits, or more.

More broadly, designers might also be interested in other ways satisfiability solvers can be used to make their lives easier. For example, designers might have an existing design they wish to debug, or may not have any golden reference circuit with which to formally compare behavior. Happily, the next chapter discusses just such an application and provides a novel, practical solution to avoid the need for a golden reference circuit.

3 A Satisfiability-Based Approach to Localizing and Correcting RTL Errors

Functional verification is often a difficult part of the digital circuit design process, and occupies up to two thirds of the design cycle [Foster, 2015]. In general, there are at least two ways to reduce the time spent on this part of the design process: Either make it easier to develop functionally-correct circuits from the beginning, or improve circuit debug and verification tools. The previous chapter took the former approach; this chapter describes a technique for the latter approach.

To better understand why we take this approach, it helps to understand the broader context in which these tools are used. Formal verification tools typically return a counterexample when verification fails, and this counterexample is used in a subsequent debugging process (i.e., error localization and correction) to understand the bug and devise a fix. This is sometimes laborious and often relies heavily on designers' expertise and experience. Tools exist to help automate error localization and correction, but most (though not all) work on the subject has either suggested repairs at the netlist level [Chung and Hajj, 1992, Chung et al., 1994], or tried to map netlist repairs back to RTL source code (e.g., [Jobstmann et al., 2005, Staber et al., 2012]), which is not always possible and can lead to incomprehensible repair suggestions.

This as problematic, as designers rarely work directly with netlists: even if tools find errors and suggest appropriate corrections in the netlist, designers must still spend an inordinate amount of time finding the true root cause at the register transfer level to be able to implement a correction they understand and can therefore have faith in. Other debugging aides that attempt to map netlist error candidates back to the RTL source level often suffer from an overabundance of false positives, meaning designer expertise is still essential. While other authors have applied satisfiability-based techniques to aid circuit debugging [Smith et al., 2004, Ali et al., 2004, Chen et al., 2010], these are mostly targeted at the gate level, which is not nearly as useful to designers. Our contribution demonstrates successful techniques to apply the automated reasoning capabilities of satisfiability solvers to locate and correct errors directly at the register transfer level, where designers typically work, in a way that gives designers confidence in the accuracy of the corrections.

In this chapter, we describe FUDGEFACTOR, introduced in the recent work, "FUDGEFACTOR: Syntax-Guided Synthesis for Accurate RTL Error Localization" by Andrew Becker, Djordje Maksimovic, David Novo, Mohsen Ewaida, Andreas Veneris, Barbara Jobstmann and Paolo Ienne and implemented by this author [Becker et al., 2015]. This is a 2QBF-SAT-based syntax-guided synthesis debugging tool for source-level error localization and correction in digital circuits. Ready availability of a tool like FUDGEFACTOR has the potential to noticeably reduce the length of the functional verification design phase by dramatically reducing engineer time wasted debugging simple errors.

3.1 Introduction

FUDGEFACTOR takes as input a buggy circuit design, at least one failing test vector, a few correct test vectors, and a list of suspect error locations. This list is provided by a state-of-the-art error localization tool that is remarkably efficient and can handle very large designs, yet despite its cutting-edge nature lacks precision. This leads to tens—or more—of fairly vague false-positive suspect locations. In our case, we use a commercial verification tool based on the work of Smith et al. [Smith et al., 2005] to obtain the list of suspect locations.

With this list as a template for potential corrections and a library of typical source code errors and associated corrections, we automatically instrument the buggy design. This instrumented buggy design is modified to allow each potential bug either to be left unchanged, or to be replaced with a set of possible corrections. FUDGEFACTOR then combines the instrumented design with the test vector(s) and solves a series of satisfiability problems to discover a source-level correction with the fewest changes that fixes the bug.

The key to this approach is to use correction rules in the instrumentation phase that describe semantically meaningful transformations: To “fudge” a design’s RTL source code to correct an error, rather than attempting to correct a gate- or transistor-level design. Minor source-level changes that correct a bug in a few representative cases without causing failures in others are highly likely to address the root cause and remove the error, so the Source Correction Candidate set FUDGEFACTOR returns (when such possible corrections are found) is usually correct.

Figure 3.1 shows the complete FUDGEFACTOR flow from a buggy RTL circuit to a (list of) suggested source-code correction(s) which fix the error(s) in the circuit. The buggy circuit must come with some test vectors and at least one of them must be failing and expose the error(s).

Effectively, FUDGEFACTOR is represented in the right part of the diagram (from the box “Instrument Buggy Circuit” to the final result) while the leftmost part can be any state-of-the-art error localization tool providing some imprecise result (thus, giving even a fairly large number of false-positive locations). We will discuss the left part of the diagram in Section 3.4 and describe the key mechanisms we use in FUDGEFACTOR in this section.

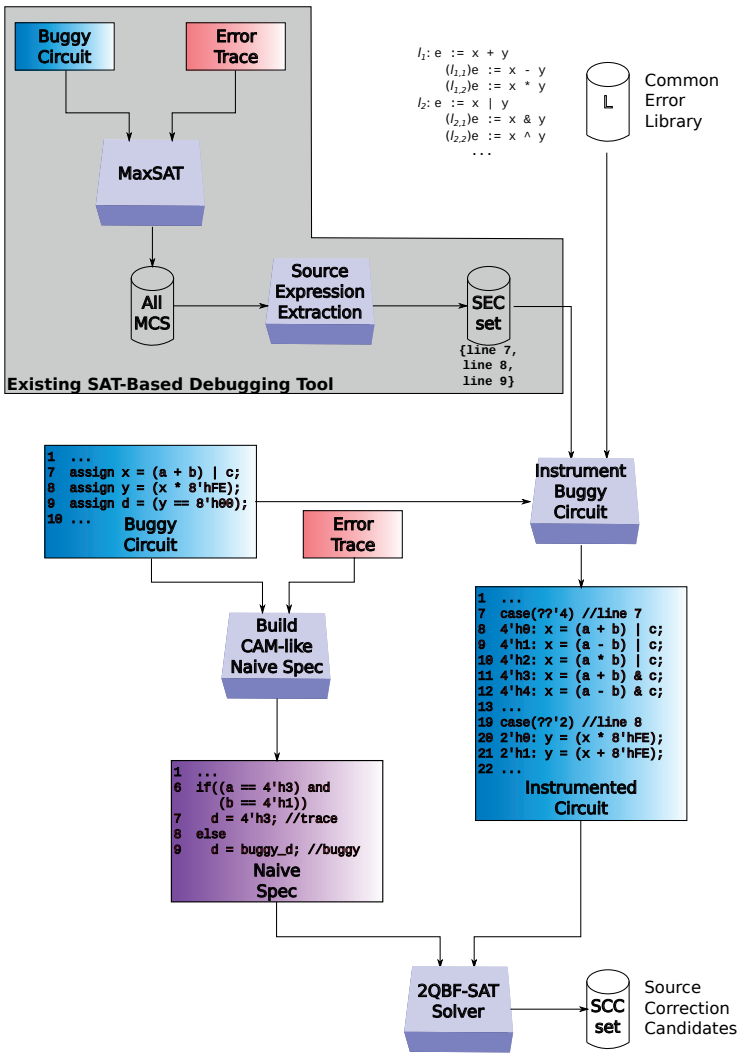


Figure 3.1 – The FUDGEFACTOR tool flow. The inputs are a buggy Verilog design and one or more error traces and the output are candidates to correct the RTL source code.

However, not all design errors are typical, “standard” mistakes that might be found in our library, and thus this approach can never be complete, regardless of the number of rules contained in the library. Still, we describe an approach for a quick, high-confidence initial debug pass that virtually eliminates a lengthy root cause analysis for a number of frequently recurring design errors.

We have tested our tool with 13 different benchmarks from 3 real-world designs available on OpenCores [OpenCores, 2015] and demonstrate here that FUDGEFACTOR suggests valid corrections for a sizable portion of the bugs within a reasonable computational time.

FUDGEFACTOR significantly owes to the approach used by Singh et al. to give meaningful automatic feedback to students of a programming course using Python [Singh et al., 2013]—in fact, the ability to “teach” the designer in which respect the design fails is exactly what drives

our efforts and distinguishes our goal. Yet, our approach in the context of digital design results in at least a couple of significant advantages: (1) Our source-level correction-rules are not at all problem-specific but empirically represent an extensible library of typical mistakes that may occur in any design, such as using a wrong compatible signal in an expression, invoking the incorrect Boolean operator, or instantiating a wrong constant. (2) The breadth of our rules is key to be able to debug arbitrary circuits but, without careful application, would naturally severely restrict our scalability. Thankfully, the existence of (often commercial) tools to approximate error location information and happen to be scalable to industrial size designs enables selective instrumentation. In other words, we only very selectively apply our generous set of correction rules to those candidate locations already suspected, and, as our experiments show, incur perfectly reasonable run times.

3.2 Related Work

Hardware debugging is a topic that has been studied extensively in the previous three decades. This field typically focuses on two related but distinct facets of the problem: finding potential error locations (at whatever level of the design), and proposing corrections which eliminate the errors.

Error Localization. Early works on design error localization were targeted at gate-level representations. Madre et al. [Madre et al., 1989] and Chung et al. [Chung and Hajj, 1992, Chung et al., 1994] proposed error localization techniques that express the problem as a set of Boolean equations. For each gate in the netlist, a Boolean equation is derived, and the existence of a solution to the equations determines if the gate is a potential error source or not. This work only located single-gate faults: faults that are the result of an erroneous gate (e.g., an `AND` instead of an `OR` gate).

Huang [Huang et al., 1998] used a simulation-based approach to find candidate error locations in combinational and sequential circuits. In this approach, a golden reference design is used to detect which of a set of random input vectors exposes faulty behavior in the design. Then, for each faulty vector, a simulation is performed with every signal in the design assumed to be stuck-at 0 or 1. A topological analysis of the circuit helps to quickly eliminate candidate signals as the suspected source of the fault if they are topologically dominated by (i.e., all path from the signal to an output flow through the topological dominator) a signal that was previously rejected as a candidate faulty signal. While this work can detect even multiple faults with reasonable precision, it's limited to detecting stuck-at faults, which are a limited class of design errors not targeted by FUDGEFACTOR.

Smith et al. [Smith et al., 2005] improved on the scalability and quality of gate-level error localization using Boolean satisfiability (SAT). Their approach represented the design as a formula under constraint by a test vector and expected response—since the buggy design by definition has an output different from the expected response, the formula is unsatisfiable. A multiplexer was added to the output of potential erroneous locations (suspects) which allowed

the solver to choose between the original fan-in or an unconstrained input. If the original fan-in made the formula unsatisfiable, then the unconstrained input would be selected. The suspects that were used to make the formula satisfiable are locations that can fix the observed error.

However, fixing design errors at the gate level produces obscure corrections that are very hard for the circuit designer to understand and therefore for the designer to trust. Our approach tackles the problem of returning meaningful corrections for the designer. Given the popularity of HDLs among hardware designers, source-level error localization has become increasingly attractive. Works by Bloem et al. [Bloem and Wotawa, 2002] and Peischl et al. [Peischl and Wotawa, 2006] discussed *Model-Based Diagnosis (MBD)* methods for error localization in VHDL descriptions. In this work, a diagnosis model describes the behavior and structure of a given RTL description. This model is then used to find conflicts between the modeled behavior and the expected output.

Several works [Chang et al., 2007b, Smith et al., 2005] adapted the concept of gate-level fault modeling to source-level error localization by mapping gates to the instantiating location in the RTL description. Our approach adopts the same concept of inserting multiplexers, but instead of having a single free signal, we insert proper error corrections based on an error library model. In this way we restrict the number of possible solutions and improve solver scalability.

Error Correction. Error localization techniques usually generate a design component set: either RTL locations, gates in the netlist or combinational paths that can be modified to correct the error. Chang et al. [Chang et al., 2007a] proposed an approach for correcting gate-level errors using signatures of candidate faulty gates. A signature is a list of bits each corresponding to the gate output for a given set of test vectors. Their approach corrects signatures and re-synthesizes them to replace the gate with one represented by the corrected signature. The idea has been applied to source-level error correction and extended to hierarchical and sequential designs [Chang et al., 2007b]. Jobstmann et al. [Jobstmann et al., 2005] suggested an approach to correct erroneous Verilog designs. Like our work, this approach assumes access to a list of suspect error locations but uses a different error and reference model. It allows corrections that can be represented by arbitrary functions in terms of the state and input variables. This leads to a very general correction model at the expense of readability and reasonability of correction suggestions. We believe that our correction rules lead to corrections that are more meaningful and much easier to understand. In addition, their approach relies on a formal specification (given in Linear Temporal Logic) that describes the desired behavior of the design. Since formal specifications are often unavailable, we focus on simulation vectors, the de facto standard technique in industry to verify digital designs. Staber et al. [Staber et al., 2012] have extended the above-mentioned repair approach to error localization by assuming that only a location that can be corrected can be an actual error location. This approach is more precise but also more expensive than other error localization approaches. It is similar to our approach as it also aims to increase the precision of error locations by searching for

correction suggestions. However, there are significant differences in the setup and underlying technique. Furthermore, our approach is a SAT-based technique, while their approach used BDD-based methodologies, which are known to be less scalable for large designs.

The related works probably most relevant to this chapter are mutation-based approaches. *Mutations* were introduced by Debroy and Wong in the software world [Debroy and Wong, 2010] and closely resemble our "fudging" rules (Section 3.3.1), but their mutations are extremely simple and limited.

More recently, Alizadeh et al. [Alizadeh et al., 2015] have used mutations to create potentially working hardware designs from a failing one; their mutations, essentially targeting signal processing designs, are a restricted and predetermined version of our rules, the latter being much more articulated and constituting an expansible library. Also, successful mutations are identified by enumeration, whereas our encoding of the problem in a Quantified Boolean Formula is more efficient and also more general: it also corrects situations where multiple rules or mutations are needed for a single bug.

A group outside the traditional hardware debugging community recently developed the idea of Syntax-Guided Synthesis (SyGuS) [Alur et al., 2013], which often employs a 2QBF-SAT problem encoding to synthesize unknown functions according to a specified grammar. The original work focuses on Linear Integer Arithmetic (LIA) theory and pure synthesis of functions, has a bespoke constraint specification system, and explores alternative problem encodings and solution strategies. We borrow the idea of grammar-level synthesis of functions, and in this work we adapt it to hardware debugging and combine it with more traditional hardware debugging techniques like vector-based simulation.

3.3 "Fudging" Buggy RTL Circuits

The approach behind FUDGEFACTOR is *syntax-guided synthesis* [Alur et al., 2013]: we tweak (or "fudge") the original buggy RTL specification in many different ways to try to synthesize a new RTL specification which is syntactically as close as possible to the buggy one yet which does not exhibit the error, and is therefore a candidate correction. In the spirit of syntax-guided synthesis, we follow the intuition that acting at the source-code level, respecting the syntactic template provided by the human designer(s) who inadvertently introduced the error in the first place, makes it possible to find good corrections much more easily. More specifically, in our application, we note how some erroneous RTL designs may be extremely "close" to the correct one in the syntactic space and yet fundamentally "far" in the netlist space. One particularly insidious example is a missing condition of assignment in a *case* construct: an omission of just a few characters in RTL can have such a dramatic effect as erroneously transforming a combinational circuit into a sequential circuit. Our approach is perfectly capable of providing *meaningful* corrections in such cases.

In this section, we explain how we instrument the buggy circuit specification given a set of

Rule	**Checker** (if the subgraph looks like...)	**Transformer** (insert these options...)
A	Signal indexing operation	Indices and ranges may be shifted to the left or right by one.
B	Incomplete case without default	Signals assigned in case get a default assignment of any compatible signal, or a pure free variable.
C	If ... If ... Else assigning the same signal	Allow use of a parallel If ... Else If ... Else with the same conditions.
D	Signal in any statement explicitly mentioned in candidate set	The signal may be replaced by any compatible signal.
E	A bitwise comparison operator	The operator may be some other bitwise comparison operator.
F	A constant value on right-hand side; not an index/range	Allow using any constant value (a pure free variable).
G	A ternary expression	The selection condition may be inverted.

Table 3.1 – The common error library rules currently implemented in FUDGEFACTOR. Note that this is by no means a list of all rules one may add, or even an attempt at capturing all of the most common RTL errors. Also note that the transformer rules do not necessarily *replace* the subgraph matched on: the transformer rules insert the *possibility* of using such a change, for which it is often necessary to add multiplexers, signals, etc. to the AST.

error rules in an empirical library and how we construct a miter whose solution results in a possible correction without needing a golden reference design.

3.3.1 Common Error Library

The key intuition of our approach is that many of the errors we make as programmers and designers are relatively predictable in nature: we may mistake one signal for another one which is electrically compatible (i.e., the same number of bits and doesn't cause a logic loop), and this may happen both on the right side of an expression (a wrong input being used in the calculation) and on the left side (the wrong signal being assigned). We might use an incorrect logic or arithmetic function by replacing, for instance, an `OR` with an `AND` or substituting a subtraction for an addition. Or, as already mentioned, we may forget some clauses in a conditional statement, leading to a variety of errors at the netlist level including the potential for circuits (or subcircuits) to become sequential when they should be purely combinational. In different contexts, researchers have already noted that this is an effective way to capture a large fraction of programming errors [Singh et al., 2013]. Self-evidently, this approach cannot capture all possible errors. For example, errors of omission (missing conditions in an expression, etc.) are unlikely to be corrected with our general rules. However, we think there is great practical value in efficiently capturing and correcting common errors and thus freeing precious designer time for concentrating on only a relatively few hard cases.

Our common error library has been developed by reflecting on our experience as RTL designers and by manually inspecting a large number of buggy designs, including student assignments and bug fixes in open-source RTL repositories. (We have excluded most of the circuits which we use as benchmarks; more details about this aspect are given in Section 3.5.) The extensi-

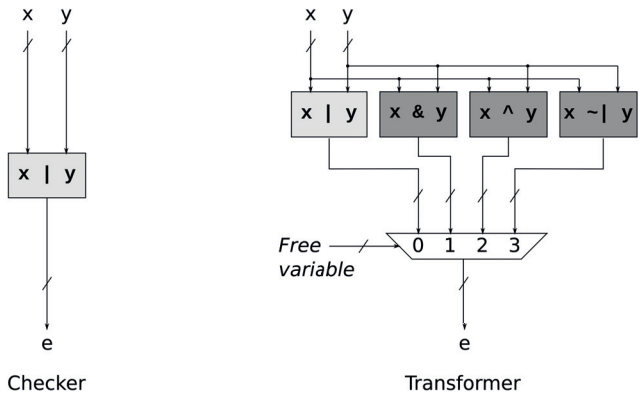


Figure 3.2 – A visualization of an error rule. The designer has written the expression $e := x \mid y$ and this rule suggests that what he or she *might have meant* was any other Boolean function (e.g., AND, XOR, NOR) instead of OR. The *rule checker* is represented in the left part of the figure and, in this elementary case, essentially says that this rule applies potentially to any OR operation. The right part of the figure is the *rule transformer*, which describes how the AST can be rewritten to allow the choice of such an alternative Boolean operator. Note that this figure shows, for convenience, the rule in the form of circuits, but rules are actually described and implemented using AST nodes and some rule-specific ad-hoc code.

ble library contains only a few very general source-code transformation rules described in Table 3.1. Although limited, it turns out that this is already very effective.

3.3.2 Error Modeling

Error rules model common designer errors as modifications to the *abstract syntax tree (AST)* obtained by parsing the input RTL. Because we work directly on the AST, our rules are not limited to identifying line-by-line modifications. Our rules can happily identify and propose corrections for errors spanning multiple lines. Each rule is composed of two different parts: The first part, the *rule checker*, determines whether the particular rule is applicable. The second part, the *rule transformer* expresses the modifications to the AST necessary to include a set of potential corrections. For example, the rule checker of the rule in Figure 3.2 checks whether an AST node represents a bitwise OR operator. If the rule checker matches a particular node of the AST, the corresponding rule transformer is executed and the AST is modified.Figure 3.2 is a simplified example of a rule one might really want to implement; in practice, the rule checker would probably match all bivariate Boolean operator nodes and allow the choice of any shows the rule checker and rule transformer for a simplified version of such a rule matching only an OR operator.

Rule checkers can perform both structural and property checks. Structural checks are based on tree isomorphism (i.e., detecting if the structure of the AST subtree matches a reference one): they detect subtrees of interest and discard cases where the rule transformer should not be applied. We implement rule checkers programmatically (they are embedded in the

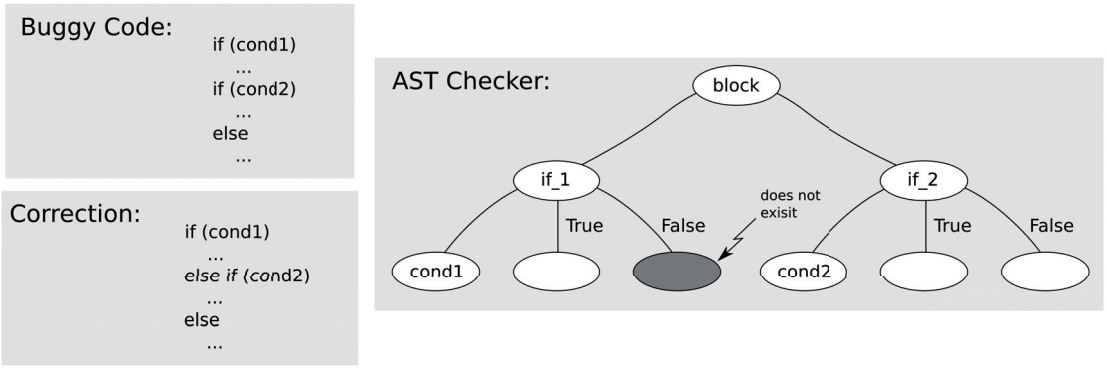


Figure 3.3 – A more complex rule matching condition. This rule checker is shown as an AST subtree to match in the design AST. It approximately corresponds to rule **C** expressing the fact that the designer might have forgotten an *else* clause in an *if* statement. This shows some of the advanced quantifiers we use in our rules, such as the fact that two *if* statement *must* exist in immediate succession within a block but the first one *must not* have already an *else* clause. The example is slightly simplified compared to the actual AST of the parser to improve readability.

FUDGEFACTOR source code), although we think that it could be possible (but not necessarily truly advantageous) to define a formal language syntax to succinctly express the conditions desired. In any case, adding to library is simple enough and only requires modifying one line before recompiling the tool.

Property checks are used to gather relevant non-structural information which is also needed to determine if the rule transformer should be applied, such as checking whether two identifiers in the matched subtree refer to the same constant value. Figure 3.3 shows the rule checker for rule **C** and shows an application of some of the matching features described above.

Rule transformers always instantiate the multiplexer structure illustrated in Figure 3.2, though not necessarily in the same AST location on which the rule matched. These multiplexers select an input depending on some free variables. The satisfiability solver will find the required assignment to these free variables that is necessary to correct the design. Some transformers include a second type of free variable—a *pure free variable*—which can be used to correct constant values (see rule **F**). For example, the condition check $x < y + 3$ can be corrected to $x < y + 5$ with this second type of free variable.

As multiple rules may be triggered on the same AST node, we propose applying the rules following a predetermined ordering roughly going from rules that are more specific to those that are more general. Although this case of conflicting rule matches does not happen with the common error library described here, Table 3.1 is ordered by a proposed priority (the first rule is checked/applied first).

3.3.3 Instrumentation of the Buggy Circuit

To implement the error rule matching and transformations above, we have modified the front-end of Yosys [Wolf and Glaser, 2013], an open source framework for RTL synthesis, to automatically instrument the buggy input circuit. We perform a bottom-up, depth-first traversal of the AST to trigger our code instrumentation. For each node in the traversal, we run each rule checker's structural and property checks around the AST location to identify whether there exists a rule in the common error library which can be applied (i.e., if any rule matches).

When a rule is triggered, the AST is modified to include the option of replacing or modifying the original AST with multiple potential corrections. All modifications result in additional primary inputs added to the faulty circuits: these free variables control whether the circuit retains the original erroneous behavior or is modified by some combination of changes caused by the rule transformers.

The word "combination" above is important: our technique works perfectly well to handle multiple simultaneous bugs, so long as they are each correctable with the available rules. To ensure the solver not only chooses free variables which give correct behavior, but also employ the minimal necessary number of changes, we also add an extra primary output to the instrumented design that is asserted when the number of non-zero free variables is less than some specified threshold, which is used in the miter to reject solutions with too many changes. This threshold is then swept, beginning with only one change allowed and and ending with a user-specified maximum number of allowed changes. We arbitrarily chose a maximum threshold of three changes for our experiments. In this way, FUDGEFACTOR finds the most succinct way to potentially correct the circuit, which may be the most natural source correction.

The next step is to construct a miter, as described in Sec. 1.3, that encodes a 2QBF-SAT problem describing the correctness constraint and the possibility of using alternative subcircuits inserted by the rule transformers. The solution to this satisfiability problem provides a concrete value for all such free variables which together render the circuit correct over all tested input vectors—if such a set of concrete values exists.

3.3.4 Miter Construction

Although the basic idea of the miter used in FUDGEFACTOR is fairly conventional for syntax-guided problems (see Figure 3.6 for a simple example), there are two aspects which are peculiar to our situation. First, in our case we assume that a reference design is not available and that the error is exposed by an error vector or trace used for functional simulation. Second, we want to control (and thus minimize) the number of individual corrections to the buggy code.

Figure 3.4 shows how to build the miter from the instrumented buggy circuit and a set of simulation traces, some of which expose the error. The resulting substitution for a golden

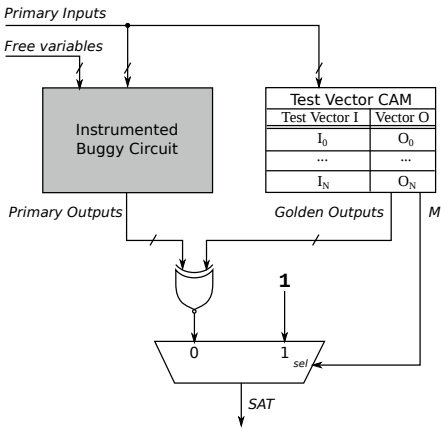


Figure 3.4 – Constructing a miter with test vectors. Since we have no functional reference, we build some golden outputs from a small subset of passing test vectors and all the failing test vectors we are trying to correct. The existence of a particular golden output for a given set of primary input is used to determine whether the output comparison is relevant or not.

reference circuit looks very similar to a Content-Addressable Memory (CAM). We add an extra multiplexer at the output of the CAM to ensure that the miter is formulated so the solver only tries to match the output to the CAM result when the input vector is in the CAM (i.e., when the *M* signal from the CAM is `0`). Thus, our miter is trivially satisfied for all input stimuli not included in the subset of simulation traces we consider.

For those input stimuli which *do* match one of the simulation traces in the CAM, the primary outputs of the template are `XOR`ed with the correct output response. Accordingly, our miter is satisfied by a given vector of free variables (i.e., by a specific set of error rules correcting the error) when the functionality of the instrumented circuit matches the correct output response for all input stimuli in our restricted domain.

One key advantage to using this construction as opposed to a golden reference, aside from the typically-limited availability of such a golden reference, is that it enhances scalability.

Of course, there is a trade-off between scalability and the ability of our method to find a real correction as opposed to simply turning the buggy circuit into another buggy circuit which only works correctly for the formerly failing vector and for a handful of other vectors—a false positive.

We discuss later our encouraging practical findings, largely dependent on the selective application of the error rules which we will describe soon in Section 3.4. Yet, irrespective of our positive results, there are two salient points: First, we aim to provide *meaningful* solutions to the designer and we assume that false solutions, such as those potentially produced using too few passing test vectors, would be immediately identified and discarded. Secondly, if this were not the case, it would be easy for the designer to tentatively implement the correction and verify with his or her standard verification flow if otherwise passing vectors now fail.

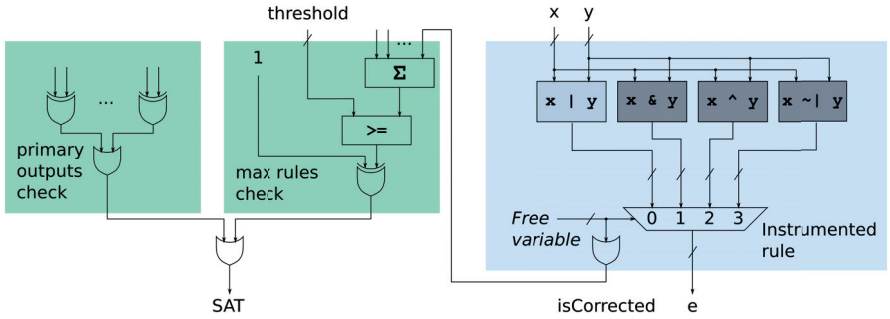


Figure 3.5 – A visualization of the 2QBF-SAT miter that helps to minimize source code interventions. The implementation of each rule transformer stores the free variables used to select a candidate change. Once all of a module's AST has been checked and transformed, these free variables are collected and their Boolean reduction is summed. The signal "isCorrected" above represents the negated Boolean reduction we actually use. A non-zero Boolean reduction (thus, a non-zero free variable) signifies that a multiplexer is configured to change the behavior of some part of the circuit. The miter then counts the number of corrections applied to the circuit and forces it to be below a fixed threshold.

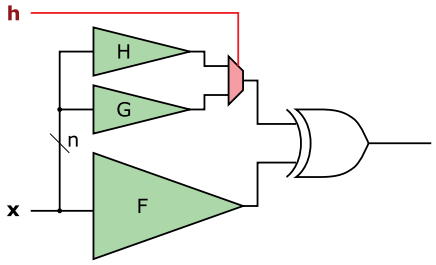


Figure 3.6 – A simplified miter for a combinational syntax-guided synthesis problem. The logic cone F represents a functional reference and cones G and H represent syntax-plausible potential implementations of F. Real miters are much more articulated than this simple example, but the basic idea remains the same.

Besides the functional equivalence constraint, we also encode a second type of constraint to force a minimum number of corrections in the buggy RTL code. Figure 3.5 includes the logic responsible for this second check, mostly in the shaded area annotated as "max rules check". We simply sweep the value of the constant *threshold* in successively formulated 2QBF-SAT satisfiability problems until we find the minimum number of changes that still induce a passing equivalence constraint. We are thereby able to find the minimal source code modification(s), which we intuit is/are closest to a natural human solution, and try our best to rule out less general but still legal solutions.

One final type of constraint may be desirable. We do not consider multiple solutions, but they can be easily handled. At each tested threshold value, multiple feasible solutions might exist for a couple reasons: either there is one or more false solutions caused by eschewing an exact golden reference design in favor of the test vector CAM, or there are simply multiple legitimate

corrections which each require the same number of RTL changes. In either case, at each solution, the previous choice for non-zero free variables can be ‘blocked’, thus excluding that same combination of RTL changes, until no more solutions exist. If multiple solutions with the same number of changes are found, the user can be presented with all of them, possibly ordered by some heuristic priority.

3.4 Selecting Areas for “Fudging”

Applying the error models described in Section 3.3 to the complete AST of a circuit may possibly identify the right correction of the buggy circuit. However, both the ability to generate any possible correction and the likelihood that the correction is the intended one may be jeopardized by this naïve implementation of our idea for a couple of reasons, of both practical and fundamental natures.

First, we deliberately selected very general rules in our common error library (Section 3.3.1). This is key in capturing sufficiently broad cases which are typical of erroneous implementations: We definitely meant to be generous with our rules. For instance, as already mentioned, we imagine the library to be extended progressively with new rules as their usefulness becomes apparent. The consequence of this “generosity” is that, were we to apply every rule on every possible AST node where it can be applied, the QBF-SAT problem would soon become intractable even for extremely simple circuits.

A second, more fundamental problem, is that an indiscriminate application of our error rules would arguably lead, in most practical cases, to multiple possible solutions, some potentially quite far (both in terms of RTL and netlist location) from the “natural” correction. We solve this issue by relying on prior work in circuit debugging and using approximate and netlist-based solutions to guide our instrumentation of the buggy specification.

3.4.1 SAT-Based Debugger

Figure 3.1 shows how we feed the output of a state-of-the-art debugger into FUDGEFACTOR. This output (also called a solution) of a SAT-based debugger [Smith et al., 2005] is a set of design components (RTL blocks, RTL code) that cause the propagation of a failure. This debugger takes as input the RTL description of the design, the expected behavior of the design over a set of test vectors, and returns an over-approximate—but not necessarily precise—set of solutions (i.e., the design component where the actual error is located is within this set). We use this tool to determine the locations on which our methodology should focus to try to correct the failure.

The details of the particular SAT-based debugger we use are not directly relevant to this work and the interested reader can refer to Smith et al. [Smith et al., 2005]. All we care for is that the solution it returns is useful to the designer in most cases but contains enough ambiguity to

require significant human analysis effort to lead to the actual error correction. Specifically, we parse and load the output of the SAT-based debugger and use this information to mark the corresponding AST nodes of the input circuit description as suspect. We then simply add one additional check when we implement the instrumentation pass described in Section 3.3.3: we only apply a rule checker if the node is marked as suspect.

3.5 Experimental Methodology

We evaluated the performance and scalability of our approach on a range of Verilog benchmarks taken from OpenCores [OpenCores, 2015]. Each benchmark has one bug either injected artificially or taken from the version control history (i.e., is organic). These buggy designs were not used to develop our common error library; they were obtained from a third party, and we do not know which bugs were injected and which are organic. We believe our results are broadly representative of how our approach works for simple bugs in realistic circuits.

As mentioned previously, we rely on a commercial verification tool based on the work of Smith et al. [Smith et al., 2005] to obtain an initial set of error candidate locations in the input Verilog. This initial set is significantly over-approximate; it contains many false positives (most of the usually dozens of candidates are not actually part of the error). We use this set in the instrumentation process as discussed in Section 3.3.3 and unroll the resulting logic with ABC [Brayton and Mishchenko, 2010] to handle sequential designs. This unrolled circuit is then passed to the CEGIS 2QBF-SAT solver [Solar Lezama, 2008].

Importantly, as mentioned in Section 3.3.4, we do not rely on the availability of a golden reference circuit: we build a miter from only three passing test vectors. The choice of three vectors is arbitrary here, and is a trade-off between avoiding trivial, incorrect solutions, and scalability. While the topic of determining which and how many vectors to include is certainly interesting, we leave a thorough investigation to future work. The results described below validate our assumption that a few test vectors are enough to properly correct most circuits with our approach: each correction found is exactly what a reasonable human designer would write, and fixes the bug most generally.

SPI Core

SPI (Serial Peripheral Interface) is a serial, synchronous, full-duplex communication protocol very widely used as a board-level interface between different devices such as microcontrollers, DACs, ADCs, and others. This core is an SPI/Microwire compliant master serial-communication controller with some additional functionality. There are four different buggy versions:

- *bug1*. This buggy version includes an incorrect signal assignment. A control register (*"we"*) in the SPI controller FIFO is assigned the wrong signal.

- *bug2*. The second buggy SPI design has a bug in the controller finite state machine. Two state transitions are swapped: specifically, the transition for state `00` is swapped with that for state `01`.

- *bug3*. The third buggy version contains an erroneous data assignment in the controller FIFO. The guard bit register is assigned the incorrect value `1` in some cases instead of the correct value `0`.

- *bug4*. The final bug is the use of an incorrect increment value. A signal in the controller FIFO is incremented by two instead of one. The very next line in the source code has an identical right-hand side, typical of a copy-paste error.

AES Core

AES (Advanced Encryption Standard) is a widely used block cipher with a block size of 128 bits and a selectable key size of 128 to 256 bits. This is a pipelined 128-bit AES design from OpenCores. This core has two buggy versions:

- *bug1*. The bug in this circuit is a missing subexpression in an assignment. An XOR operation of three signals is instead an XOR of only the first two.

- *bug2*. This buggy circuit contains an incorrect signal assignment. One XOR operation references the wrong signal. This signal is the same as the one used in the operation immediately above it, again indicating a probable copy-paste error.

Integer Divider Core

The *Integer Divider Core* is a parameterizable non-restoring signed-by-unsigned integer division core. In our experiments we used a 16-bit dividend and an 8-bit divisor. This design comes with seven different buggy circuits:

- *bug1*. The bug erroneously clips a signal range by one, and concatenates a two-bit constant instead of a one-bit constant. It is difficult to see how this error would be likely to occur, or how it could be corrected with a general rule.

- *bug2*. The bug is an erroneously switched set of function parameters; their order should have been reversed. Because both parameters can be changed to compatible signals, this can be corrected.

- *bug3*. In this version, the arguments of a function are both reversed, but consist of array-indexing expressions. Our rules do not capture the possibility of reversing the operands per se, although this could conceivably be corrected with another fairly-general rule.

- *bug4*. The bug references the wrong array for computing the divisor. Instead of reading the array *s_pipe*, the designer made the mistake of reading from array *d_pipe*—a typo off by one key on a keyboard.

- *bug5*. The index to the array *s_pipe* is off by one.

- *bug6*. In this circuit, the 4 least significant bits in a certain signal are moved to the 4 most significant positions. In other words, the signal is right-rotated by 4 bits.

- *bug7*. In this buggy version, the designer used the wrong second signal in the concatenation of two signals.

Buggy Design	# RTL Changes	Solved?	Oracle Soln.?	Fixing Rule(s)	Applied Rules	Total AST Size	# Matched AST Nodes	SLOC
spi_bug1	1	✓	✓	D	ABDEF	2968	20	271
spi_bug2	–	×	–	–	BD	2964	2	266
spi_bug3	–	×	–	–	DEF	2968	10	266
spi_bug4	1	✓	✓	F	ABDF	2968	13	266
aes_bug1	–	×	–	–	ADFG	5080	19	467
aes_bug2	1	✓	✓	D	ABDG	5251	33	467
div_bug1	–	×	–	–	ADF	2486	13	163
div_bug2	2	✓	✓	DD	AD	2478	8	165
div_bug3	–	×	–	–	ADF	2486	13	165
div_bug4	1	✓	✓	D	ADF	2502	10	165
div_bug5	–	×	–	–	ADF	2516	15	168
div_bug6	–	×	–	–	ADF	2528	20	165
div_bug7	2	✓	✓	DD	ADF	2510	12	165
cpu_bug1	1	✓	✓	B	BDG	3842	4	530
cpu_bug2	1	✓	✓	C	CDEF	3846	5	531

Table 3.2 – Those experiments listed above the break were provided by a third-party and not used to develop rules; those below the break are contrived, but show meaningful results. Note that we correct nearly half of the non-contrived experiments. Note also that all solutions are indeed those which an oracle would provide: exactly what any reasonable human designer would provide. “# Matched Nodes" lists how many AST nodes matched one (or more) of our rules. Finally, “SLOC" represented the lines of RTL source code (excluding comments, etc.).

MIPS CPU

The MIPS CPU is available on GitHub [Mahler, 2015]. We used the CPU design to develop rules, prototype our tool flow, and validate our ability to actually solve the problems we formulate. We injected simple errors that designers commonly make and which we believe traditional debugging tools would have difficulty with:

- *bug1.* In this buggy version, the designer left out a default case when assigning a forwarding source. This causes the creation of an entirely different circuit: this logic is no longer combinational.
- *bug2.* Here, the designer wrote an *if* condition when he or she should have written an *else if* condition.

3.6 Experimental Results

Tables 3.2 and 3.3 summarize the experimental results. The *“# Free Var. Bits"* column gives the total number of free variables used in the instrumented design (including both the control signals of all multiplexers and all pure free variables representing constants). The *“Solver Time"* column shows the cumulative solver time spent on each experiment. For example, those experiments which failed include the solver time used sequentially for all three attempted threshold values (1, 2, and 3). The *“# RTL Changes"* column describes the number of error candidate locations in the Verilog which needed fixing (for the benchmarks where a correction was found)—in other words, it is the minimum number of multiplexer free variables (“isCorrected” in Figure 3.5) which need to be non-zero in order to correct the bug. The *“Solved?"*

Buggy Design	# Free Var. Bits	Total Solver Time (*s*)	# Golden Gates	Unroll Frames	Blowup
spi_bug1	92	1.90	14468	20	2.94x
spi_bug2	8	1.69	14468	20	1.20x
spi_bug3	35	2.23	14468	20	1.90x
spi_bug4	65	1.66	14468	20	2.14x
aes_bug1	373	18.71	86878	6	1.07x
aes_bug2	62	517.40	86878	6	1.29x
div_bug1	33	32.28	96767	48	2.30x
div_bug2	20	71.47	96767	48	2.12x
div_bug3	30	21.82	96767	48	2.28x
div_bug4	26	78.90	96767	48	2.24x
div_bug5	37	49.05	96767	48	3.20x
div_bug6	32	17.75	96767	48	1.99x
div_bug7	30	101.46	96767	48	3.15x
cpu_bug1	12	87.53	34294	15	2.28x
cpu_bug2	46	60.05	34294	15	2.56x

Table 3.3 – More information on the experiments. We show the total number of free variable bits inserted, the total solver time, the size (in And-Invert gates as reported by ABC) of the associated golden reference design, the number of frames it was unrolled, and the total blowup (i.e., how much larger the instrumented circuit is than the unrolled golden reference design.

column reports if the solver was actually able to find a solution with three or fewer changes. Note that even with our relatively sparse common error library, FUDGEFACTOR was able to correct nearly half the simple bugs in the third-party designs.

The *"Fixing Rule(s)"* column describes which rule(s) were essential to correct the bug. In this column, we see that one rule appears with striking regularity: rule **D** (see Table 3.1). This should come at no surprise, as this is one of the most general rules in our library. *"Applied Rules"* lists all rules which were employed in the instrumentation phase for each experiment. Finally, *"Oracle Sol."* indicates whether the correction returned matches that which an oracle would give: if the changes were what any reasonable designer would do, we say, "yes" here. Importantly, all of the solutions found were indeed "oracle solutions". Although we do not, and will never, solve every bug, FUDGEFACTOR reports *no* false positives while maintaining a reasonable true positive rate. We should also emphasize that the true positive rate is artificially lowered by our decision to develop the rules with only a limited set of examples and mostly based on our intuition as designers: as mentioned, we have treated all buggy designs above the break as a clean test-set which has not been used to develop rules. On the other hand, in practice, the extensibility of the common error library is a fundamental part of our approach and many (but not all) of the unsolved designs could be fixed by developing additional general rule.

These tables also include some information which can be useful in determining how practical our approach is and validating our use of the SAT-based debugger to compute an over-approximate error set; our general rules would not scale if they matched many more nodes. As rule **D** shows, our strength is in using fairly general rules, but this comes at a cost: Without

hints of where to look, we would be forced to use less general rules and fundamentally limit our ability to find bugs.

3.7 Conclusions

Since designers introduce bugs to designs in the language they use for the design, we formulate the problem of error localization and correction in a buggy RTL circuit as an RTL syntax-guided synthesis problem. This problem essentially reduces to reasoning about the circuit and many potential syntactic cousins of that circuit, in order to find a correct circuit with minimal syntactic distance from the buggy specification.

To "fudge" the buggy specification into a rich variety of possible alternate circuits, we use an empirical library of error models and corresponding correction rules that tries to capture common errors humans make. Although our rules are quite general and produce a very generous set of alternate versions, we use them sparingly by leveraging other over-approximate, better-scalable bug localization tools. We have shown, though a controlled test set that were not used to develop the initial set of rules, that we can correct a reasonably large set of errors and, most strikingly, in all cases we can correct, we obtain *exactly* the RTL correction a human designer would have produced. As the library of common errors is extensible, we think that the success rate could be improved significantly with acceptable impact on runtimes.

This technique is clearly not a complete solution—it will never find all possible bugs; yet, we believe this novel application of satisfiability to localizing and correcting errors in digital circuits shows promise. The relative speed of satisfiability solvers' reasoning about these problems can be invaluable to help automatically identify intuitive and immediately understandable solutions to simple design mistakes. This frees up designers' time to focus on more complicated design issues that require human creativity.

However, not all bugs are necessarily functional, and just as satisfiability solvers can effectively reason about circuits to discover functional errors, they can also be used to reason about certain security properties those circuits might have. The next chapter explores two techniques that use 2QBF-SAT solvers to reason about possible circuit transformations that reduce the size of certain models used to check security properties. Because these models are typically too large to be useful, and their effective optimization is a global optimization problem with many variables, satisfiability-based techniques to automatically analyze and optimize these models might be essential to bringing this kind of security analysis into the mainstream.

4 Using Satisfiability to Optimize GLIFT Model Circuits

As recent decades have seen an increased focus on software security and a concomitant hardening of software, less-noticed hardware flaws have become an increasingly attractive target for attackers. Unfortunately, circuit designers still largely lack tools that enable the analysis of large designs for security flaws, even as design complexity continues to explode. Further, such security analyses need to consider many more details than just the functional behavior of a circuit, as information that must be protected can leak through so-called "side channels", or extra-functional properties of the actual implementation of the circuit (e.g., timing, power draw, etc.). Information flow tracking (IFT) models provide an approach to verifying a hardware design's adherence to security properties related to isolation (the absence of functional and side channels for information flow) and reachability. These are especially important because humans perform poorly at the task of reasoning about the global state of every gate under any given input condition, which is exactly what is needed to verify these security properties. Humans need help.

However, reasoning about circuits' functional properties is difficult enough, and existing precise IFT models are far more complex than the modeled design itself. Unfortunately, this means that these models are usually too complex to actually use, even with reasoning aids like Boolean SAT solvers; it is common for SAT solver queries to time out, even for IFT models of relatively small designs and when verifying relatively simple properties. It is possible to create less complex models, but these come at the cost of a severe loss of precision—they frequently indicate that some property fails when in fact it holds. Consequently, verification using these less-precise models requires extensive additional manual investigation and seriously undermines the utility of IFT techniques.

This chapter describes work contained in two recent papers that develops 2QBF-SAT based automated reasoning procedures to simplify *Gate-Level Information Flow Tracking* (GLIFT) models, and therefore to make SAT queries incorporating them more tractable. The first is, "Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking" by Wei Hu, Andrew Becker, Armita Ardeshiricham, Yu Tai, Paolo Ienne, Dejun Mu, and Ryan Kastner, and for which this author developed the first 2QBF-SAT-based optimization

method to reduce to GLIFT model complexity without sacrificing any precision [Hu et al., 2016a]. The second is, "Arbitrary Precision and Complexity Tradeoffs for Gate-Level Information Flow Tracking" by Andrew Becker, Wei Hu, Yu Tai, Philip Brisk, Ryan Kastner, and Paolo Ienne, and for which this author devised and implemented a 2QBF-SAT-based algorithm to accept some fine-grained, controlled, disciplined sacrifices in model precision to achieve even greater GLIFT model complexity reductions [Becker et al., 2017]. This final method allows using the most appropriate precision/complexity trade-off for the design size and available computing resources, meaning it is now possible to create models that are not too complex to be usable, and which offer more precision (fewer false positives) than was previously practical.

4.1 Introduction

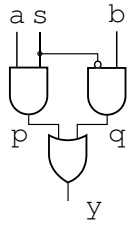
The constant increase in semiconductor hardware design complexity and the many millions of logic gates in modern designs practically ensures that digital circuits will contain security flaws even with the most ardent efforts to avoid them. In addition, supply chains are typically opaque [Bloom et al., 2012], and methods for sabotage can be so stealthy [Becker et al., 2013], that malicious design modifications could remain undetected for years. It is thus not surprising that attacks exploiting hardware design flaws are increasingly common, and the target scope includes everything from personal mobile devices [Kocher et al., 2018, Lipp et al., 2018, Genkin et al., 2013] to air defense radar systems [Adee, 2008].

Automated analysis methods that can verify a system adheres to high-level security specifications could eliminate the possibility of certain exploitable flaws. Information Flow Tracking (IFT) models [Bidmeshki and Makris, 2015, Zhang et al., 2015], for example, can help to verify non-interference [Goguen and Meseguer, 1982], or that if one attaches security labels (i.e. 'high' or 'low' security) to inputs and outputs, 'low'-security outputs are unaffected by 'high'-security inputs. This means IFT models can be used to check properties like isolation and reachability [Goguen and Meseguer, 1982], useful, for example, to detect hardware Trojans [Bloom et al., 2012].

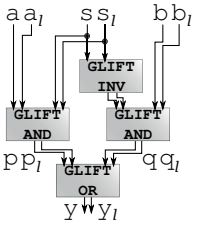
4.1.1 GLIFT

Of particular interest are gate level (GLIFT) models [Tiwari et al., 2009] which add a "taint" label to each signal in the raw design netlist and model how tainted information can flow gate-to-gate from inputs to outputs. GLIFT models are interesting for hardware designers because they capture information which enables the verification of important properties of a specific circuit implementation related to confidentiality, integrity, and logical side channels [Oberg et al., 2011].

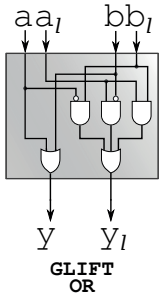
Designers usually want to check a number of these properties, and so once a GLIFT model is constructed, the designer formulates a number of Boolean SAT queries incorporating that model in order to check each property. This ability to query a GLIFT model and get proofs



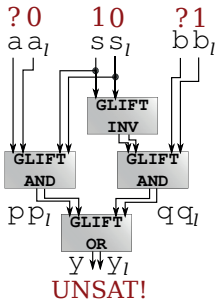
(a) An example original design netlist.



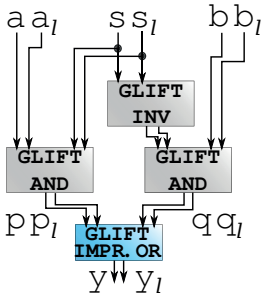
(b) The corresponding GLIFT model for the design in (a), as built by the "constructive method"



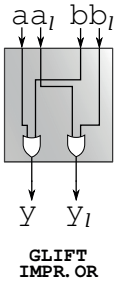
(c) A white-boxed GLIFT OR cell.



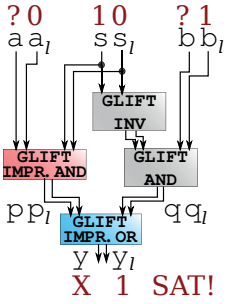
(d) An illustration of a query on the GLIFT model in (b).



(e) A precisely simplified version of the GLIFT model.



(f) A white-boxed imprecise "all-OR" GLIFT cell.



(g) An imprecisely simplified version of the GLIFT model and an associated false-positive flow.

Figure 4.1 – An overview of GLIFT cells, models, and queries.

of invulnerability to classes of attacks is extremely tantalizing. In addition, the ability to enumerate flows (i.e., one assignment to the inputs and input labels that causes one output to be considered tainted) helps to diagnose and correct those security flaws when they are detected.

To understand how these models are built and used in practice, consider Fig. 4.1. Figure 4.1(a) shows a simple logic circuit (a MUX) and Fig. 4.1(b) shows how the typical method for constructing GLIFT models, the "constructive mapping" approach, replaces each gate in the netlist with a GLIFT cell that implements the very same logic functionality but also includes extra inputs (a_l, s_l and b_l) and outputs (y_l) to label and track tainted flows. Figure 4.1(c) details the GLIFT OR cell, which expresses that the output is tainted either if both inputs are tainted or if one input is tainted and the other is at the logic value 0 (that is, it exposes the state of the tainted input to the output). A typical Boolean SAT query on this GLIFT model, albeit a trivial one in this elementary example, is shown in Fig. 4.1(d): The user asks whether there is any condition under which a tainted input b may leak to the output given the knowledge of some of the inputs and labels; the answer is clearly "no" in this case (and hence the query is

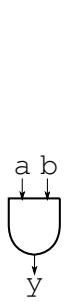
UNSAT) because $s = 1$ and the output is logically connected to a (remember that the circuit is a MUX). The notion of taint is entirely a construct of the user; for example, the user might mark some input (say, a secret key to a cryptographic function) as tainted (by assigning 1 to the corresponding label input) in order to ensure that this information cannot flow to an output (say, a 'done' signal output of that cryptographic function).

Fig. 4.1(e) shows the result of our precise simplification approach [Hu et al., 2016a], where the GLIFT OR cell at the circuit output is replaced with a lower-cost imprecise version shown in Fig. 4.1(f). The simplification is possible because the internal signals p and q are mutually exclusive (due to the circuit structure and signal s), so some of the input combinations (like when p and q are both 1) are not actually possible to observe. Thus, the original precise GLIFT OR cell can be replaced with an imprecise version without any change in the functionality—even though taken alone, this imprecise GLIFT cell over-approximates the taint propagation of a GLIFT OR cell. Therefore, the resulting simplified model is smaller but the propagated label value is still perfectly correct under all conditions.

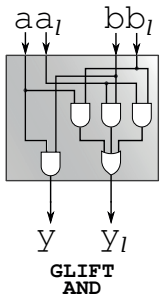
Finally, notice that in Fig. 4.1(g) the same simplification in the taint propagation logic is applied to one of the GLIFT AND cells. Again, the simpler imprecise cell over-approximates the label (is 1 when it should be 0), but in this case the label may propagate to one of the model's label outputs. This GLIFT model is now imprecise because that same SAT query is now satisfiable: the GLIFT model now reports a *false positive*: a reported information flow that does not actually exist. It is worth emphasizing that this does not compromise the security of the model in any way. Instead, it burdens the designer to assess whether each reported flow is indeed real.

To better understand the relation between GLIFT cell precision and false-positive reported flows, consider a gate in a netlist, like the AND gate in Fig. 4.2(a), the corresponding precise GLIFT AND cell in Fig. 4.2(b), and its truth table in Fig. 4.2(d). Note that the functional logic (fan-in to the output y) in the precise GLIFT AND cell is identical to the functional logic in an "all-OR" GLIFT AND cell, shown in Fig. 4.2(c). However, also note the difference in the taint propagation logic (fan-in to the output y_l): the precise GLIFT AND cell has much more complex logic than the imprecise "all-OR" GLIFT AND cell, which only uses a single OR gate to propagate labels—hence the description as an "all-OR" GLIFT AND cell. The imprecision introduced by this simplified taint propagation logic can be seen in the differences between the truth tables for the two GLIFT cells, shown in Fig. 4.2(d) and Fig. 4.2(e): The latter, representing the imprecise GLIFT AND cell, has identical functional behavior, but contains a number of 1-valued cells in the label column that are 0-valued cells in the corresponding column of the former truth table. Each such cell represents an introduced false-positive flow.

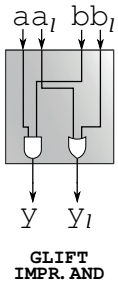
Unfortunately, a single SAT query on a GLIFT model can take days, or even longer, for moderate- to large-sized designs. Designers are thus typically forced to use an "all OR" model, where this simplification (replacing the correct taint propagation logic with a simple OR gate) is applied everywhere. This makes it possible to query models of more complex designs, but



(a) Part of a circuit to be modeled.



(b) A precise GLIFT AND cell.



(c) An imprecise "all-OR" GLIFT AND cell.

a	a_l	b	b_l	y	y_l
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	1
1	1	1	1	1	1

(d) Truth table for the precise GLIFT AND cell.

a	a_l	b	b_l	y	y_l
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	1	0	1
1	1	1	0	1	1
1	1	1	1	1	1

(e) Truth table for the imprecise "all-OR" GLIFT AND cell.

Figure 4.2 – Precise and imprecise GLIFT cells and truth tables.

there is no control whatsoever on introduced false positive flows.

4.1.2 Practical GLIFT

The state of the art approach for handling bigger designs with GLIFT is to use simple Boolean OR to propagate labels [Suh et al., 2004]. Because each GLIFT cell's propagation function is so simple, these models are much less complex. However, this comes at a cost: a Boolean OR propagation function over-approximates information flows. Because in this approach the entire GLIFT model is composed of these cells, the over-approximation is compounded. In other words, while this approach yields solvable models, it sacrifices a great deal of precision to achieve simplicity. This imprecision is a serious problem, because every 'false positive' information flow (the result of model imprecision) requires laborious manual investigation to check the flow's authenticity.

Unfortunately, although designers might want to pick some intermediate trade-off between complexity and precision, designers wishing to use GLIFT models of their designs must typically choose between either precise, but unusably complex models or unusably imprecise, but practical models.

This chapter describes automation techniques for circuit designers to make principled trade-offs between the precision and complexity of GLIFT models. Specifically, we propose three new 2QBF-SAT problem formulations and describe algorithmically how to apply a solver to iteratively generate a simplified GLIFT model either with no imprecision or with some target level of imprecision. Our first formulation uses a 2QBF-SAT solver to determine if some minimum number of the GLIFT cells can be replaced with less complex cells without altering the model behavior under any conditions. A simple search procedure then repeatedly attempts to increase that minimum number until the solver fails to find a solution, and in this way finds the maximum number of simplifying GLIFT cell substitutions. Our second formulation allows imprecision by forcing the solver to choose a set of bit vectors to exclude from the precise equivalence constraint; this approach works in principle, but suffers from severe scalability issues. Our third formulation addresses this shortcoming by instead forcing the solver to choose a set of bit vector *patterns*, including don't-cares, subject to additional constraints on the allowed number of extra imprecise flows. We analyze the trade-off space for an example set of designs and show how our approach can be used to create GLIFT models that trade-off between precision and complexity in a reasonably controllable way.

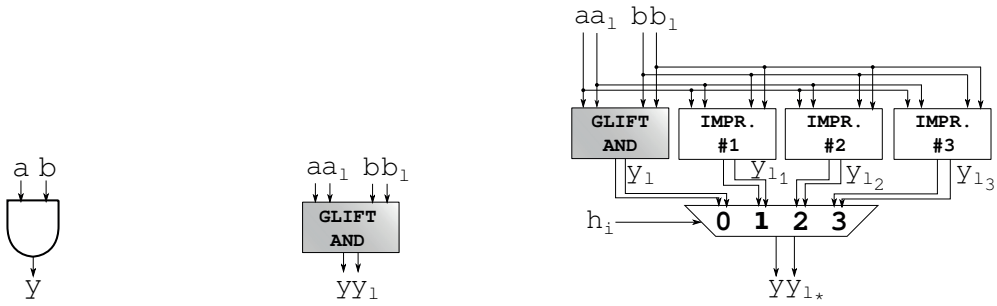
4.2 Precise GLIFT Model Simplification

It is impractical to build perfectly precise GLIFT models for circuits larger than toy examples [Hu et al., 2012]. The constructive mapping heuristic for building GLIFT models demonstrated, for example, in Fig. 4.1, is similar to technology mapping [Hu et al., 2011, 2016b] and is a popular and tractable approach, but comes at the expense of a small amount of introduced false-positive flows. However, even these models are often *still* too complex to be useful.

In this section, we explore a 2QBF-SAT problem encoding that exploits the internal don't-care conditions inside these constructively-mapped GLIFT models to enable a 2QBF-SAT solver to reason about which simplifying GLIFT cell substitutions can be introduced without introducing any additional false-positive flows.

4.2.1 Instrumented Model Construction

Precisely simplifying the model requires first constructing a so-called "instrumented model", incorporating it in a 2QBF-SAT problem formulation, and iteratively driving the solver to find the solution with the most simplifying GLIFT cell substitutions. The instrumented model is constructed in a slightly different constructive mapping process that replaces each GLIFT cell with a GLIFT "supercell" (Fig. 4.3c), where a multiplexer selects a taint propagation function



(a) Part of a circuit to be modeled.

(b) The GLIFT cell that corresponds to the logic gate in (a).

(c) A so-called GLIFT supercell. h_i tunes the (im)precision of the supercell.

Figure 4.3 – An overview of how to construct GLIFT models from circuit gates with the constructive mapping approach, and how to construct instrumented GLIFT models from GLIFT models.

according to a select line `hi` whose value will eventually be determined by the solver. The choice of propagation functions determines the generated model's precision and complexity.

While so far we have only discussed single Boolean `OR` gates as imprecise taint propagation functions, this is rather the extreme. Other alternative taint propagation functions are possible, each with varying costs. For example, the precise taint propagation logic for an `AND` gate is $y_l = a \cdot b_l + b \cdot a_l + a_l \cdot b_l$, but $y_l = b_l + b \cdot a_l$ and $y_l = a_l + a \cdot b_l$ are also correct (i.e., do not cause any false negatives in the flow analysis), albeit imprecise. However, both of these are more precise than $y_l = a_l + b_l$.

By allowing a choice of alternative functions, the solver has potentially more freedom to choose alternative functions elsewhere without changing overall model functionality; although the choice of propagation function is a local substitution, it affects the global context. This kind of circuit reasoning is extraordinarily difficult for humans to perform, and is very well suited to the 2QBF-SAT problem encoding.

The eventual solution specifies a concrete value for each `hi` in the instrumented model, the two-bit signal that selects the appropriate propagation function for a given supercell. Once the selection has been made, the instrumented model is converted to RTL and each `hi` is replaced with its constant value determined by the solver. Simple constant propagation eliminates the unselected alternative GLIFT cells and the multiplexer, leaving only the one GLIFT cell with the solver-chosen propagation function for each original gate.

Simplification as 2QBF-SAT

The 2QBF-SAT problem (see Sec. 1.3 for a brief introduction) is formulated so the acceptance function $\phi(\vec{h}, \vec{i})$'s output is `1` when both of the following conditions are satisfied:

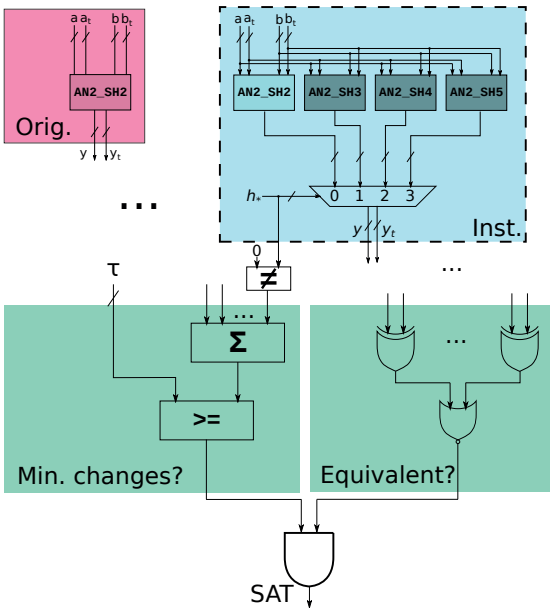


Figure 4.4 – A visualization of the 2QBF-SAT miter circuit for precise GLIFT model simplification. Note the two conditions: on the left, that $\vec{h}$ (the concatenation of all h_*) corresponds to at least τ replacements (i.e., non-zero choices for h_*); on the right, that with these replacements, the GLIFT logic is functionally identical to the original. Note that the "Inst" logic in the dashed box in the upper-right is not connected to the equivalence check: only GLIFT primary outputs are checked for equivalence.

- when $\vec{h}$—the concatenation of all supercells' select lines—configures at least some minimum number τ of supercells to use a locally-imprecise propagation function,

- *and* when the instrumented model's output label values are identical to those of the original precise GLIFT model.

Fig. 4.4 visualizes the components of the 2QBF-SAT miter that is constructed from the instrumented and original GLIFT models. The shaded box in the upper-left represents cells in the original GLIFT model. The shaded box with a dashed border in the upper-right represents the GLIFT supercells in the instrumented model. The shaded box on the lower-left represents logic that ensures some minimum number of supercells select simplifying substitutions. The final shaded box on the lower-right represents the equivalence checking logic; this is the logic that ensures that regardless of the simplifying GLIFT cell substitutions made, the resulting simplified model is functionally identical to the original model under all input conditions.

By iteratively solving and adjusting τ at each iteration, the GLIFT model with the most possible simplifying GLIFT cell replacements will be found.

4.3 Imprecise GLIFT Model Simplification

Precise GLIFT simplification, as described in the preceding section, exploits internal redundancy created by the constructive model construction method to simplify some GLIFT cells without affecting global model precision. However, the strict equivalence constraint still yields costly implementations. In this section, we relax the strict equivalence constraint, which exposes more simplification opportunities at the expense of some 'false-positive' detected flows in addition to those that are already present in the constructive GLIFT model. In other words, an imprecise GLIFT model may report a flow (i.e., an output with label `1`) when the original model reports that no such flow exists (i.e., the same output is labeled `0`). We describe techniques to formulate the 2QBF-SAT constraints to allow false positives for only some subset(s) of input combinations, where the solver automatically chooses the best such subsets (it is also possible to restrict false positives to only a subset of output labels, or to partially specify which subsets of input combinations are allowed to produce false positives). By simply changing one parameter to these constraints, the user can explore trade-offs between GLIFT model precision and complexity.

4.3.1 Explicit Acceptance by Bit Vectors

Let `(x)` denote a bit vector of length `|x|` whose bit string is given by `x`. A slot is a bit vector `(s)` containing an "input vector" and an "output mask", and describes a model input combination that may trigger a false positive, along with which output label(s) which may report a false-positive flow. When generating an imprecise GLIFT model, the designer specifies some number `N` of slots to use, thereby providing a degree of control over the amount of imprecision that may be added to the model. However, the *contents* of the slots are determined by the solver; these slots are existentially quantified in the 2QBF-SAT formulation. Still, if the user wishes to partially specify some content of the slots (i.e., assign some constant values instead of letting the solver choose), that is also possible.

Imprecision Acceptance Criteria

Fig. 4.5 illustrates several relevant aspects of the imprecise acceptance criteria. In this example, there are $N = 2$ slots, each of which has a corresponding output mask. For each bit in the output mask, a value of `1` indicates that a false positive is allowed in the corresponding output label, and a value of `0` indicates that a false positive is not allowed there.

Three errors occur in Fig. 4.5 that would not be accepted: First, a bit vector not matching any slot yields a false positive. Second, a bit vector that *does* match a slot yields a false positive in an output label whose corresponding bit in the output mask has a value of `0`. Third, the model generates a false negative in an allowable bit position, rather than a false positive. Our problem formulation ensures that these types of errors do not occur.

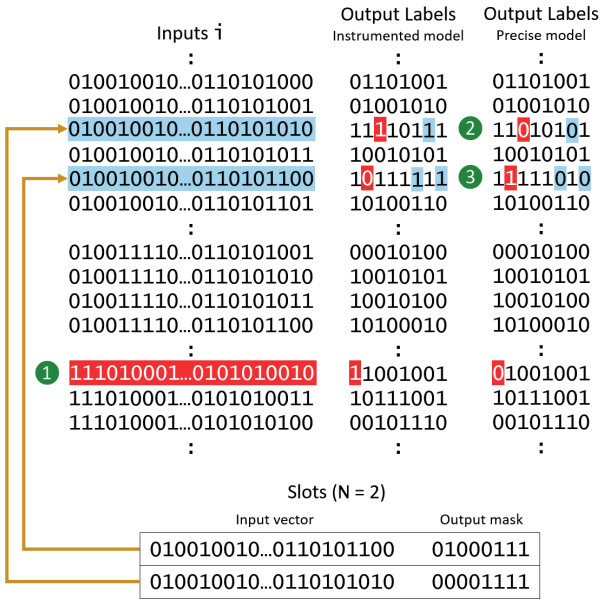


Figure 4.5 – The imprecision acceptance criteria for the explicit method. This is an example of an invalid supercell configuration, showing how the slots determine which imprecision-generating inputs are accepted. Here the instrumented model is not a valid approximation due to three problems (indicated in negative, red): (1) The instrumented model's outputs change under three input vectors but there are only two slots. (2) When the first input vector is applied, one false positive is not in a position allowed by the slot's output mask. (3) With the second input vector, one of the changes is not a false positive, but a false negative.

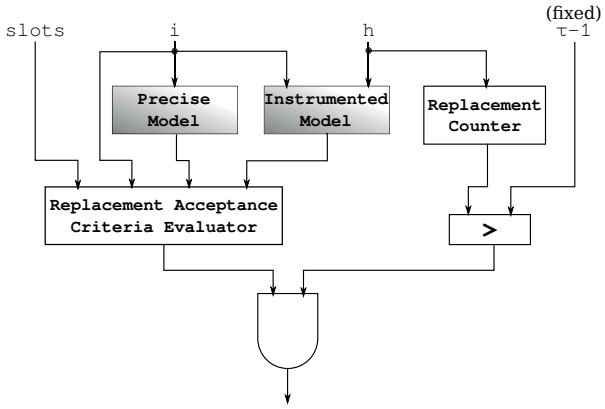


Figure 4.6 – A visualization of the 2QBF-SAT miter for imprecise GLIFT model simplification using the explicit method. The satisfiability solver tries to force the output of this circuit to 1 for all possible values of the primary inputs `i` under the constraint that the configuration found must use at least τ imprecise GLIFT cells. The solution, if one exists, provides an assignment for $\vec{h}$ (encoding the chosen configurations for the supercells) and `slots`, which limits global model imprecision. That limit is effected by the Replacement Acceptance Criteria Evaluator, whose operation is visualized in Fig. 4.5.

This is demonstrated in the acceptance function depicted in Fig. 4.6, which is a visualization of the ϕ function described earlier. The rules of this acceptance function can also be stated as: "it must be precise, except if the input combination is explicitly excluded from the equivalence constraint, that any imprecision is a false positive occurring in an allowable output label, and at least τ locally-imprecise GLIFT cells are substituted." The Replacement Acceptance Criteria Evaluation function ensures that the constraints encoded in the slots (whose contents are themselves determined by the solver) are checked. For input bit vectors that don't match a slot, the imprecise and original GLIFT models must produce identical labels; in case of a match, the labels may be identical or a false positive—but only if that false positive appears in an allowable output label. The user also provides an integer parameter, τ, which is a lower bound on the number of locally-imprecise GLIFT cell replacements to be made; the Replacement Counter and comparator (>) ensure that this lower bound is achieved. The `AND` gate at the bottom ensures that both criteria are satisfied: (1) the imprecise GLIFT model properly accounts for all $2^{|i|}$ slot and non-slot bit vectors; and (2) at least τ locally-imprecise GLIFT cells are used instead of precise GLIFT cells.

We encode GLIFT gate replacement configurations exactly as in the previous method, which was shown in Fig. 4.3(c). The Replacement Counter and comparator (>) ensure that a sufficient number of imprecise GLIFT supercells are chosen.

Our technique encodes the same GLIFT gate replacement configurations in $\vec{h}$ as the previous precise method, along with the contents of the slots, and the Replacement Counter and comparator ensures that enough GLIFT supercells have been configured to be imprecise than the required minimum threshold τ.

2QBF-SAT Formulation

The added elements of imprecision require an updated 2QBF-SAT problem formulation that goes beyond the precise formulation introduced in Section 4.2. The formulation for the function illustrated in Fig. 4.6 follows.

$$\exists(h, slots) \in \{0,1\}^m . \forall i \in \{0,1\}^n : \phi(h, slots, i) \tag{4.1}$$

Notice that the function ϕ in Equation 4.1 has no parameter τ, which is shown as an input in Fig. 4.6. We fix the value of τ for each 2QBF-SAT problem instance; we iteratively adjust τ and re-solve, using the by-now familiar binary search method to generate a sequence of models with progressively more replacements, eventually converging on a model that maximizes the number of replaced GLIFT cells while still adhering to the constraints described above.

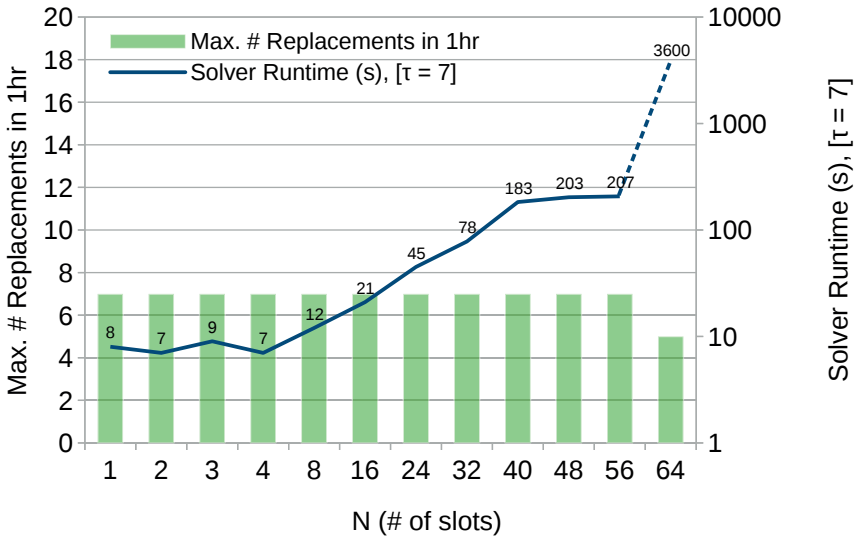


Figure 4.7 – A chart showing with columns (axis on left) the maximum number of GLIFT cell replacements and the 2QBF-SAT solver runtime to find those replacements for the `too_large` experiment with various `N` values (i.e., numbers of slots), given a 1-hour solver timeout, using the explicit method. The data at `N=64` shows that the solver was not even able to find the same seven replacements it could with fewer slots; it could only find five. Note how the solver runtime increases rapidly, and yet we find no additional GLIFT cell replacements.

Solver Runtime

Fig. 4.7 reports the runtime of the 2QBF-SAT solver as a function of the number of slots (`N`) provided by the user, given a one-hour time limit. For $N \leq 56$, the solver was able to replace seven cells in 207 seconds or less; for $N = 64$, the solver could only find five replacements within the allotted hour, so a dashed line is shown to the timeout. These results indicate that acceptance by bit vectors scales poorly.

4.3.2 Acceptance by Patterns

Our solution is to calculate acceptance not by input bit vectors, but by *patterns* of bit vectors. Acceptance by patterns allows each slot to encode allowable false positive flows for multiple bit vectors. A pattern includes one or more don't-care values encoded by an `X` in place of an individual bit, as shown in Fig. 4.8; a pattern with `j` don't-cares covers 2^j distinct input combinations.

At a high level, the 2QBF-SAT problem formulation given below is identical to that of the explicit method above.

$$\exists(h, slots) \in \{0,1\}^m . \forall i \in \{0,1\}^n : \phi(h, slots, i). \tag{4.2}$$

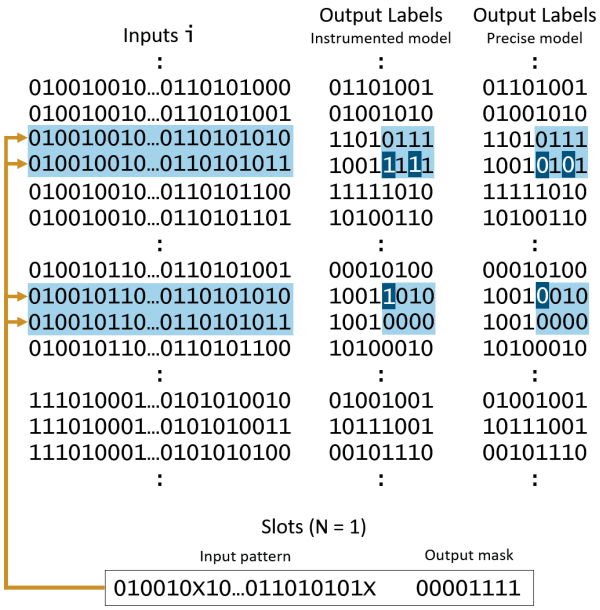


Figure 4.8 – A visualization of the acceptance criteria for the pattern imprecise simplification method. This figure shows how changes to the GLIFT model truth table (i.e., false positives) are allowed in the pattern method and how the estimated upper bound on additional false positives is computed. Individual input vectors are now replaced by patterns including don't-cares. Acceptable false positives (three in the example, indicated in negative, dark blue) must match at least one of the input patterns' covered rows and also in the output mask's columns.

Again, the contents of the slots are determined by the solver.

The difference between the two approaches lies in the exact formulation of the function ϕ, which is visualized below in Fig. 4.8. This figure, representing the new miter circuit representing a new acceptance function, is similar to the function shown in Fig. 4.6, but with a few key differences. The slot encoding allows for don't-care bits (not shown); the user specifies a parameter `MaxFP` which provides an upper limit on the number of unique input combinations that can be covered by patterns (note that multiple patterns may cover the same input combination); and a "Bound Evaluator" component, which enforces the aforementioned upper bound. This provides the user an extra degree of freedom in addition to the number of slots. Specifically, `MaxFP` can be interpreted as an upper bound on the number of output label cells in the truth table describing the model that may become a false positive (i.e., turn into `1`).

The truth table cell coverage for a given slot is computed as: $2^{\texttt{HW(input\_mask)}} \cdot \texttt{HW(output\_mask)}$ where `HW` is the Hamming weight function. The coverage value for each slot is then summed and compared to the provided threshold `MaxFP`.

The appropriate parameter value for `MaxFP` varies from design to design, and some sense of the number of *existing* flows in the design can help to bound the desired number of false-positive flows. Our approach is to simulate the original precise model with a relatively small number

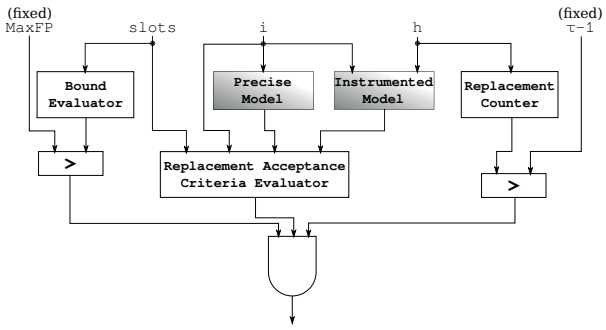


Figure 4.9 – A visualization of the 2QBF-SAT miter circuit for imprecise GLIFT model simplification using the patterns method. The satisfiability solver addresses essentially the same problem as the explicit method with two important differences: (1) The semantics of the slots are now changed to contain don't-care patterns, and the acceptance function is changed accordingly. (2) The upper bound on how imprecise the `h` can validly make the model is now limited by the constant parameter `MaxFP`. The criteria used in this circuit are illustrated in Fig. 4.8.

(2^{20}) of uniformly random input and input label values. We then scale the resulting number of observed `1`-valued output labels over all input combinations to the full size of the model input space $2^{|I|}$, where $|I|$ is the number of inputs and input labels. This provides a rough estimate of the number of original flows, allowing for quick calibration of the `MaxFP` parameter. As an example, the `alu4` benchmark circuit has 28 model inputs, so its `MaxFP` parameter is 256 times the number of flows sampled in the precise model. This is not a hard upper bound, and could easily overestimate the number of precise flows when scaled to the size of the full model input space.

4.4 Experimental Results

To demonstrate the potential of these approaches, we gathered a number of GLIFT benchmark circuits mostly derived from the IWLS benchmark suite [IWLS, 2005] and tested a script that automatically replaced GLIFT cells with supercells, iteratively constructed an appropriate 2QBF-SAT miter, invoked a 2QBF-SAT solver, generated the resulting simplified circuit, and continued searching for a more optimized circuit, if possible. This search procedure is sketched in pseudo-code in Fig. 4.10.

Due to the complexity of some benchmarks, the voluminous number of different configurations, and limited computing resources, some benchmarks were tested only with the precise simplification method.

```
delta := infinity
tau := None
N := //... (for the imprecise methods only)
MaxFP := //... (for the imprecise methods only)
solution = Solve(tau, N, MaxFP) //(precise w/o N, MaxFP)
min_fail := 0

if not solution:
    exit
while delta > 1:
    delta := (min_fail - max_succ)
    tau := max_succ + (delta / 2)
    solution := Solve(tau, N, MaxFP) //(or w/o N, MaxFP)
    if solution:
        max_succ := size(get_replacements(solution))
    else:
        min_fail := tau
```

Figure 4.10 – Pseudo-code for the solution space exploration algorithm used to find the best solution. This algorithm finds the GLIFT supercell configuration that results in the maximum number of replacements possible given the values for `N` false-positive explicit/pattern slots and a `MaxFP` bound on the percent of false-positives in the generated model.

4.4.1 Precise Simplification

Each of the benchmarks in Table 4.1 was simplified with the precise simplification method. The Area columns shows the original and resulting circuit area (as reported by ABC after mapping to the `mcnc` generic standard cell library) before and after precise simplification of the GLIFT model. Note that this library is purely synthetic; these numbers should only be compared in relation to each other, and not in absolute terms. The Cells/Updated columns show the number of GLIFT cells in the original model and the number that are simplified in the final resulting model. Finally, the Total Solver Time column shows the aggregate amount of 2QBF-SAT solver time spent during the entire simplification procedure, including failed

Table 4.1 – Complexity of GLIFT models before and after simplification.

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	2211	337	113	121
alu4	4706	4081	701	291	845
C3540	17014	14876	2477	851	134604
C5315	24019	21160	3706	1422	84778
C7552	20783	18260	3693	1716	67856
des	32295	26665	4672	2120	91381
i10	19912	18082	3119	951	63090
pair	13860	13613	2175	446	8283
t481	231	129	47	17	9
too_large	2118	2067	281	7	41
ttt2	1108	1073	168	29	10
x1	2132	2071	300	4	30

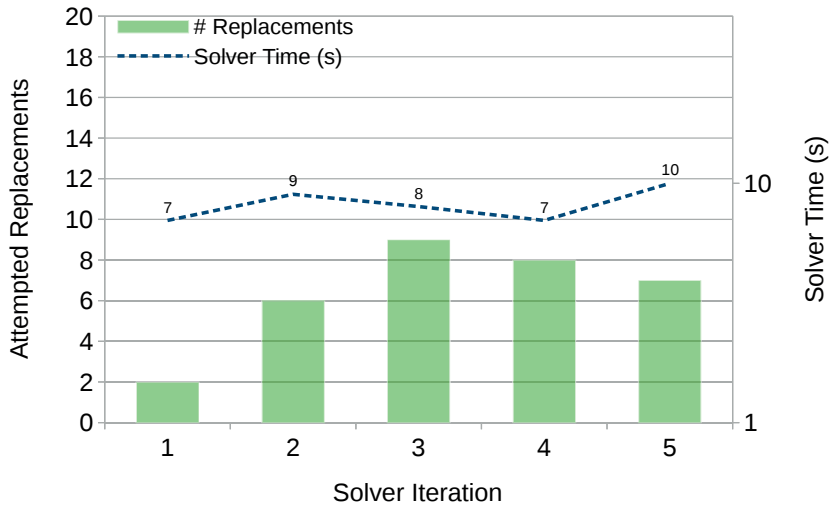


Figure 4.11 – The solver progression visualized for the precise simplification of the `too_large` benchmark. Each column represents a 2QBF-SAT instance for one iteration of the optimization procedure. The attempted constraint on the minimum number of replacements is shown in the height of each column; the associated solver time is shown by the dashed line.

instances (i.e., those with a `tau` that is too high). These results show a modest but significant reduction in model complexity. Notably, while solver time grows with model complexity, the structure of the circuit can have a large effect on the solver performance. Simply put, some models are more difficult to optimize than others. All experiments ran on Xeon E5-2680 v3 processors with at least 64GiB of available RAM, with no 2QBF-SAT instance timeout, using Yices 2.4.2 [Dutertre, 2014] in “exists-forall" mode.

Fig. 4.11 shows how the solving procedure illustrated in Fig. 4.10 works on the `too_large` benchmark. Columns, and the left vertical axis, describe the attempted `tau`, or minimum number of replacements, encoded in that iteration’s 2QBF-SAT miter. The dashed line shows the time spent by the 2QBF-SAT solver for each iteration. In some iterations, the 2QBF-SAT solver fails to find a solution. In this case, the `tau` parameter was too high; the next iteration will attempt a lower minimum number of replacements.

This benchmark is quickly solved in all iterations. Figure 4.12 shows how the solving procedure progresses for a much more complex GLIFT model: the `C7552` benchmark. Again, columns and the left vertical axis describe the iteration’s attempted `tau` value, and the dashed line describes the associated 2QBF-SAT solver time.

While this more complex model requires much more solver time to simplify, these figures show that this approach is reasonably scalable, and that solver times tend to be fairly consistent from iteration to iteration.

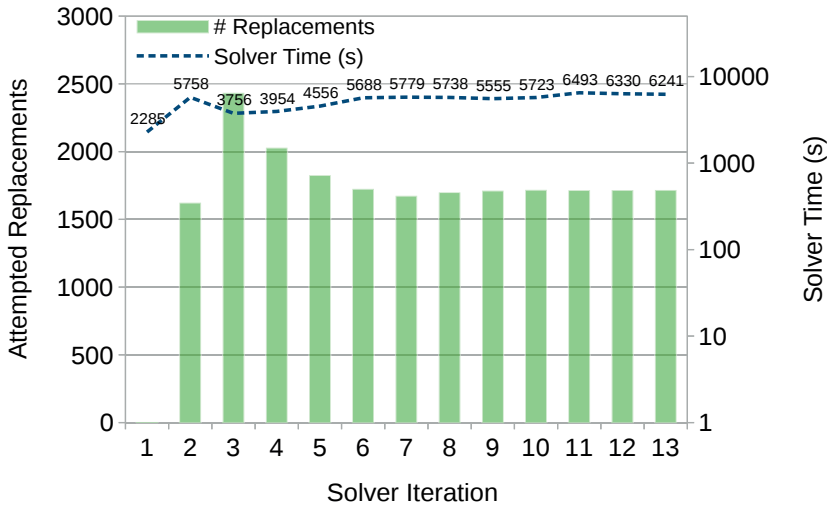


Figure 4.12 – The solver progression visualized for the precise simplification of the much more complex C7552 benchmark.

4.4.2 Imprecise Simplification

To empirically verify our claim that we can generate circuits with arbitrary trade-offs between added false positives and complexity, we must have a method to measure at least an estimate of the actual number of additional false-positive flows produced by a given imprecisely simplified model. To estimate, we use a set of designs from the standard IWLS benchmark set, and for each we use the same pseudo-randomly generated 2^{20} model input vectors used for estimating MaxFP, which were generated using Linear Feedback Shift Registers (LFSRs) with periods longer than 2^{20}. Then we simply count the number of flows detected and subtract the number of flows detected for the same sample with the precise model.

Due to the time and expense of an exhaustive exploration of the possible configuration space for each experiment, we employed a binary search method to find the maximum possible number of replacements given N (the number of slots) and MaxFP, described by the pseudo-code in Fig. 4.10, and ran multiple experiments varying MaxFP to estimate 80%, 60%, 40%, 20%, 10%, 5%, 2%, and 1% false positive rates. All experiments ran on Xeon E5-2680 v3 processors with at least 64GiB of available RAM, with a 2QBF-SAT instance timeout of 1 hour, using Yices 2.5.1 [Dutertre, 2014] in “exists-forall" mode.

The tables in Fig. 4.14 show the results of optimizing GLIFT models of various IWLS benchmarks. Each iteration of the optimization procedure ran with a timeout of one hour. While some of the reported runtimes are large, in the vast majority of cases the solver runtime is dominated by a few timed-out instances. A number of the C7552 instances failed to solve within the timeout on the first iteration; therefore, no optimized model was found. These data show clearly that varying the MaxFP parameter is effective at reducing area and increasing the number of replacements. Some instances, however—for example, the t481 instances with

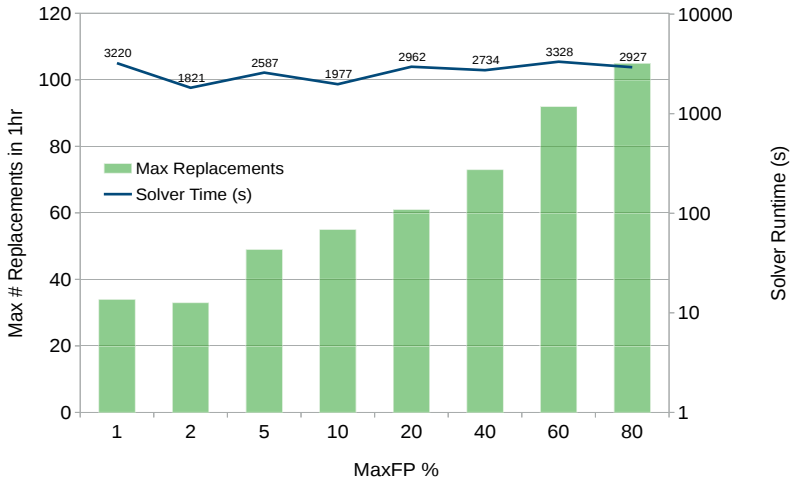


Figure 4.13 – A visualization of solver performance and effectiveness of the patterns imprecise simplification method. Similar to Fig. 4.7, but with `MaxFP` on the horizontal axis as the user-controlled variable, with two pattern slots, for `too_large`. Note how many more replacements (columns, left axis) are found than with the 'explicit' method, and the runtime (line, right axis) stability.

40% and 20% `MaxFP` parameters—show an interesting artifact: despite having the same (or in other instances, even fewer) number of GLIFT cell replacements, the area is reduced with a lower `MaxFP` bound. This is due to the fact that the procedure optimizes for number of GLIFT cell replacements, and not for area reduction. Future work might try to weight GLIFT supercell choices, in order to optimize more directly for area reduction; however, this may increase the 2QBF-SAT instance complexity and thus increase solver runtime.

Discussion

Fig. 4.13 serves as a counterpoint to Fig. 4.7. With the 'patterns' technique, we can find many times the number of replacements as 'explicit', and with more reasonable runtime, too. Some experiment instances used almost the entire allotted hour of solver time while others finished within minutes.

In Fig. 4.15 we show the actual area reduction achieved versus the measured false positive rate. The results reported here are only for the results of the search algorithm in Fig. 4.10 with two pattern slots, not intermediate steps. While this is still a busy chart, one can see how generally a higher measured false positive rate corresponds with a bigger area reduction. One can also see that the region between the "all-`OR`" data points is not quite covered. This is likely due to our approach maximizing the *number* of imprecise GLIFT cells, not the *simplicity* of imprecise GLIFT cells. We also suspect the visible "noise" in the `ttt2` data is due to the same cause. Future work will explore, for example, weighting each supercell option; this weight could be the number of gates in the chosen propagation function to potentially make a better proxy for

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	1495	337	313	22
alu4	4706	3113	701	616	313
C880	2116	1566	361	280	16970
C7552	20783	12962	3693	3167	19271
t481	231	116	47	24	26
too_large	2118	1789	281	105	15251
ttt2	1108	855	168	103	10050
x1	2132	1639	300	181	17145

(a) 80% `MaxFP`, N = 2

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	1617	337	310	49
alu4	4706	3006	701	609	1184
C880	2116	1637	361	243	17646
C7552	20783	18303	3693	1298	27113
t481	231	118	47	24	40
too_large	2118	1730	281	92	18757
ttt2	1108	917	168	81	12726
x1	2132	1641	300	157	12466

(b) 60% `MaxFP`, N = 2

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	1723	337	298	356
alu4	4706	3332	701	536	25228
C880	2116	1910	361	178	18670
C7552	20783	N/A	3693	N/A	3600
t481	231	136	47	22	51
too_large	2118	1854	281	73	19272
ttt2	1108	994	168	69	18194
x1	2132	1789	300	117	18857

(c) 40% `MaxFP`, N = 2

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	1830	337	253	15820
alu4	4706	3711	701	429	26018
C880	2116	2034	361	129	9938
C7552	20783	N/A	3693	N/A	3600
t481	231	122	47	22	71
too_large	2118	1954	281	61	15899
ttt2	1108	1011	168	52	9872
x1	2132	1900	300	76	20380

(d) 20% `MaxFP`, N = 2

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	1902	337	201	21543
alu4	4706	3836	701	376	35219
C880	2116	2065	361	119	16722
C7552	20783	N/A	3693	N/A	3600
t481	231	187	47	19	69
too_large	2118	1851	281	55	17363
ttt2	1108	1011	168	47	14452
x1	2132	2049	300	37	9803

(e) 10% `MaxFP`, N = 2

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	2066	337	171	23878
alu4	4706	4039	701	341	27765
C880	2116	2075	361	112	13116
C7552	20783	N/A	3693	N/A	3600
t481	231	206	47	19	197
too_large	2118	1940	281	49	13285
ttt2	1108	1024	168	44	18137
x1	2132	2030	300	25	11900

(f) 5% `MaxFP`, N = 2

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	2068	337	149	15726
alu4	4706	3941	701	319	30687
C880	2116	2038	361	106	17967
C7552	20783	N/A	3693	N/A	3600
t481	231	160	47	19	201
too_large	2118	1887	281	41	17211
ttt2	1108	1049	168	42	16803
x1	2132	2044	300	21	14743

(g) 2% `MaxFP`, N = 2

Benchmark	Area		Cells/Updated		Total Solver
	orig	simpl	orig	simpl	Time (s)
alu2	2480	2142	337	138	19091
alu4	4706	4056	701	309	24002
C880	2116	2053	361	99	21151
C7552	20783	19181	3693	1010	59537
t481	231	129	47	19	304
too_large	2118	1930	281	34	14392
ttt2	1108	1031	168	40	17372
x1	2132	2043	300	15	9966

(h) 1% `MaxFP`, N = 2

Figure 4.14 – Results for GLIFT models of various benchmarks optimized with the imprecise 'patterns' method, for varying `MaxFP` parameters and with two slots.

simplicity. Still, overall, these data show that we do effectively trade off complexity (by proxy of area) and the false positive rate for the generated GLIFT models.

In Fig. 4.16, we show the measured additional false positive rates for the same experiments versus the `MaxFP` parameter used. Here one can clearly make out that increasing the bounded false positive rate generally induces more aggressive imprecision.

Together, Fig. 4.15 and Fig. 4.16 show that not only can we trade off complexity and imprecision, but we have a controllable and flexible method to do so, as well.

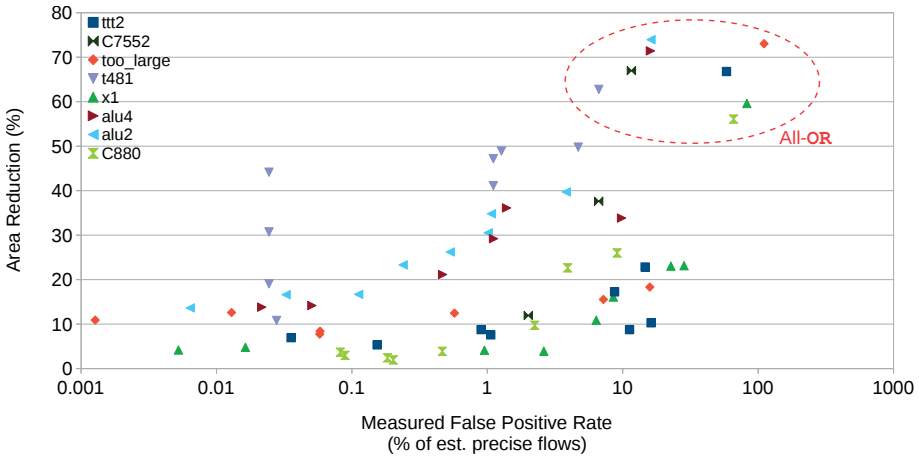


Figure 4.15 – A comparison of area reduction to the measured additional false positive rate (as a percentage of the original number of flows) among the sampled 2^{20} input vectors, given two pattern slots. Allowing extra false positives reduces the model's area, and we can generate models with arbitrary imprecision. The corresponding "all-OR" models are highlighted.

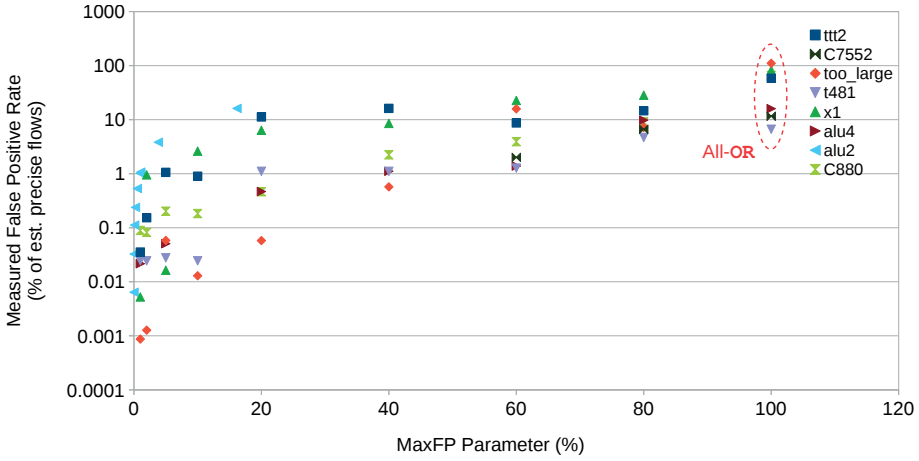


Figure 4.16 – The measured additional false positive rate (on a log axis) among the sampled 2^{20} input vectors versus the MaxFP parameter. The MaxFP bound is loose, but clearly effective at controlling the actual false positive rate. The corresponding "all-OR" models are again highlighted.

4.5 Related Work

Denning was one of the first researchers to take an information-theoretic approach to reasoning about security [Robling Denning, 1982]. McLean [McLean, 1990] and Gray [Gray III, 1992] pioneered the formalization of security properties using an information flow model. More recent research has applied information flow analysis across different layers of the computer system stack [Sabelfeld and Myers, 2003, Krohn et al., 2007, Suh et al., 2004, Tiwari et al., 2009, Zhang et al., 2012, 2015]. A number of these use information flow analysis to build secure hardware. For example, Tiwari et al. proposed a fine granularity information flow tracking method that enforces non-interference at the Boolean gate level [Tiwari et al., 2009]. Chiricescu et al. incorporated hardware assisted fine-grained flow tracking into a secure computer architecture in order to dynamically check security properties specified at the software level [Chiricescu et al., 2013].

Using the correct abstraction is an important factor in reducing the complexity of the security analysis [Li et al., 2010]. For instance, we could model the system at the register transfer level (RTL), and use RTL information flow analysis tools such as [Li et al., 2013, Zhang et al., 2012, 2015]. This would allow a designer to assign a one bit label to an entire multi-bit variable. That would speed up analysis, but the results would not present any bit-level flows. However, there are many scenarios that require a lower level (gate level) IFT model, like detecting some hardware Trojans [Hu et al., 2016b]. Again, picking the correct level of abstraction is an important decision for system security modeling, and one that provides a complementary approach that could be used to further inform the trade-off between the precision and complexity in GLIFT models.

Precision and complexity in GLIFT models are known to be contradictory modeling goals [Hu et al., 2012]. In practice, formally verifying properties of GLIFT models is intractable without accepting very low precision. Although the internal redundancy common in constructively-generated GLIFT logic has previously been noted [Hu et al., 2011, 2012], before the work in this chapter, designers not wishing simply to abandon GLIFT in favor of a higher-level model had few options, and most often used very imprecise models with “all `OR`" taint propagation [Bidmeshki and Makris, 2015].

There is some relevant work in the logic optimization domain. Mishchenko et al. use the complete don’t-care set for logic optimization [Mishchenko and Brayton, 2005]. Their technique uses a Boolean SAT solver to compute a complete don’t-care set in local reconvergent fanout regions and leverages these don’t-care conditions to optimize the design. This does not achieve the global optimization possible with the work presented in this chapter. Further, general logic synthesis tools significantly and uncontrollably change the structure of the design. As a result, the security labels of internal signals may be synthesized away, and information may flow in different ways. Our techniques preserve the security labels while optimizing the design and only introducing controlled amounts of imprecision when desired.

4.6 Conclusions

Gate-level information flow tracking offers the promise of verifying important security properties at the Boolean gate level. Unfortunately, precise GLIFT models are often too complex to practicably use for verifying security properties. Previous work mostly involved extreme simplifications like reducing all GLIFT taint propagation to `OR`; more recent work introduced some limited means of trading a small amount of precision for a reduction in complexity, but without any controllability. We present three methods to simplify GLIFT models: One precise simplification method that introduces no additional false positive flows, and two imprecise methods that allow a limited and controlled amount of imprecision in exchange for the ability to simplify the model further. These latter methods are the first methods known to the author to systematically generate imprecise GLIFT models with a controllable trade-off between precision and complexity, potentially allowing the use of more precise models than previously was possible and reducing the manual verification burden. While imprecision does not reduce security, it adds the burden of manually verifying all reported flows. Excessively imprecise models are of limited use because the false positive rate is very high, so the signal-to-noise ratio of the model queries is very low. In future work we hope to reduce the complexity of the false positive rate calculation logic while sacrificing as little controllability as possible, to assign weights to alternative propagation functions, and to demonstrate empirically how imprecise GLIFT models can help speed up verification in the real world.

The kind of bit-precise global optimization problem that this simplification represents is both extremely difficult for humans to reason about effectively and perfectly suited to automated reasoning by a 2QBF-SAT solver. This application shows definitively that the kind of automated circuit reasoning that 2QBF-SAT solvers can provide can be extremely useful for circuit analysis and optimization tasks and that designers and hardware security engineers should have easy access to them.

At this point, both the utility of 2QBF-SAT solvers in a variety of circuit design tasks and the overarching similarity between this and the previous applications contained in this thesis should be clear. Each of these applications shares common themes, including the need to formulate a 2QBF-SAT miter, the need to solve that miter, and the need to integrate the solver's results into some user-defined process that eventually generates a circuit. The applications in each of the preceding chapters were implemented in a diverse array of programming languages and with bespoke scripting systems. However, a significant amount of the labor involved in implementing these applications could have been avoided if there were some language available that was flexible enough to read, construct, and manipulate circuits and easily construct and solve 2QBF-SAT miters. Happily, the next chapter presents just such a language.

5 Solver-Aided Circuit Design and Optimization with Nasadiya

Then even nothingness was not, nor existence...
— Rigveda (10.129.1)

Wings are a constraint that makes it possible to fly.
— Robert Bringhurst

In Chapter 2, we explored how SKETCHILOG can be useful to create designs in the presence of explicit design uncertainties like missing constant values and small logic fragments. While this functionality is indeed useful, it only begins to scratch the surface of what is possible with a tight integration of circuit design and satisfiability solvers. The preceding chapters give some ideas of what is possible; these are somehow all independent applications that apply the same underlying satisfiability solver technology. What these applications all share are a few common requirements: the need to formulate a 2QBF-SAT miter circuit appropriate to the application, the need to transparently transform the internal circuit representation into a format comprehensible by a 2QBF-SAT solver, and the need to use the solver results to further some circuit analysis or generation problem.

In this chapter, we develop Nasadiya, a more powerful extension of the Scala-hosted Chisel domain-specific language [Bachrach et al., 2012] designed to address key limitations inherent in SKETCHILOG and provide a comprehensive framework for manipulation, analysis, and generation of circuits with the effortless assistance of 2QBF-SAT solvers. SKETCHILOG enables only one inflexible and implicit application of satisfiability to circuit design: finding "hole" values that induce functional equivalence between a circuit being designed and a "golden" reference circuit. In contrast, Nasadiya offers much more flexibility, and, thanks to the experience gained in the implementation of the preceding chapters, enables a novel approach to combinational circuit design that we refer to as *Solver-Aided Design*, which uses satisfiability solvers to help drive the design space exploration and generation of a design.

The purpose of solver-aided design is to help bridge the gap between designers' need to satisfy

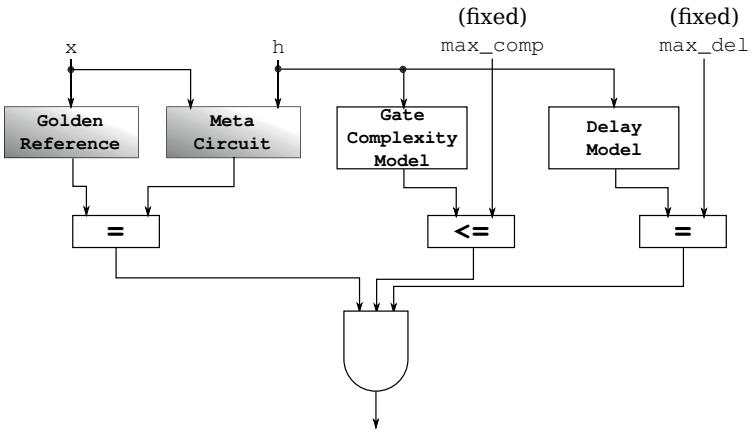


Figure 5.1 – An example 2QBF-SAT miter that can be easily built and solved in Nasadiya.

design constraints—whether those constraints are functional (e.g., equivalence checking) or extra-functional (circuit delay, area, side-channel resistance, etc.)—and the promise of satisfiability solvers to analyze circuit designs and determine how (if possible) to satisfy those constraints. Because the specific constraints designers must satisfy vary widely with design domain and goals, and also with how those constraints relate to decisions to be made in the design, Nasadiya aims primarily to facilitate the integration of satisfiability solvers into the design source code itself and leaves the designer the responsibility to direct *how* it is used.

Essentially, Nasadiya is a language with two goals in mind: generating Verilog descriptions of digital circuits, and constructing and solving 2QBF-SAT miters including those circuits and potential extra-functional models. Figure 5.1 visualizes such a miter; this miter includes not only a constraint on functional equivalence of the "Meta Circuit" (or sketch) and the "Golden Reference", but also rejects certain solver assignments to `h` if they cause the "Gate Complexity Model" or the "Delay Model" extra-functional models automatically built by Nasadiya to exceed specified fixed bounds.

Nasadiya provides a library and supporting syntax features which together make it easier for the designer to the specify how to construct these 2QBF-SAT miters (including building and referencing extra-functional models), to transparently translate those miters to a format comprehensible to a satisfiability solver, and to solve those miters and interpret the results appropriately in order eventually to emit the desired Verilog design.

Nasadiya is a fairly simple language extension to Chisel, and represents the synthesis of what the author learned from the implementation of applications in the previous chapters about obstacles in formulating applying 2QBF-SAT problems and solutions in various circuit design, debugging, and optimization contexts. It is both easy to use and expressive enough to describe every application in the preceding chapters of thesis.

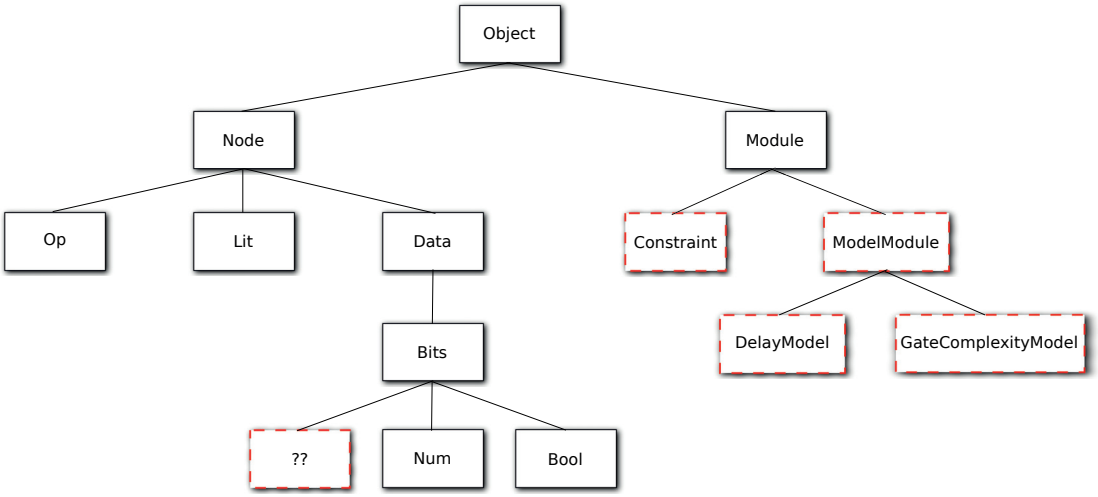


Figure 5.2 – Part of the Nasadiya object and class hierarchy, with red dashed boxes for those elements not included in standard Chisel.

5.1 Nasadiya

Nasadiya extends SketchiLog by introducing new objects, subclasses, and library functions.

Fig. 5.2 shows part of the object and class hierarchy in Nasadiya, with those parts not included in Chisel highlighted with a dashed red border. The primary additions are the `Module` subclasses `ModelModule` (along with its subclasses) and `Constraint`. However, Nasadiya also includes the standard hole support found in SketchiLog, including the `??` "raw hole" construct and the `either choose ... or` "meta mux" constructions to assist the designer to express architectural freedom. Nasadiya also provides the `Nasadiya` object, which contains the primary interface to the satisfiability solver backend; the complete interface is listed in Table 5.1. These additions represent key features that are treated more extensively below.

Fig. 5.3(b) shows the typical design flow for solver-aided design, where the *elaborate* phase from the standard Chisel flow in Fig. 5.3(a) is replaced with an *elaborate, translate, solve* loop. Unlike a more traditional design flow, where the elaborated design may be subjected to subsequent characterization and validation phases whose results then inform the next iteration of design, solver-aided design puts these phases inside design elaboration itself. This tight integration of modeling/characterization, constraint/miter formulation, translation to a format accepted by the satisfiability solver, solving, and further elaboration is the heart of solver-aided design.

To better conceptualize how these modules come together in solver-aided design, consider the toy example in Fig. 5.4. This figure shows a hypothetical 3-input multiplexer, a realistic logic gate representation of its implementation, and a more optimal architecture valid only under certain conditions. In Nasadiya, with the aid of a satisfiability solver and simple constraints, it's possible to utilize the more optimal architecture represented in Fig. 5.4(c)

Nasadiya Interface Function	Functionality
`init(): Unit`	Set up (or reset) the Nasadiya context to begin elaborating modules and instantiate and solve problems.
`elaborate(m: Module, s: String = null): Unit`	Instantiate the module `m`, using the provided hole assignments in the solution bit vector `s` (if provided), and, if `m` is a `Constraint` instance, generate the appropriate input/output interface signals.
`solve(p: => Constraint, t: Int): Option[String]`	Instantiate the `p` `Constraint` module, translate it to a representation accepted by the satisfiability solver, and invoke the satisfiability solver with the provided timeout `t` (in seconds, if provided). If a solution (i.e., a concrete value for each hole) is found within `t` seconds that satisfies the constraint expressed by `p` for all inputs, it is returned.

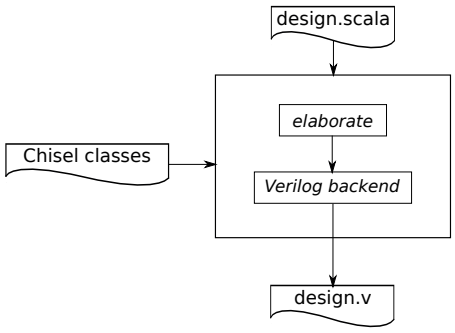
Table 5.1 – The public, designer-facing interface for the `Nasadiya` object.

automatically wherever possible, without the need for a designer to determine explicitly when such a substitution is permissible. Such a solver-aided design will formulate constraints over the design containing these modules, solve the constraints, and automatically determine which implementation to emit for each module instance, all without any need for manual designer specification or intervention.

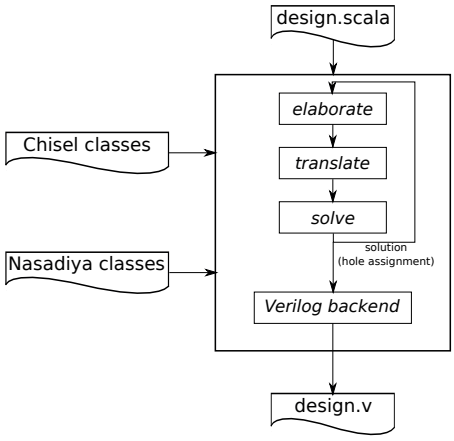
This could be achieved with the use of the construction demonstrated in Fig. 5.5 in place of the original 3-input multiplexer in Fig. 5.4(a). In this construction, the potential implementations of the multiplexer shown in Fig. 5.4(a) and Fig. 5.4(b) are selected by a special multiplexer selected directly by holes. We refer to this kind of hole-selected as a “meta mux”, because in the final design elaborated with concrete hole assignments, such multiplexers do not exist: These meta muxes exist only to express the possibility of selecting each of the sub-circuits appearing on the corresponding input. In this way, a designer can express local options for circuit architecture. Ensuring the validity of those options—or maximizing the global utility of chosen options—is left to the designer and the solver to ensure with the appropriate constraint formulation.

5.1.1 Integrated Modeling Library

The first key to solver-aided design is an easy-to-use and extensible hole-sensitive modeling facility. In order to make meaningful choices between design options, the impacts these options have on the resulting circuit must be modeled in a way that quantifies the utility of an option to the underlying solver. This allows designers to formulate constraints over extra-functional properties like pre-synthesis estimates of critical path length and circuit area. While there is little inherent novelty in the construction of these models, integrating modeling into the design language itself saves designers from the tedium, complexity, and engineering overhead of ad-hoc scripts with all their mundane details (e.g., parsing). In



(a) The standard Chisel flow is a straightforward design elaboration process. The top-level design module defined in `design.scala` is instantiated in the *elaborate* phase, and is written in Verilog to `design.v` by the *Verilog backend*.



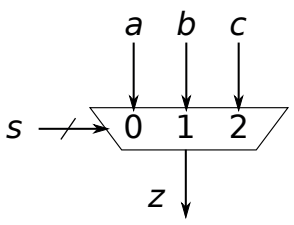
(b) The Nasadiya flow allows an intermediate elaborate-translate-solve loop, where design modules are instantiated, a satisfiability problem is formulated using them, and the problem is translated to a format suitable for the satisfiability solver. The result of the satisfiability solver is then used to drive further decisions in construction of the design. The resulting fully-specified module is then written by the *Verilog backend* to `design.v`.

Figure 5.3 – A diagram showing a high-level overview of the difference in the flow for a regular Chisel design (Figure (a)) and for solver-aided design with Nasadiya (Figure (b)).

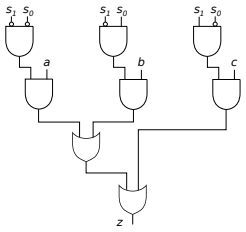
contrast, integrating modeling into the language gives direct first-class access to the object hierarchy describing the circuit to be modeled, greatly simplifying model construction and enabling extensibility for end users to customize provided modeling functions to suit their needs.

Because these models are logic circuits that take holes as inputs and produce one number (as a bit vector) as output, these model construction facilities are well-suited to design-space exploration with satisfiability solvers. In addition to saving the tedium typical of ad-hoc scripts, their instant availability alongside a satisfiability solver may provide an alternative to the kind of tedious manual architecture analysis typical [Aktan et al., 2015] for designers looking to optimize circuits for various extra-functional properties. Rather than manually formulating an algebraic description of gate depth in terms of explicit global parameters, designers might rely on ready-made models sensitive to locally-embedded architectural choices encoded by holes.

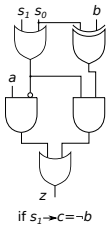
For example, consider the potentially optimized multiplexer contained in Fig. 5.4. Note that the architecture in Fig. 5.4(b) is always correct, while the architecture in Fig. 5.4(c) is better, but only correct under certain conditions. With only a standard equivalence check, as in SKETCHILOG, there is no way to ensure the more optimal architecture in Fig. 5.4(c) is ever chosen; even *if* the conditions for its equivalence to Fig. 5.4(b) are satisfied, *both* Fig. 5.4(b) and



(a) The module, a multiplexer with 3 inputs (assume that `s` can never be 3) whose architecture might be optimized with the help of a satisfiability solver.



(b) A reasonable example of how this module might naturally be implemented, with (excluding inverters) 8 gates and 4 gate delays from s_1 to the output.



(c) A possible optimized implementation, which is valid if $s_1 \implies c = \neg b$. Excluding inverters, this architecture has only 5 gates and 3 gate delays from s_1 to the output.

Figure 5.4 – A simple toy example of a module and two possible implementations for which solver-aided design might be useful.

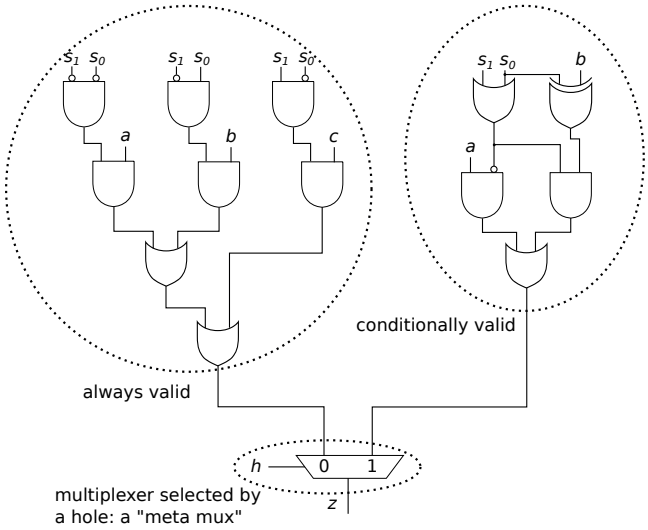


Figure 5.5 – A “meta mux” construction that could be used in the place of the three input multiplexer in Fig. 5.4(a), selecting between the always-valid implementation from Fig. 5.4(b) on input `0` and the conditionally-valid implementation from Fig. 5.4(c) on input `1`. This is called a “meta mux” because the final multiplexer and one of the implementations will be eliminated once the satisfiability solver assigns a concrete value to the hole h on the select line. In other words, this “meta mux” is used only to express design options, and will not actually be present in the final design.

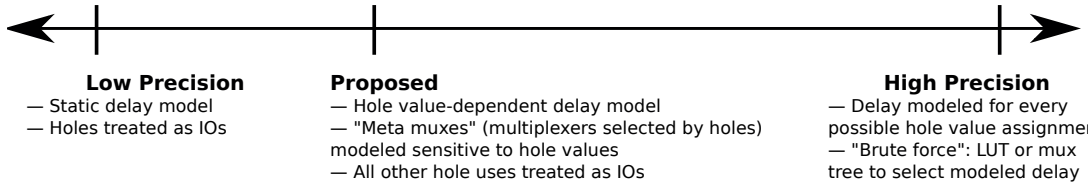


Figure 5.6 – A visualization of the trade-off space for hole-based circuit delay modeling. Nasadiya's approach is detailed as "Proposed".

Fig. 5.4(c) are equivalent. By providing integrated modeling that can in turn be constrained, it's relatively easy to ensure that the more optimal architecture of Fig. 5.4(c) is chosen where possible, because its models will be superior to the architecture in Fig. 5.4(b).

Integrated modeling makes it possible to apply satisfiability solvers to discriminate between equally correct choices with different extra-functional utilities, and the modeling functionality provided by Nasadiya means designers can immediately make use of them rather than spending time to develop bespoke models. The overall effect is to expand the applicability of satisfiability to different design constraints, to reduce mental burden, and to enable a greater focus on improving or developing the design.

Delay Modeling

Perhaps the most obvious kind of model is a delay model. Nasadiya provides an extensible and ready-to-use delay modeling functionality with the `DelayModel` subclass of the `ModelModule` class.

This `DelayModel` class is constructed with respect to some desired component to be modeled. In principle, it performs the rough equivalent to a run-of-the-mill pre-synthesis static delay analysis—but with a twist. Because each different assignment for a hole value can induce radically different resulting design architectures, this delay model needs to be sensitive to the special nature of holes in modules—optimizing a function only makes sense if that function actually varies over its inputs. In other words, hole values here are the 'knobs' for a solver to 'turn' in order to try to satisfy a constraint imposed by some function of those hole values. Clearly there is nothing to be accomplished by turning those knobs if they are completely disconnected from that constraint.

However, there is a small complication: Model construction must trade off between precision and model complexity. Further, this trade-off must be done along two distinct axes: hole-based design delay model complexity vs. precision, and typical static delay model complexity vs. precision.

The first trade-off axis is unique to solver-aided design with holes. Because holes are only an intermediate part of the design—they do not exist in the final generated design—and yet their values may effectively change the circuit structure, we have established that the delay

```
class NewDelayModel()(mod: Module) extends DelayModel()(mod) {
  override def score_node(n: ChiselNode): Int = {
    n match{
      case op: Op => {
        if ("&|^~!".contains(op.operation)) {
          return 1
        }
        else if(op.operation == "=="){
          return 2 //XOR the inputs, then NOR those bits.
        }
        else return 0 //some other Op node, like wire manipulation
      }
      case m: Mux => {
        if (isMetaMux(m)) return 0 //a "meta mux"
        else return 3 //a regular 2-input multiplexer
      }
      case _ => return 0 //some other kind of Node, like a type node
    }
  }
}
```

Figure 5.7 – An example demonstrating how to extend the delay modeling functionality in Nasadiya. While the provided `score_node` function here is simple and, for exposition purposes, neither accurate nor perfectly syntactically correct, it really is this simple to override the node scoring functionality in order to customize Nasadiya's model construction.

model of such a design should be sensitive to the values assigned to the holes. Ideally, the delay model should model with perfect precision, given a set of hole values, the static critical path delay in the circuit resulting from specializing the design with those hole assignments.

In practice, the trade-off space for this axis is visualized in Fig. 5.6. On the left, we see the lowest-precision (but also lowest-complexity) approach, which is to simply treat holes as circuit inputs in a traditional static timing analysis [Devadas et al., 1991]. This simplistic kind of model may of course give very different results from those of a model of the final generated circuit; it is not hole-sensitive and therefore unsuitable for solver-aided design.

On the right we see the highest-precision (but also highest-complexity) approach, which is to perform fully precise static delay modeling of the design (e.g., [Bahar et al., 1994]) *for each possible value for the holes*, and then to select among them with "brute force", using a look-up table or multiplexer tree. Clearly, for non-trivial designs with even a modest number of hole bits, such a model is unfeasible either to create or to use.

Towards the left, labeled "Proposed", we see the point in the trade-off that Nasadiya targets. This approach embeds hole-selected multiplexers directly into the delay model, reflecting one key use of holes: to choose between different potential circuit structures, which may have different delays. If the modeled design uses a multiplexer selected directly by a hole (a meta mux), the different sub-circuits on each input will be modeled and those sub-circuit models will be selected by that same hole with a multiplexer in the delay model.

Lower-level uses of holes, however, like bare `AND` gates with one hole input, will not be treated specially: Such holes will be considered like any other circuit input and the value of that hole will not affect the modeled delay at the `AND` gate. It's worth explicitly emphasizing that this

means that Nasadiya's delay modeling is not fully precise for any arbitrary use of holes. If the hole on the input of an `AND` gate is assigned value 0, that `AND` gate should have a modeled delay of 0. However, in this case, because Nasadiya's delay models are only sensitive to hole values at meta muxes, Nasadiya would model the worst case delay *regardless of that hole value*, and thus the modeled delay will be the sum of the delay for this gate and the modeled delay of whatever signal appears on the `AND` gate's other input.

The second axis is familiar to any designer who ever built a delay model. For example, designers could model delay by simply counting the gates along a path, by also distinguishing between different gate types, by additionally considering estimated post-synthesis gate capacitance, or any of a number of other precision-enhancing effects. Considering these effects increases the complexity of the model, but also increases its precision.

In Nasadiya, the delay model uses a simple heuristic by default: one two-input gate has delay 1. However, the designer may provide his or her own delay calculation function, like the example shown in Fig. 5.7, where he or she can instead assign any arbitrary delay to each Chisel node element in the Chisel object graph. For clarity, note that Fig. 5.7 is merely an example. The default Chisel node scoring function built into Nasadiya handles more Chisel node types and with greater accuracy, and while it is more verbose, it is similarly simple. In such a case of a custom node scoring function, the rest of the delay model construction remains unchanged. Even designers wishing to use different (likely more precise) delays can still transparently take full advantage of the rest of the hole-sensitive delay modeling infrastructure, saving the designer from having to develop the entire modeling algorithm and supporting code from scratch.

We believe our approach represents an agreeable trade-off in the space described by Fig. 5.6: such "meta mux" structures are the most natural primitives for expressing alternative (sub-)circuit architectures, the complexity of resulting models is limited, and crucially, unlike traditional static delay modeling, the estimated delay computed is sensitive to value changes in the signals representing holes. This value sensitivity for certain signals is essential to determine the conditions (i.e., concrete values assigned to hole signals) under which sub-circuits would be effectively cut off from the rest of the design; this may radically alter the design's delay, so it is crucial for model accuracy.

On the other hand, while the model must be precise enough to account for potentially radical variation under different hole assignments, there is limited utility to increasing precision. These models are constructed based on the Chisel node hierarchy representing the design—they are pre-*elaboration* models, not just pre-synthesis models. While delay estimates produced by these models should still be strongly correlated with those of models of the post-physical-layout design, there are diminishing returns to increasing precision of inherently imprecise models.

A sketch of the algorithm Nasadiya uses to build delay models is shown in Fig. 5.8. This function is used to perform a bottom-up (from outputs to inputs) depth-first traversal of

```
def traverse(n: ChiselNode) = {
    if(isMetaMux(n)){
        inputs = n.inputs.filter(n.select_line)
        foreach(in <- inputs) traverse(in)
    } else {
        foreach(in <- n.inputs) traverse(in)
    }

    node_score = score_node(n) //user-extensible scoring function
    calculated_del = List()

    if(type(n) == Op and "&|^~".contains(n.operation)){
        for(i <- 0 until n.width){
            in_bits_delays = foreach(in <- n.inputs) delay_nodes(in)(i)
            calculated_del += AddModelNode(
                                (ConstModelNode(node_score),
                                MaxModelNode(in_bits_delays)))
        }
    }
    else if(type(n) == Op and n.operation == "=="){
        in_bits_delays = List()
        foreach(in <- n.inputs) in_bits_delays += delay_nodes(in)
        calculated_del += AddModelNode(
                            (ConstModelNode(node_score),
                            MaxModelNode(in_bits_delays)))
    }
    else if(type(n) == Mux){
        select_node = n.inputs(0)
        in0 = n.inputs(1)
        in1 = n.inputs(2)

        if(isMetaMux(n)){
            for(i <- 0 until n.width){
                calculated_del += MuxModelNode(
                                    LogicModelNode(select_node),
                                    delay_nodes(in0)(i),
                                    delay_nodes(in1)(i))
            }
        }
        else{
            sel_del = MaxModelNode(delay_nodes(select_node))
            for(i <- 0 until n.width){
                i0_del = delay_nodes(in0)(i)
                i1_del = delay_nodes(in1)(i)
                calculated_del += AddModelNode(
                                    (ConstModelNode(node_score),
                                    MaxModelNode((sel_del, i0_del, i1_del))))
            }
        }
    }
    else if(type(n) == Op and n.operation == "##"){
        foreach(in <- n.inputs) calculated_del += delay_nodes(in)
    }
    else if(type(n) == Extract){
        in = n.inputs(0)
        idx_hi = n.inputs(1)
        idx_lo = n.inputs(2)
        foreach(i <- idx_lo to idx_hi) calculated_del += delay_nodes(in)(i)
    }
    else{
        if(n.inputs.size == 0){
            //n is an IO node: probably 0 delay
            foreach(i <- 0 until n.width) calculated_del += ConstModelNode(node_score)
        }
        else{
            //type nodes, etc.  Assume no delay; just pass through the delay from the input
            calculated_del = delay_nodes(n.inputs(0))
        }
    }

    delay_nodes(n) = calculated_del
}
```

Figure 5.8 – Pseudo-code for the depth-first traversal function used to build delay models.

Delay Calculation Node	Value
`ConstModelNode(d: Int)`	d
`AddModelNode(l: List[ModelNode])`	$\sum_{i=0}^{\lvert l\rvert-1} l_i$
`MaxModelNode(l: List[ModelNode])`	$\max_{0\le i\le\lvert l\rvert} l_i$
`MuxModelNode(s: LogicModelNode, a: ModelNode, b: ModelNode)`	$\begin{cases} a & s=0 \\ b & s=1 \end{cases}$
`LogicModelNode(n: ChiselNode)`	$n \in \{0,1\}$

Table 5.2 – A list of delay calculation nodes used to build abstract delay models. `LogicModelNode` is a special case, which is used to embed a Boolean logic expression inside the delay model itself, and is used to express the select line of each `MuxModelNode`.

the Chisel object graph, and associates delay calculation nodes (see Table 5.2 below) with each object in the Chisel object graph. After calling this function on each output of the Chisel `Module` to be modeled, a subsequent pass on the resulting delay calculation graph implements each delay calculation node in Chisel, creating the actual Boolean logic for the `DelayModel ModelModule`; this `ModelModule` can then be treated like any other Chisel `Module`.

However, there are a few complicating points: First, so-called "meta mux" structures (i.e., multiplexers selected by holes) are treated specially; second, the delay analysis is bit-precise, while each Chisel node may represent elements of varying bit-widths. Thus, each Chisel object maps to a list of delay calculation nodes—one for each bit in the valid range of the Chisel object—and meta muxes encountered map to delay calculation nodes that select delays of their input sub-circuits according to the value of the hole(s) on the select line, rather than mapping to delay calculation nodes that calculate the maximum delay of their input sub-circuits, summed with the delay of the multiplexer itself.

A table of the available delay calculation nodes is shown in Table 5.2. These delay calculation nodes are, for the most part, straightforward. The `MuxModelNode` and `LogicModelNode` delay calculation nodes would not normally be used in static delay modeling: these node types exist solely to handle the "meta mux" structures that are treated specially by Nasadiya's delay modeling. For example, the "meta mux" in Fig. 5.5 would be modeled by implementing a `MuxModelNode` that selects the delay from the inputs of the "meta mux" according to the value of a `LogicModelNode` that replicates the select line of the "meta mux". Thus, under a concrete hole configuration that selects input `0` of the "meta mux"—meaning only the sub-circuit on input `0` *actually* exists in the resulting circuit—the delay calculation node for that "meta mux" represents only the delay for the input `0` sub-circuit. The delay calculation nodes for the input `1` sub-circuit are irrelevant when the select line of the "meta mux" does not select input `1`. Of course, in the alternative case when the concrete hole value on the "meta mux" selects its input `1`, the `MuxModelNode` selects the delay calculation node for input `1` and ignores the delay calculation nodes for input `0`.

Gate Complexity Modeling

Nasadiya also provides another type of modeling as a part of its standard library: gate complexity modeling. This type of model is used to determine the sum total, given a hole configuration, of gate complexities in the design. In other words, this type of modeling first determines which gates will actually exist in the final design (so, will not be eliminated by constant propagation of concrete hole assignments through "meta muxes"), then sums the node score for those nodes which are determined actually to exist in the final design.

By default, Nasadiya calculates gates' scores by a simple heuristic: a gate's score is equal to its number of inputs. Thus, a two-input `AND` gate has score 2; an inverter has score 1. As with Nasadiya's provided delay models, the designer may provide his or her own scoring function to model gate complexity differently without needing to re-implement the entire model.

Here again we are forced to choose a point in the trade-off space between model complexity and precision. Nasadiya uses the same approach for gate complexity modeling as for delay modeling: special "meta mux" constructions are treated specially, while other uses of holes are treated as regular circuit inputs.

The gate complexity modeling algorithm in Nasadiya can be separated into two phases. First, each Chisel node in the design is mapped to `PathConditions`, or concrete assignments to holes for which that Chisel node would actually exist in the final design, along with the range of bit indices of that Chisel node for which the `PathCondition` applies. Second, all `PathConditions` are aggregated, Chisel nodes are sorted into classes of nodes that are exposed under the same `PathConditions`, and summing the node scores of all class members to determine the weight of each class. The final model value is computed by summing the weights of those classes whose `PathConditions` are `true` given concrete hole values.

The basic idea behind these `PathConditions` is best demonstrated with the aid of Fig. 5.9. Note that the circuit shown in this figure has only one output, `io_z` on the far left. Immediately preceding this output is a "meta mux" selected by the hole h. As explained earlier, this "meta mux" structure means that only one of the sub-circuits (inside either of the shaded boxes labeled, "Weight 11" and "Weight 20") on the "meta mux" inputs will actually exist in the final circuit. In this example, the "Weight 11" sub-circuit has the `PathCondition` $h == 1$ and valid bit range `[0, 0]`, while the "Weight 20" sub-circuit has the `PathCondition` $h == 0$ and valid bit range `[0, 0]`. When a concrete assignment to the hole signal selecting the "meta mux" selects a sub-circuit, it satisfies one of these `PathConditions`, and we say that that sub-circuit is *exposed*.

Figure 5.10 highlights how the sub-circuit on input `1` of the "meta mux" is exposed when the hole selecting the "meta mux", h, has a concrete value of `1`, thus satisfying the `PathCondition` $h == 1$. Under this hole assignment, the final circuit will have only this sub-circuit (labeled "Weight 11"); the other unexposed sub-circuit and the "meta mux" itself will be eliminated by constant propagation. Many designs, however, have more than one output, more than one

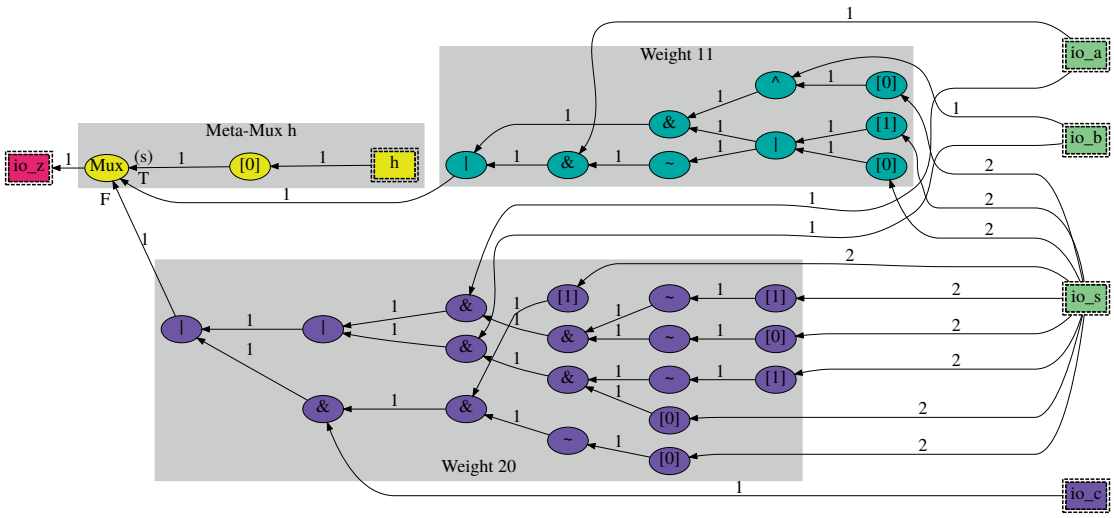


Figure 5.9 – A visualization of the Chisel node graph for the "meta mux" construction in Fig. 5.5, overlaid with the classes comprising the gate complexity model. Nodes, representing the Chisel nodes in the module for Fig. 5.5, that have the same color belong to the same class. Edges are labeled with the width of the signal connecting the two elements. Note that it is possible for different bit indices of the same node to belong to different classes, if different hole values might expose some bit indices of a node but not others. Weights of classes exposed under a given hole assignment are summed to produce the model value.

"meta mux", and may only partially expose some Chisel nodes—only some bit indices of those nodes will be exposed. This is why the first phase of the gate complexity modeling algorithm maps each Chisel node to a set of `PathConditions`, the satisfaction of any of which will expose the node, and why `PathConditions` are only valid for a specified range of bit indices.

While the analysis of the provided design is rather involved, the final model is simple. Figure 5.11 visualizes the final circuit structure of the gate complexity model for the same example circuit. The model is implemented merely as a sum of the weights of all classes, where each class weight is gated by a mux selected by the mutual disjunction of the associated set of `PathConditions`. In this case, each class only had one `PathCondition` in its set; if a class has more, each `PathCondition` is `OR`'ed to compute the final value of the select line for the class's associated mux.

Figure 5.12 sketches the first phase of the algorithm used to build gate complexity models: mapping Chisel nodes to `PathConditions`. This depth-first traversal of the Chisel node graph begins at the outputs, with an initial `PathCondition` of 1 (i.e., always exposed) and a bit range including all bit indices for that output. For example, if there is an output `io_z` that is three bits wide, the traversal begins with a `PathCondition` of 1 (this output will exist in the final design regardless of the hole values for which it is specialized) and a valid bit range of `[2, 0]` (all bit indices of this node will be visible).

As the traversal continues towards the inputs, the `PathCondition` changes only when crossing

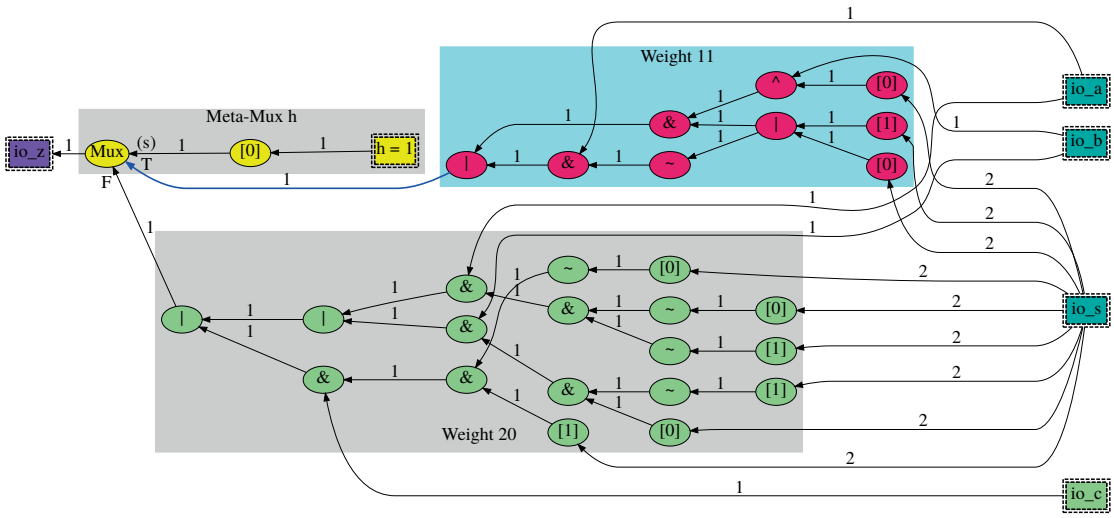


Figure 5.10 – A visualization of the Chisel node graph and gate complexity model information created for the "meta mux" construction in Fig. 5.5 when the hole *h* has a concrete value of `1`, thus selecting the conditionally-valid implementation of Fig. 5.4(c). Note that only the corresponding complexity class (with weight 11) is highlighted: this is the only part of the circuit that actually survives to the final design.

a "meta mux": The `PathCondition` for the sub-circuit on its inputs becomes the conjunction of the current `PathCondition` with the select line or its inverse, as appropriate. This is because for the sub-circuit on one of the "meta mux" inputs to be exposed, the `PathCondition` that exposes the output of the "meta mux" must be true *in addition to* the select line selecting that sub-circuit.

The valid bit range is updated more frequently during traversal. Wire concatenation and extraction must be accounted for by shifting the current valid bit range as appropriate. The select line of a regular multiplexer (not a "meta mux", but a multiplexer that is an architectural part of the final design) is treated specially: this one-bit signal always has the valid bit range `[0, 0]`, because it must be exposed to select *any* bit(s) of either of the inputs.

5.1.2 Arbitrary Constraint Specification

Integrated modeling facilities can be very useful, but without the ability to specify exactly what he or she wishes to constrain, the designer cannot make much use of them. Thus the second key feature provided by Nasadiya is support for construction of arbitrary constraints in a simple and straightforward manner.

In the limited context of designing a circuit for which a golden reference circuit is available, mere equivalence constraints are useful for finding a valid circuit design. However, in this context, when a golden reference circuit is available, the overarching goal motivating the design of the new circuit is usually not just to find a *valid* circuit, but to find a *better* circuit.

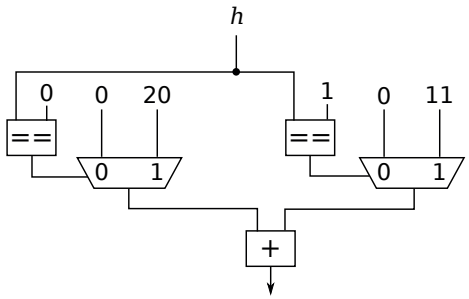


Figure 5.11 – A visualization of the gate complexity model created for the "meta mux" construction in Fig. 5.5. Note the simple structure: the model is just a sum of all weights of selected classes from Figs. 5.9 and 5.10.

While SKETCHILOG is well suited to the former, the latter is a kind of solver-aided design for which no pre-existing tools are well suited.

The essential insight here is that what is most useful to a designer is not just to find concrete values for uncertainties, but to allow meaningful choice between architectural options. Exactly what meaning these choices have depends entirely on the designer's goal. Allowing the designer to specify arbitrary constraints—quite possibly formulated with the aid of the aforementioned modeling facilities—enables discrimination between valid solutions and more fully unlocks the potential of satisfiability solvers to aid reaching specific parts of the design space.

To support this functionality, Nasadiya provides a `Module` subclass called `Constraint`. This `Constraint` subclass is to be extended by the designer to implement the desired constraints, and represents the miter circuit that will be provided to the satisfiability solver. This special component provides behind-the-scenes support to make the designer's job easy.

Part of this support is the `addModule()` function, which takes a `Module` as a parameter and automatically re-connects that `Module`'s inputs to the miter inputs; when `addModule()` is called multiple times with `Modules` with identical input names, those inputs are connected to the same miter inputs. In addition, `Modules` containing holes reference the same hole signals by default, although this can be disabled. Combined with the transparent ability to functionally constrain those `Modules`' outputs with comparison operators (`===`, `<=`, `>`, etc.), it is easy to compare the behavior of multiple circuits—or meta circuits containing holes and/or "meta muxes"—under the same input conditions. Specifying the functionality of the miter output (effectively, the actual constraint implemented) is achieved by assigning the desired logic function to a special signal, called `assert_ok`.

These features are demonstrated in Fig. 5.13, which builds delay and gate complexity models of a `MetaCircuit Module` with "meta muxes", and constrains its functionality to equivalence with a `GoldenReferenceCircuit Module`, and also constrains the modeled delay and gate complexity to provided targets.

```
def traverse(n: ChiselNode, pc_rng: PathCondition) = {
    node_pcs(n) += pc_rng
    pc = pc_rng.first
    range = pc_rng.second

    if(type(n) == Extract){ //Update ranges across wire extraction nodes
        in = n.inputs(0)
        idx_hi = n.inputs(1)
        idx_lo = n.inputs(2)

        new_rng = conjoin([range.top + idx_lo, range.bot + idx_lo], [idx_hi, idx_lo])
        traverse(in, (pc, new_rng))
    }
    else if(type(n) == Op and n.operation == "##"){ //Wire concatenation
        left = n.inputs(0)
        l_sz = left.width
        right = n.inputs(1)
        r_sz = right.width

        left_rng = conjoin([range.top - r_sz, range.bot - r_sz], [l_sz - 1, 0])
        right_rng = conjoin(range, [r_sz - 1, 0])

        traverse(left, (pc, left_rng))
        traverse(right, (pc, right_rng))
    }
    else if(type(n) == Mux){ //Multiplexer
        select_node = n.inputs(0)
        in0 = n.inputs(1)
        in1 = n.inputs(2)

        if(isMetaMux(n)){ //Update ranges across meta muxes
            new_pc0 = pc & ~select_node
            new_pc1 = pc & select_node
            new_rng0 = conjoin(range, [in0.width - 1, 0])
            new_rng1 = conjoin(range, [in1.width - 1, 0])

            traverse(in0, (new_pc0, new_rng0))
            traverse(in1, (new_pc1, new_rng1))
        }
        else{ //Regular multiplexer--handle select line specially
            traverse(select_node, (pc, [0, 0]))
            traverse(in0, pc_rng)
            traverse(in1, pc_rng)
        }
    }
    else{ //All other node types: just pass through the path condition and range
        for(in <- n.inputs) traverse(in, pc_rng)
    }
}
```

Figure 5.12 – Pseudo-code for the depth-first traversal function used to build gate complexity models.

5.1.3 Virtualized Solver Access

Finally, the keystone of the solver-aided design approach enabled by Nasadiya is the explicit facility for invoking a satisfiability solver *during design elaboration itself*, a feature we call *virtualized solver access*. Embedding a satisfiability solver as a first-class participant in the design process allows designers to formulate and solve multiple satisfiability problems to inform architectural choices during design creation, and, crucially, eliminates the typical overhead of shuffling circuit representations around (e.g., from structural Verilog to formats comprehensible by the satisfiability solver).

Coupled with the ability to specify arbitrary constraints, this particularly enables iterative design techniques, where constraints are progressively relaxed or tightened, allowing auto-

```
class DelayComplexityThreshold()(delay: Int, max_complexity: Int) extends Constraint {
    val uut = addModule(new MetaCircuit())
    val spec = addModule(new GoldenReferenceCircuit())

    val delay_model = addModule(new DelayModel()(uut))
    val gate_complexity_model = addModule(new GateComplexityModel()(uut))

    val correct = uut === spec
    val delay_ok = delay_model === delay // '<=' also OK
    val complexity_ok = gate_complexity_model <= max_complexity
    assert_ok := correct & delay_ok & complexity_ok
}
```

Figure 5.13 – An example showing an arbitrary constraint over a circuit and its models.

```
class GateComplexityThreshold()(max_complexity: Int) extends Constraint {
    val uut = addModule(new MetaCircuit())
    val spec = addModule(new GoldenReferenceCircuit())

    val gate_complexity_model = addModule(new GateComplexityModel()(uut))

    val correct = uut === spec
    val complexity_ok = gate_complexity_model <= max_complexity
    assert_ok := correct & complexity_ok
}
```

Figure 5.14 – An example constraint that might be used as part of a strategy to maximize the number of conditionally-valid three-input multiplexers from Fig. 5.4(c) used in `MetaCircuit`.

matic design-space exploration. By handing control over solver invocation to the designer, the designer is freed from the limited applicability of SketchiLog. This hands-off approach turns Nasadiya into a fully general language suitable for nearly any application of satisfiability solvers to combinational circuit design.

As an example, suppose that the `MetaCircuit Module` above contains multiple instances of the "meta mux" construction from Fig. 5.5, and the designer wishes to use the optimized version (that appears on input `1` of this "meta mux", and is shown in Fig. 5.4(c)) of the three-input multiplexer whenever possible, without affecting the circuit functionality. Also suppose that the `GoldenReferenceCircuit Module` above contains an identical implementation, except it uses the always-valid three-input multiplexer implementation from Fig. 5.4(b) rather than the "meta mux" construction from Fig. 5.5.

In this case, the designer might use a constraint like the one in Fig. 5.14, which is broadly similar to 5.13. With this constraint, the designer can employ Nasadiya's gate complexity modeling functionality to help implement the most possible optimized three-input multiplexers: Because the optimized (and only conditionally-valid) three-input multiplexers have a lower modeled gate complexity than the always-valid original three-input multiplexers from Fig. 5.4(b), it stands to reason that finding a concrete assignment to "meta mux" hole values that minimizes `max_complexity` will also find the assignment to hole values that maximizes the number of optimized three-input multiplexers used.

However, a single solution to this constraint will not achieve the objective of minimizing the gate complexity model value: The satisfiability solver only reports if the constraint is

```
def BisectionDriver(): Unit = {
    var minSuccess = 0
    var maxFailure = 0
    var max_complexity = pow2(ModelingParams.max_model_width) - 1
    while (maxFailure == 0 || minSuccess - maxFailure > 1) {
        Nasadiya.init()
        println("Now trying complexity: " + max_complexity)

        val instance = new GateComplexityThreshold()(max_complexity)
        Nasadiya.solve(instance) match {
            case None => {
                maxFailure = max_complexity
                println("Failed to find a solution!")
            }
            case Some(sol) => {
                minSuccess = max_complexity
                println("Found solution \"" + sol + "\" of complexity at most " + minSuccess)
            }
        }
        max_complexity = maxFailure + abs(minSuccess - maxFailure) / 2
    }
}
```

Figure 5.15 – An example driver that uses virtualized solver access to iteratively minimize the `GateComplexityThreshold` constraint's `max_complexity` parameter with a binary search-based approach.

satisfiable, *given a single `max_complexity` parameter*. Thus, it is necessary to use an iterative process, where the same constraint is solved multiple times with varying values for the `max_complexity` parameter.

Figure 5.15 shows an example of the driver code to find the hole configuration for the `GateComplexityThreshold` constraint's `MetaCircuit` module (see `uut` in Fig. 5.14) that minimizes the gate complexity estimated by the `GateComplexityThreshold` constraint's `GateComplexityModel` module (see `gate_complexity_model` in Fig. 5.14), while ensuring the `GoldenReferenceCircuit` and the resulting solved `MetaCircuit` are equivalent.

5.2 Related Work

As SketchiLog was inspired by Sketch [Solar-Lezama et al., 2006], it's natural that Nasadiya is also closely related to it. The embedding of a 2QBF-SAT solver (although implicit in Sketch) and transparent construction of miters to complete programs or designs at the programming language level came, for this author, from Sketch.

Other recent work has explored more explicit and flexible ways to integrate satisfiability solving into programming languages; for example, metaSMT [Riener et al., 2017] offers a handy solver abstraction layer in a domain-specific language embedded in C++. Other authors have integrated the Z3 satisfiability solver into Scala [Köksal et al., 2011]. The transparent and natural availability of powerful automated reasoning systems directly in widely-used programming languages reduces the effort required to integrate satisfiability solving into applications. However, unlike the Nasadiya language introduced in Chapter 5, these are not designed to reason about or produce hardware circuits, and only support quantifier-free logic:

2QBF-SAT problems are not natively expressible or solvable.

More typically, language-level integrations of satisfiability solvers focus on specifying constraints for or otherwise aiding debugging or formal verification of software. KLEE [Cadar et al., 2008] is a symbolic execution engine that uses satisfiability solvers to reason about execution paths in software programs. While KLEE is most often used for test case generation, debugging, and reverse engineering, it also offers an Application Programming Interface (API) for software programmers to harness KLEE's solver capabilities directly in their code: the provided `klee_assume()` function allows programmers to specify arbitrary constraints over values in the program, which the solver (invoked transparently by the execution engine) proves are satisfied before continuing symbolic execution (or reporting an error and halting execution otherwise). This limited solver integration allows programmers the use the power of a satisfiability solver to prove invariants and find magic values that satisfy arbitrary conditions in some limited cases. Automated reasoning aids like this can be very helpful in the software development process. However, while specifying constraints is supported, user code cannot change the manner in which the satisfiability solver is applied: user code does not support first-class access to the solver, and there is no support for 2QBF-SAT, both of which limit the tool's utility.

Liquid Haskell [Vazou et al., 2014] integrates a programming language's type system with user-supplied invariants and employs a satisfiability solver to guarantee those invariants hold. Many others provide facilities to specify program pre- and post-conditions, loop invariants, constrained types, verifying compilers, etc. to help program verification [Brady, 2013, Pearce and Groves, 2013]. Π-Ware is a language hosted in Agda [Flor et al., 2015] that enables one specification to be both executable and synthesizable (i.e., to hardware); it also integrates the type system with a satisfiability solver and allows the expression and implicit verification of constraints over dependent types, helping to ease design of hardware that is "correct by construction".

5.3 Case Study: Power-Efficient Parallel Prefix Adders

To demonstrate concretely how all these facilities can be used to solve real-world problems, in this section we develop a solver-aided design generator in Nasadiya for a power-efficient parallel prefix adder. The architecture design space for binary adders is well-developed [Zimmermann, 1998, Ercegovac and Lang, 2004]. For minimal delay, modern adders typically use some sort of parallel prefix computation scheme [Ercegovac and Lang, 2004]; one of the most common is the *Ling Adder* [Naffziger, 1996, Ling, 1966].

As described in Sec. 2.4.1, in a parallel prefix adder, each carry-in bit to a full adder that computes a final sum bit is calculated by the prefix tree. This contrasts with a *ripple-carry* adder, where each carry-in bit is computed by the full adder in the next least significant bit position. While the critical path delay through a parallel prefix adder is significantly less than the critical path delay through a ripple-carry adder, parallel prefix adders use considerably

more power.

Recent research from Aktan et al. has explored the design space for so-called *sparse* parallel prefix adders [Aktan et al., 2015]. A sparse parallel prefix adder is distinguished from a regular parallel prefix adder by computing only *some* of the carry-in bits to the final full adders; those bit positions not calculated by the prefix tree use as carry-in the carry-out signal from the next least significant bit position, as in a ripple-carry adder. By reducing the number of carry-in bits calculated by the prefix tree, power and area are reduced. However, too much sparsity can increase the critical path delay if the ripple-carry chains in the final full adders are too long.

In order to determine the sparsity that minimizes the power consumed by the adder but that does not increase critical path delay, the authors developed by hand analytic models of the delay, gate complexity, and wire complexity of a Ling parallel prefix adder according to the specified level of sparsity.

5.3.1 Design

We show how Nasadiya can help automatically find design architectures that optimize a given objective function, and how Nasadiya's provided modeling functionality can be used to avoid potentially man-months of labor building analytically models. To this end, we re-implemented the Ling adder design Aktan et al. studied in Nasadiya, sketching the sum block that depends on adder sparsity, and used Nasadiya's built-in modeling capabilities to automatically generate models of the circuit delay and complexity. These automatically generated models obviate the need for manual analysis: with solver-aided design these implicitly parameterized models can be integrated directly into a constraint specification that is then solved during the design elaboration.

In effect, we show how Nasadiya's support for holes, constraint specification and solving, and automatic hole-sensitive model generation can be used easily to achieve the same goal that required months of manual human analysis.

The top-level module, called `LingKoggeStonePar`, and whose generator code is listed in Fig. 5.16, builds a mostly-standard complete (i.e., no sparsity) Ling adder. However, the `either choose` construct is used to select between two possible implementations of the final block of full adders: Either ripple-carry adders are used (as described above), or non-sparse parallel prefix adders are used (as described by Aktan et al., and a better choice for large sparsities). The effects of this design choice are fully captured in the automatically-created models; thus, assuming the correct formulation of constraints around those models, the solution will automatically choose the "best" implementation.

However, this design implements a *complete* prefix tree. The sparsity value is effectively sketched, however, in `RCSparseSumBlock` and `PPSparseSumBlock`. The code for building the ripple-carry implementation of the block of final full adders is listed in Fig. 5.17. It is this component which is responsible for determining sparsity values and implementing the

```
object LingKoggeStonePar{
    def apply(w: Int, a: Bits, b: Bits, cin: Bits): (Bits, Bits) = { //(sum, cout)
        val ph_sigs = new ArrayBuffer[ArrayBuffer[Prefix]]
        (0 until log2Up(w) + 1) map { _ => ph_sigs += new ArrayBuffer[Prefix] }

        val cout_sigs = new ArrayBuffer[Bits]
        val g = new ArrayBuffer[Bits]
        val t = new ArrayBuffer[Bits]

        //build g and t:
        for(i <- 0 until w + 1){
            if(i > 0){
                g += a(i-1) & b(i-1)
                t += a(i-1) | b(i-1)
            } else {
                g += cin
                t += Bits("b1", 1)
            }
        }

        //build first level of carry tree:
        for(i <- 0 until w + 1){
            if(i == 0) ph_sigs(0)      += prefix(t(0), g(0)) //carry in
            else if(i == 1) ph_sigs(0) += prefix(t(i-1), g(i) | g(i-1)) //white rhomboid
            else ph_sigs(0)   += prefix(t(i-1) & t(i-2), g(i) | g(i-1)) //black rhomboid
        }

        //flesh out each level in the adder:
        for(cur_level <- 1 until log2Up(w) + 1){
            //connect cur_level's prefixes:
            for(i <- 0 until pow2(cur_level)){
                val ph = ph_sigs(cur_level - 1)(i)
                val ph_h = get_H(ph)
                val ph_t = get_t(ph)
                ph_sigs(cur_level)  += ph //white circles / buffers
            }//nothing more to do for these bits; they inherit what was already computed

            for(i <- pow2(cur_level) to w){
                val left = ph_sigs(cur_level - 1)(i)
                val right = ph_sigs(cur_level - 1)(i - pow2(cur_level))
                val ph = prefix(left, right) //black circles
                val ph_h = get_H(ph)
                val ph_t = get_t(ph)
                TWLModel.set_input_wire_cost(ph_h, 2 * pow2(cur_level) + 1)
                TWLModel.set_input_wire_cost(ph_t, 2 * pow2(cur_level) + 1)
                ph_sigs(cur_level) += ph
            }
        }

        //Prepare carry signals for the sum block:
        for (i <- 0 until w + 1) {
            if (i == 0) cout_sigs += cin else if (i != w) {
                val ti = t(i)
                val hi = get_H(ph_sigs.last(i))
                cout_sigs += ti & hi
            }
        }

        //Calculate carry-out:
        val cout = t(w) & get_H(ph_sigs.last(w))

        //Make the final choice between each option's bit positions and calculate the sum
          bits:
        val rc_sum = RCSparseSumBlock(w, a, b, cout_sigs)
        val pp_sum = PPSparseSumBlock(w, a, b, cout_sigs)
        val sum = either choose rc_sum or pp_sum //sketching: let the solver choose!
        return (sum, cout)
    }
}
```

Figure 5.16 – The code for building a Ling adder suitable for solver-aided design and optimization. `RCSparseSumBlock` and `PPSparseSumBlock` implement sketched modules which select with "meta muxes" only some of the prefix tree results. Thus the exact architecture is determined by the sketched design's hole values.

```
def apply(w: Int, a: Bits, b: Bits, cin: IndexedSeq[Bits]): Bits = {
        val sum_0 = new ArrayBuffer[Bits]
        val sum_1 = new ArrayBuffer[Bits]
        val sum_out = new ArrayBuffer[Bits]
        val carry_out_0 = new ArrayBuffer[Bits]
        val carry_out_1 = new ArrayBuffer[Bits]

        //We will build two chains of full adders, like a Carry Select adder, but with each
        //fadd carry-in supplied not by const "0" or const "1", but by a MUX selecting the
        //appropriate constant (0 or 1) *or* the previous full adder's carry-out.

        //Build both carry chains:
        for (i <- 0 until w) {
            val cin_0 = if (i == 0) Bits("b0", 1) else {
                val in0 = Bits("b0", 1)
                val in1 = carry_out_0.last

                either choose in0 or in1 //sketching: let the solver choose!
            }

            val cin_1 = if (i == 0) Bits("b1", 1) else {
                val in0 = Bits("b1", 1)
                val in1 = carry_out_1.last

                either choose in0 or in1 //sketching: let the solver choose!
            }

            val (sum_0_i, cout_0_i) = FullAdder(a(i), b(i), cin_0)
            val (sum_1_i, cout_1_i) = FullAdder(a(i), b(i), cin_1)
            sum_0 += sum_0_i
            sum_1 += sum_1_i
            carry_out_0 += cout_0_i
            carry_out_1 += cout_1_i
        }

        //Build the final MUXes to select the real sum bits:
        for(i <- 0 until w) {
            //Now select among the possible sparse carry signals, to determine which signal
            //to use as the select line to the final "carry select" Mux:
            val idx = ??(log2Up(cin.size)) //sketching: create an explicitly named hole
            val select_line = either choose cin using idx //sketching: let the solver choose!
            sum_out += Mux(select_line, sum_1(i), sum_0(i))
        }

        return Cat(sum_out.reverse.head, sum_out.reverse.tail: _*)
    }
}
```

Figure 5.17 – The code to build the ripple-carry implementation of the sparse sum block for the `LingKoggeStonePar` module of Fig. 5.16. This is the code that sketches the adder's sparsity.

correct set of carry chains to compute the final sum. First, at each index, two full adders are instantiated (one for the `0` side and one for the `1` side of a carry-select adder). Each full adder's carry-in signal is the output of a "meta mux" that chooses between the appropriate constant value (signifying a choice to start a new carry chain because the corresponding carry from the prefix tree has been selected) or the carry-out signal of the full adder in the previous bit position (signifying a choice to continue the carry chains, because this position is 'sparse': the corresponding carry from the prefix tree is not selected).

In this way, the effective sparsity of the circuit is sketched, so the sparsity of the resulting adder design is implicitly encoded in the values the solver chooses for the holes selecting these "meta muxes".

Finally, a row of multiplexers selects between the `0` and `1` sides of the generated carry-select

```
class ComplexityThreshold()(min_delay: Int, tau: Int) extends Constraint {
    val uut = addModule(new LingKoggeStonePar()(w = ppadder_globals.w))
    val spec = addModule(new RippleCarryAdder()(w = ppadder_globals.w))

    val delay_model = addModule(new DelayModel()(uut))
    val gate_complexity_model = addModule(new TGCModel()(uut))
    val wire_complexity_model = addModule(new TWLModel()(uut))

    val correct = uut === spec
    val complexity_ok = (gate_complexity_model + (wire_complexity_model >> UFix(1))) <= UFix(
      tau)
    val delay_ok = delay_model === UFix(min_delay) // '<=' also OK
    assert_ok := correct & complexity_ok & delay_ok
}
```

Figure 5.18 – The `Constraint` used to minimize modeled circuit power: the constraint's single primary output is true when the circuit is equivalent to a simple ripple-carry adder, when its modeled delay meets the minimum specified delay `min_delay`, and when its modeled gate and wire complexities meet the maximum specified `tau`.

structure, with the index of the selecting carry signal from the prefix tree determined by the solver. In this way, when the solver chooses to connect the carry-in signals of bit position `i` to the carry-out signals of bit position `i - 1` (i.e., when bit position `i` is sparse), the solver will also choose to select the final multiplexer with the same prefix tree-provided carry signal as was used for the rest of the carry chain.

5.3.2 Constraints

The design is constrained by one of two constraints: a `ComplexityThreshold`, as described in Fig. 5.18, and a `DelayThreshold` which is similar but does not include the gate or wire complexity models and requires only correctness and a modeled delay value that meets the specified threshold `tau`. Compare the constraint described by the code in Fig. 5.14 and the constraint described by the code in Fig. 5.18. While the constraints differ in function, modules, and models used, their descriptions are both immediately plain and very succinct: Neither constraint description occupies more than a dozen lines of code.

The process by which these constraints are used iteratively to find the optimal value for `tau` should by now be familiar. Figure 5.19 shows the driver code used to iteratively formulate and solve these constraints, until the minimum possible modeled complexity is found given the minimum possible modeled delay.

The solutions that result from this search will be treated below; those solutions near the minimum complexity value should correspond to concrete Ling adder designs with sparsity configurations that minimize circuit power, as the gate and wire complexity models are designed to serve as a proxy to circuit power [Aktan et al., 2015].

```
object PPAdderDriver {
    def apply(): Unit = {
        Nasadiya.init()

        var minSuccess: Int = 0
        var maxFailure: Int = 0
        var cur_delay_tau: Int = pow2(ModelingParams.max_model_width) - 1
        var cur_comp_tau: Int = pow2(ModelingParams.max_model_width) - 1
        val timeout_sec = 60 * 60 * 1 //1h solver timeout
        var s: String = null

        //Do a binary search to find the minimum value for the delay:
        while (maxFailure == 0 || minSuccess - maxFailure > 1) {
            println("Now trying delay: " + cur_delay_tau)
            val instance = new DelayThreshold()(cur_delay_tau)
            Nasadiya.solve(instance, timeout_sec) match {
                case None => maxFailure = cur_delay_tau; println("Failed at " + maxFailure)
                case Some(sol) => {
                    s = sol
                    val adder = new LingKoggeStonePar()(w = width)
                    Nasadiya.elaborate(adder, s)
                    val delay_model = new DelayModel()(adder)
                    minSuccess = delay_model.evaluate(s)

                    println("Success: " + minSuccess + "!\n" + s)
                    println(s)
                }
            }

            cur_delay_tau = maxFailure + abs(minSuccess - maxFailure) / 2
            Nasadiya.init()
        }
        val min_delay = minSuccess
        println("Minimum delay achieved: " + min_delay)

        //Same for comp:
        minSuccess = 0
        maxFailure = 0
        while (maxFailure == 0 || minSuccess - maxFailure > 1) {
            println("Now trying complexity: " + cur_comp_tau)
            val instance = new ComplexityThreshold()(min_delay, cur_comp_tau)
            Nasadiya.solve(instance, timeout_sec) match {
                case None => maxFailure = cur_comp_tau; println("Failed at " + maxFailure)
                case Some(sol) => {
                    s = sol

                    val adder = new LingKoggeStonePar()(w = width)
                    Nasadiya.elaborate(adder, s)
                    val gate_complexity_model = new TGCModel()(adder)
                    val gate_comp = gate_complexity_model.evaluate(s)
                    val wire_complexity_model = new TWLModel()(adder)
                    val wire_comp = (wire_complexity_model.evaluate(s)) / 2
                    minSuccess = gate_comp + wire_comp

                    println("Success: " + minSuccess + ",G=" + gate_comp + ",W=" + wire_comp)
                    println(s)
                }
            }

            cur_comp_tau = maxFailure + abs(minSuccess - maxFailure) / 2
            Nasadiya.init()
        }
        val min_comp = minSuccess
        assert(s != null, "No solution found.")

        println("Minimum comp achieved: " + min_comp)
        println("Best solution:\n" + s)
    }
}
```

Figure 5.19 – The driver code used to discover solutions that optimize the `ComplexityThreshold` constraint, to find the lowest-delay Ling adder with sparsity choices that minimize modeled circuit power.

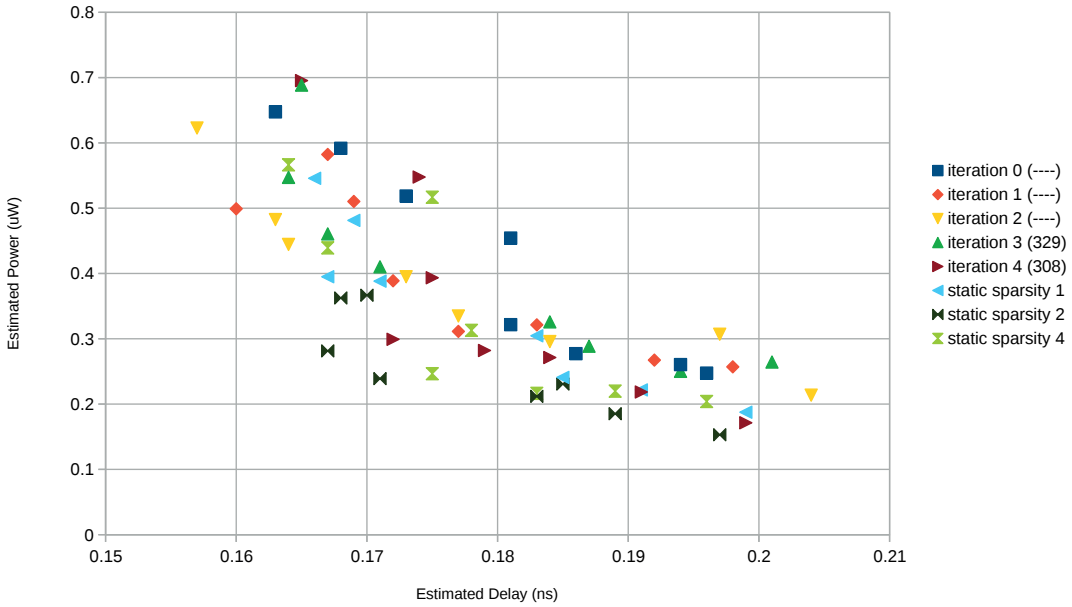


Figure 5.20 – Power (μW) vs. delay (ns) for all intermediate 8-bit adders listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. Curves are also included for statically-generated adders with four different sparsities for comparison. Note that successive iterations find usually-Pareto-dominant architectures.

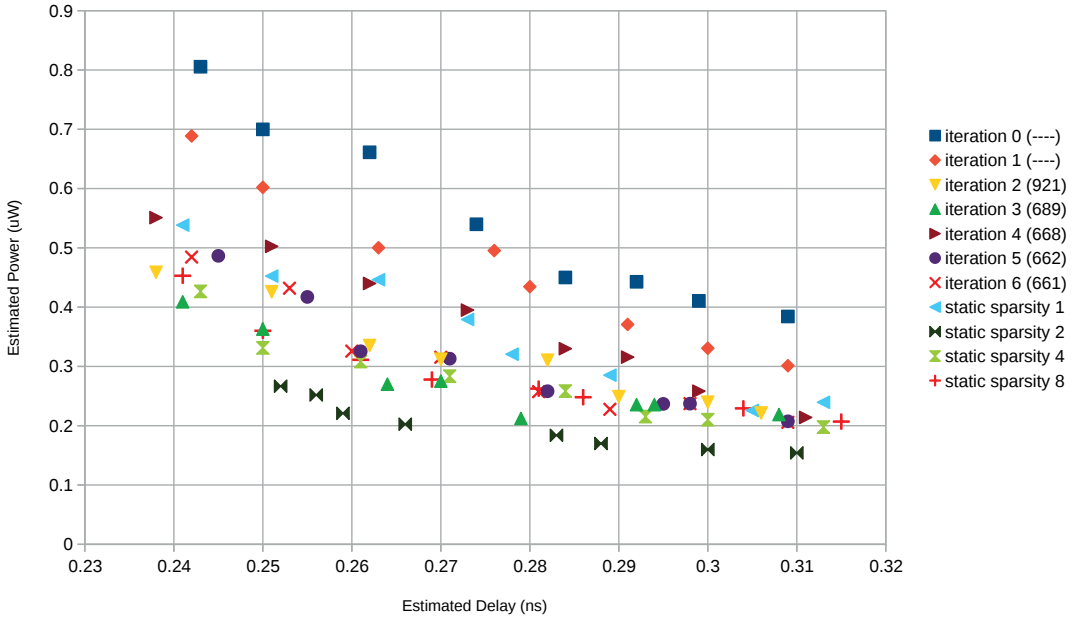


Figure 5.21 – Power (μW) vs. delay (ns) for all intermediate 16-bit adders listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. Curves are also included for statically-generated adders with four different sparsities for comparison. Again note that successive iterations find usually-Pareto-dominant architectures.

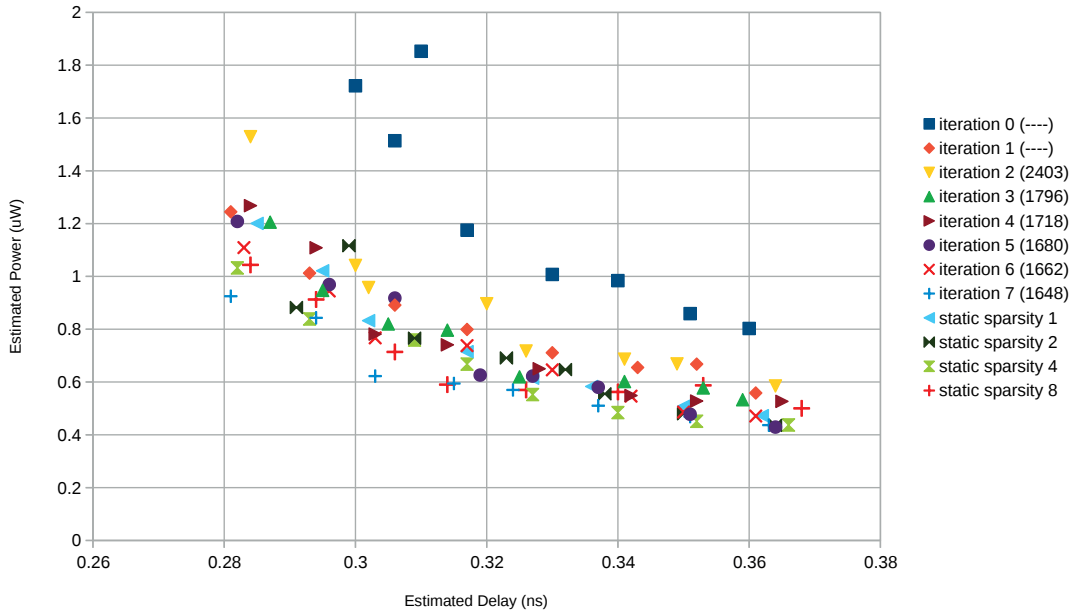


Figure 5.22 – Power (μW) vs. delay (ns) for all intermediate 32-bit adders listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. Curves are also included for statically-generated adders with four different sparsities for comparison. Note that not only do successive iterations find usually-Pareto-dominant architectures, but in some cases and at some points in the design space, the discovered architectures are superior to manually-optimized architectures.

5.3.3 Evaluation

To evaluate the effectiveness of solver-aided design for generating resulting adders that do indeed minimize power, as the models are intended to represent, we run the driver in Fig. 5.19 three times on an Intel Xeon™ E5-2698 v4 with 128GiB of memory, each time with a per-iteration solver timeout of 3600 seconds (one hour). The solver was Yices 2.5.1 [Dutertre, 2014].

We then collected the intermediary and final resulting circuits (each corresponding to a solution found in the driver), synthesized them with Synopsys DesignCompiler, placed and routed them in Cadence Innovus 16.1 using the general-purpose TSMC 40nm technology. Power, area, and delay were estimated with the aid of switching activity simulation in Synopsys VCS [Synopsys, 2018c], either with exhaustive simulation (8-bit and 16-bit), or simulation with 2^{16} random input vectors (32-bit).

Figures 5.20, 5.21, and 5.22 show the final post-place-and-route power estimations for adders generated by the driver in Fig. 5.19 for instance sizes of 8-, 16-, and 32-bits, respectively. These figures show that as the driver progresses, the resulting adders generally become better: the adders typically use less power for a given delay target. Further, the best adder generated by the solver-aided driver is generally about as good as the best statically-generated adder with some sparsity (sparsity 1 means no sparsity).

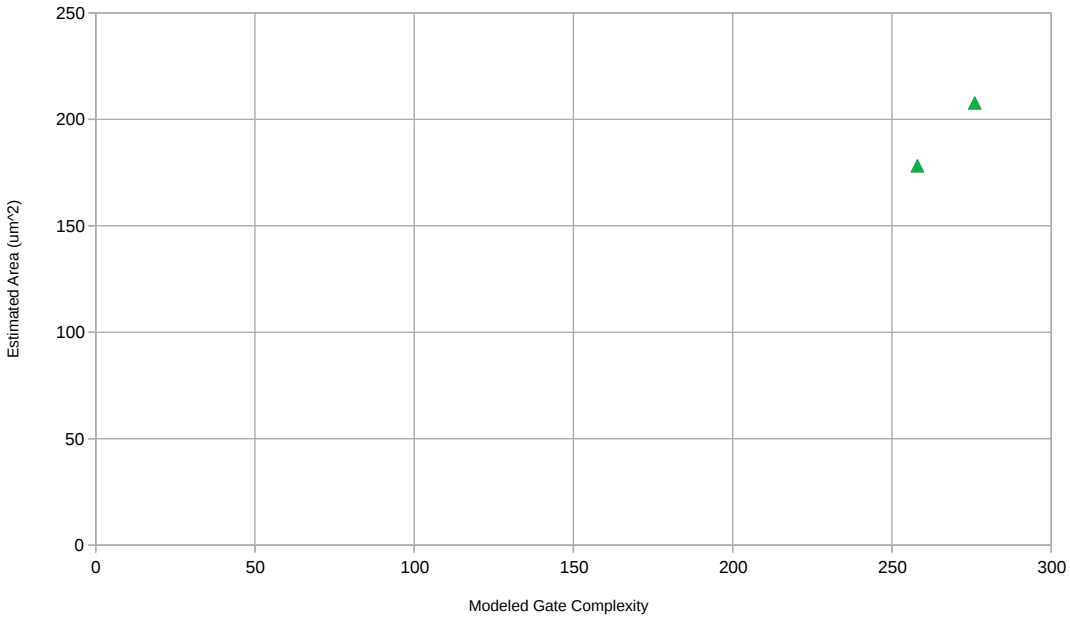


Figure 5.23 – Area (μm^2) vs. modeled gate complexity for all intermediate 8-bit adders, listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. Because the design is only 8 bits and there is relatively limited architectural freedom, only two intermediate solutions were found. However, these two points comport with the theory that our modeling is somewhat accurate; estimated area increases with model complexity.

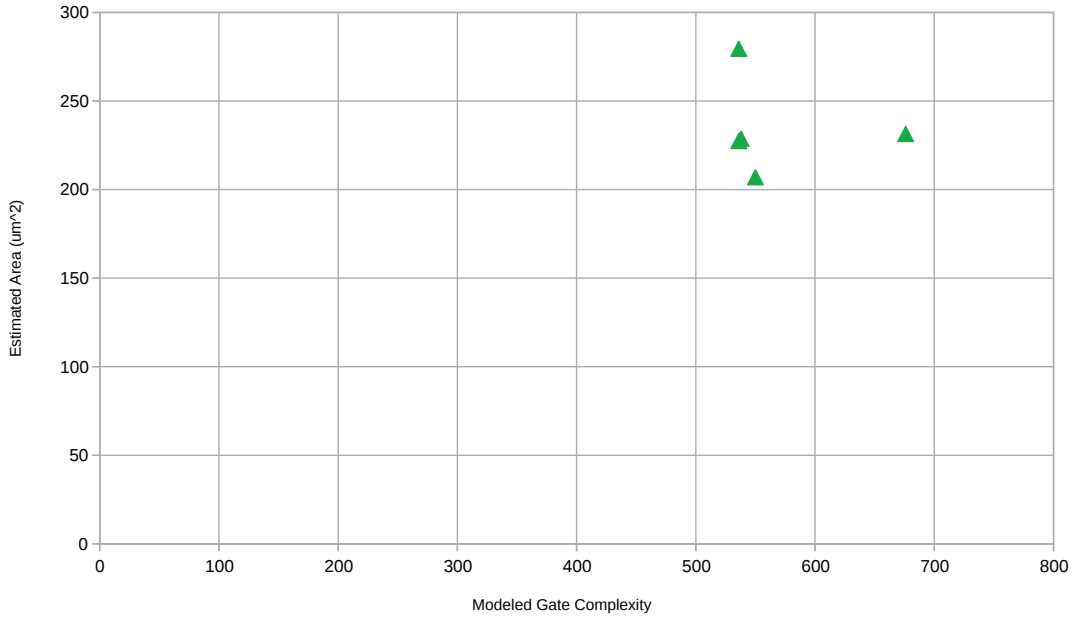


Figure 5.24 – Area (μm^2) vs. modeled gate complexity for all intermediate 16-bit adders, listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. This chart shows the inherent limitations of such a simplistic model; there is substantial "noise".

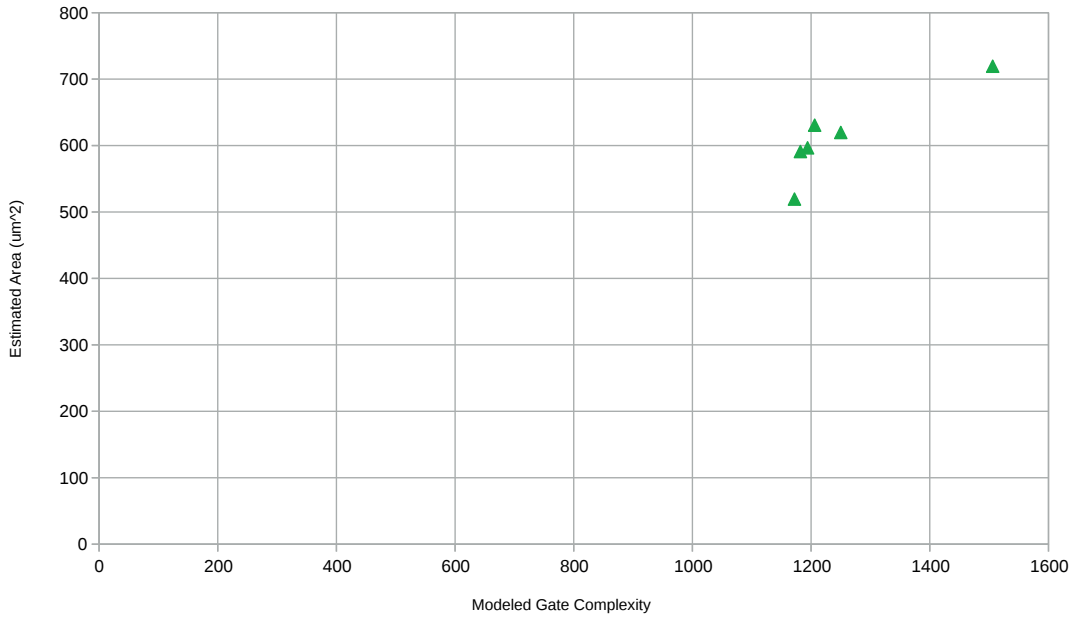


Figure 5.25 – Area (μm^2) vs. modeled gate complexity for all intermediate 32-bit adders, listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. This chart shows that despite some "noise", the simplistic model still roughly comports with expected behavior.

Note in particular that in Fig. 5.22, the final solution generated is very close, and almost always superior, to the static sparsity-4 adder. This static sparsity-4 adder is the architecture Aktan et al. determined to be optimal with their analytic model. The way the final sum block sketches the sparsity actually exposes even more design freedom to the solver than exists in that analytic model: an adder can use multiple sparsities for different parts of the adder. The results here show that Nasadiya sometimes not only can replicate the results of manually-derived analytic models, it can even do better.

Figures 5.23, 5.24, and 5.25 show the geometric mean of the area estimated by Innovus for the adders at the `ComplexityThreshold tau` values on the horizontal axis for 8-, 16-, and 32-bit instance sizes. Figure 5.24 shows a lot of noise around the minimum complexity value. It's likely that wiring effects on the small 16-bit instance make fine gradations between gate complexity essentially useless. For the 32-bit instance, however, Fig. 5.25 shows that post-place-and-route area estimated by Innovus rises with the modeled pre-synthesis gate complexity. This validates the use of a pre-synthesis gate complexity model as a reasonable representative of the post-place-and-route circuit area, at least for moderately-sized designs.

Figures 5.26, 5.27, and 5.28 show the geometric mean of the power estimated by Innovus for the adders at the `ComplexityThreshold tau` values on the horizontal axis for 8-, 16-, and 32-bit instance sizes. Again, Fig. 5.27, the chart for the 16-bit instances, shows considerable noise around the minimum model complexity value. However, Fig. 5.28, the chart for the 32-bit instances, shows that the complexity model has great correlation with the estimated post-place-and-route circuit power.

Figure 5.29 shows the time it takes the satisfiability solver to solve each successive constraint. Failed constraints (whether proven unsatisfiable or simply timed out at one hour) are discarded here. These data are merely intended to show the scalability of this approach: Even complex problems like extra-functional architecture optimization of this 32 bit adder can yield high-quality solutions in reasonable time. A user-defined timeout means the maximum effort spent is adjustable, and, once a known-satisfiable complexity threshold `tau` value is discovered, there are a bounded number of remaining solver iterations. That means the algorithm will complete eventually, and it's possible to make reasonable worst-case estimates of the total time required to find the more optimal solution.

5.4 Conclusion

In this chapter, we describe Nasadiya, a more powerful extension of the Scala-hosted Chisel domain-specific language [Bachrach et al., 2012] designed to address key limitations inherent in SKETCHILOG and enable a more flexible integration of satisfiability into the digital circuit design and optimization process. We show how an integrated modeling library, a flexible constraint specification facility, and virtualized solver access—giving designers the ability to formulate and solve arbitrary constraint satisfaction problems inside the design generator itself—can together make Nasadiya a powerful and general tool for solver-aided design and

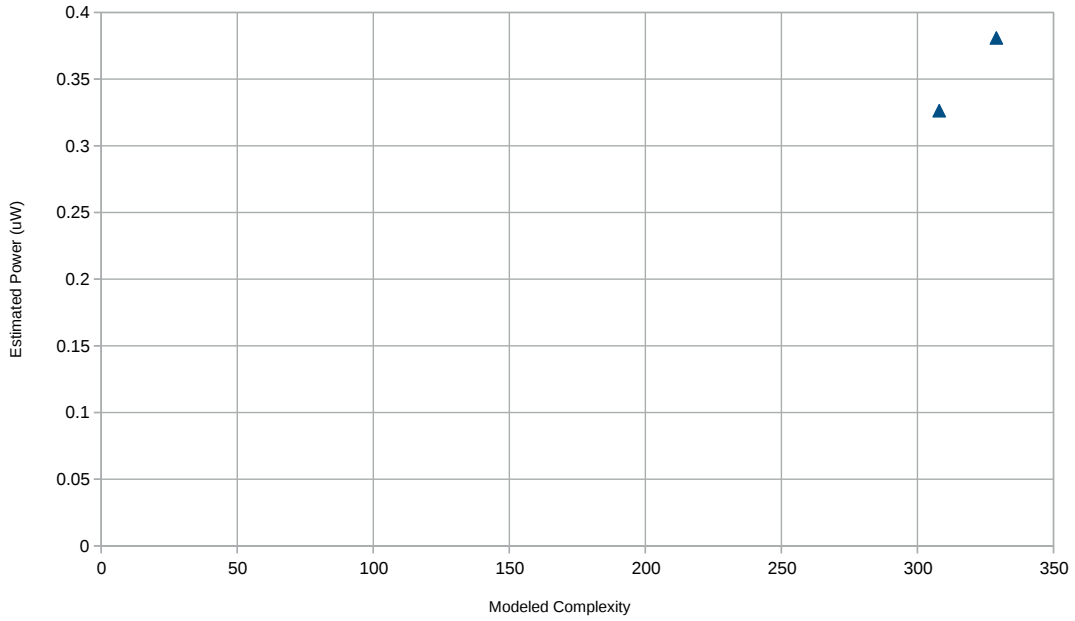


Figure 5.26 – Power (μW) vs. modeled total complexity for all intermediate 8-bit adders, listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. Because power is so closely linked to gate area, it's no surprise that these results look similar to the area results.

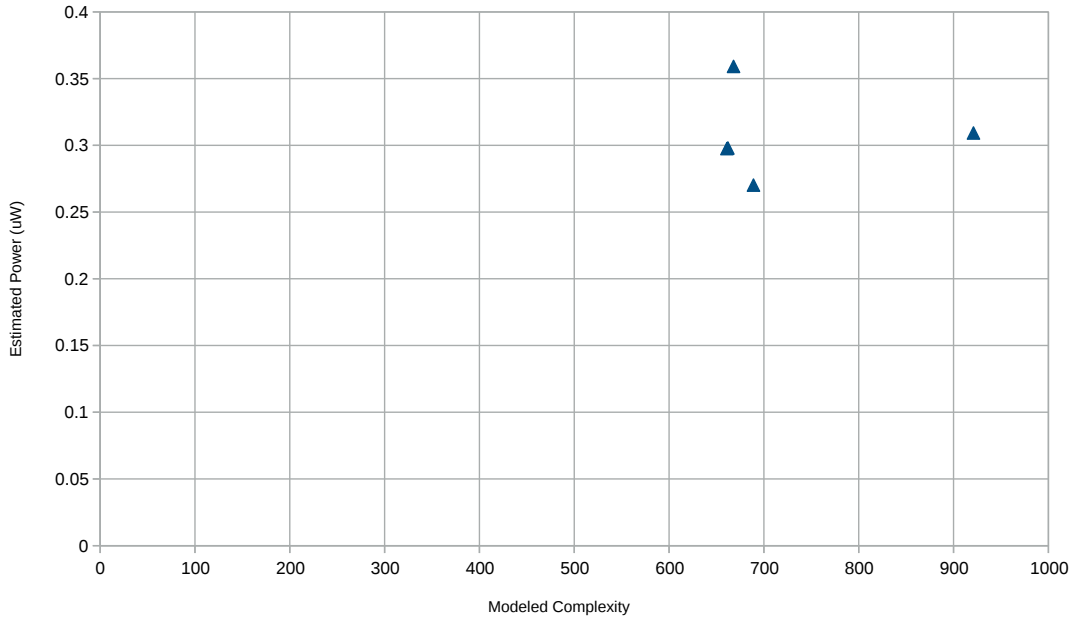


Figure 5.27 – Power (μW) vs. modeled total complexity for all intermediate 16-bit adders, listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. Again, this chart has a striking similarity to its corresponding area chart.

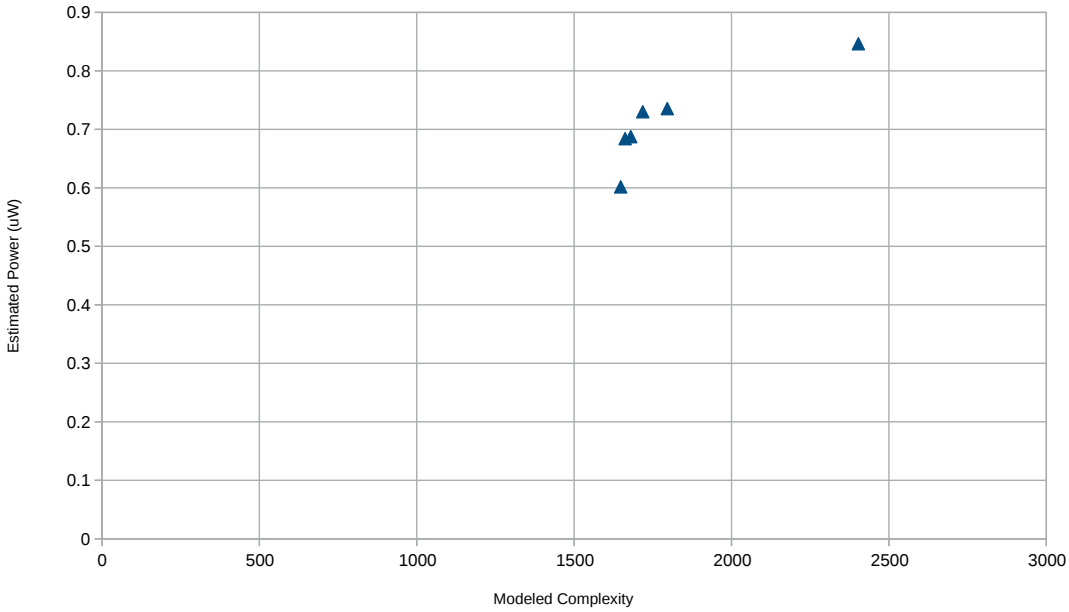


Figure 5.28 – Power (μW) vs. modeled total complexity for all intermediate 32-bit adders, listed by the value of the `tau` parameter to the `ComplexityThreshold` the adder satisfies. Again, this chart has a striking similarity to its corresponding area chart.

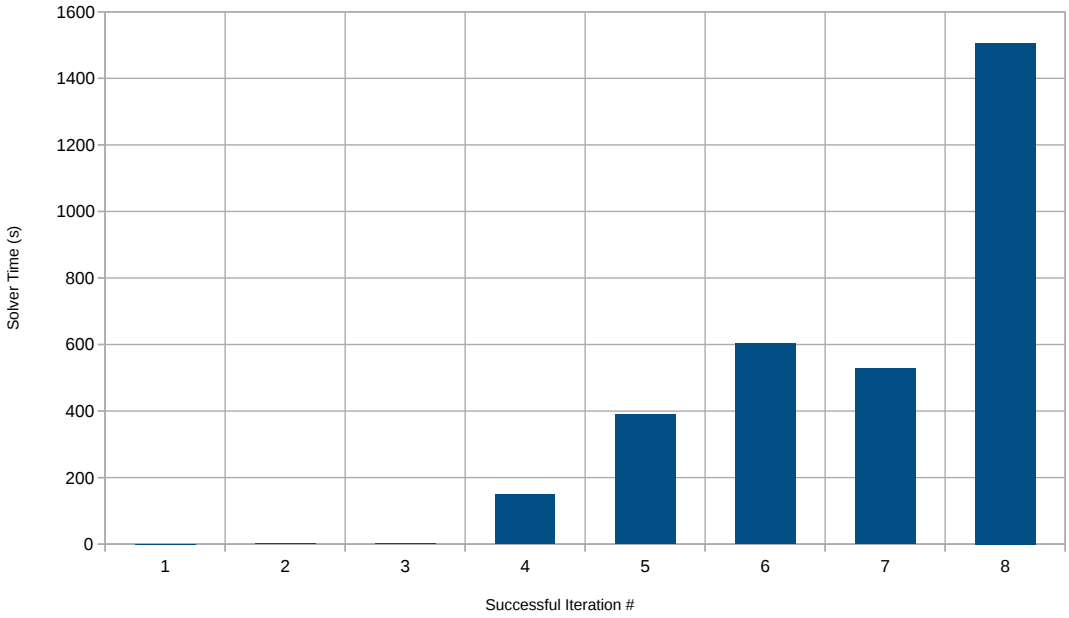


Figure 5.29 – A plot showing solver time (s) vs. iteration for the 32-bit adder experiment. Only successfully solved iterations are included on the horizontal axis.

other advanced reasoning about circuits.

We also demonstrate Nasadiya on a case study: specifying an implicitly parameterized (i.e., sketched) adder architecture and using these key features to automatically build delay and

complexity models, formulating constraints over the adder and these extra-functional models, and driving iterated application of a satisfiability solver to quickly discover adder architectures that minimize an objective function for circuit power. The resulting adder is shown to be roughly equally good as the adder other researchers constructed after many hours of manual analysis, without requiring any manual analysis itself. Satisfiability solvers and the fewer than 2000 lines of delay and gate complexity model construction code completely described in pseudo-code in this section together make it feasible to apply raw computational brute force to meaningfully optimize an arbitrary sketched combinational design and provides an automated alternative to arduous manual analysis of even reasonably-complex designs.

In short, Nasadiya is a relatively straightforward extension to the Chisel design language that is both general and powerful, and allows arbitrary applications of 2QBF-SAT to circuit design, debugging, and optimization.

Future work might more thoroughly explore the trade-off space described in Fig. 5.6, presenting designers with configurable model precision. Future work could also explore applications to sequential designs, determining the right way to enable designers to express what sequential equivalence means in the context of a specific design. More sophisticated iterated constraint satisfaction algorithms than that in, e.g., Fig. 5.19 are also an obvious area for potential improvement. One possible approach to even faster results might be to begin iterations with a reduced timeout, find the most optimal solution possible with that solver timeout, then exponentially increase the solver timeout in subsequent solver loops bisecting a smaller search space. Another possible approach might parallelize the iterated constraint satisfaction process by speculating N `tau` parameters, invoke solvers in parallel on N remote computers to test these values, then using a dynamic job scheduler to interpret results and manage remote jobs to continue to search N points simultaneously in the remaining search space. Such improvements and additions to Nasadiya could be very useful and make it an even more attractive platform for solver-aided design.

6 Conclusion

Exploding design complexity has made it increasingly difficult for digital circuit designers to develop and maintain complete and accurate mental models of their designs. This has negative implications for virtually the entire hardware ecosystem: it is more difficult to create correct circuits, to find and fix errors, and to optimize circuits for specific design constraints. Most attempts to fix these problems revolve around raising the level of abstraction and generating circuits from higher-level descriptions. Unfortunately, there is a good reason most designers work directly at the RTL level: it offers an unparalleled level of control, and designers can usually achieve much better implementations than high-level synthesis systems.

Satisfiability is a broad class of decision procedures that determine whether some input assignments can make some propositional logic formula to be true ("satisfiable"). Traditional Boolean SAT has been widely used in CAD for over a decade to aid automatic physical implementation of RTL designs. Because of the very close relation between propositional logic and digital circuits, satisfiability solvers are well-suited to difficult reasoning problems around digital circuits. Recent advances in other, more powerful, forms of satisfiability show promise to enable even more powerful kinds of reasoning about digital circuits.

In this thesis, we describe a number of novel applications of 2QBF-SAT to various problems in the digital circuit design process, and introduce a domain-specific language well-suited to these and other applications. In Chapter 2, we describe SketchiLog a language for developing RTL generators that supports "sketching" combinational circuits, or allowing certain features of the design implementation to be left unspecified. With the aid of a satisfiability solver and a golden reference circuit, SketchiLog determines how to complete the unspecified parts of the design in such a way that the sketch is proven to be functionally equivalent to the golden reference under any input condition. Since most arithmetic or other datapath components are typically combinational and have simple and obviously-correct albeit inefficient implementations, SketchiLog is particularly useful for developing guaranteed-correct hand-optimized components. The support for filling holes make corner cases easy to handle, while the expressive language still allows designers to control the lowest-level details in their designs.

Chapter 3 expands this idea of sketching to circuit debugging, with two key innovations. First, the responsibility to insert holes to express alternative functionality is moved from the designer to the debugging tool, which uses a known erroneous input and response along with a list of suspected error locations and a library of common syntax-level errors in order automatically to permute the buggy design to potentially fix one of those common errors in the design. Second, unlike datapath components, full designs typically do not have a readily-available golden reference circuit. Thus, we developed an approach to use only a few input vectors and known-correct responses in order to loosely constrain the functionality of the design being debugged. Finally, we show that this technique can be used to find a significant number of simple errors in a sample set of various designs, and that despite the lack of a complete golden reference, all discovered errors are localized and corrected with the canonical, semantically-correct source code changes in all tested cases.

In addition to reasoning about circuits being designed and debugged, we show how the same kind of satisfiability problem can be used to optimize circuits in Chapter 4. Here, we focus on two methods to optimize GLIFT models—which are themselves circuits—in order to reduce their complexity and help make GLIFT analysis feasible for real-world sized circuits. The first method is an exact method: at least a certain number of GLIFT cells are replaced while satisfying the constraint that the transformed GLIFT model be functionally identical to the original GLIFT model. We also introduce a bisection-based iterative solving procedure to progressively narrow the search for the maximum possible number of replacements. The second method is an inexact method: it allows the optimized GLIFT model not to match exactly the original GLIFT model's functionality, but only under controlled conditions (e.g., only under certain patterns of inputs) and with some maximum amount of induced false-positive detected flows. This kind of intricate reasoning about circuits is extraordinarily difficult for a human: not only does potential modified functionality need to be evaluated for all possible input values, but local changes in one part of the model can affect other local changes. In other words, this is a global optimization problem with hundreds to thousands of free variables. This is the first known method to allow arbitrary trade-offs between GLIFT model precision and complexity and shows how powerful 2QBF-SAT can be for automated reasoning about circuits.

Finally, all these ideas are synthesized in Chapter 5, which describes a language for solver-aided RTL design generation and optimization that is general and flexible enough to implement any of the applications developed in this thesis. As the previous chapters demonstrate, satisfiability can be integrated into the design process in a number of different ways and to achieve different goals. Nasadiya provides full, transparent integration of the satisfiability solver directly into the design itself: arbitrary constraints are simple to write, can easily reference design components, and are transparently translated into miters. Solving is a simple matter of one function call, rather than a tedious and involved process of wrangling textual formats acceptable to the solver and parsing solver output and exit codes. Further, Nasadiya provides simple but extensible hole-sensitive model creation facilities. In this way, designers can automatically build a component that models the pre-synthesis delay or area of some sketched component.

All combined, this amounts to a powerful and flexible language well suited for virtually any integration of 2QBF-SAT and circuit design, debug, or optimization. As a demonstration, we examine a case study using Nasadiya to automatically optimize the energy of a sketched parallel prefix adder design. We demonstrate that the integrated extra-functional modeling capabilities and integrated satisfiability solver can generate parallel prefix adder architectures nearly as good (and sometimes better) as those resulting from a laborious and time-consuming manual analysis and optimization effort. In contrast, Nasadiya only requires a sketched design and a few lines of code to drive the solver to find the best implementations.

Circuit design, debug, and optimization are all perennial challenges that are only compounded by ever-increasing design sizes and complexities. We formulate a number of novel satisfiability-based approaches to tackling certain problems in this space, show that 2QBF-SAT solver integration can enable new, time-saving ways to reason about well-constrained circuit analysis problems, and develop a language particularly designed to facilitate this integration. Hopefully, the scalability of our applications, continuing progress in satisfiability solver algorithms, and the availability of this language will convince others of the potential advantages to embracing this emerging technology and enable many more designers to take advantage of the powerful assistance satisfiability solvers can provide to many aspects of the digital circuit design, debug, and optimization processes.

Bibliography

Sally Adee. The hunt for the kill switch. *Spectrum, IEEE*, 45(5):34–39, May 2008. ISSN 0018-9235. doi: 10.1109/MSPEC.2008.4505310.

Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, 6 1978. ISSN 0018-9340. doi: 10.1109/TC.1978.1675141.

Mustafa Aktan, Dursun Baran, and Vojin G. Oklobdzija. Minimizing energy by achieving optimal sparseness in parallel adders. In *22nd IEEE Symposium on Computer Arithmetic, ARITH 2015, Lyon, France, June 22-24, 2015*, pages 10–17, 2015. doi: 10.1109/ARITH.2015.13.

Moayad F. Ali, Andreas Veneris, Sean Safarpour, and Ralf Drechsler. Debugging sequential circuits using boolean satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 204–209, San Jose, CA, USA, November 2004.

B. Alizadeh, P. Behnam, and S. Sadeghi-Kohan. A scalable formal debugging approach with auto-correction capability based on static slicing and dynamic ranking for RTL datapath designs. *IEEE Transactions on Computers*, 64(6):1564–78, June 2015.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: Constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 1216–1225. ACM, 2012. ISBN 978-1-4503-1199-1.

R. I. Bahar, H. Cho, G. D. Hachtel, E. Macii, and F. Somenzi. Timing analysis of combinational circuits using adds. In *Proceedings of European Design and Test Conference EDAC-ETC-EUROASIC*, pages 625–629, Feb 1994. doi: 10.1109/EDTC.1994.326813.

Andrew Becker, David Novo, and Paolo Ienne. SketchiLog: Sketching combinational circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014. doi: 10.7873/DATE.2014.165.

Andrew Becker, Djordje Maksimovic, David Novo, Mohsen Ewaida, Andreas G. Veneris, Barbara Jobstmann, and Paolo Ienne. Fudgefactor: Syntax-guided synthesis for accurate RTL error localization and correction. In *11th International Haifa Verification Conference, HVC'15*, pages 259–275, Nov. 2015. doi: 10.1007/978-3-319-26287-1_16.

Andrew Becker, Wei Hu, Yu Tai, Philip Brisk, Ryan Kastner, and Paolo Ienne. Arbitrary precision and complexity tradeoffs for gate-level information flow tracking. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. doi: 10.1145/3061639.3062203.

Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. In *the 15th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'13, pages 197–214. Springer-Verlag, 2013. ISBN 978-3-642-40348-4. doi: 10.1007/978-3-642-40349-1_12.

Mohammad-Mahdi Bidmeshki and Yiorgos Makris. VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation. In *2015 IEEE International Symposium on Circuits and Systems, ISCAS 2015, Lisbon, PT, May 24-27, 2015*, pages 29–32, 2015. doi: 10.1109/ISCAS.2015.7168562.

Roderick Bloem and Franz Wotawa. Verification and fault localization in VHDL programs. *Journal of the Telematics Engineering Society*, 2:30–33, 2002.

Gedare Bloom, Eugen Leontie, Bhagirath Narahari, and Rahul Simha. Hardware and security: Vulnerabilities and solutions, 2012.

George Boole. *The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning*. Cambridge University Press, 1847. doi: 10.1017/CBO9780511701337.010.

George Boole. *An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities*. Cambridge University Press, 1854. doi: 10.1017/CBO9780511693090.014.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23, 09 2013.

Robert K. Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, volume 6174, pages 24–40. Springer, July 2010.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

Raul Camposano. From behavior to structure: High-Level Synthesis. *IEEE Design and Test of Computers*, 7(5):8–19, October 1990.

Kai Hui Chang, Igor Markov, and Valeria Bertacco. Fixing design errors with counterexamples and resynthesis. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 944–949, Yokohama, JP, January 2007a.

Kai Hui Chang, Ilya Wagner, Valeria Bertacco, and Igor Markov. Automatic error diagnosis and correction for RTL designs. In *Proceedings of the High Level Design Validation and Test Workshop*, pages 65–72, Irvine, CA, USA, November 2007b.

Yibin Chen, Sean Safarpour, João Marques-Silva, and Andreas Veneris. Automated design debugging with maximum satisfiability. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, November 2010.

S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, G. T. Sullivan, A. Thomas, J. Tov, C. M. White, and D. Wittenberg. Safe: A clean-slate architecture for secure systems. In *Technologies for Homeland Security (HST), 2013 IEEE International Conference on*, pages 570–576, Nov 2013. doi: 10.1109/THS.2013.6699066.

Pi Yu Chung and Ibrahim N. Hajj. ACCORD: Automatic catching and correction of logic design errors in combinational circuits. In *Proceedings of the International Test Conference*, pages 742–751, Baltimore, MD, USA, September 1992.

Pi Yu Chung, Yi Min Wang, and Ibrahim N. Hajj. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(3):320–332, September 1994.

Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011. ISSN 0278-0070. doi: 10.1109/TCAD.2011.2110592.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. ISBN 0-13-451832-2.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi: 10.1145/800157.805047.

B. Jack Copeland. The modern history of computing. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2017 edition, 2017.

Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, pages 65–74, April 2010.

Srini Devadas, Kurt Keutzer, and Sharad Malik. Delay computation in combinational logic circuits: theory and algorithms. In *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, pages 176–179, Nov 1991. doi: 10.1109/ICCAD.1991.185224.

Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

Milos D. Ercegovac and Tomas Lang. *Digital Arithmetic*. Morgan Kaufmann, San Francisco, CA, USA, 2004.

João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. Pi-Ware: Hardware Description and Verification in Agda. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:27, Dagstuhl, DE, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-030-9. doi: 10.4230/LIPIcs.TYPES.2015.9. URL http://drops.dagstuhl.de/opus/volltexte/2018/8479.

H.D. Foster. Trends in functional verification: A 2014 industry study. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6, June 2015.

Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013. https://eprint.iacr.org/2013/857.

N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne. Making domain-specific hardware synthesis tools cost-efficient. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 120–127, Dec 2013. doi: 10.1109/FPT.2013.6718341.

E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. URL www.qbflib.org.

Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20, 1982. doi: 10.1109/SP.1982.10014.

Evgueni Goldberg, Mukul Prasad, and Robert Brayton. Using sat for combinational equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '01, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press. ISBN 0-7695-0993-2.

James W Gray III. Toward a mathematical foundation for information flow security. *Journal of Computer Security*, 1(3):255–294, 1992.

Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461 – 466, jan. 2003. doi: 10.1109/ICVD.2003.1183177.

Eldon C. Hall. *Journey to the Moon: The History of the Apollo Guidance Computer*. American Institute of Aeronautics & Astronautics, 1996. ISBN 156347185X.

Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. Theoretical fundamentals of gate level information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(8):1128–1140, Aug 2011. ISSN 0278-0070. doi: 10.1109/TCAD.2011.2120970.

Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. On the complexity of generating gate level information flow tracking logic. *IEEE Transactions on Information Forensics and Security*, 7(3):1067–1080, June 2012. ISSN 1556-6013. doi: 10.1109/TIFS.2012.2189105.

Wei Hu, Andrew Becker, Armita Ardeshiricham, Yu Tai, Paolo Ienne, Dejun Mu, and Ryan Kastner. Imprecise security: Quality and complexity tradeoffs for hardware information flow tracking. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, pages 95:1–95:8, New York, NY, USA, 2016a. ACM. ISBN 978-1-4503-4466-1. doi: 10.1145/2966986.2967046.

Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. Detecting hardware trojans with gate-level information-flow tracking. *Computer*, 49(8):32–40, August 2016b. ISSN 0018-9162.

Shi Yu Huang, Kwang Ting Cheng, Kuang Chien Chen, and Juin Yen J. Lu. Fault-simulation based design error diagnosis for sequential circuits. In *Proceedings of the 35th Design Automation Conference*, pages 632–637, San Francisco, CA, USA, June 1998.

IWLS. IWLS benchmarks ver. 3.0, 2005. http://iwls.org/iwls2005/benchmarks.html.

Peter Jamieson, Kenneth B. Kent, Farnaz Gharibian, and Lesley 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*, pages 149–156, 2010.

Mikolas Janota, Charles Jordan, Will Klieber, Florian Lonsing, Martina Seidl, and Allen Van Gelder. The QBF gallery 2014: The qbf competition at the FLoC Olympic games. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:187–206, 2014 (published 2016).

Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *Proceedings of the 17th International Conference on Computer Aided Verification*, pages 226–238, Edinburgh, Scotland, UK, July 2005.

Charles Jordan and Martina Seidl. The QBF gallery 2014, a competitive evaluation of QBF tools, 2014. URL http://qbf.satisfiability.org/gallery/qbf-gallery-2014-talk-long.pdf.

Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of Z3: Integrating SMT and programming. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 400–406, Heidelberg, DE, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22438-6.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294293.

Yung-Te Lai, Sarma Sastry, and Massoud Pedram. Boolean matching using binary decision diagrams with applications to logic synthesis and verification. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers Processors*, pages 452–458, Oct 1992. doi: 10.1109/ICCD.1992.276313.

Xun Li, Mohit Tiwari, Ben Hardekopf, Timothy Sherwood, and Frederic T. Chong. Secure information flow analysis for hardware design: Using the right abstraction for the job. In *the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 8:1–8:7, New York, NY, USA, 2010. ISBN 978-1-60558-827-8. doi: 10.1145/1814217.1814225.

Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Position paper: Sapper – a language for provable hardware policy enforcement. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, pages 39–44, New York, NY, USA, 2013. ISBN 978-1-4503-2144-0. doi: 10.1145/2465106.2465214.

H. Ling. High speed binary parallel adder. *IEEE Transactions on Electronic Computers*, EC-15 (5):799–802, Oct 1966. ISSN 0367-7508. doi: 10.1109/PGEC.1966.264571.

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

Chris Lomont. Fast inverse square root, 2003. URL http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf.

Florian Lonsing, Martina Seidl, and Allen Van Gelder. The QBF Gallery: Behind the scenes. *Artif. Intell.*, 237:92–114, 2016.

Chris A. Mack. Fifty years of moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, May 2011. ISSN 0894-6507. doi: 10.1109/TSM.2010.2096437.

Jean C. Madre, Olivier Coudert, and Jean P. Billon. Automating the diagnosis and the rectification of digital errors with PRIAM. In *Proceedings of the International Conference on Computer-Aided Design*, pages 30–33, Santa Clara, CA, USA, November 1989.

Jeremiah Mahler. A MIPS CPU written in Verilog. https://github.com/jmahler/mips-cpu, 2015. [Accessed: 24-April-2015].

Scott Mahlke, Rajiv Ravindran, Michael Schlansker, Robert Schreiber, and Timothy Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-20(11):1355–71, November 2001.

Wojciech Maly. Cost of silicon viewed from vlsi design perspective. In *31st Design Automation Conference*, pages 135–142, June 1994. doi: 10.1145/196244.196311.

Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, March 1971. ISSN 0001-0782. doi: 10.1145/362566.362568.

Konstantinos Markantonakis, Michael Tunstall, Gerhard Hancke, Ioannis Askoxylakis, and Keith Mayes. Attacking smart card systems: Theory and practice. *Information Security Technical Report*, 14(2):46 – 56, 2009. ISSN 1363-4127. doi: https://doi.org/10.1016/j.istr.2009.06.001. URL http://www.sciencedirect.com/science/article/pii/S136341270900017X. Smart Card Applications and Security.

João P. Marques-Silva and Karem A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. ISSN 0018-9340. doi: 10.1109/12.769433.

Grant Martin and Gary Smith. High-Level Synthesis: Past, present, and future. *IEEE Design and Test of Computers*, 26(4):18–24, July–August 2009.

John McLean. Security models and information flow. In *1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 180–187, 1990.

Alan Mishchenko and Robert K. Brayton. Sat-based complete don't-care computation for network optimization. In *Design, Automation and Test in Europe*, pages 412–417, March 2005. doi: 10.1109/DATE.2005.264.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, June 2001. doi: 10.1145/378239.379017.

S. Naffziger. A sub-nanosecond 0.5 /spl mu/m 64 b adder design. In *1996 IEEE International Solid-State Circuits Conference. Digest of TEchnical Papers, ISSCC*, pages 362–363, Feb 1996. doi: 10.1109/ISSCC.1996.488718.

Razvan Nane, Vlad Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35:1–1, 12 2015.

Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information flow isolation in I2C and USB. In *Design Automation Conference (DAC)*, pages 254 –259, June 2011.

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima, Walnut Creek, CA, USA, second edition, 2010.

OpenCores. OpenCores database. http://www.opencores.org, 2015. [Accessed: 24-April-2015].

Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Design*. Oxford University Press, New York, NY, USA, second edition, 2010.

David J. Pearce and Lindsay Groves. Whiley: A platform for research in software verification. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, pages 238–248, Cham, 2013. Springer International Publishing. ISBN 978-3-319-02654-1.

Bernhard Peischl and Franz Wotawa. Automated source level error localization in hardware designs. *Journal of IEEE Design & Test in Computers*, 23(1):8–19, January 2006.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75293.

Andreas Raabe and Rastislav Bodík. Synthesizing hardware from sketches. In *DAC*, pages 623–624. ACM, 2009. ISBN 978-1-60558-497-3.

Heinz Riener, Finn Haedicke, Stefan Frehse, Mathias Soeken, Daniel Grosse, Rolf Drechsler, and Goerschwin Fey. metasmt: focus on your application and not on solver integration. *International Journal on Software Tools For Technology Transfer*, 19(5):17. 605–621, 2017.

Dorothy Elizabeth Robling Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121.

Samir Sapra, Michael Theobald, and Edmund Clarke. Sat-based algorithms for logic minimization. In *Proceedings 21st International Conference on Computer Design*, pages 510–517, Oct 2003. doi: 10.1109/ICCD.2003.1240948.

Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723, Dec 1938. ISSN 0096-3860. doi: 10.1109/T-AIEE.1938.5057767.

Henry Maurice Sheffer. A set of five independent postulates for boolean algebras, with application to logical constants. *Transactions of the American Mathematical Society*, 14(4): 481–488, 1913. ISSN 00029947. URL http://www.jstor.org/stable/1988701.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, Seattle, WA, USA, June 2013.

Alexander Smith, Andreas Veneris, and Anastasios Viglas. Design diagnosis using Boolean satisfiability. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 218–23, Yokohama, JP, January 2004.

Alexander Smith, Andreas Veneris, Moayad Fahim Ali, and Anastasios Viglas. Fault diagnosis and logic debugging using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-24(10):1606–21, October 2005.

Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, UC Berkeley, Dec 2008. URL http://eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 404–415. ACM, 2006. ISBN 1-59593-451-0.

Stefan Staber, Barbara Jobstmann, and Roderick Bloem. Finding and fixing faults. *Journal of Computer and System Sciences*, 78(2):441–460, March 2012.

G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGOPS Oper. Syst. Rev.*, 38(5):85–96, October 2004. ISSN 0163-5980. doi: 10.1145/1037949.1024404.

Synopsys. DesignCompiler, 2018a. URL https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html.

Synopsys. DesignWare, 2018b. URL https://www.synopsys.com/designware-ip.html.

Synopsys. VCS, 2018c. URL https://www.synopsys.com/verification/simulation/vcs.html.

Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *international conference on Architectural support for programming languages and operating systems*, ASPLOS'09, pages 109–120, New York, NY, USA, 2009. ISBN 978-1-60558-406-5.

Grigori S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. ISBN 978-3-642-81955-1. doi: 10.1007/978-3-642-81955-1_28.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628161.

Yuxin Wang, Peng Li, and Jason Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 199–208, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2671-1. doi: 10.1145/2554688.2554780. URL http://doi.acm.org/10.1145/2554688.2554780.

Clifford Wolf and Johann Glaser. Yosys - A free Verilog synthesis suite. In *Proceedings of 21st Austrian Workshop on Microelectronics*, Linz, AT, October 2013.

Lee C. Y. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959. doi: 10.1002/j.1538-7305.1959.tb01585.x.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110, New York, NY, USA, 2012. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254078.

Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 503–516, New York, NY, USA, 2015. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694372.

Reto Zimmermann. *Binary adder architectures for cell-based VLSI and their synthesis*. PhD thesis, ETHZ, Zürich, CH, 1998.

Andrew Becker

Interests/Objective

My interests span broad areas including security and privacy, FPGAs and reconfigurable architectures, embedded systems, verification and synthesis, processor architecture, high-level synthesis, and more. I am particularly interested in applying my experience in these fields to improve the state of hardware and software security and to enable more trustworthy computing.

Relevant Skills/Tools

Languages
Proficiency: C, C++, Scala, Chisel, Verilog, Python, bash
Competency: Java, C#, PHP, SQL, various assembly

Hardware
RTL and FPGA design (Xilinx and Altera/Intel), embedded system design, Yosys, ABC, Synopsys DesignCompiler and Formality

Software
SAT and SMT solvers (Yices, CEGIS, Z3), various IDEs (IntelliJ IDEA, Visual Studio, Vim), version control/project mgmt. (git, GitLab and GitHub), compiler design (Flex and Bison), GCC, gdb, binutils, IDA Pro, general GNU userland

Work Experience

Doctoral Assistant
09/2011 to 09/2017
Processor Architecture Lab
École Polytechnique Fédérale de Lausanne (EPFL) — Lausanne, VD, CH
Advisor: Prof. Paolo Ienne

Research Assistant
10/2007 to 06/2011
Embedded Systems Lab
University of California, Riverside (UCR) — Riverside, CA, USA
Advisor: Prof. Frank Vahid

Publications

A. Becker, W. Hu, Y. Tai, P. Brisk, R. Kastner, P. Ienne. **"Arbitrary Precision and Complexity Tradeoffs for Gate-Level Information Flow Tracking"**, *Design Automation Conference (DAC)*, June 2017.

W. Hu, A. Becker, A. Ardeshiricham, Y. Tai, P. Ienne, D. Mu, R. Kastner. **"Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking"**, *International Conference on Computer Aided Design (ICCAD)*, November 2016.

A. Becker, D. Maksimović, D. Novo, M. Owaida, A. Veneris, B. Jobstmann, P. Ienne. **"FudgeFactor: Syntax-guided synthesis for accurate RTL error localization and correction"**, *Haifa Verification Conference (HVC)*, November 2015.

A. Becker, D. Novo, P. Ienne. **"SKETCHILOG: Sketching combinational circuits"**, *Design, Automation and Test in Europe (DATE)*, March 2014.

A. Becker, D. Novo, P. Ienne. **"Automated circuit elaboration from incomplete architectural descriptions**", *Asilomar Conference on Signals, Systems, and Computers*, November 2013.

A. Becker, S. Sirowy, F. Vahid. **"Just-in-Time Compilation for FPGA Processor Cores***", IEEE Electronic System Level Synthesis Conference (ESLsyn)*, June 2011.