Automatic Term-Level Abstraction

by

Bryan Brady

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor Robert K. Brayton
Professor Alper Atamtürk

Spring 2011

Automatic Term-Level Abstraction

Abstract

Automatic Term-Level Abstraction

by

Bryan Brady

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Recent advances in decision procedures for Boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT) have increased the performance and capacity of formal verification techniques. Even with these advances, formal methods often do not scale to industrial-size designs, due to the gap between the level of abstraction at which designs are described and the level at which SMT solvers can be applied. In order to fully exploit the power of state-of-the-art SMT solvers, abstraction is necessary. However, applying abstraction to industrial-size designs is currently a daunting task, typically requiring major manual efforts. This thesis aims to bridge the gap between the level at which designs are described and the level at which SMT solvers can reason efficiently, referred to as the term level.

This thesis presents automatic term-level abstraction techniques in the context of formal verification applied to hardware designs. The techniques aim to perform abstraction as automatically as possible, while requiring little to no user guidance. Data abstraction and function abstraction are the foci of this work. The abstraction techniques presented herein rely on combining static analysis, random simulation, machine learning, and abstraction-refinement in novel ways, resulting in more intelligent and scalable formal verification methodologies.

The data abstraction procedure presented in this work uses static analysis to identify portions of a hardware design that can be re-encoded in a theory other than the theory of bit vectors, with the goal of creating an easier to reason about verification model. In addition, the data abstraction procedure can provide feedback that can help the designer create hardware designs that are easier to verify.

The function abstraction procedures described in this work rely on static analysis, random simulation, machine learning, and counterexample-guided abstraction-refinement to identify and abstract functional blocks that are hard for formal tools to reason about. Random simulation is used to identify functional blocks that will likely yield substantial performance increases if they were to be abstracted. A static analysis-based technique, ATLAS, and a separate technique, CAL, based on a combination of machine learning and counterexample-guided abstraction-refinement, are then used to compute conditions under which it is precise to abstract. That is, functional blocks are abstracted in a manner that avoids producing spurious counterexamples.

Experimental evidence is presented that proves the efficacy and efficiency of the data and function abstraction procedures. The experimental benchmarks are drawn from a wide range of hardware designs including network-on-chip routers, low-power circuits, and microprocessor designs.

To Mom, Dad, and Ashley

Acknowledgments

First, and foremost, I would like to thank my advisor, Sanjit A. Seshia. His contagious enthusiasm, intense motivation and determination, and seemingly infinite patience, have set an example that I can only hope to achieve. His willingness to give feedback, brainstorm, and discuss research ideas late into the night or early hours of the morning, in the office or on the phone, have been invaluable during my tenure as a graduate student. Knowing how hard Sanjit works is enough to motivate me to work harder; he truly leads by example. I am proud to have had the opportunity to work with Sanjit.

I would also like to thank Robert Brayton and Alper Atamtürk for serving on my dissertation committee and qualifying examination committee. Additionally, I would like to thank Ras Bodik for serving on my qualifying examination committee.

Many people have influenced my work in a positive way, through feedback, brainstorming sessions, and sound advice. However, a few deserve special mention. I would like to thank Randy Bryant for his continued collaboration during my graduate career, the critical feedback he provided on the automatic data and function abstraction techniques, and for creating the Y86 processor benchmark that I have used (and become intimately familiar with) over the last several years. I would also like to thank John O'Leary for mentoring me during my internship at Intel and providing me with the opportunity to evaluate my data abstraction tool on a real-world design. His feedback, during and after my internship, helped refine the data abstraction technique and spawn an interest in automatic function abstraction techniques.

Special thanks to Ruth Gjerde and Sandy Weisberg for deciphering departmental policies and procedures, the persistent reminders about paperwork I had forgotten, and, most importantly, the many wonderful conversations.

Thanks to Dan Holcomb, Wenchao Li, Susmit Jha and Rhishikesh Limaye for the intriguing conversations, critical feedback, and all the fun times in the lab. Thanks to the many friends, climbing partners, and roommates for the adventures outside of school that helped keep my sanity intact.

From my days back in Pittsburgh, I would like to thank Steve Jacobs for the solid foundation in digital logic and all of the hard homework and exam questions that kept me up so many nights. Thanks to Steve Levitan for introducing me to verification for the first time and for being my unofficial advisor. Thanks to Rob Santoro, Sam Dickerson, and the many other friends from Pittsburgh who made those late nights in Benedum so enjoyable.

Most importantly, I would like to thank my parents, Mike and Michele, and my sister, Ashley, for your love and support through all these years. I am grateful that you taught me to work hard, to take pride in my work, and to enjoy my work. But, above all, I am grateful that you are my family; I could not have done this without you.

Finally, I would like to thank my girlfriend, Bridget, for making the last year and a half, my best year and a half.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Computer systems are ubiquitous in the world today. They are used in a variety of applications from personal communication devices to mission critical systems such as automotive and avionic control systems. Depending on the application, an error could be a minor inconvenience or a major catastrophe, possibly endangering lives. Therefore, there is an ethical and moral obligation to correctly design and implement computer systems.

Design and verification engineers have an arsenal of tools with which they can verify computer systems. These verification tools come in two main flavors: *simulation* and *formal*. Simulation-based verification involves simulating a design with input vectors and comparing the output against known results for those same input vectors. In formal verification, a mathematical model of the design is created along with properties that the design must satisfy and automated reasoning techniques are then used to determine whether the design satisfies the properties in all situations.

Consider the classic example of an elevator design, where the elevator is not supposed to move when the doors are open. To verify this elevator design with simulation-based verification would require simulating the elevator many times and checking in each case if the doors are ever open while the elevator is moving. At the end of this verification process, we know that the elevator never moves when the doors are open for all of the individual test cases that we tested. In small designs, it might be possible to test every possible case. However, in most realistic cases there are far too many combinations of inputs to test the entire input space. Simulation-based techniques can be helpful in finding bugs, but not proving their absence. Various techniques exist that attempt to quantify the amount of the search-space tested by the simulation vectors by computing what is known as a *coverage metric*. A coverage metric is a measure of confidence that the system is correct.

In situations where absolute guarantees of correctness are required, design and verification engineers turn to formal verification. Formal techniques are able to not only find bugs, but prove the absence of bugs. However, formal methods don't always scale well to large designs and often require hand-crafted abstractions, which can be tedious and error-prone to create. This dissertation focuses on increasing the scalability of formal techniques by automatically applying abstraction to portions of designs that are hard for formal techniques to reason about.

1.1 Formal Verification

Formal verification involves mathematically proving properties about designs. While there are many variations of formal verification, such as model checking and equivalence checking, each variant shares the following three requirements: (i) a mathematical model of the design; (ii) the property to be verified, and (iii) a model of the environment in which the design will be operating. We present below three formal verification techniques that we use in the remainder of this thesis.

Model checking. The goal of model checking is to determine whether a design D satisfies a property P, typically expressible in temporal logic, while operating in environment E. In model checking, the design, property, and environment are modeled symbolically. The model checking algorithm attempts to prove that design D satisfies property P in all states. This is accomplished by computing the set of reachable states $S_0, S_1, ...$ and checking that each state in S_i satisfies property P. In the absence of a counterexample, the set of reachable states are computed until a fix-point is reached (i.e., $S_i = S_{i+1}$), at which point it has been proven that design D satisfies property P. Computing the set of reachable states is a major challenge for model checking algorithms. A weaker form of model checking, where it is not necessary to compute the entire set of reachable states, is bounded-model checking (BMC). In BMC, the design is initialized to some known state and is then unrolled for a bounded number of cycles. For a design to pass BMC, it must satisfy the property in each time frame.

Equivalence checking. Equivalence checking involves proving that two designs have equivalent functionality. There are two types of equivalence checking: *combinational* and *sequential*. Combinational equivalence checking involves proving that two designs produce the same output for any input. Sequential equivalence checking involves proving that two designs produce the same output for any sequence of inputs. In either case, there are two versions of a design, with the same symbolic inputs connected to both designs, and the goal is to prove that the designs always produce the same output. If two designs are *equivalent*, they can be used interchangeably. Equivalence checking is a process used frequently throughout the design of hardware systems. Hardware is typically described using a high-level hardware description language (HDL) such as Verilog or VHDL. Next, a variety of logic synthesis algorithms, such as retiming and technology mapping are applied to the high-level HDL. After each synthesis step, it is necessary to ensure that the functionality of the design hasn't changed. Equivalence checking is typically performed after each step of synthesis.

Correspondence checking. Correspondence checking, introduced by Burch and Dill [31], is a formal verification technique used to verify the control logic of pipelined microprocessor designs. In correspondence checking, a pipelined microprocessor is verified against a sequential (meaning that instructions are executed one after another) version of the same processor. The sequential processor is essentially a formal model of the instruction set architecture. The property that correspondence checking aims to verify is that the pipelined processor

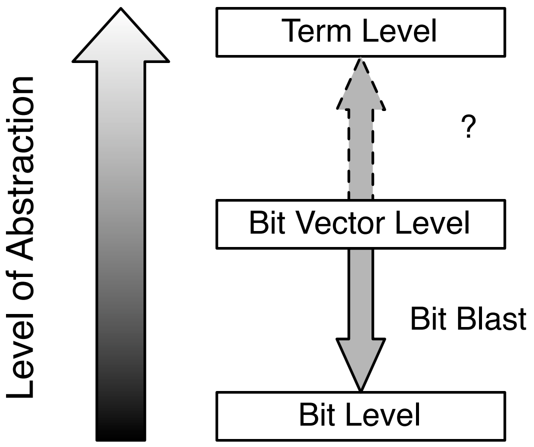


Figure 1.1: **Levels of abstraction.** Bit-level modeling is the lowest level of abstraction. Bit-vector level modeling is one step above bit-level modeling, however, it is precise in the sense that no information is abstracted away. Term-level modeling is the highest level of abstraction, where datapaths and functional blocks can be represented abstractly.

refines the sequential version of the same processor. In other words, the pipelined processor can mimic all of the behaviors present in the sequential version. Since processor designs are some of the main benchmarks used in this dissertation, we describe this technique in more detail in Section 2.5.

1.2 Abstraction Layers in Formal Verification

Regardless of the type of formal verification being applied, the same underlying reasoning mechanisms can be used. The type of reasoning mechanism depends on the level of abstraction used to model the system being verified. While there are many levels of abstraction within which the design and property to be tested can be modeled, this work focuses on three levels: *bit-level* modeling, *bit-vector-level* modeling, and *term-level* modeling. The aforementioned levels of abstraction are listed in increasing levels of abstraction as illustrated in Figure 1.1.

Bit level modeling. At the lowest level, the entire design can be modeled with individual bits, using propositional logic. In this case, higher-level constructs such as addition or multiplication circuits are modeled solely with Boolean logic gates, at the bit level. Solvers that operate at the bit level, such as Boolean satisfiability (SAT) solvers and Binary Decision Diagram (BDD) packages, can be quite powerful and have seen tremendous performance and capacity increases over the last decade.

BDDs, in their current form, were introduced by Bryant in 1986 [25]. He showed how to reduce BDDs into a canonical form by using a series of simplifications that are applied during the creation of a BDD. Further optimizations, such as using a unique table for fast access to existing nodes and adding complemented edges to represent negation, were presented in [14]. Even the most state-of-the-art BDD packages, such as CUDD, an efficient BDD package created by Somenzi [64], suffer from exponential time and space requirements in the worst case.

The Boolean satisfiability (SAT) problem, and algorithms to solve SAT problems, have been around for several decades [35, 34]. Even with these early works, the field of SAT solving did not see major advances until the late 1990s and early-to-mid 2000s [55, 57, 39, 40, 38, 67]. SAT solving has since received much attention, and as a result, additional progress has been made [39, 7, 12, 47]. The problem of Boolean satisfiability is NP-complete, thus, it is no surprise that problems exist that exhibit worst-case behavior for even the best SAT solvers. Additionally, certain circuit structures, such as those with many exclusive-or gates, tend to hinder the performance of bit-level solvers. To combat these problems, a higher level of abstraction can be employed.

Satisfiability Modulo Theories. The Satisfiability Modulo Theories (SMT) problem is a decision problem over formulas in first-order logic coupled with decidable first-order background theories. SMT solvers have become the umbrella term for solvers that reason at a level of abstraction above the bit level [9, 10]. There are many background theories supported by SMT solvers, however, they are not all compatible with one another. This work focuses on two background theories, the theory of bit vectors, and logic of equality with uninterpreted functions [10, 9, 33].

Bit-vector level modeling. The bit-vector level, also referred to as the word level, is the level of abstraction in which most hardware designs are modeled. At the word level, datapath signals are modeled with groups of bits, or bit vectors, and operations are defined over these bit vectors. The most obvious advantage that the bit-vector level has over the bit level is the reduction in model size. For example, the number of gates within a multiplier grows quadratically with the size of the datapath, however, when represented at the bit-vector level, a multiplier has constant size. Another advantage, which can be exploited for major performance increases, is that word-level reasoning procedures take into account properties that are lost at the bit level [48]. One such example is the commutativity of multiplication as we show in Chapter 3. Bit-vector solvers have received a lot of attention over the last several years [27, 28, 44, 43, 21] and there are many efficient solvers available [48, 23, 22, 37]. While bit-vector solvers outperform bit-level solvers in many cases, there are still situations where further abstraction is required. In these situations, term-level abstraction can be employed.

Term-level modeling. Term-level abstraction is a modeling technique where datapath signals are represented with symbolic *terms* and precise functionality is abstracted away with *uninterpreted functions*. Term-level abstraction has been found to be especially useful in microprocessor design verification [46, 31, 52, 54]. The precise functionality of units such as instruction decoders and the ALU are abstracted away using *uninterpreted functions*, and decidable fragments of first-order logic are employed in modeling memories, queues, counters, and other common data structures. While several SMT solvers support some, or all, of the background theories used in term-level abstraction, UCLID [53, 29] and SAL [36, 11] are two of the only full-fledged term-level verification systems available.

A major challenge of any abstraction technique, including term-level abstraction, is in determining what functionality to abstract. On the one hand, constructing these models by hand is a tedious process prone to errors, hence automation is essential. On the other hand, automatically abstracting all bit-vector signals to terms and all operators to uninterpreted functions results in too coarse an abstraction, in which properties of bit-wise and finite-precision arithmetic operators are obscured, leading to a huge number of spurious counterexamples. While such spurious counterexamples can in many cases be eliminated by selectively abstracting only parts of the design to the term level, manual abstraction requires detailed knowledge of the RTL design and the property to be verified. It is difficult for a human to decide what functional blocks or operators to abstract in order to obtain efficiency gains and also avoid spurious counterexamples.

1.3 Thesis Contributions

The problem addressed by this thesis is the automatic application of term-level abstraction to hardware designs with the goal of creating easier to verify models.

The main contributions of this thesis include:

1. A data abstraction technique, optionally guided by user-provided type annotations, that re-encodes portions of a bit-vector design at the term-level. While this technique can be fully automated, user-provided type annotations are used to provide feedback to the verification engineer (Chapter 4);

2. A random-simulation-based technique used for the identification of abstraction candidates (Chapter 5);

3. An automatic approach to function abstraction based on a combination of random simulation and static analysis. Random simulation is used to identify abstraction candidates. Then, static analysis is used to compute conditions under which it is precise to abstract (Chapter 5), and

4. An automatic function abstraction technique based on machine-learning, random simulation, and counterexample-guided abstraction-refinement. Abstraction candidates are identified using random simulation. Interpretation conditions are then learned by applying machine learning techniques to the spurious counterexamples that arise (Chapter 6).

1.4 Thesis Overview

This thesis is broken down into two main parts. The first part of this thesis discusses relevant background material. Chapter 2 introduces basic notation, modeling techniques, and levels of abstraction commonly used in formal methods. This includes the basics of bit-level, word-level, and term-level modeling and examples of term-level abstraction techniques being applied to simple circuits. Chapter 3 discusses the decision procedures used to solve problems at the various levels of abstraction, as well as discussing the strengths and weaknesses of each approach.

The second part of this thesis presents our approaches to data and function abstraction. Our data abstraction procedure, based on joint work with R. E. Bryant and S. A. Seshia [16], is presented in Chapter 4. Chapter 5 presents our automatic approach to function abstraction based on random simulation and static analysis. The techniques discussed in Chapter 5 are based on collaborations with R. E. Bryant, S. A. Seshia, and J. W. O'Leary [17]. An alternative approach to automatic function abstraction, based on a combination of machine learning, random simulation, and counterexample-guided abstraction-refinement, is presented in Chapter 6. This technique is the result of joint work with S. A. Seshia [19]. Finally, in Chapter 7, we summarize the techniques and results presented in this thesis and give avenues for future work.

Part I

Background

Chapter 2

Modeling Hardware Systems

The level of detail required to model a hardware system is highly dependent, if not dictated, by the task being performed. For example, SPICE-level models are often used for transistor-level simulation. In this type of simulation, the thickness of wires, the type of dielectric material, and other physical details are used to get an extremely accurate analog picture of how the design is operating. On the other end of the spectrum, in Network-on-Chip (NoC) simulation, processing elements are modeled as black-boxes, ignoring all of the physical details, and keeping only those logical details that control the interaction with the network, or neighboring nodes.

Register-transfer level (RTL) descriptions are often the most authoritative models of hardware systems. RTL descriptions precisely capture the logical details of a circuit, while ignoring most of the physical implementation details. Synthesis tools are used to transform the RTL description into lower-level models, such as SPICE-level models for transistor-level simulation. While RTL descriptions are at a much higher level than transistor-level descriptions, or even Boolean descriptions, they still present a major verification hurdle due to the size and complexity of the circuits being described. In order to perform most formal verification techniques, further abstraction is necessary. The challenge with using abstraction lies in determining what components to abstract and under what conditions.

The remainder of this chapter is organized as follows. Preliminary definitions and notation used throughout this thesis are presented in Section 2.1. The basics of term-level abstraction are introduced in Section 2.2 and term-level modeling is described in Section 2.3. Examples of how term-level abstraction can be applied to a network-on-chip router and a fragment of a processor pipeline in Section 2.4. Finally, we expand our description of correspondence checking in Section 2.5.

2.1 Preliminaries

Word-level netlists are the basic building blocks we use to represent RTL designs.

Definition 2.1. A *word-level netlist* $\mathcal{N}$ is a tuple $(\mathcal{I}, \mathcal{O}, \mathcal{C}, \mathcal{S}, \mathit{Init}, \mathcal{A})$ where

(a) $\mathcal{I}$ is a finite set of input signals;

(b) $\mathcal{O}$ is a finite set of output signals;

(c) $\mathcal{C}$ is a finite set of intermediate combinational (stateless) signals;

(d) $\mathcal{S}$ is a finite set of intermediate sequential (state-holding) signals;

(e) *Init* is a set of initial states, i.e., initial valuations to elements of $\mathcal{S}$, and

(f) $\mathcal{A}$ is a finite set of assignments to outputs and to sequential and combinational intermediate signals. An assignment is an expression that defines how a signal is computed and updated. We elaborate below on the form of assignments.

First, note that input and output signals are assumed to be combinational, without loss of generality. Moreover, although the signals in the designs we consider can all be modeled as bit vectors of varying sizes, it is useful to distinguish Boolean signals for the control logic from bit-vector valued signals modeling the datapath. Boolean and bit-vector signals are defined in Section 2.1.1 and Section 2.1.2, respectively.

Memory is typically modeled as a flat array of bit-vector signals in an RTL design. However, modeling memory in this way leads to extremely large verification models. For instance, state-of-the-art microprocessors typically have several megabytes of on-chip cache and rely on gigabytes of off-chip memory. Representing such memories precisely is beyond the capacity of any formal tool. Therefore, we assume that all memories are modeled abstractly, as described in Section 2.2.3.

Definition 2.2. A *combinational assignment* is a rule of the form $v \leftarrow e$, where v is a signal in the disjoint union $\mathcal{C} \uplus \mathcal{O}$ and e is an expression that is a function of $\mathcal{C} \uplus \mathcal{S} \uplus \mathcal{I}$. Combinational loops are disallowed.

We differentiate between combinational assignments based on the type of the right-hand side expression and write them as follows:

$$b \leftarrow bool \;\mid\; v \leftarrow bv$$

Here *bool* and *bv* represent Boolean and bit-vector expressions in a word-level netlist.

Definition 2.3. A *sequential assignment* is a rule of the form $v := e$, where v is a signal in $\mathcal{S}$ and e is an expression that is a function of $\mathcal{C} \uplus \mathcal{S} \uplus \mathcal{I}$.

Again, we differentiate between sequential assignments based on type, and write them as follows (where b, b_a are any Boolean signals and v, u are any bit-vector signals):

$$b := b_a \;\mid\; v := u$$

Note that we assume that the right-hand side of a sequential assignment is a signal; this loses no expressiveness since we can always introduce a new signal to represent any expression.

Modern hardware systems are typically described in hierarchical fashion. Hierarchically structured designs promote the reuse of components across multiple designs, help divide the design workload by providing boundaries between components, and aid design and verification engineers in understanding the overall design. Hardware description languages, such as Verilog [1] and VHDL [2], provide constructs (e.g., modules and entities) to describe designs hierarchically. We capture this hierarchical structure formally using a *word-level design*. In Chapter 5, we show how the hierarchical structure can be exploited to aid in abstraction.

Definition 2.4. A *word-level design* $\mathcal{D}$ is a tuple $(\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, where $\mathcal{I}$ and $\mathcal{O}$ denote the set of input and output signals of the design, and the design is partitioned into a collection of N word-level netlists.

Definition 2.5. A *well-formed* design is a design such that:

(a) every output of a netlist is either an output of the design or an input to some netlist, including itself;

(b) every input of a netlist is either an input to the design or exactly one output of a netlist.

We refer to the netlists $\mathcal{N}_i$ as *functional blocks*, or *fblocks*.

Consider the example RTL design illustrated in Figure 2.1. The top-level module is A. The remaining modules, B, C, and D, are modules instantiated within A. Note that A, B, C, and D are fblocks. Let $\mathcal{D}_{\mathsf{A}}$ be the word-level design associated with module A. Then, $\mathcal{D}_{\mathsf{A}} = (\mathcal{I}, \mathcal{O}, \{\mathcal{N}_{\mathsf{A}}, \mathcal{N}_{\mathsf{B}}, \mathcal{N}_{\mathsf{C}}, \mathcal{N}_{\mathsf{D}}\})$, where $\mathcal{I} = \{i_0, i_1, i_2, i_3\}$, $\mathcal{O} = \{o_0, o_1, o_2, o_3\}$ and the word-level netlists associated with A, B, C, and D are $\mathcal{N}_{\mathsf{A}}$, $\mathcal{N}_{\mathsf{B}}$, $\mathcal{N}_{\mathsf{C}}$, and $\mathcal{N}_{\mathsf{D}}$, respectively.

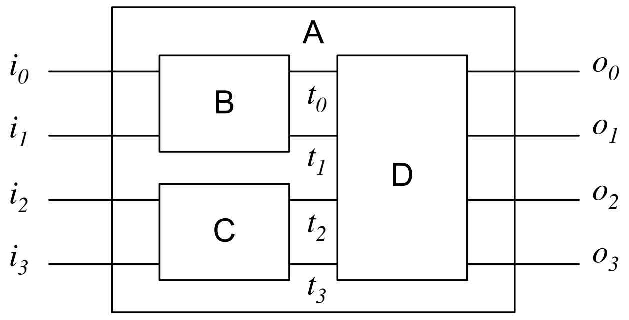


Figure 2.1: **A hierarchical word-level design.** Fblock A is the top-level word-level design and B, C, and D are fblocks contained within A.

2.1.1 Bit-Level Modeling

A Boolean signal (*bit*), b, can take values from the set of Boolean values $\mathcal{B} = \{\textbf{false}, \textbf{true}\}$. A Boolean function, $F_{bool} : \mathcal{B}^n \rightarrow \mathcal{B}$, maps a set of n bits $\{b_1, b_2, ..., b_n\}$ to a single Boolean output $b_{out} = F_{bool}(b_1, b_2, ..., b_n)$. Boolean functions can be represented with Boolean formulas.

Boolean formulas are constructed using Boolean variables, constants, and three propositional operators: negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). In addition to the aforementioned operators, a grouping operator is included in our definition to enforce specific order of operations when needed, and is denoted by parenthesis. A Boolean formula is generated by the grammar in Figure 2.2.

$$\begin{array}{lllllllll} \phi_{bool} & ::= & \textbf{false} & \mid & \textbf{true} & \mid & b \in \mathcal{B} & & \\ & \mid & \neg\phi_{bool} & \mid & \phi_{bool} \wedge \phi_{bool} & \mid & \phi_{bool} \vee \phi_{bool} & \mid & (\phi_{bool}) \end{array}$$

Figure 2.2: **Syntax for Boolean formulas.** A grammar describing Boolean formulas (ϕ_{bool}).

A few examples of Boolean formulas are $\neg a \vee b$, $(a \wedge \neg b) \vee (\neg a \wedge b)$, and $(a \vee \neg b) \wedge (\neg a \vee b)$, where a and b Boolean signals.

In the digital hardware design and verification community, the negation, conjunction, and disjunction operators are often referred to as the **not**, **and**, and **or** operators, respectively. Furthermore, they are denoted by $\overline{a}$, ab or $a \cdot b$, and $a + b$, respectively. For the remainder of this thesis, we will use these terms and notation interchangeably when it is convenient to do so.

In addition to the propositional operators defined above, there are several derived operators that are used frequently in digital hardware design and verification, so we include them here. These additional operators are **nand**, **nor**, **xor**, **xnor**, **implies**, and **equiv**. Figure 2.3 lists the Boolean operations along with their corresponding symbols and propositional representations. A few examples of Boolean formulas using the derived operators are:

$$\overline{a \cdot b} + \overline{c \cdot d} + \overline{e \cdot f}$$

$$a \oplus b \oplus c$$

$$(a \cdot b) + (c \cdot (a \oplus b))$$

Hardware systems, or logic circuits, are often represented graphically with a network of interconnected logic gates. For the remainder of this thesis, the term *circuit* is used to denote

Gate	Boolean Expression	Propositional Expression
not a	$\overline{a}$	$\neg a$
a **and** b	$a \cdot b$	$a \wedge b$
a **nand** b	$\overline{a \cdot b}$	$\neg(a \wedge b)$
a **or** b	$a + b$	$a \vee b$
a **nor** b	$\overline{a + b}$	$\neg(a \vee b)$
a **xor** b	$a \oplus b$	$(a \wedge \neg b) \vee (\neg a \wedge b)$
a **xnor** b	$\overline{a \oplus b}$	$(a \wedge b) \vee (\neg a \wedge \neg b)$
a **implies** b	$a \Rightarrow b$	$\neg a \vee b$
a **equiv** b	$a \iff b$	$(a \wedge b) \vee (\neg a \wedge \neg b)$

Figure 2.3: **Boolean operators.** The Boolean operators along with their corresponding symbols and propositional expressions.

a logic circuit or a hardware system. An example of a circuit represented at the bit level with a Boolean formula is the 1-bit full adder shown in Figure 2.4. Consider the datapaths encoding the *sum* and c_{out} signals. The corresponding Boolean formulas, represented as combinational assignments (Definition 2.2), are:

$$\begin{aligned} sum &\leftarrow c_{in} \oplus (a \oplus b) \\ c_{out} &\leftarrow (a \cdot b) + (c_{in} \cdot (a \oplus b)) \end{aligned}$$

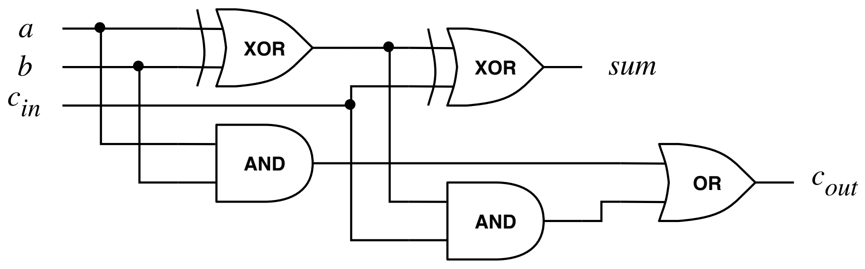


Figure 2.4: **A 1-bit full adder circuit.** a, b, c_{in} are Boolean signals representing the input arguments, and the carry-in signal, respectively. *sum* and c_{out} represent the sum and carry-out signals, respectively.

2.1.2 Bit-vector Level Modeling

A bit-vector signal (*bit vector*), bv, is a sequence of bits $\langle b_1, b_2, ..., b_N \rangle$, where N is the length of bv, and $b_i \in \mathcal{B}$ for all b_i. We use $length(bv)$ to denote the length, or *bit-width*, of the bit-vector signal bv. Let $\mathcal{BV}$ be the set of all finite-length bit-vector signals and $\mathcal{BV}_N \subset \mathcal{BV}$ be the set of all bit-vector signals of length N. A bit-vector function, $F_{bv} : \mathcal{BV}^m \rightarrow \mathcal{BV}$, maps a set of m bit vectors $\{bv_1, bv_2, ..., bv_m\}$ to a single bit vector $bv_{out} = F_{bv}(bv_1, bv_2, ..., bv_m)$. Bit-vector expressions are used to represent bit-vector functions.

Bit-vector expressions (ψ) and formulas (ϕ_{bv}) are constructed from bit-vector constants, signals, and a set of bit-vector operators. A bit-vector constant of length N with value x, where $x \in \mathbb{N}$ is a non-negative integer, is denoted $bv\langle x \rangle_N$. The set of bit-vector components we use, shown in Figure 2.5, is a representative subset of commonly-used operators supported by most bit-vector solvers. Furthermore, the techniques presented in this work can easily be extended to other bit-vector operators. The text based representations (shown in **bold** in Figure 2.5) are used when referring to operators in the text and in figures, in which case, the size of the operator will be obvious from the context or irrelevant. We write formulas and expressions using the corresponding symbolic representation.

Note that all operators are assumed to be unsigned, unless explicitly stated otherwise. A subscripted bit-vector operator denotes the bit-width of the operator. For instance, $a +_4 b$ represents a 4-bit wide addition, where $length(a) = length(b) = 4$. Any time that a bit-vector operator is explicitly sized, the input arguments must have the same length as the operator. A bit-vector operator without a subscript takes the size of the arguments,

Type	Function	Symbol	Description
Arithmetic	**bvneg**	$-\psi_1$	Bit-vector negation
	bvadd	$\psi_1 +_N \psi_2$	Bit-vector addition
	bvsub	$\psi_1 -_N \psi_2$	Bit-vector subtraction
	bvmul	$\psi_1 \times_N \psi_2$	Bit-vector multiplication
	bvdiv	$\psi_1 \div_N \psi_2$	Bit-vector division
Shifting	**bvlsl**	$\psi_1 \ll_N \psi_2$	Logical shift left
	bvlsr	$\psi_1 \gg_N \psi_2$	Logical shift right
	bvasr	$\psi_1 \ggg_N \psi_2$	Arithmetic shift right
Bitwise Logical	**bvnot**	$\sim \psi_1$	Bit-wise not
	bvand	$\psi_1 \ \& \ \psi_2$	Bit-wise and
	bvor	$\psi_1 \mid \psi_2$	Bit-wise or
	bvxor	$\psi_1 \otimes \psi_2$	Bit-wise xor
Relation	**bveq**	$\psi_1 = \psi_2$	Bit-vector equality
	bvneq	$\psi_1 \neq \psi_2$	Bit-vector inequality
	bvgt	$\psi_1 > \psi_2$	Bit-vector greater than
	bvgte	$\psi_1 \geq \psi_2$	Bit-vector greater than or equal
	bvlt	$\psi_1 < \psi_2$	Bit-vector less than
	bvlte	$\psi_1 \leq \psi_2$	Bit-vector less than equal
Bit-manipulation	**extract**	$\psi[msb{:}lsb]$	Bit-vector extraction
	concat	$\psi_1 \bullet \psi_2$	Bit-vector concatenation
	sx	$\psi <S\ w$	Bit-vector sign extension
	zx	$\psi <Z\ w$	Bit-vector zero extension
Conditional	**ite**	$ITE_N(\phi, \psi_1, \psi_2)$	Bit-vector if-then-else

Figure 2.5: **Bit-vector operators.** Bit-vector operators along with their respective symbols and descriptions.

which must be equal (e.g., $length(a \ \& \ b) = length(a) = length(b)$). Bit-vector relations are Boolean expressions with arguments of equal width. The bit-width of the left-hand side of combinational and sequential assignments is determined by the size of the right-hand side.

The extraction operator is parameterized by non-negative integers *msb* and *lsb* where $msb \geq lsb$. The size of an extraction operation is $msb - lsb + 1$. For example, let $x = \langle x_0, x_1, ..., x_{15} \rangle$ be a bit-vector with $length(x) = 16$, then $x[3{:}0] = \langle x_0, x_1, x_2, x_3 \rangle$, where $length(x[3{:}0]) = 4$. A concatenation operator takes two arguments, x and y. The size of a concatenation is the sum of the bit-widths of each input argument, $length(x) + length(y)$. The sign- and zero- extension operators are parameterized by a non-negative integer w. They each take an argument with length $\leq w$, and produce a bit-vector of size w.

We also define bvop, bvmanip and bvrel to be an arbitrary bit-vector arithmetic operator, bit manipulation operator, or relation, respectively, for cases when it doesn't matter what specific operation is being performed. bvop can be any one of the arithmetic, shift, or logical operators, bvmanip can be any of the bit-manipulation operators and bvrel can be any relation operator listed in Figure 2.5.

A bit-vector formula, ϕ_{bv}, extends the definition of Boolean formulas to include bit-vector relations. Bit-vector expressions and formulas are described by the grammar in Figure 2.6.

$$
\begin{array}{llllllll}
\psi & ::= & bv\langle x\rangle_n & \mid & bv \in \mathcal{BV} & & \\
 & \mid & \textbf{bvneg}\ \psi & \mid & \textbf{bvnot}\ \psi & & \\
 & \mid & \psi\ \textbf{bvadd}\ \psi & \mid & \psi\ \textbf{bvsub}\ \psi & & \\
 & \mid & \psi\ \textbf{bvmul}\ \psi & \mid & \psi\ \textbf{bvdiv}\ \psi & & \\
 & \mid & \psi\ \textbf{bvlsl}\ \psi & \mid & \psi\ \textbf{bvlsr}\ \psi & \mid & \psi\ \textbf{bvasr}\ \psi \\
 & \mid & \psi\ \textbf{bvand}\ \psi & \mid & \psi\ \textbf{bvor}\ \psi & \mid & \psi\ \textbf{bvxor}\ \psi \\
 & \mid & \textbf{extract}\ \mathit{msb}\ \mathit{lsb}\ \psi & \mid & \textbf{concat}\ \psi\ \psi & & \\
 & \mid & \textbf{sx}\ w\ \psi & \mid & \textbf{zx}\ w\ \psi & & \\
 & \mid & \textbf{ite}\ \phi\ \psi\ \psi & & & & \\
 & & & & & & \\
\phi_{bv} & ::= & \phi_{bool} & \mid & (\phi_{bv}) & & \\
 & \mid & \neg\phi_{bv} & \mid & \phi_{bv} \wedge \phi_{bv} & \mid & \phi_{bv} \vee \phi_{bv} \\
 & \mid & \psi\ \textbf{bveq}\ \psi & \mid & \psi\ \textbf{bvgt}\ \psi & \mid & \psi\ \textbf{bvlt}\ \psi \\
 & \mid & \psi\ \textbf{bvneq}\ \psi & \mid & \psi\ \textbf{bvgte}\ \psi & \mid & \psi\ \textbf{bvlte}\ \psi
\end{array}
$$

Figure 2.6: **Syntax for bit-vector expressions and formulas.** A grammar describing bit-vector expressions ψ and bit-vector formulas ϕ_{bv}.

Examples of bit-vector formulas are

$$(a \times_{32} b) = (b \times_{32} a)$$

$$\mathit{ITE}_{32}(a > b, a, b) > (a +_{32} b)$$

$$\mathit{bv6}_{16} = \mathit{bv1}_{16} + \mathit{bv2}_{16} + \mathit{bv3}_{16}$$

where $a, b \in \mathcal{BV}_{32}$.

An example of a 4-bit addition circuit modeled at the bit-level and the corresponding bit-vector model is shown in Figure 2.7. Each full adder (FA) component in Figure 2.7(a) is implemented with the circuit shown in Figure 2.4. Without any optimizations, this circuit is represented with 20 logic gates and the size is linear in the number of input bits. The bit-vector model is shown in Figure 2.7(b). Regardless of the bit-width of the addition, a bit-vector addition is represented with a single operator parameterized by a non-negative integer. While it is possible for the bit-vector model to be converted into its bit-level equivalent, modeling operators at the bit-vector level requires less space and allows decision procedures to exploit higher-level properties such as commutativity and associativity.

An example of a word-level netlist $\mathcal{N}_{ex}$ is shown in Figure 2.8. $\mathcal{N}_{ex}$ is a 4-bit counter with reset functionality. On each clock cycle, c is updated with the sequential assignment $c := y$, unless the *reset* signal is asserted, in which case, $c := \mathit{bv0}_4$. The formal word-level description for $\mathcal{N}_{ex}$ is $\mathcal{I} = \{\mathit{reset}\}$, $\mathcal{O} = \{\mathit{out}\}$, $\mathcal{C} = \{x, y\}$, $\mathcal{S} = \{c\}$, $\mathit{Init} = \{c := \mathit{bv0}_4\}$, $\mathcal{A} = \{x \leftarrow c +_4 \mathit{bv1}_4, y \leftarrow \mathit{ITE}_4(\mathit{reset}, \mathit{bv0}_4, x), c := y, \mathit{out} \leftarrow c\}$. The corresponding word-level design is $\mathcal{D}_{ex} = (\mathcal{I}, \mathcal{O}, \{\mathcal{N}_{ex}\})$.

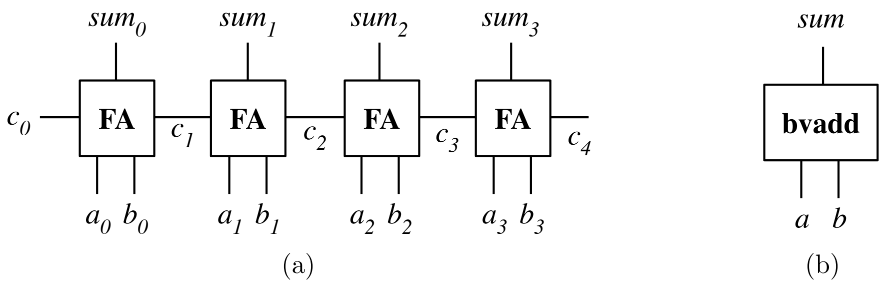


Figure 2.7: **Comparison of bit-level and bit-vector addition circuits.** A bit-level and bit-vector level model of a 4-bit addition circuit.

An example of a property that can be proved on the counter circuit is $(reset) \implies (next(out) == bv0_4)$. This states that *out* must be equal to $bv0_4$ on the cycle *after* the *reset* signal is asserted. The notation $next(x)$ represents the next state of a signal and is defined formally in Section 3.5.

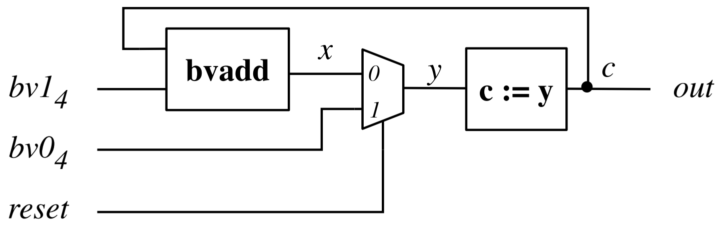


Figure 2.8: **Example word-level design.** A 4-bit counter with reset functionality.

Creating a verification model, or word-level design, from an RTL description is a straightforward process. In most cases, it is possible to directly and automatically translate RTL descriptions in languages such as Verilog [1] and VHDL [2] into bit-vector expressions. This translation is discussed further in Chapter 3

2.2 Term-Level Abstraction

Term-level abstraction is a technique used to abstract word-level designs in a formal logic. The main goal of term-level abstraction is to create verification models that are easier to reason about than the original, word-level designs. Term-level abstraction is especially powerful when applied to designs with data-insensitive control flow, where the specific values of data have limited influence on the control logic.

Term-level abstraction relies on a combination of techniques, such as representing terms over an abstract domain, the logic of equality with uninterpreted functions (EUF), and a restricted form of lambda expressions. Modeling signals over an abstract domain allows us to create smaller verification models. Uninterpreted functions allow us to model complex circuitry abstractly which can help improve verification performance. Lambda expressions provide the constructs necessary to model data structures such as memories and queues. The three main flavors of term-level abstraction – data abstraction, function abstraction, and memory abstraction – are defined in the remainder of this section.

2.2.1 Data Abstraction

In data abstraction, bit-vector expressions are modeled as abstract *terms* that are interpreted over a suitable domain (typically a subset of $\mathbb{Z}$). Data abstraction is effective when it is possible to reason over the domain of abstract terms far more efficiently than it is to do so over the original bit-vector domain. The most basic form of data abstraction is to model terms using *the logic of equality*.

In equality logic, an abstract *term* is an expression over an abstract domain $\mathcal{Z} \subseteq \mathbb{Z}$. Formulas are constructed in equality logic using equality between terms and the usual Boolean connectives. Equality logic expressions (τ) and formulas (ϕ_E) are generated by the grammar shown in Figure 2.9, where v is a variable over $\mathcal{Z}$ and $term\langle x\rangle$ is a symbolic constant, where $x \in \mathcal{Z}$.

$$\begin{array}{lcccccc}
\phi_E & ::= & \neg\phi_E & \mid & (\phi_E) & & \\
 & \mid & \phi_E \wedge \phi_E & \mid & \phi_E \vee \phi_E & \mid & \tau = \tau \\
 & & & & & & \\
\tau & ::= & v \in \mathcal{Z} & \mid & term\langle x\rangle & \mid & \textbf{ite } \phi_E\ \tau\ \tau
\end{array}$$

Figure 2.9: **Syntax for equality logic expressions and formulas.** A grammar describing expressions and formulas in equality logic.

Examples of formulas in equality logic are

$$a = b \wedge b = c \wedge a \neq c$$

$$a = b \wedge a = 2$$

where a, b, and c are abstract terms. Note that we use $a \neq b$ as shorthand for $\neg(a = b)$.

The power of data abstraction comes from the fact that abstract terms do not have a specific size. In many practical situations, word-level designs use far more bits than are necessary in order to prove a given property. Equality logic allows us to take advantage of the *finite model property* which states that every satisfiable formula in this logic has a satisfying interpretation of finite size. Additionally, we can compute the bound on the size of variables in equality logic formulas [51, 9]. We discuss the finite model property and how it is used in encoding equality logic formulas as propositional formulas in Section 3.4.2. Figure 2.10 illustrates the difference between modeling with bit-vectors and terms. While bit-vector signals such as $\langle x_n, ..., x_2, x_1\rangle$ in Figure 2.10(a) have an associated size and encoding (e.g., unsigned, two's complement), term signals such as x in Figure 2.10(b) have no associated size or encoding.

Notice the similarity between equality logic and the fragment of the theory of bit-vectors that includes only bit-vector equalities over variables and constants. The only difference is

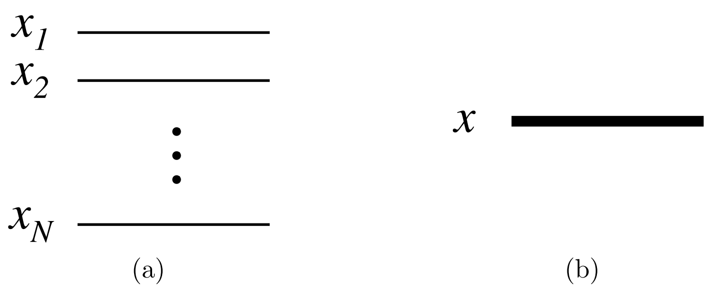


Figure 2.10: **Comparison between bit-vector and term signals.** Part (a) shows a bit-vector signal x of length N where each bit is represented precisely and part (b) shows a term-level representation of x where there is no associated size.

the type of the basic expression. Equality logic formulas are constructed using equalities between abstract terms, not bit vectors. This similarity can be exploited by representing portions of bit-vector formulas that contain only bit-vector equalities in equality logic, instead of the theory of bit-vectors. The intuition behind this abstraction is that formulas will require fewer bits to represent abstractly in equality logic, than precisely with bit-vectors, and this will lead to easier-to-verify models.

2.2.2 Function Abstraction

When function abstraction is employed, portions of a word-level design are treated abstractly as "black-boxes" using *uninterpreted* functions. Individual operators or entire netlists can be abstracted using uninterpreted functions.

An *uninterpreted function*, *UF*, is a function that is constrained only by functional consistency. Functional consistency (also called functional congruence) states that a function must evaluate to the same value when applied to equal arguments

$$\forall x_1, ..., x_n y_1, ..., y_n . x_1 = y_1 \wedge ... \wedge x_n = y_n \implies UF(x_1, ..., x_n) = UF(y_1, ..., y_n)$$

where x_i and y_i for all i are terms.

Unlike the Boolean and bit-vector functions described in Section 2.1.2, an uninterpreted function can be applied to arguments of any type, and yield an expression of any type. Let $\mathcal{V} = \mathcal{B} \uplus \mathcal{BV} \uplus \mathcal{Z}$ be the set of all values a signal can take. An uninterpreted function $UF : \mathcal{V}^n \rightarrow \mathcal{V}$, maps a set of n values to a single value. We differentiate uninterpreted functions by the type of their output. Uninterpreted bit-vector functions map $\mathcal{V}^n \rightarrow \mathcal{BV}$, uninterpreted term functions map $\mathcal{V}^n \rightarrow \mathcal{Z}$, and uninterpreted predicates map $\mathcal{V}^n \rightarrow \mathcal{B}$. Thus, uninterpreted functions can be added to both equality logic and the theory of bit-vectors.

The logic of equality with uninterpreted functions (EUF) is an extension of equality logic where expressions can now be the result of uninterpreted function applications and formulas can be the result of uninterpreted predicate applications. Uninterpreted functions can also be added to the theory of bit-vectors. The theory of bit-vector with uninterpreted functions is referred to as BVUF. BVUF allows one to represent precise, bit-vector operators abstractly, with the intention of creating verification models that are smaller and/or easier-to-verify than the original bit-vector model.

Formulas and expressions in EUF are described by the grammar shown in Figure 2.11. Formulas and expressions in BVUF extend the bit-vector grammar shown in Figure 2.6. Only these extensions to the bit-vector grammar are shown in Figure 2.12. Also note that it is possible to use a combination of EUF and BVUF, where the input arguments to uninterpreted functions can have any type $t \in \mathcal{V}$.

$$
\begin{array}{rclcccc}
\phi_{UF} & ::= & \textbf{false} & | & \textbf{true} & | & v \in \mathcal{B} \\
& | & \neg\phi_{UF} & | & \phi_{UF} \wedge \phi_{UF} & | & \phi_{UF} \vee \phi_{UF} \\
& | & \tau_{UF} = \tau_{UF} & | & UF_{bool}(t_1, t_2, ..., t_n) & | & (\phi_{UF}) \\
\\
\tau_{UF} & ::= & v \in \mathcal{Z} & | & term\langle x \rangle & & \\
& | & ITE(\phi,\ \tau_{UF},\ \tau_{UF}) & | & UF_{term}(t_1, t_2, ..., t_n) & &
\end{array}
$$

Figure 2.11: **Syntax for expressions and formulas in EUF.** Formulas in EUF are Boolean combinations of equalities between terms where terms are constants, variables, or the result of uninterpreted function applications.

$$
\begin{array}{rcl}
\phi_{UF} & ::= & UF_{bool}(bv_1, bv_2, ..., bv_n) \\
\\
\psi_{UF} & ::= & UF_{bv}(bv_1, bv_2, ..., bv_n)
\end{array}
$$

Figure 2.12: **Syntax for expressions and formulas in BVUF.** BVUF extends the grammar shown in Figure 2.6. Only these extensions are shown here.

Examples of formulas that are in EUF are

$$t_1 = f(t_2, t_3) \ \wedge\ t_1 = f(t_3, t_2)$$

$$t_1 = t_2 \ \wedge\ t_2 = g(t_2) \ \wedge\ h(b_1)$$

where $t_i \in \mathcal{Z}$, $b_i \in \mathcal{B}$, $f \in \mathcal{Z}^2 \rightarrow \mathcal{Z}$, $g \in \mathcal{Z} \rightarrow \mathcal{Z}$, and $h \in \mathcal{B} \rightarrow \mathcal{B}$.

Examples of formulas in BVUF are

$$bv_1 = f(bv_2, bv_3) \ \wedge\ bv_1 = f(bv_3, bv_2)$$

$$bv_1 +_N bv_2 = g(bv_1, bv_2)$$

where $bv_i \in \mathcal{BV}_N$ and $f, g \in \mathcal{BV}_N^2 \rightarrow \mathcal{BV}_N$.

The function abstraction techniques in this work focus on BVUF. The main goal of replacing a bit-vector fblock with an uninterpreted function is to reduce the complexity of the circuit being verified. It is often more efficient to reason about uninterpreted functions than bit-vector operators such as multiplication and division.

It is always sound to replace a combinational fblock with an uninterpreted function in the sense that if the abstracted model is valid, then so is the original, word-level model. This is the case because an uninterpreted function contains more behaviors than the original, precise function (i.e., it is an over-approximation).

Note that soundness only holds for replacing a combinational fblock with an uninterpreted function. In order for soundness to hold for a sequential fblock, we must take into account a bounded amount of history for the inputs of the fblock. Furthermore, the sequential fblock must be acyclic. The key point here is that the uninterpreted function must be an over-approximation of the acyclic, sequential fblock being replaced for the abstraction to be sound. In order to over-approximate a sequential fblock, every possible unique sequence of inputs must be accounted for. Thus, if there are n latches in a path within a sequential fblock, then we must account for history up to a depth of 2^n. We discuss this further in Section 5.2. For the remainder of this thesis, it is assumed that any sequential fblock being abstracted or being considered for abstraction is acyclic.

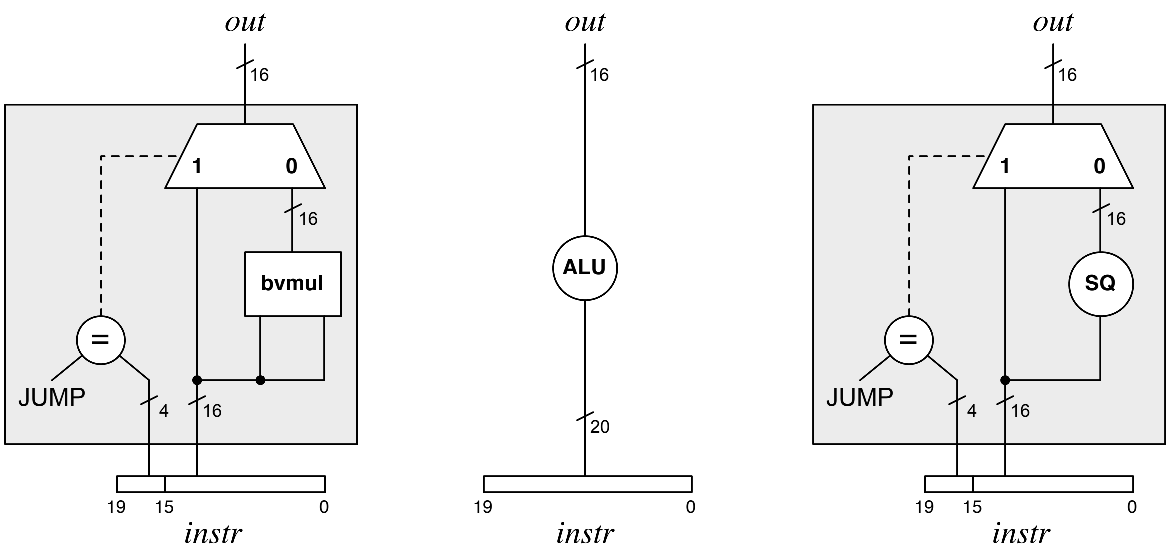
Note that even with a sound abstraction, the additional behaviors introduced by uninterpreted functions can lead to spurious counterexamples. Spurious counterexamples are counterexamples that occur in the abstracted model but not the original model.

ALU Example We illustrate the concept of function abstraction using a toy ALU design. Consider the simplified ALU shown in Figure 2.13(a). Here a 20-bit instruction is split into a 4-bit opcode and a 16-bit data field. If the opcode indicates that the instruction is a jump, then the data field contains a target address for the jump and is simply passed through the ALU unchanged. Otherwise, the ALU computes the square of the 16-bit data field and generates as output the resulting 16-bit value.

Using very coarse-grained term-level abstraction, one could abstract the entire ALU module with a single uninterpreted function (ALU) , that maps the 20-bit instruction to a 16-bit output, as shown in Figure 2.13(b). However, we lose the precise mapping from *instr* to *out*.

Such a coarse abstraction is quite easy to perform automatically. However, this abstraction loses information about the behavior of the ALU on jump instructions and can easily result in spurious counterexamples. In Section 2.4, we will describe a larger equivalence checking problem within which such an abstraction is too coarse to be useful.

Suppose that reasoning about the correctness of the larger circuit containing this ALU design only requires one to precisely model the difference in how the jump and squaring instructions are handled. In this case, it would be preferable to use a partially-interpreted ALU model as depicted in Figure 2.13(c). In this model, the control logic distinguishing the handling of jump and non-jump instructions is precisely modeled, but the datapath is abstracted using the uninterpreted function SQ. However, creating this fine-grained abstraction by hand is difficult in general and places a larger burden on the designer. A main goal of this thesis is to mitigate this burden.



(a) Original word-level ALU (b) Fully uninterpreted ALU (c) Partially interpreted ALU

Figure 2.13: **Three versions of an ALU design with varying levels of abstraction.** Boolean signals are shown as dashed lines and bit-vector signals as solid black lines. The uninterpreted function ALU in part (b) has type $\mathcal{BV}_{20} \rightarrow \mathcal{BV}_{16}$, while SQ in part (c) has type $\mathcal{BV}_{16} \rightarrow \mathcal{BV}_{16}$.

2.2.3 Memory Abstraction

While this work does not address automatic memory abstraction, memory abstraction is considered to be a form of term-level abstraction [17] and is employed in the case studies used throughout this thesis. Thus, a brief discussion of memory abstraction is warranted.

The goal of memory abstraction is to accurately represent only as many memory locations as necessary, instead of representing an entire memory as a flattened array of bit vectors. The memory abstraction technique we use relies on a restricted form of *lambda expressions* described by Seshia *et al.* [29, 53, 63]. Lambda expressions are a versatile construct that are especially useful in modeling data structures, such as stacks, queues, and random-access memory (RAM). We describe how to model random-access memory (RAM) with lambda expressions. The interested reader is referred to [63] for a description of other data structures that can be modeled with lambda expressions.

A *memory expression* is a function expression M that maps addresses to values. A memory expression has the type $\mathcal{V}^n \rightarrow \mathcal{V}$, where n is the dimensionality of the memory. We define memory operations **read** and **write** to represent reading from and writing to memories. A read operation on a 1-dimensional memory M at address v is encoded as the function application $M(v)$ and is denoted $\textbf{read}(M, v)$, where $M \in \mathcal{V} \rightarrow \mathcal{V}$, $v \in \mathcal{V}$, and $\textbf{read}(M, v) \in \mathcal{V}$. A write operation on memory M with address v and data d, denoted

$\mathbf{write}(M, v, d)$, is modeled with the lambda expression

$$M' := \lambda addr.ITE(addr = v, d, M(addr))$$

where $M' \in \mathcal{V} \to \mathcal{V}$. Note that while the result of a memory read operation is a *value* of type $\mathcal{V}$, the result of a memory write operation is a new *memory expression* with type $\mathcal{V} \to \mathcal{V}$.

Memory reads (writes) are typically modeled using combinational (sequential) assignments. Consider the effect of a sequence of n memory write operations,

$$\begin{gathered}\mathbf{write}(M, v_1, d_1)\\ \mathbf{write}(M, v_2, d_2)\\ \vdots\\ \mathbf{write}(M, v_n, d_n)\end{gathered}$$

where data d_i is written to address v_i in memory M on clock cycle i. The above sequence of memory writes is synonymous with the following sequence of sequential assignments:

$$\begin{aligned}M_1 &:= \lambda addr.ITE(addr = v_1, d_1, M_0)\\ M_2 &:= \lambda addr.ITE(addr = v_2, d_2, M_1)\\ \vdots &:= \qquad\qquad \vdots\\ M_n &:= \lambda addr.ITE(addr = v_n, d_n, M_{n-1})\end{aligned}$$

where M_0 is the initial state of memory M and M_i is the state of M after the i-th write operation. The initial state M_0 is modeled with a read-only memory (i.e., a memory without an associated write operation).

In the example given above, we use a 1-dimensional memory, however, it is possible to have multidimensional memories where the address argument v would be replaced with a tuple of address arguments $(v_1, v_2, ..., v_n)$ such that $v_i \in \mathcal{V}$ for $i = 1, ..., n$. In this case, we let V denote the multidimensional address into a memory M.

In addition to modeling RAM, it is possible to model other memory elements, such as queues and stacks, and arbitrary Boolean, bit-vector and term expressions using lambda expressions. The only restriction placed on the usage of lambda expressions is that the arguments must be Boolean, bit-vector or term expressions. Thus, it is impossible to express iteration or recursion [63]. Lambda expressions can be used to mimic the behavior associated with a hierarchical construct (e.g., module, entity). However, it is important to note that, lambda expressions must be removed from the formula before invoking a solver. Due to the restriction placed on lambda arguments (i.e., they must be Boolean, bit-vector, or term signals), *beta substitution* can be used to eliminate lambda expressions from the formula [63]. In some cases, this procedure can incur significant overhead within decision procedures.

Thus, it is prudent to use lambda expressions only when necessary (e.g., abstracting a large memory), especially when the same behavior can be obtained by other means (e.g., using a module).

As with uninterpreted functions, memories can have arguments of any type and can yield values of any type. We differentiate between memory expressions based on the type of the data being stored in the memory. The grammar describing memory expressions is shown in Figure 2.14. The syntax for a memory read or memory write is the same regardless of the type of the output. Thus, we show only the general form in Figure 2.14 instead of listing the syntax for each type.

$$\begin{array}{lcl} M_t & ::= & \mathbf{write}(M_t, V, d_t) \quad | \quad C_t \quad | \quad M_t \\ \\ e_t & ::= & \mathbf{read}(M_t, V) \end{array}$$

Figure 2.14: **Syntax for memory expressions.** M_t is a memory expression with type t, where $t \in \{\mathcal{B}, \mathcal{BV}, \mathcal{Z}\}$. C_t denotes a constant memory expression of type t. V denotes the address argument(s) to the memory read and write operators, where $V \in \mathcal{V}^n$, and d_t is the data being written to memory during a memory write. e_t is the expression with type t resulting from a read from memory M_t

2.3 Term-Level Modeling

Term-level modeling involves using abstraction to hide implementation details in the hope of creating easier-to-verify models. Informally, term-level models are extensions of word-level models where expressions can be represented precisely with bits and bit vectors as described in Section 2.1 or abstractly with terms using the abstraction techniques described in Section 2.2.

Definition 2.6. A *term-level netlist* is a generalization of a word-level netlist where expressions can be from the syntax shown in Figure 2.15.

Term-level netlists can also contain sequential and combinational assignments to memory variables. Assignments of this form are also allowed within a word-level netlist. As memory abstraction is not the focus of this thesis, we do not consider a netlist where memory abstraction is the only form of abstraction to be a term-level netlist.

Definition 2.7. A *strict term-level netlist* is a term-level netlist that satisfies one or more of the following properties:

(a) There exists a signal $s \in \mathcal{I} \uplus \mathcal{O} \uplus \mathcal{C} \uplus \mathcal{S}$ such that s has type $\mathcal{Z}$.

(b) There exists an assignment $a \in \mathcal{A}$ such that the right-hand side of a is of the form: $UF_{bool}(V)$, $UF_{bv}(V)$, or $UF_{term}(V)$, where $V \in \mathcal{V}^n$ for some n.

$$
\begin{array}{lllllll}
\phi & ::= & \mathbf{false} & \mid & \mathbf{true} & & \\
 & \mid & b \in \mathcal{B} & \mid & (\phi) & & \\
 & \mid & \neg\phi & \mid & \phi \wedge \phi & \mid & \phi \vee \phi \\
 & \mid & \psi\ \mathbf{bveq}\ \psi & \mid & \psi\ \mathbf{bvgt}\ \psi & \mid & \psi\ \mathbf{bvlt}\ \psi \\
 & \mid & \psi\ \mathbf{bvneq}\ \psi & \mid & \psi\ \mathbf{bvgte}\ \psi & \mid & \psi\ \mathbf{bvlte}\ \psi \\
 & \mid & \tau = \tau & \mid & \tau \neq \tau & & \\
 & \mid & \mathit{UF}_{\mathit{bool}}(V) & \mid & \mathbf{read}(M_{\mathit{bool}}, V) & & \\
 & & & & & & \\
\psi & ::= & bv\langle x\rangle_n & \mid & bv \in \mathcal{BV} & \mid & (\psi) \\
 & \mid & \mathbf{bvneg}\ \psi & \mid & \mathbf{bvnot}\ \psi & & \\
 & \mid & \psi\ \mathbf{bvadd}\ \psi & \mid & \psi\ \mathbf{bvsub}\ \psi & & \\
 & \mid & \psi\ \mathbf{bvmul}\ \psi & \mid & \psi\ \mathbf{bvdiv}\ \psi & & \\
 & \mid & \psi\ \mathbf{bvlsl}\ \psi & \mid & \psi\ \mathbf{bvlsr}\ \psi & \mid & \psi\ \mathbf{bvasr}\ \psi \\
 & \mid & \psi\ \mathbf{bvand}\ \psi & \mid & \psi\ \mathbf{bvor}\ \psi & \mid & \psi\ \mathbf{bvxor}\ \psi \\
 & \mid & \mathbf{extract}\ \mathit{msb}\ \mathit{lsb}\ \psi & \mid & \mathbf{concat}\ \psi\ \psi & & \\
 & \mid & \mathbf{sx}\ w\ \psi & \mid & \mathbf{zx}\ w\ \psi & & \\
 & \mid & \mathit{UF}_{\mathit{bv}}(V) & \mid & \mathbf{read}(M_{\mathit{bv}}, V) & \mid & \mathbf{ite}\ \phi\ \psi\ \psi \\
 & & & & & & \\
\tau & ::= & \mathit{term}\langle x\rangle & \mid & t \in \mathcal{Z} & & \\
 & \mid & \mathit{UF}_{\mathit{term}}(V) & \mid & \mathbf{read}(M_{\mathit{term}}, V) & \mid & \mathbf{ite}\ \phi\ \tau\ \tau \\
 & & & & & & \\
M_{\mathit{bool}} & ::= & \mathbf{write}(M_{\mathit{bool}}, V, d) & \mid & C_{\mathit{bool}} & \mid & M_{\mathit{bool}} \\
 & & & & & & \\
M_{\mathit{bv}} & ::= & \mathbf{write}(M_{\mathit{bv}}, V, d) & \mid & C_{\mathit{bv}} & \mid & M_{\mathit{bv}} \\
 & & & & & & \\
M_{\mathit{term}} & ::= & \mathbf{write}(M_{\mathit{term}}, V, d) & \mid & C_{\mathit{term}} & \mid & M_{\mathit{term}}
\end{array}
$$

Figure 2.15: **Syntax for formulas and expressions used in term-level netlists.** Boolean expressions are denoted with ϕ, bit-vector expressions with ψ, and term expressions with τ. Uninterpreted functions, memory expressions, and constant memory expressions are differentiated by the type of their output and are denoted with UF_t, M_t, and C_t, respectively, where $t \in \{\mathit{bool}, \mathit{bv}, \mathit{term}\}$. V denotes a tuple of arguments to an uninterpreted function or memory operation where each $v \in V$ has type $\mathcal{V}$. n denotes the length of a bit-vector constant, while x denotes the value of a bit-vector or term constant. w denotes the size of a bit-vector signal after sign-/zero- extension is performed, while msb and lsb denote the most and least significant bits of a bit-vector extraction. Recall that certain bit-vector operators can be annotated with size information. Refer to Figure 2.5 for this size information.

In other words, Definition 2.7 states that data abstraction has been performed on at least one signal or that function abstraction has been performed on at least one fblock. The set of all word-level netlists is a (strict) subset of the set of all (strict) term-level netlists.

Definition 2.8. A *pure term-level netlist* $\mathcal{P} = (\mathcal{I}, \mathcal{O}, \mathcal{C}, \mathcal{S}, \mathit{Init}, \mathcal{A})$ is a term-level netlist such that each signal $s \in \mathcal{I} \uplus \mathcal{O} \uplus \mathcal{C} \uplus \mathcal{S}$ has type $\mathcal{Z}$.

The notion of a pure term-level netlist comes into play when performing data abstraction. In this situation, because each signal is being interpreted abstractly over $\mathcal{Z}$, it is possible to encode the verification problem more concisely. Chapter 4 discusses this procedure in detail.

Definition 2.9. A *term-level design* $\mathcal{T}$ is a tuple $(\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, where at least one fblock $\mathcal{N}_i$ is a strict term-level netlist.

Given a word-level design $\mathcal{D} = (\mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\})$, we say that $\mathcal{T}$ is a *term-level abstraction* of $\mathcal{D}$ if $\mathcal{T}$ is obtained from $\mathcal{D}$ by replacing some word-level fblocks $\mathcal{N}_i$ by strict term-level fblocks $\mathcal{N}_i'$.

2.4 Modeling Examples

In this section we present examples that illustrate how data and function abstraction can be used to create easier-to-verify designs. We show how data abstraction can be applied to a chip-multiprocessor router in Section 2.4.1 and how function abstraction can be performed on a fragment of a pipelined processor.

2.4.1 Chip-Multiprocessor Router

Network-on-chip (NoC) architectures are the backbone of modern, multicore processors, serving as the communication fabric between processor cores and other on-chip devices such as memory. It is important to prove that individual routers and networks of interconnected routers operate properly. We show how data abstraction can be used to reduce the size of the verification model of a chip-multiprocessor (CMP) router which leads to smaller verification runtimes.

The CMP router design [59] we focus on is part of an on-chip interconnection network that connects processor cores with memory and with each other. The main function of the router is to direct incoming packets to the correct output port. Each packet is made up of smaller components called *flits*. There are three kinds of flits: a *head flit*, which reserves an output channel, one or more *body flits*, which contain the data payload, and a *tail flit*, which signals the end of the packet. The anatomy of a flit is depicted in Figure 2.16. The two least-significant bits represent the flit type, the next 6 most-significant bits represent the destination address, and the 24 most significant bits contain the data payload.

The CMP router consists of four main modules, as shown in Figure 2.17. The *input controller* buffers incoming flits and interacts with the *arbiter*. Upon receipt of a head flit,

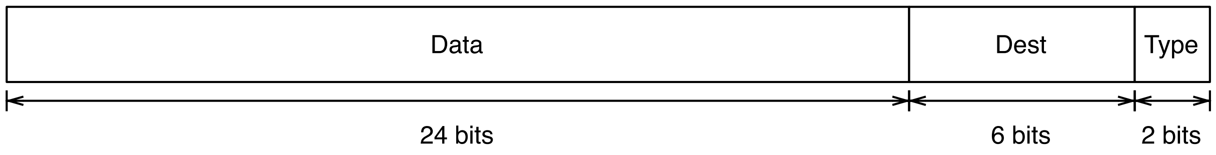


Figure 2.16: **Anatomy of a flit.** A flit contains a 24-bit data payload, a 6-bit destination address, and a 2-bit type.

the input controller requests access to an output port based on the destination address contained in the head flit. The arbiter grants access to the output ports in a fair manner, using a simple round-robin arbitration scheme. The remaining modules are the *encoder* and *crossbar*. When the arbiter grants access to a particular output port, a signal is sent to the input controller to release the flits from the buffers, and at the same time, an allocation signal is sent to the *encoder* which in turn configures the *crossbar* to route the flits to the appropriate output port.

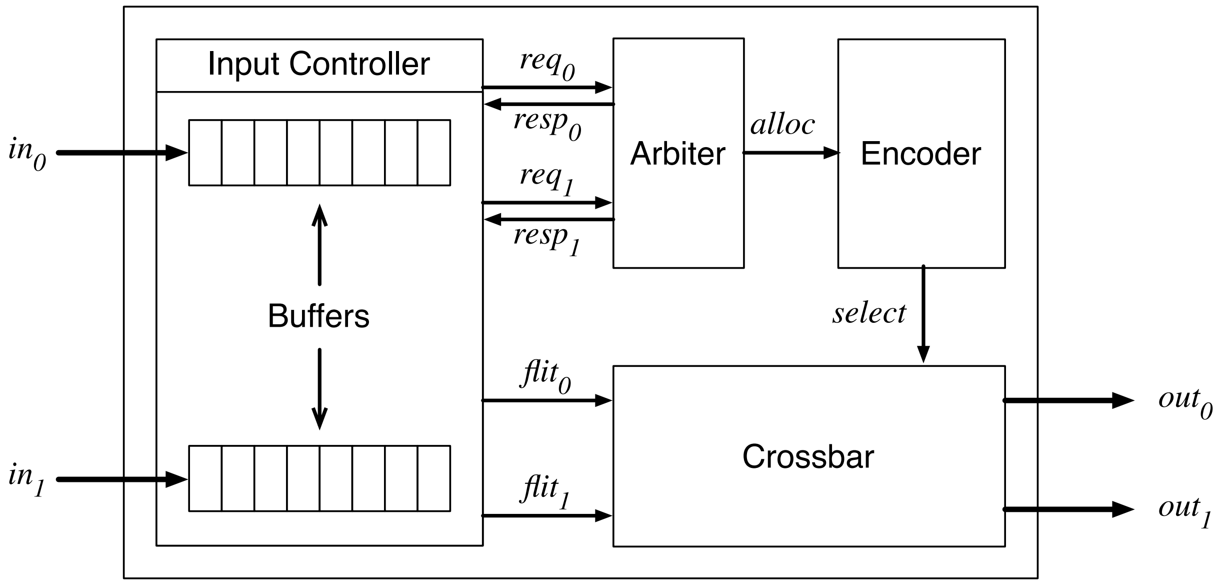


Figure 2.17: **Chip-Multiprocessor (CMP) Router.** There are four main modules: the input controller, the arbiter, the encoder, and the crossbar.

Consider a word-level representation of the CMP router, where each packet is represented with 32-bits. Examples of correctness properties that one might wish to prove on the CMP router are: 1) packets are being routed to the correct output port and 2) packets spend no more than N clock cycles within a router. For example, if packet p is sent to input port in_0 of router R then it must appear on output port out_1 within N clock cycles. This property can be formalized as a disjunction of equalities,

$$\phi_{cmp} \equiv \bigvee_{i=1}^{N} in_{0,1} = out_{1,N}$$

where $in_{i,j}$ ($out_{i,j}$) denotes the packet on input (output) port i on clock cycle j.

The property ϕ_{cmp} is expressible in equality logic, which leads to the obvious question: is it possible to encode the CMP router where the flit datapath signals are terms, instead of bit vectors? Unfortunately, it is not possible to do so because the router requires that the destination and type fields of a flit be represented precisely. The routing logic uses bit-vector inequality relations to determine the correct output channel and the input controller uses the flit type to coordinate its interaction with the arbiter. While it is not possible to encode the entire flit datapath as a term, it is possible to encode the data payload as a term. This is possible because the only operation performed on the data portion of a flit is equality. Thus, a flit would then be represented with 8-bits and a term. Decision procedures for equality logic are then responsible for computing the appropriate number of bits to accurately represent the data payload.

As long as data abstraction is only applied to the top 24 bits of the flit datapath, the resulting abstracted model will be valid if and only if the word-level model is valid. Let $\mathcal{D}_{cmp}$ be the word-level CMP router and $\mathcal{T}_{cmp}$ be the term-level CMP router in which the top 24 bits are abstracted with a term. Then $\mathcal{D}_{cmp} \models \phi_{cmp} \iff \mathcal{T}_{cmp} \models \phi_{cmp}$

In Chapter 4, we show how this process can be automated and show that it results in a smaller, easier-to-verify CMP router model.

2.4.2 Processor Fragment

Function abstraction is especially useful when verifying data insensitive properties. One such example is in microprocessor design verification, where it is possible to abstract functionality such as the ALU and branch prediction logic to create easier-to-verify models [31, 52]. In this section we give an example of how function abstraction can be applied to the domain of processor verification. Figure 2.18 illustrates an equivalence checking problem between two versions of a processor fragment.

Consider Design A. This design models a fragment of a processor datapath. PC models the program counter register, which is an index into the instruction memory denoted as IMem. The instruction is a 20-bit word denoted $instr$, and is an input to the ALU design shown earlier in Figure 2.13(a). The top four bits of $instr$ are the operation code. If the instruction is a jump instruction (i.e., $instr$[19:16] equals JUMP), then the PC is set equal to the ALU output out_A; otherwise, it is incremented by 4.

Design B is virtually identical to Design A, except in how the PC is updated. For this version, if $instr$[19:16] equals JUMP, the PC is directly set to be the jump address $instr$[15:0].

Note that we model the instruction memory as a read-only memory using an uninterpreted function IMem. The same uninterpreted function is used for both Design A and Design B. We also assume that Designs A and B start out with identical values in their PC registers.

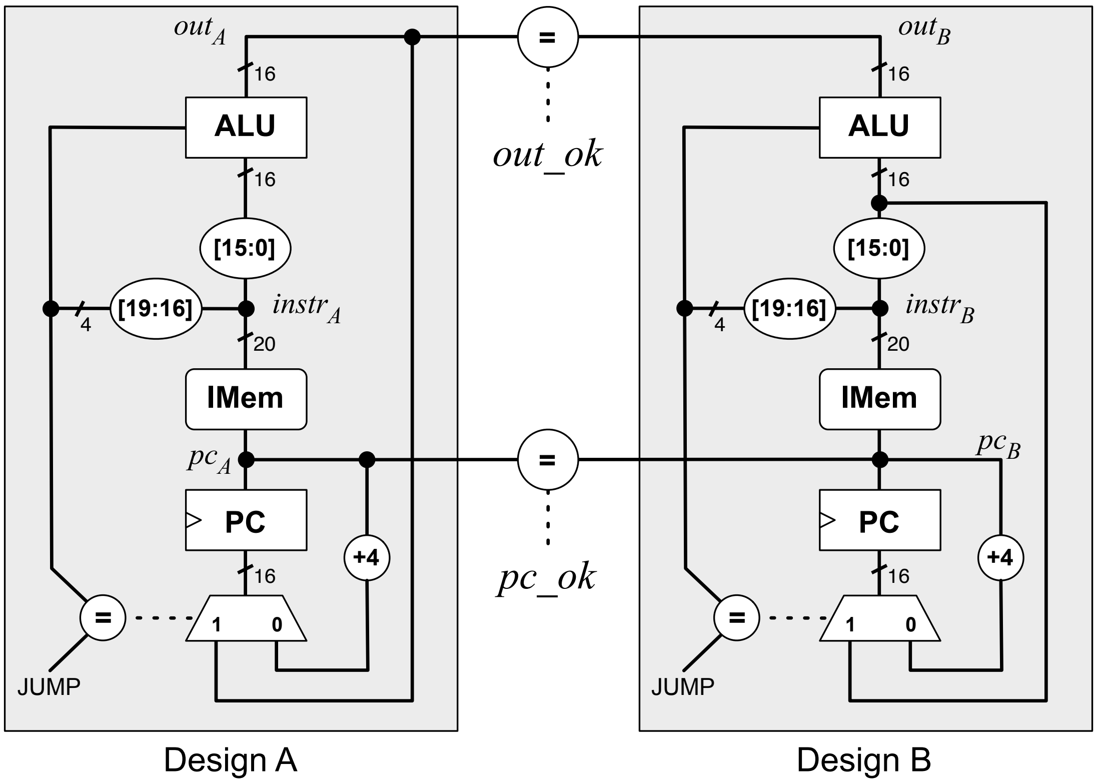


Figure 2.18: **Equivalence checking of two versions of a portion of a processor design.** Boolean signals are shown as dashed lines and bit-vector signals as solid lines.

The two designs are equivalent if and only if their outputs are equal at every cycle, meaning that the Boolean assertion $out\_ok \wedge pc\_ok$ is always **true**.

It is easy to see that this is the case. By inspection of Figure 2.13(a), we know that out_A always equals $instr[15{:}0]$ when $instr[19{:}16]$ equals JUMP. The question is whether we can infer this without the full word-level representation of the ALU.

Consider what happens if we use the abstraction of Figure 2.13(b). In this case, we lose the relationship between out_A and $instr[19{:}16]$. Thus, the verifier comes back to us with a spurious counterexample, where in cycle 1 a jump instruction is read, with the jump target in Design A different from that in Design B, and hence pc_A differs from pc_B in cycle 2.

However, if we instead used the partial term-level abstraction of Figure 2.13(c) then we can see that the proof goes through, because the ALU is precisely modeled under the condition that $instr[19{:}16]$ equals JUMP, which is all that is necessary.

The challenge is to be able to generate this partial term-level abstraction automatically. We describe two methods to solving this problem in Chapters 5 and 6.

2.5 Correspondence Checking

Recall that correspondence checking, introduced by Burch and Dill [31] and described in Section 1.1, is a formal verification technique used to verify the control logic of pipelined microprocessor designs.

In correspondence checking, a pipelined microprocessor (F_{impl}) is verified against a sequential (meaning instructions are executed one after another) version (F_{spec}) of the same processor. The sequential processor is essentially a formal model of the instruction set architecture. We refer to F_{impl} as the *implementation* version of the processor (i.e., the pipelined version) and F_{spec} as the *specification* version of the processor (i.e., the sequential version that the implementation is being verified against).

Correspondence checking relies on two functions: *flush* and *project*.

1. *flush* is used to execute the instructions in the pipeline and update the processor state (i.e., commit the instruction) after each instruction is executed. *flush* is implemented by issuing no-ops to the pipeline for enough cycles so that all the instructions present in the pipeline are executed and committed. Whenever the pipeline is being flushed no new instructions are injected into the pipeline.

2. *project* is used to project the state of F_{impl} into F_{spec}. The state of a processor includes a register file, program counter, condition codes, and memory. *project* is typically used only once during correspondence checking to initialize the sequential processor's state to be consistent with the state of the pipelined processor after the pipeline has been flushed. Note that the pipelined implementation must be flushed before the projection takes place, in order for the instructions in the pipeline to be executed and committed to the implementation state, before the implementation state is projected into the specification state.

Figure 2.19 illustrates the structure of a correspondence checking problem. A high-level description of correspondence checking is as follows.

The state of F_{impl} is initialized to an arbitrary, ideally reachable, state. F_{impl} is flushed for enough cycles so that all of the instructions within the pipeline are committed to the processor state. The state of F_{impl} is then projected into the state of F_{spec} and the state of F_{spec} is stored in S_0. Next, an instruction is injected into and executed on the specification machine. After the instruction is committed to the specification state, the updated specification state, F'_{spec}, is stored in S_1. Next, the same instruction that was injected into F_{spec} is now injected into and executed on F_{impl}. The injected instruction is then committed to the implementation state by flushing for an appropriate number of cycles and the updated implementation state, F'_{impl}, is stored in I. The final step of correspondence checking is to determine whether the implementation refines the specification. This refinement checking problem is formulated as: $I = S_0 \vee I = S_1$. Note that it is necessary to check $I = S_0$ because the pipelined processor can stall, in which case, no instruction is committed to the F_{impl} state. The interested reader is referred to [31] for a more in-depth description of correspondence checking.

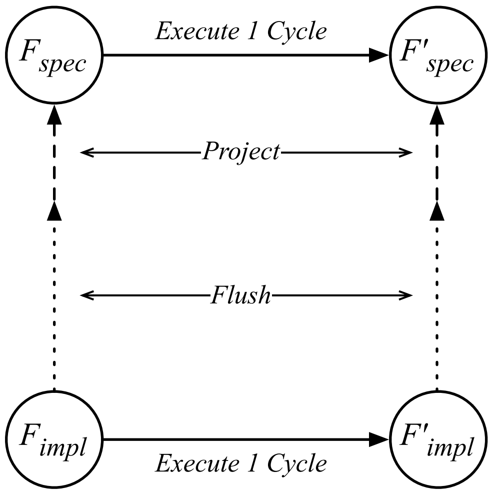


Figure 2.19: **Correspondence checking diagram.** Correspondence checking determines whether a pipelined version of a processor refines the instruction set architecture specification of the same processor.

Chapter 3

Solving Techniques

The Boolean satisfiability (SAT) problem is a decision problem over formulas in propositional logic and is fundamental to higher-level reasoning. The Satisfiability Modulo Theories (SMT) problem is a decision problem over formulas in first-order logic coupled with decidable first-order background theories [10, 44]. Examples of such background theories are the theories of linear arithmetic over the reals or integers, bit-vectors, uninterpreted functions, and arrays. This work relies on the theory of bit-vectors and the logic of equality with uninterpreted functions.

Most formal verification tools encode the verification problem as a SAT or SMT problem. Efficient solvers are then used to determine whether or not the design satisfies the property, by means of searching for a satisfying assignment of the underlying SAT or SMT problem.

In recent years, both SAT and SMT solvers have seen great performance advances and are being adopted by users across both academia and industry. SAT solvers are well-studied and highly optimized, and have a small footprint due to the relatively few constructs present in a propositional formula [7, 40, 38, 39, 12, 47]. The main advantage SMT solvers have over traditional SAT solvers are the theory-specific properties present at the SMT level which are lost at the bit level. Exploitation of these properties can lead to drastic performance improvements. For example, consider the problem where we wish to prove that $a \times_N b = b \times_N a$. Trying to prove this equality at the bit level is a challenging problem even with relatively small bit-widths. If $a \times_N b = b \times_N a$ is represented at a higher level, such as with bit vectors or integers, instead of at the bit level, proving equivalence becomes trivial. We discuss this further in Section 3.2.2.

The remainder of this chapter describes decision procedures for the various levels of abstraction used throughout this thesis and presents the strengths and weaknesses of each approach.

3.1 Terminology

A Boolean *literal* l is a variable b or its complement $\overline{b}$ (where $b \in \mathcal{B}$ and $\overline{b} \in \mathcal{B}$). A *clause* is a disjunction of literals. For example, if a, b, and c are Boolean variables, then a, $\overline{b}$, and c are literals and $(a + \overline{b} + c)$ is a clause. The *cone-of-influence* (*COI*) of a signal s in a circuit is defined as the set of all signals in the transitive fanin of s, including s. We denote the *COI* of s by *COI*(s). The set of signals in *COI*(sum) is $\{a, b, c_{in}, g_1, g_2\}$ and is shown in bold in Figure 3.1.

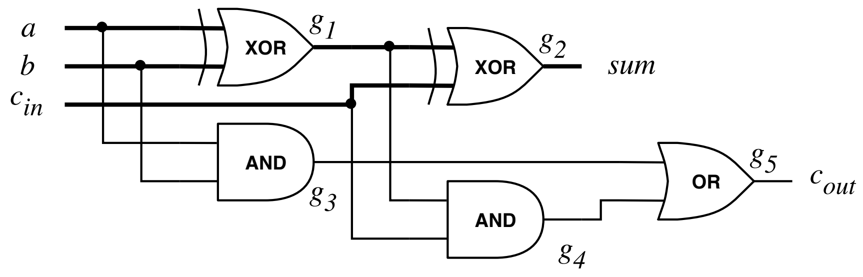


Figure 3.1: **Cone-of-influence (COI) illustration.** The cone-of-influence for the *sum* signal.

3.2 Boolean satisfiability (SAT)

The *Boolean satisfiability* (SAT) problem is the problem of deciding whether a Boolean formula ϕ is satisfiable. A Boolean formula ϕ is considered satisfiable if and only if there exists an assignment of Boolean values to the support variables $x_1, x_2, ..., x_n$ such that ϕ to evaluates to **true**. Formally, the SAT problem is defined as $\exists x_1, x_2, ..., x_n \in \mathcal{B}$ such that $\phi(x_1, x_2, ..., x_n) =$ **true**? Consider the example formulas in Equation 3.1. $\phi_1(a, b)$ is unsatisfiable because for any valuation of a and b one of the clauses evaluates to **false** which causes the conjunction to evaluate to **false**. $\phi_2(a, b)$ is satisfiable because there exists an assignment to a and b that causes $\phi_2(a, b)$ to evaluate to **true**. One such assignment is $a =$ **true** and $b =$ **false**.

$$\begin{aligned} \phi_1(a,b) &= (\overline{a} \vee \overline{b})(\overline{a} \vee b)(a \vee \overline{b})(a \vee b) \\ \phi_2(a,b) &= (\overline{a} \vee \overline{b})(a \vee b) \end{aligned} \tag{3.1}$$

Most modern, state-of-the-art SAT solvers [7, 12, 39, 38, 47, 40] operate on Boolean formulas in *conjunctive-normal form* (CNF). CNF is a restricted form of a Boolean formula where the formula is a conjunction of disjunctions

$$\bigwedge_i (\bigvee_j l_{ij})$$

where l_{ij} is the j-th literal of the i-th clause. ϕ_1 and ϕ_2 in Equation 3.1 are both in CNF. A CNF formula is generated by the grammar shown in Figure 3.2.

Using CNF does not restrict the type of logic circuits we can represent, in fact, any Boolean formula can be expressed in CNF. The Tseitin encoding is a technique that allows the conversion between an arbitrary circuit to an equivalent representation in CNF.

$$\begin{array}{llclcl} \textit{literal} & ::= & b & | & \neg b & b \in \mathcal{B} \\ \textit{clause} & ::= & \textit{clause} \vee \textit{literal} & | & \textit{literal} & \\ \phi_{CNF} & ::= & \phi_{CNF} \wedge \textit{clause} & & & \end{array}$$

Figure 3.2: **Conjunctive Normal Form (CNF) grammar.** A Boolean formula in conjunctive normal form consists of a conjunction of clauses, where each clause is a disjunction of literals.

3.2.1 Tseitin Encoding

The *Tseitin encoding* is a technique used to encode a logic circuit into an equivalent representation in CNF.

The Tseitin encoding operates by first introducing a new variable for each logic gate in the original circuit. Next, several clauses are created that relate the output of each logic gate to the inputs of that same gate. The conjunction of these clauses is true if and only if the inputs and output of the associated gate abide by the definition of the gate. For example, consider a *not* gate with output b and input a, the clauses are: $(\overline{a}+\overline{b})$ and $(a+b)$. As you can see, the conjunction of these clauses is true if and only if $a = \overline{b}$. Table 3.1 shows the clauses generated for each operator. The final step of the Tseitin encoding is to conjoin the clauses for all gates in the circuit along with the single-literal clause corresponding to the output signal of interest.

Gate	**Clauses**
$b =$ **not** (a)	$(\overline{a}+\overline{b})$, $(a+b)$
$c = a$ **and** b	$(\overline{a}+\overline{b}+c)$, $(a+\overline{c})$, $(b+\overline{c})$
$c = a$ **nand** b	$(\overline{a}+\overline{b}+\overline{c})$, $(a+c)$, $(b+c)$
$c = a$ **or** b	$(a+b+\overline{c})$, $(\overline{a}+c)$, $(\overline{b}+c)$
$c = a$ **nor** b	$(a+b+c)$, $(\overline{a}+\overline{c})$, $(\overline{b}+\overline{c})$
$c = a$ **xor** b	$(a+b+\overline{c})$, $(a+\overline{b}+c)$, $(\overline{a}+b+c)$, $(\overline{a}+\overline{b}+\overline{c})$
$c = a$ **xnor** b	$(a+b+c)$, $(a+\overline{b}+\overline{c})$, $(\overline{a}+b+\overline{c})$, $(\overline{a}+\overline{b}+c)$
$c = a$ **implies** b	$(a+c)$, $(\overline{b}+c)$, $(\overline{a}+b+\overline{c})$
$c = a$ **equiv** b	$(\overline{a}+\overline{b}+c)$, $(\overline{a}+b+\overline{c})$, $(a+\overline{b}+\overline{c})$, $(a+b+c)$

Table 3.1: **Clauses generated during Tseitin encoding.** Clauses generated during the Tseitin encoding for each gate type.

Consider the circuit in Figure 3.1. Performing the Tseitin encoding on the *sum* signal generates new variables and clauses for gates g_1 and g_2, because they are in $COI(sum)$. Table 3.2 lists the clauses generated for each gate in $COI(sum)$.

Let ϕ_{sum} be the formula representing the satisfiability problem for the signal *sum*, shown in Equation 3.2. ϕ_{sum} is true if and only if *sum* evaluates to 1 for some assignment to a, b, and c_{in}. Note that in addition to the clauses generated for gates g_1 and g_2, the single-literal clause (g_2) is included. Without this clause, the formula could be satisfiable with $g_2 =$ **false**.

g_1	g_2
$(a + b + \overline{g_1})$	$(g_1 + c_{in} + \overline{g_2})$
$(a + \overline{b} + g_1)$	$(g_1 + \overline{c_{in}} + g_2)$
$(\overline{a} + b + g_1)$	$(\overline{g_1} + c_{in} + g_2)$
$(\overline{a} + \overline{b} + \overline{g_1})$	$(\overline{g_1} + \overline{c_{in}} + \overline{g_2})$

Table 3.2: **Example of Tseitin encoding.** The clauses generated during the Tseitin encoding of *sum*.

ϕ_{sum} is satisfiable and a satisfying assignment is $a = b =$ **false** and $c_{in} =$ **true**.

$$\begin{aligned}\phi_{sum} = {} & (a + b + \overline{g_1}) \cdot (a + \overline{b} + g_1) \cdot (\overline{a} + b + g_1) \cdot (\overline{a} + \overline{b} + \overline{g_1}) \cdot \\ & (g_1 + c_{in} + \overline{g_2}) \cdot (g_1 + \overline{c_{in}} + g_2) \cdot (\overline{g_1} + c_{in} + g_2) \cdot (\overline{g_1} + \overline{c_{in}} + \overline{g_2}) \cdot \\ & (g_2)\end{aligned} \tag{3.2}$$

3.2.2 Disadvantages of Boolean Satisfiability

Modeling circuits at the bit level is the lowest-level logical representation of a circuit. Verification tools operating at the bit level, especially SAT solvers, have seen tremendous performance and capacity increases in the last 10 years. We are able to represent circuits with millions of logic gates and prove interesting properties about them. However, even with the recent performance increases, bit-level techniques fall short on many industrial-scale circuits and circuits that contain operators that are challenging to reason about at the bit level.

A multiplication circuit is an example of a circuit that is hard to reason about at the bit level. Let ϕ_{Mult} denote the bit-level formula corresponding to $a \times_N b = b \times_N a$, where a and b are bit vectors of size N. That is, ϕ_{Mult} is valid if and only if $a \times_N b = b \times_N a$. For relatively small bit-widths state-of-the-art SAT solvers will prove ϕ_{Mult} valid in a reasonable amount of time. However, even the best SAT solvers won't prove this equivalence for multipliers with large bit-widths. Table 3.3 shows the runtimes for 5 leading SAT solvers: glucose [7], MiniSAT [40, 38], MiniSAT2 [39], PicoSAT [12], and PrecoSAT [47]. As you can see, for instances with small bit-widths ϕ_{Mult} is proven valid in only a few seconds. However, larger bit-widths prove to be much more challenging for SAT solvers.

For multipliers with a relatively small bit-width (≤ 12), using exhaustive simulation can be more efficient. SAT solvers can not distinguish between primary inputs and intermediate variables, whereas simulation only takes into account the primary inputs, reducing the search-space from that of a SAT problem. However, even exhaustive simulation can not handle multipliers with large bit-widths. The limiting factor is the number of simulation vectors required. Shown in Table 3.4 are the runtimes for the simulation of ϕ_{Mult} with $N = 16$. As expected, the runtime is exponential in the number of input bits, hence, using exhaustive simulation for circuits with wide datapaths in unrealistic.

Representing $a \times_N b = b \times_N a$ at a higher level would allow us to reason about the multiplication operation itself, instead of reasoning about Boolean operators. The commutativity property of multiplication, which is lost at the bit level, allows higher-level solvers

Solver	Bit-width						
	4	6	8	10	12	14	16
glucose	0.00	0.09	3.76	132.94	5696.93	T.O.	T.O.
MiniSAT	0.00	0.13	2.73	85.86	2826.51	T.O.	T.O.
MiniSAT v2	0.00	0.09	2.83	53.90	2164.52	T.O.	T.O.
PicoSAT	0.00	0.07	2.04	47.67	1119.63	T.O.	T.O.
PrecoSAT	0.01	0.10	2.97	127.57	4384.95	T.O.	T.O.

Table 3.3: **Runtimes for checking the commutativity of multiplication.** For small bit-widths SAT solvers can prove that $a \times_N b = b \times_N a$

Multiplier Bit-width	#Vectors	Runtime (sec)
4	2^8	0.01
6	2^{12}	0.28
8	2^{16}	0.85
10	2^{20}	22.30
12	2^{24}	536.38
14	2^{28}	12022.61
16	2^{32}	T.O.

Table 3.4: Simulation runtime of 16-bit multiplier for varying number of simulation vectors.

to immediately prove the the equivalence of $a \times_N b$ and $b \times_N a$ (i.e., $a \times_N b = b \times_N a$ is valid). Bit-level reasoning techniques are not sufficient when reasoning about larger, more complex circuits.

3.3 Word-Level Reasoning

In order to be consistent with our definition of a word-level design, we consider a word-level reasoning technique to be a technique that reasons about formulas defined by the grammar shown in Figure 2.6. Therefore, a word-level technique, in the context of this thesis, does not reason about abstract terms or uninterpreted functions.

Word-level decision procedures are often layered techniques that rely heavily on a process known as rewriting. Rewriting takes advantage of high-level, theory-specific rules that dictate how formulas can be simplified. In addition to rewriting, abstraction-refinement techniques [21, 22, 27, 28, 43] can be employed within a bit-vector decision procedure. Finally, when word-level techniques are exhausted, bit-vector problems are translated into equivalent bit-level representations using a technique called bit-blasting, after which bit-level techniques are employed.

3.3.1 Rewriting

A main component to any state-of-the-art bit-vector decision procedure is a simplification process called rewriting [8]. Rewriting engines are based on a set of rules that dictate how certain formulas can be simplified, or rewritten.

Rewriting is not limited to high-level theories such as bit vectors or integers. For example, the laws of Boolean logic can be used to simplify a Boolean formula before invoking a SAT solver. Figure 3.3 lists a set of properties of Boolean logic that can serve as simplification rules.

Property	Rewrite Rules	Property	Rewrite Rules
Absorption	$x \wedge (x \vee y) = x$ $x \vee (x \wedge y) = x$	Annihilator	$x \wedge \textbf{false} = \textbf{false}$ $x \vee \textbf{true} = \textbf{true}$
Idempotence	$x \vee x = x$ $x \wedge x = x$	Identity	$x \vee \textbf{false} = x$ $x \wedge \textbf{true} = x$
Complementation	$x \wedge \neg x = \textbf{false}$	Complementation	$x \vee \neg x = \textbf{true}$

Figure 3.3: **Boolean simplification rules.** The axioms of Boolean logic can be used to simplify Boolean formulas.

Consider the following example where the formula ϕ_{SAT1} can be proven to be satisfiable using only the simplification rules listed in Figure 3.3. Starting from our original formula ϕ_{SAT1}, we obtain Equation 3.4 by applying the complementation rule. Next, we obtain Equation 3.5 by applying the identity property to Equation 3.4. Finally, we obtain **true** by applying the complementation property to Equation 3.5. We deem problems that can be solved with simplification rules alone to be trivially solvable (i.e., ϕ_{SAT1} is trivially satisfiable).

$$\begin{aligned}
\phi_{SAT1} &= a \wedge (b \vee \neg b) \vee \neg a \wedge (b \vee \neg b) && (3.3)\\
&= a \wedge (\textbf{true}) \vee \neg a \wedge (\textbf{true}) && (3.4)\\
&= a \vee \neg a && (3.5)\\
&= \textbf{true} && (3.6)
\end{aligned}$$

An example of a formula where simplification rules alone are not enough to decide satisfiability is ϕ_{SAT2} shown in Equation 3.7. This example differs from the previous in the second step when the identity and annihilator properties are applied resulting in $\phi_{SAT2} = a$. Note that this problem is not trivially solvable and a proper satisfiability engine must be invoked.

$$\begin{aligned}
\phi_{SAT2} &= a \wedge (b \vee \neg b) \vee \neg a \wedge (b \wedge \neg b) && (3.7)\\
&= a \wedge (\textbf{true}) \vee \neg a \wedge (\textbf{false}) && (3.8)\\
&= a && (3.9)
\end{aligned}$$

Bit-vector decision procedures solvers are equipped with an analogous set of rewriting rules [48, 43, 22, 44]. There are far too many rewriting techniques to discuss here, so instead we list commonly used techniques that convey the spirit of rewriting.

The most simple form of bit-vector rewriting is constant propagation. Constant propagation involves computing a concrete value of an input or output to a bit-vector operator. Consider a bit-vector multiplication where one of the inputs is $\mathit{bv0}_N$. The result of this multiplication is $\mathit{bv0}_N$. The bitwise logical-AND operation has the same property. $\mathit{bv0}_N$ is a controlling value for both the multiplication and bitwise logical-AND operators. This is a special form of constant propagation where the input to the operator dictates the output.

Another class of rewriting rules aims to eliminate redundancies by analyzing the structure of bit-vector expressions. For instance, consider an *ITE* node where the conditional argument is **true** or **false**. In this case, we can replace the *ITE* with the appropriate value. Another situation involving the *ITE* operator is when the "then" and "else" branches are equal.

A more subtle rewriting rule exploits the commutativity property. Before creating an operator that has the commutativity property, such as addition or multiplication, the decision procedure checks to see if an equivalent operator has already been created. For example, assume we create the node $a \times_N b$. Later, if we try to create the node $b \times_N a$, we first check to see if either $a \times_N b$ or $b \times_N a$ have been created already. If so, we return the existing node instead of creating a new node. Taking this example a step further, assume we wish to create the equality $a \times_N b = b \times_N a$. First we create $a \times_N b$. Then we attempt to create $b \times_N a$ but because it has already been created, we end up with $a \times_N b$ again. Finally, when we create the equality, we have $a \times_N b = a \times_N b$ which simplifies to **true**.

The last form of rewriting we discuss is equality propagation. Equality propagation involves substituting values into expressions. Consider the following fragment of a word-level design.

$$\begin{aligned}
v_1 &\leftarrow a +_N b \\
v_2 &\leftarrow v_1 +_N c \\
v_3 &\leftarrow v_2 +_N d \\
c &\leftarrow v_3 > \mathit{bv0}_N
\end{aligned}$$

This portion of a bit-vector design would be rewritten as $c \leftarrow a +_N b +_N c +_N d > \mathit{bv0}_N$. The benefit of such a translation is that the intermediate variables v_i are removed from the underlying SAT problem. This can lead to drastic reduction in the size and complexity of the SAT problem.

3.3.2 Abstraction-Refinement

Abstraction-refinement within a bit-vector decision procedure involves solving successive under- and over- approximations of the original formula.

Let ϕ_{under} be an under-approximation of ϕ, meaning that if ϕ_{under} is satisfiable so is ϕ. Let ϕ_{over} be an over-approximation of ϕ, meaning that if ϕ_{over} is unsatisfiable so is

ϕ. The intuition behind computing approximations is that it should be easier to find a satisfying assignment of an under-approximation and easier to prove the unsatisfiability of a over-approximation.

There are many existing heuristics [21, 22, 27, 28, 43] that compute under- and over-approximations of bit-vector formulas. A thorough discussion of all such approximation techniques is beyond the scope of this thesis. Instead, we discuss briefly the approximation techniques used in our own bit-vector decision procedure [27, 28].

The abstraction-refinement loop implemented within the UCLID bit-vector decision procedure operates as follows. First, for each bit-vector variable v_i in the support of the formula ϕ_{bv}, an encoding size s_i is chosen, where $0 \leq s_i \leq \mathit{length}(v_i)$. Next, an under-approximation ϕ_{under} of ϕ_{bv} is generated where each variable v_i is encoded with s_i Boolean variables and then ϕ_{under} is encoded as a SAT problem. If ϕ_{under} is satisfiable then the process terminates. Otherwise, the unsatisfiable core is used to generate an over-approximation ϕ_{over} of ϕ_{bv}. A key point here is that in the over-approximation ϕ_{over} the variables v_i are encoded as a SAT problem with the full number of bits, otherwise, ϕ_{over} would be unsatisfiable. If ϕ_{over} is unsatisfiable, then so is ϕ_{bv} and the process terminates. Otherwise, if ϕ_{over} is satisfiable, then it must be the case that at least one Boolean variable v_i is assigned a value that can not be encoded with s_i Boolean variables. In this case, it is necessary to increase s_i for v_i and repeat the process.

Additional techniques for computing under- and over- approximations can be found in [21, 22, 27, 28, 43].

3.3.3 Bit-blasting

Bit-blasting is a technique that converts a word-level formula into its corresponding bit-level representation. There are two general methods of bit-blasting. The first is a two-phase approach where a word-level formula is converted into an equivalent bit-level formula then the bit-level formula is encoded as a CNF formula [27, 28] using, for example, the Tseitin transformation as described in Section 3.2.1. An alternative approach converts a word-level formula directly into a CNF formula [43].

An advantage to the two-phase approach is that propositional simplifications can be applied before the formula is encoded to CNF. A disadvantage of the two-phase approach is that the intermediate propositional formula could be large, and time consuming to generate.

3.3.4 Limitations

After all the rewriting rules and theory-specific optimizations have been employed, most bit-vector solvers resort to bit-blasting. In this case, there are generally two limiting factors: the size of the datapaths present in the problem and the number of hard to reason about operators present after all the optimizations have been employed. Consider a bit-vector problem where a single multiplier with a very large bit-width remains after the word-level simplifications have been applied. Bit-blasting such a circuit will most likely result in an extremely large and hard to solve SAT problem. This problem is only compounded when many bit-vector operators remain after the theory specific optimizations have been applied.

3.4 Decision Procedures for EUF

There are two main options when solving EUF formulas. The first option involves combining an EUF algorithm, such as congruence closure, with a SAT solver. The second option involves transforming the EUF formula into a propositional formula.

In the first approach, the EUF formula is abstracted by replacing each equality with a fresh Boolean variable. A SAT solver is then used to determine if the abstracted formula (also known as the Boolean *skeleton*) is satisfiable. If a satisfying assignment is found, an EUF algorithm (e.g., congruence closure) is used to determine if there is a corresponding assignment to the equality predicates [51]. If no satisfying assignment to the equality predicates exist, then a new satisfying assignment to the Boolean skeleton is found (if it exists), and the process is repeated. A more detailed description of this approach can be found in [51].

In this work, we rely on the second approach, where the EUF formula is transformed into propositional logic. The translation from EUF to propositional logic, requires two main procedures: eliminating function applications and determining how many Boolean variables are required to encode each term. These techniques are described in Sections 3.4.1 and 3.4.2.

It is important to note, however, that while we use the second approach, where EUF formulas are translated into propositional logic, the techniques described in this work are not fundamentally limited to such an approach. Our reliance on this approach stems from the use of UCLID [53, 18, 63] as our underlying verification environment. UCLID, as described in Section 3.5, provides a rich description language, as well as a symbolic simulation engine which enables the solving of a variety of verification problems.

Before describing function elimination and small-domain encoding, we introduce some notation. Let ϕ_{UF} be an EUF formula containing function applications, ϕ_{Elim} be an equi-satisfiable equality-logic formula without function applications, and ϕ_{Prop} be the propositional formula encoding ϕ_{Elim}

3.4.1 Eliminating Function Applications

Before translating an EUF formula into propositional logic, function applications must be removed. There are two main procedures for eliminating function applications. The first is the classic technique introduced by Ackermann [3]. The second, more recent, approach was introduced by Bryant *et al.* [24]. We describe Ackermann's technique below, using an example.

Starting with ϕ_{UF}, the goal is to compute the equi-satisfiable formula ϕ_{Elim} such that all function applications have been removed. Let $f : \mathcal{Z} \to \mathcal{Z}$ be a function used in ϕ_{UF} and let $a, b, c \in \mathcal{Z}$ be terms appearing in ϕ_{UF}. Assume that the following function applications appear in ϕ_{UF}: $f(a)$, $f(b)$, and $f(c)$. Ackermann's method proceeds as follows. First, create the formula ϕ_{Fresh} by replacing each function application in ϕ_{UF} with a fresh variable. In this case, we would replace $f(a)$ with $v_{f(a)}$, $f(b)$ with $v_{f(b)}$, and $f(c)$ with $v_{f(c)}$ to create

ϕ_{Fresh}. Next, the set of functional consistency constraints are generated for f:

$$a = b \Longrightarrow v_{f(a)} = v_{f(b)}$$
$$a = c \Longrightarrow v_{f(a)} = v_{f(c)}$$
$$b = c \Longrightarrow v_{f(b)} = v_{f(c)}$$

Let ϕ_{FCC} denote this set of functional consistency constraints. Then $\phi_{FCC} \implies \phi_{Fresh}$ is the equi-satisfiable formula ϕ_{Elim} that contains no function applications. In general, this process is repeated for each function in ϕ_{UF}. Note that the number of functional consistency constraints grows quadratically with the number of applications of a given function.

We omit a description of Bryant's technique and refer the interested reader to [24, 9].

3.4.2 Small-Domain Encoding

The last step in converting ϕ_{UF} to ϕ_{Prop} is to determine the number of bits required to accurately represent each term in ϕ_{Elim}.

Let ϕ be an equality logic formula with support variables $x_1, x_2, ..., x_n \in \mathcal{Z}$. Consider the worst case situation where each x_i must be distinct:

$$\phi \equiv \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} i \neq j \implies x_i \neq x_j$$

In order for ϕ to be **true**, there must be n distinct values, so that each x_i is different. It is possible represent n values with $log_2(n)$ bits. This leads to a solution space size of $\mathcal{O}(n^n)$ [9]. After determining the number of Boolean variables necessary to accurately represent each term variable in ϕ_{Elim}, the term variables are replaced with a vector of Boolean variables of length $log_2(n)$ (i.e., a bit-vector x with $length(x) = log_2(n)$).

An optimization to this technique is to group terms into equivalence classes so that terms appearing in the same equality are grouped into the same class. Each equivalence class can then be treated separately when computing the number of bits necessary to encode each term [9]. Pnueli *et al.* describe another optimization, called *range allocation*, however, the details of this technique are beyond the scope of this work and we refer the interested reader to [60].

3.5 UCLID

UCLID is a verification framework capable of reasoning about systems expressible with a variety of logics. Originally focused on infinite-state systems, UCLID now provides support for modeling at the bit-, word-, and term-level [53, 18, 63]. The UCLID system comprises the UCLID specification language and the verification engine itself. A UCLID model is essentially a description of a transition system, where the user provides the initial and next state functions for state variables [18, 20, 63]. State variables can be any of the base types or function types as defined in Chapter 2. Thus, state variable x can have type t, where $t \in \{\mathcal{V}, \mathcal{V}^n \to \mathcal{V}\}$.

The main component of UCLID is a decision procedure for a decidable fragment of first-order logic, including the logic of equality with uninterpreted functions, integer linear arithmetic, finite-precision bit-vector arithmetic, and constrained lambda expressions. The decision procedure operates by translating the input formula to an equi-satisfiable SAT or SMT formula on which it invokes a SAT or SMT solver, respectively [18, 20]. UCLID is equipped with a symbolic simulation engine which provides a mechanism to model a variety of verification problems such as inductive invariant checking, bounded-model checking (BMC), equivalence checking, and correspondence checking [20, 63].

There are certain situations that require reasoning about signals across time frames. For example, in the example shown in Figure 2.8, the property we wish to prove is $(\mathit{reset}) \implies (\mathit{next}(\mathit{out}) == \mathit{bv0_4})$. This property requires reasoning about the value of *out* in the time-frame after the *reset* signal is asserted. Therefore, in order to reason across time frames, we define polymorphic operators $\mathit{init}()$, $\mathit{prev}()$, and $\mathit{next}()$ to represent the state of a signal in the initial state, the previous state, and next state, respectively.

Let x be a signal with type $t \in \{\mathcal{V}, \mathcal{V}^n \to \mathcal{V}\}$ and let x_i denote the value of signal x in time frame i. Assume that the verification problem in question requires the model to be unrolled for N cycles, that is, there are N time frames in addition to the initial time frame $(0 \leq i \leq N)$.

Definition 3.1. $\mathit{init}(x_i)$ denotes the initial state of signal x_i and is equivalent to x_0.

Definition 3.2. $\mathit{next}(x_i)$ denotes the next state of signal x_i and is equivalent to x_{i+1}.

Definition 3.3. $\mathit{prev}(x_i)$ denotes the previous state of signal x_i and is equivalent to x_{i-1}.

Note that the value of $\mathit{init}()$ is the same regardless of the time frame in which it is used. This differs from $\mathit{prev}()$ and $\mathit{next}()$ where the value is dependent upon the time frame in which it is used. Also note that $\mathit{prev}(x_0) = \mathit{init}(x_0) = x_0$ and $\mathit{next}(x_N)$ is undefined, thus, it is up to the user to ensure that the verification model is unrolled for enough cycles so that every use of the next state operator is defined.

3.6 Challenges

While bit-vector SMT solvers have outperformed bit-level reasoning techniques in many cases, there are still problems that even the best SAT and bit-vector SMT solvers can not solve in a reasonable amount of time. This thesis focuses on these hard cases.

The word level is typically the level at which hardware designs are modeled. However, in some specialized cases, such as high performance microprocessors, designs are hand-optimized at the bit level. This bit-level optimization is usually necessary in order to satisfy timing and other performance constraints. While it is somewhat easier to perform abstraction on a word-level design, compared to an equivalent design at the bit level, design size is a limiting factor for current abstraction techniques.

A major challenge for any abstraction technique is in deciding what to abstract. Constructing abstract verification models by hand is a tedious and error prone process. Moreover, this requires maintaining multiple models of the same design, one describing the actual design, the other describing the verification model. Maintaining multiple models of the same design poses several challenges. Changes to the design will likely require an associated change to the verification model. If a bug arises in the verification model, it is necessary to determine if it is an actual bug in the design, a bug due to an abstraction introduced in the verification model, or simply an error introduced in the construction of the verification model that isn't present in the actual design. Tracking down the root cause of a bug can be a challenging endeavor, however, it is an unavoidable challenge associated with hardware design. Maintaining multiple models of a design multiplies the effort required in determining the source of an error. While this is an obvious drawback to maintaining multiple models of a design, an even greater drawback comes from manually introducing abstraction into the verification model. Creating a precise abstraction (i.e., an abstraction that does not introduce spurious counter-examples) often requires intimate knowledge of the design. It is often the case that introducing abstraction will result in spurious counterexamples which must be dealt with manually. Thus, automatic abstraction procedures are desired.

The challenge with automatic abstraction procedures is how to choose the abstraction granularity. Andraus and Sakallah [6] were the first to propose an automatic abstraction techniques operating directly on RTL. These techniques rely on counterexample-guided abstraction-refinement (CEGAR) and start by abstracting each word-level operator with an uninterpreted function. While CEGAR-based approaches have shown promise, the main drawback is the number of CEGAR iterations required. As we discuss in Section 4.7, CEGAR-based approaches can require many iterations to learn very simple properties. On the other end of the spectrum, it is possible to abstract larger functional blocks with uninterpreted functions. However, this poses its own challenges. Module boundaries are an obvious starting point, however, industrial-scale designs can have hundreds or thousands of module instantiations. Deciding which modules should be abstracted is a non-trivial problem. Thus, more intelligent abstraction procedures are necessary.

Bit-vector SMT solvers represent only a small portion of the SMT solvers available. Certain hardware design patterns can be reasoned about more efficiently in theories other than the theory of bit-vectors, such as logic of equality with uninterpreted functions. This work enables the use of SMT solvers that reason over these other theories by automatically abstracting bit-vector designs to those theories.

Part II

Automatic Abstraction

Chapter 4

Data-Abstraction

In this chapter we present a data abstraction technique based on type inference. *Type qualifiers*, optionally provided by the designer, are used to selectively abstract portions of a word-level design to the term level. This lightweight approach can yield substantial performance improvements, while requiring little to no designer input. Type inference rules are used to check that the annotations provided by the designer can be applied safely, that is, without introducing spurious counterexamples. In the absence of designer-provided type qualifiers, an automatic technique is used to compute a maximal, sound, and complete, term-level abstraction. We present experimental results that show that type qualifier-based, selective term-level abstraction can be effective at scaling up verification.

4.1 Type Qualifiers

We propose the use of *type qualifiers* to guide the process of abstracting to the term level. Type qualifiers are widely used in the static analysis of software [42], from finding security vulnerabilities to the detection of races in concurrent software, to enable programmers to easily guide the analysis. Even standard languages such as C and Java have keywords such as `const` that qualify standard types such as `ints`.

In our case, type qualifiers indicate what can and cannot be abstracted. The widespread effectiveness of type qualifiers in software leads us to believe that they would be effective in incorporating a designer's insights for abstraction, without placing an undue burden on the designer. In our context, a type qualifier specifies the part of a bit-vector signal that should be abstracted with a term.

A type qualifier is always associated with an underlying logic. In this work, the underlying logic is *logic of equality with uninterpreted functions (EUF)*. Thus, a type qualifier specifies which portion of a bit-vector signal should be encoded as a term $t \in \mathcal{Z}$.

Definition 4.1. A type qualifier $TQ_v = \{[w_1 : w'_1], [w_2 : w'_2], ..., [w_k : w'_k]\}$ is a set of non-overlapping sub-ranges of $[v-1 : 0]$ specifying portions of the bit-vector signal v that should be abstracted with a term, where v is a bit-vector signal with *length*(v).

Our approach is depicted in Figure 4.1. We start with Verilog RTL, the specification to be verified, and, optionally, with designer-provided type qualifier annotations. The Verilog-to-UCLID abstraction tool, v2ucl, then performs type inference and type checking. If the designer-provided annotations are inconsistent, warnings are generated, which indicate to the designer what operators are prohibiting abstraction. Otherwise, v2ucl automatically creates a term-level model, with the specification included in it. Note that while the tool v2ucl takes Verilog as input, the techniques presented in this work are not limited to Verilog.

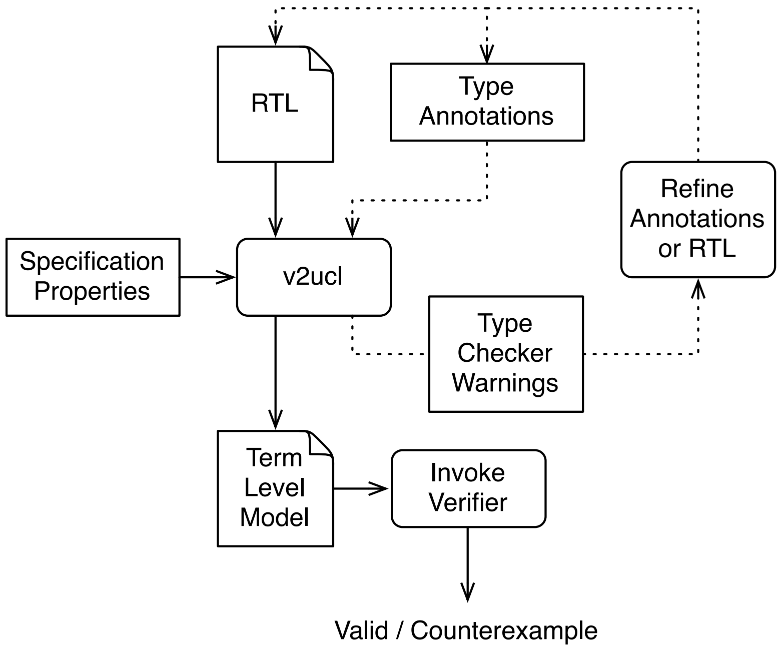


Figure 4.1: **An overview of v2ucl.** v2ucl takes Verilog RTL and optional type-qualifiers as input and produces a term-level model. The type checker issues warnings when the type qualifiers specified are not consistent with the design. This information can be used to refine the design to make it amenable for verification or correct mistakes in the type qualifiers.

Given a Verilog RTL description $\mathcal{R}$, optionally annotated with type qualifiers, and a specification S, v2ucl performs an automatic analysis in the following steps:

1. *Compute Equivalence Classes:* Partition the set of signals and their extracted portions into equivalence classes such that signals that appear together in assignments, relational comparisons, or functional operations in $\mathcal{R}$ or S end up in the same equivalence class.

2. *Compute Maximal Term Abstraction:* For each equivalence class, we compute the maximal term abstraction (defined in Section 4.3) that is common to all signals in that equivalence class.

3. *Check Type Annotations for Consistency:* If the designer has provided any type annotations, we check those for consistency with the computed abstraction. If the designer's

annotations are more abstract, an error message is generated. If they are less abstract, V2UCL can either use the automatically inferred abstraction, or use the abstraction provided by the user. In either case, a warning is issued.

4. *Create UCLID Model:* Signals that have associated term abstractions are encoded in the UCLID model with combinations of term and bit-vector variables, and a hybrid UCLID model is generated and verified.

We describe each of these stages in more detail in Sections 4.2-4.5 below.

4.2 Compute Equivalence Classes

This step is performed by computing equivalence classes of bit-vector expressions defined in the RTL with the goal of giving signals in the same equivalence class the same term abstraction.

Before computing equivalence classes, we construct a word-level design $\mathcal{D}_{RTL} = \mathcal{R} \| S$, by composing Verilog RTL model $\mathcal{R}$ and specification S. We make the reasonable assumption that there is a 1-1 mapping from Verilog operators to the word-level operators listed in Figure 2.5, thus, the translation from Verilog description to $\mathcal{D}_{RTL}$ is straightforward. Note that it is important to include the specification S in the analysis, because bit-vector operations might be performed on a node in S even if it is not performed in $\mathcal{R}$. If the bit-vector representation of a signal is important for determining the truth value of S, this information must also be retained in the model.

The process of constructing equivalence classes is as follows. First, each bit-vector signal is placed in its own singleton equivalence class. Denote the equivalence class containing signal v_i by $\mathcal{E}(v_i)$. Next, equivalence classes are merged according to the rules listed in Table 4.1. This is accomplished by iterating through the signals in $\mathcal{D}_{RTL}$ and applying the appropriate equivalence class update rule.

Notice that there are no update rules for bit-manipulation operators (i.e., nothing is merged). This omission prevents signals with different bit-widths from belonging to the same equivalence class. We construct equivalence classes with the idea that each signal belonging to an equivalence class will be abstracted in the same manner, and hence, have the same type. If we merged equivalence classes through bit-manipulation operators, we would have signals of different bit-widths (i.e., different types) within the same equivalence class.

After iterating through all of the signals in $\mathcal{R}$, we are left with a set of equivalence classes $\mathcal{ES}$. Each equivalence class $\mathcal{E} \in \mathcal{ES}$ can have at most one unique type qualifier associated with it. Thus, for all bit-vector signals $x, y \in \mathcal{E}$, if TQ_x and TQ_y exist, then $TQ_x = TQ_y$.

We denote the type qualifier associated with the equivalence class $\mathcal{E}$ by $TQ_{\mathcal{E}}$.

Assignment Type	Expression	Update Rule
Bit-vector arithmetic	$v \leftarrow \mathsf{bvop}(v_1, \ldots, v_k)$	Merge $\mathcal{E}(v), \mathcal{E}(v_1), \ldots, \mathcal{E}(v_k)$
Bit manipulation	$v \leftarrow \mathsf{bvmanip}(v_1, \ldots, v_k)$	Merge Nothing
Bit-vector relations	$\mathsf{bvrel}(v_1, \ldots, v_k)$	Merge $\mathcal{E}(v_1), \ldots, \mathcal{E}(v_k)$
Equality	$v_1 = v_2$	Merge $\mathcal{E}(v_1), \mathcal{E}(v_2)$
Sequential/combinational assignment	$v \leftarrow u$ $v := u$	Merge $\mathcal{E}(v), \mathcal{E}(u)$
Multiplexor assignment	$v \leftarrow \mathit{ITE}(b,\, v_1,\, v_2)$	Merge $\mathcal{E}(v), \mathcal{E}(v_1), \mathcal{E}(v_2)$
Memory operations	$v \leftarrow \mathbf{read}(M, u)$ $M := \mathbf{write}(M, u, v)$	Merge $\mathcal{E}(M), \mathcal{E}(v)$
Uninterpreted Function	$v_x \leftarrow \mathit{UF}(v_{x_1}, \ldots, v_{x_k})$ $v_y \leftarrow \mathit{UF}(v_{y_1}, \ldots, v_{y_k})$	Merge $\mathcal{E}(v_x), \mathcal{E}(v_y)$ Merge $\mathcal{E}(v_{xi}), \mathcal{E}(v_{yi})\ \forall i = 1 \ldots k$

Table 4.1: **Rules for merging equivalence classes.** u and v denote bit-vector expressions, b denotes a Boolean expression, M denotes a memory expression, and UF is an uninterpreted bit-vector function or predicate.

4.3 Compute Maximal Abstraction

For each equivalence class, we compute the *maximal term abstraction* common to all signals in that equivalence class. The maximal term abstraction for a signal v with range $[w-1:0]$ is a sequence of non-overlapping sub-intervals $[w_1 : w_1']$, $[w_2 : w_2']$, …, $[w_k : w_k']$ of $[w-1:0]$ such that the fanouts of those sub-ranges of v do not appear in any bit-vector arithmetic operation, but those of the sub-ranges $[w, w_1]$, $[w_1', w_2]$, …, $[w_k', 0]$ do (and hence the latter intervals cannot be abstracted to terms).

The first step is to filter out equivalence classes that cannot be abstracted. For each equivalence class, if any element is used with a bit-vector operator excluding extraction or concatenation, compared with a bit-vector constant, or compared with non-equality relational comparison (such as $<$), we will not abstract any signals in that equivalence class, since that operation cannot be modeled in equality logic.

At this point, the only equivalence classes remaining will contain nodes whose fanouts go into assignments, conditional assignments, equality (equals, not equals), or extraction/concatenation operators. Note that all signals within an equivalence class will have the same bit-width. The next step will determine if a consistent term abstraction exists for each remaining equivalence class.

The main challenge is in dealing with extractions and concatenations. Let us first consider concatenations. Our approach is to treat concatenations like extractions, by tracking the sub-intervals that make up the result of a concatenation. For example, if we had the assignment $\texttt{y} = \texttt{y1} \bullet \texttt{y2}$, with `y1` being 8 bits and `y2` being 4 bits, we treat `y` as if it had ranges [11:4] and [3:0] extracted from it. Next we compute how extractions refine the bit-range of each signal.

For each equivalence class $\mathcal{E}$, let the bit-width of any signal in that class be w. For each bit-vector signal $v_j \in \mathcal{E}$, let I_j be the set of all ranges extracted from v_i. Thus, I_j is a set of sub-intervals of the range [w-1:0]. We take the intersection of the sub-intervals in all I_j, corresponding to each signal in that equivalence class, resulting in a maximally refined partition $P_{\mathcal{E}}$ of the interval [w-1:0]. By filtering out all of the intervals in $P_{\mathcal{E}}$ that appear in bit-vector operators (i.e., extract, concatenation), we are left with the maximal term-level abstraction for class $\mathcal{E}$ (i.e., the sub-intervals that can be modeled with terms).

4.4 Determine Final Abstraction

The goal of this step is to determine what abstraction should be used for each equivalence class $\mathcal{E} \in \mathcal{ES}$.

Definition 4.2. A type qualifier $TQ_{\mathcal{E}}$ *legal* if and only if for each sub-range $r \in TQ_{\mathcal{E}}$ there exists a sub-range $s \in P_{\mathcal{E}}$ such that r is contained within s.

Definition 4.3. A type qualifier $TQ_{\mathcal{E}}$ *maximal* if and only if $TQ_{\mathcal{E}} = P_{\mathcal{E}}$.

For each equivalence class $\mathcal{E} \in \mathcal{ES}$ that has an associated type qualifier $TQ_{\mathcal{E}}$, this step determines:

1. If $TQ_{\mathcal{E}}$ is legal.

2. If $TQ_{\mathcal{E}}$ is maximal.

If $TQ_{\mathcal{E}}$ is legal, it is used to create a term-level model. If $TQ_{\mathcal{E}}$ is not legal, a warning is issued and the equivalence class $\mathcal{E}$ is not abstracted. If $TQ_{\mathcal{E}}$ is not maximal, a warning is issued and $\mathcal{E}$ is abstracted according to $TQ_{\mathcal{E}}$.

If an equivalence class does not have an associated type qualifier, then no abstraction is performed, but a warning is issued to inform the designer than it is possible to abstract the equivalence class.

It is important to note that v2ucl can operate completely automatically, without any guidance from the user, by using the maximal abstraction computed in Section 4.3. However, it is also important to allow the designer to convey their intuition about how the design should be abstracted.

A discrepancy between how a design can be abstracted and how a designer thinks it should be abstracted, can be helpful in tracking down possible errors. If a designer-supplied type qualifier is the same as the maximal abstraction computed in Section 4.3, then this is an extra assurance that the circuit is designed the way they expected. If the type qualifier is too aggressive, meaning that it specifies to abstract a portion of a signal that can not be abstracted, errors are issued to the designer. The designer can then figure out why the specified abstraction does not work. If the type qualifier is conservative, meaning that it is a legal, but not maximal, abstraction, then instead of using the maximal abstraction, we issue warnings and use the user-supplied abstraction. One possible reason for this is that during the design cycle, the designer will be verifying parts of the design before other parts are complete. If the designer knows that a certain signal will be connected to another signal that can not be abstracted, they may not want to abstract this portion of the design, so they can reuse this verification model at a later time.

4.5 Create UCLID Model

A term-level model is created from the word-level model $\mathcal{R}$ in a straightforward way. Each signal in $\mathcal{R}$ that has an associated legal type qualifier is split into several sections, some of which will be encoded as terms, while the rest remain bit-vectors.

For example, let $x \in \mathcal{E}$ be a bit-vector signal with $\mathit{length}(x) = N$ and let $TQ_{\mathcal{E}} = \{[w_1 : w'_1], [w_2 : w'_2], ..., [w_k : w'_k]\}$ be the type qualifier associated with $\mathcal{E}$. Then signal x is encoded with k terms and at most $k+1$ bit-vectors. There are at most $k+1$ bit-vectors because there can be a bit-vector before and after every term. Note that there are at least $k-1$ bit-vectors because there must be a bit-vector between every term. Figure 4.2 depicts how signal x is partitioned into a series of terms and bit-vectors.

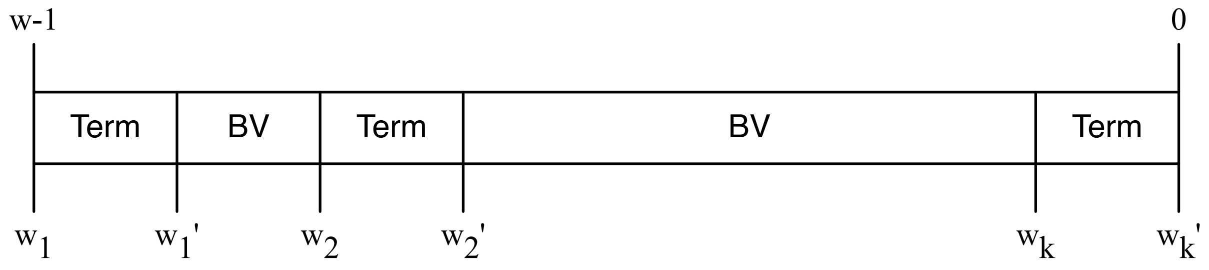


Figure 4.2: **Term-level encoding of a signal.** Each sub-range $[i : i']$ is encoded as a term. The sub-ranges not encoded as terms are encoded as bit-vectors.

Generating a term-level model from a word-level model based on type qualifiers requires to types of modifications to the word-level design.

1. Every assignment, whether combinational or sequential, including multiplexor assignments and assignments to and from memory, are split into at most $2k+1$ assignments, one assignment for each section of the newly partitioned signal.

2. Every equality is transformed into a conjunction of at most $2k+1$ equalities, one conjunct for each section of the newly partitioned signal. This translation is depicted in Figure 4.3.

After the term-level circuit is created by applying the above-mentioned modifications, the portions of the circuit corresponding to the term segments are pure term-level netlists.

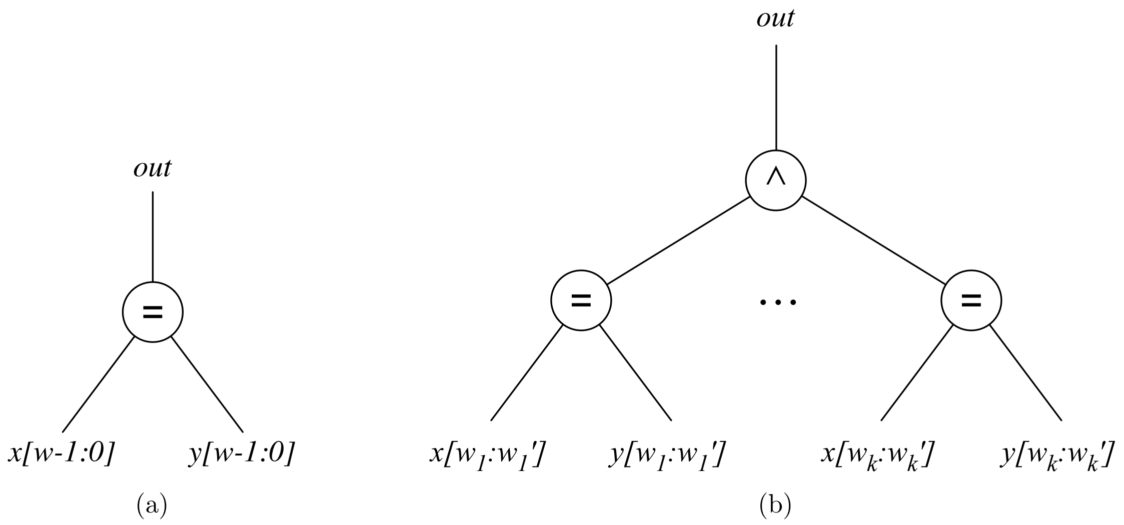


Figure 4.3: **Equality operator transformation.** (a) Original word-level equality operator. (b) Equality operators are transformed by instantiating an equality operator for each segment of the newly partitioned signal and then conjoining the equality operators together.

4.6 Case Study: CMP Router

We present experimental results using the CMP router described in Section 2.4.1.

4.6.1 Experimental Setup

Two models were created of the CMP router design described in Section 2.4.1. The first model is a word-level design that contains no abstraction aside from using lambda expressions to model queues [29]. Note that the word-level design was created automatically by V2UCL using Verilog models of the CMP router provided by Peh *et al.* [59] as input. It is possible to disable abstraction within V2UCL. In this case, V2UCL translates the Verilog description directly into a bit-vector UCLID model.

The second model of the CMP router is a term-level design that was automatically produced by V2UCL. In fact, no annotations were necessary. The type qualifier associated with the maximal abstraction of the flit datapath is $TQ_{flit} = [31{:}8]$. Intuitively, the reason TQ_{flit} is maximal is that the bottom 8-bits as shown in Figure 2.16 are extracted from the flit datapath and used in precise bit-vector operations. Bits [7:2] can't be abstracted with terms because they are used in bit-vector inequalities which determine the proper output port for a flit. Bits [1:0] are compared against bit-vector constants, and hence, they can not be abstracted with terms, either.

The environment of the router was manually modeled in UCLID. The environment injects one packet onto each input port, with the destination of the packets modeled by an symbolic destination field in the respective head flit; this models scenarios of having packets destined for different output ports as well as for the same output port (resulting in contention to be resolved by the arbiter). Bounded-model checking (BMC) was used to check that starting from a reset state, the router correctly forwards both packets to their respective output ports within a fixed number of cycles that depends on the length of the packet. If both packets are headed for the same output port, the lower priority packet must wait until all flits of the other packet have been copied to the output. This property was verified successfully by UCLID using purely bit-vector reasoning and also using the hybrid term-level reasoning. Note that in a purely term-level model, where the entire flit datapath is modeled as a single term, the router generates spurious counterexamples as the correctness depends on routing logic that performs bit extractions and comparisons on the flit header.

We ran experiments with both models for varying packet size. Recall that a packet comprises several flits. We used UCLID as the verification system for this experiment. For the bit-vector model, the bit-vector decision procedure within UCLID is invoked. For the term-level model, containing both term and bit-vector datapaths, UCLID reasons in equality logic for the term-abstracted datapaths and with bit-vector arithmetic for the rest. There were no uninterpreted functions in this model.

4.6.2 Experimental Results

Figure 4.4 compares the verification runtimes for the word-level model with those for the term-level model. We plot two curves for each type of model: one for the time taken by the SAT solver (MiniSAT), and another for the time taken by UCLID to generate the SAT problem. We can see that, for the pure bit-vector model, the time taken by the SAT solver increases rapidly with packet size, whereas the SAT time for the hybrid model increases much more gradually. Using the hybrid version achieves a speed-up of about 16X in SAT time. The improvement in time to encode to SAT is more modest. This shows that data abstraction can help increase the capacity of formal verification tools.

To evaluate whether the improvement was entirely due to reduction in SAT problem size, we plotted the ratio of speedup in SAT time along with the ratio of reduction in problem size. These plots are shown in Figure 4.5. We can see that the speedup in time does not track the reduction in problem size (which is largely constant for increasing packet size), indicating that the abstraction is assisting the SAT engine in other ways.

4.7 Related Work

The first automatic term-level abstraction tool was Vapor [6], which aimed at generating term-level models from Verilog. The underlying logic for term-level modeling in Vapor is CLU (Counter arithmetic, Lambda expressions, and Uninterpreted functions), which originally formed the basis for the UCLID system [29]. Vapor uses a counterexample-guided abstraction-refinement (CEGAR) approach [6]. Vapor has been since subsumed by the Reveal system [4, 5] which differs mainly in the refinement strategies employed within the CEGAR loop.

Both Vapor and Reveal start by completely abstracting a Verilog description to the UCLID language by modeling all bit-vector signals as abstract terms and all operators as

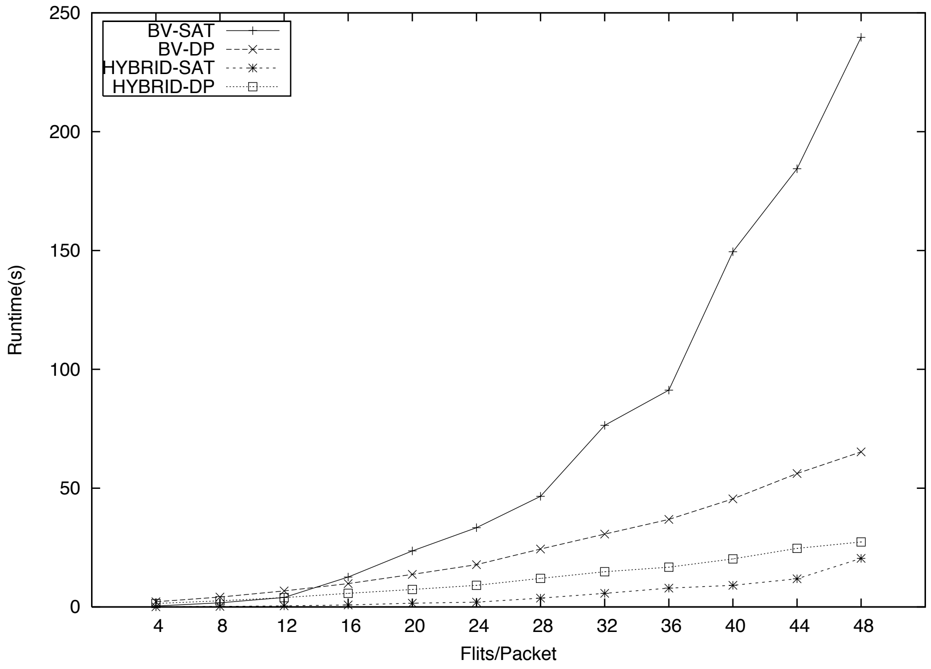


Figure 4.4: **Runtime comparison for increasing packet size in CMP router.** The runtimes for the word-level and term-level CMP designs are compared for increasing packet size. The time for the SAT solver is indicated by "SAT" and the time to generate the SAT problem by "DP."

uninterpreted functions. Next, verification is attempted on the abstracted design. If the verification succeeds, the tool terminates. However, if the verification fails, it checks whether the counterexample is spurious using a bit-vector decision procedure. If the counterexample is spurious, a set of bit-vector facts are derived, heuristically reduced, and used on the next iteration of term-level verification. If the counterexample is real, the system terminates, having found a real bug.

The CEGAR approach has shown promise [5]. In many cases, however, several iterations of abstraction-refinement are needed to infer fairly straightforward properties of data, thus imposing a significant overhead. For instance, in one example, a chip multiprocessor router [59], the header field of a packet must be extracted and compared several times to determine whether the packet is correctly forwarded. If any one of these extractions is not modeled precisely at the word level, a spurious counterexample results. The translation is complicated by the need to instantiate relations between individually accessed bit fields of a word modeled as a term using special uninterpreted functions to represent concatenation and extraction operations.

Our approach to selective term-level data abstraction is distinct from that of Andraus and Sakallah, for the following reason. Our approach is selective. Instead of abstracting all datapath signals to the term level, we abstract only those portions of the design specified by the designer. If the designer-provided annotations are incorrect, a type checker emits warnings and avoids performing the specified abstractions. When annotations are not provided,

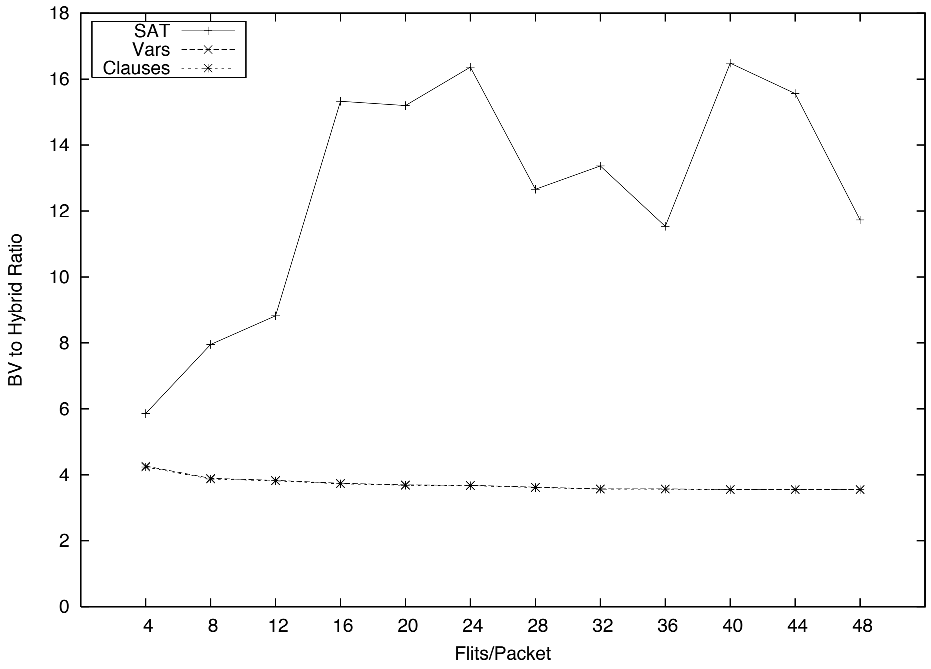


Figure 4.5: **Runtime reduction versus reduction in SAT problem size.** The reduction in runtime is labeled “SAT” and is shown in a solid line. The reduction in the number of variables and clauses in the SAT instance are shown in dashed lines and are denoted “Vars” and “Clauses”, respectively. We compare the reduction in runtime with the reduction in problem size, going from a word-level model to a term-level model, for increasing packet size. Note that the curves for the number of variables and the number of clauses coincide.

we abstract only the portions of the design expressible in EUF.

Johannesen presented an automated bit-width reduction technique which was used to scale down design sizes for RTL property checking [49, 50]. A static data-flow technique is used to partition the datapath signals based on the usage of the individual bits. Bits that are used in a symmetric way are abstracted to take the same value, while preserving satisfiability. As with the approach we present, one is able to compute a satisfying assignment for the original circuit from a satisfying assignment of the reduced circuit.

More recently, Bjesse presented a technique which is an extension to Johannesen’s work [13]. The main difference is that Bjesse includes the partitioning of the initial states, ensures that the current- and next-state partitions correspond, and bit-blasts operators such as inequalities which are left untouched in Johannesen’s work.

Our work, developed concurrently with and independent of Bjesse’s [13], is different from that work and Johannesen’s [49, 50] in that our abstraction is not limited to merely reducing bit-widths of variables. Instead, we encode the different partitions of the circuit in different logical theories. In this work, we use two theories: finite-precision bit-vector arithmetic (BV) and the logic of equality with uninterpreted functions (EUF), but the ideas are not restricted to BV and EUF. The use of logical theories allows us to use one of several techniques to encode abstracted variables, not just a single procedure; e.g., we are able to use for EUF the technique of positive equality [24] that encodes some variables into the SAT problem with constant values. Furthermore, neither Johannesen's nor Bjesse's reduction technique has a way to incorporate user insight for abstraction. In our work, we allow the user to use type qualifiers to convey his/her intuition as to how the circuit should be abstracted. If the computed abstraction does not coincide with the designer's intuition, V2UCL will warn the user and generate suitable feedback. The output from V2UCL can guide the user to write annotations or rewrite the RTL in a way that substantially reduces verification effort.

4.8 Summary

In this chapter we presented an automatic data abstraction technique called V2UCL. V2UCL takes as input an RTL design, a property to be proven over this design, and optional abstraction information (in the form of type qualifiers). If type qualifiers are provided, V2UCL determines if they are legal, and if so, abstracts the design accordingly. If the annotations are illegal or overly conservative, V2UCL will issue a warning. V2UCL can be configured to perform the maximal abstraction without relying on guidance from the user. We presented experimental evidence that V2UCL can create smaller, easier-to-verify models.

A binary distribution of V2UCL, the original Verilog models of the CMP router, bit-vector and hybrid UCLID models of the CMP router, and the experimental data from which the results are based, can be obtained from `http://uclid.eecs.berkeley.edu/v2ucl`.

Chapter 5

Automatic Function Abstraction

This chapter describes ATLAS[1], an approach to automatic function abstraction. ATLAS is a hybrid approach, involving a combination of random simulation and static analysis, and operates in three main stages. First, fblocks that are candidates for abstraction are identified using random simulation. Next, static analysis is used to compute conditions under which the fblocks must be modeled precisely. Finally, a term-level model is generated where each fblock is replaced with a partially interpreted function.

We provide evidence that shows ATLAS can be used to speed up equivalence and refinement checking problems.

5.1 Identifying Candidate Fblocks

Word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$ are derived from RTL designs in languages such as Verilog and VHDL. In such languages, modules defined by the designer provide natural boundaries for function abstraction.

Consider the flat word-level netlist obtained from an RTL design after performing all module instantiations. Every module instance corresponds to a functional block, or fblock, of the flat netlist. However, only some of these fblocks are of interest for function abstraction.

The first important notion in this regard is that of *isomorphic fblocks*.

Definition 5.1. Two fblocks $\mathcal{N}_1 = (\mathcal{I}_1, \mathcal{O}_1, \mathcal{S}_1, \mathcal{C}_1, \mathit{Init}_1, \mathcal{A}_1)$ and $\mathcal{N}_2 = (\mathcal{I}_2, \mathcal{O}_2, \mathcal{S}_2, \mathcal{C}_2, \mathit{Init}_2, \mathcal{A}_2)$ are said to be *isomorphic* if there is a bijective function φ such that $\varphi(\mathcal{I}_1, \mathcal{O}_1, \mathcal{S}_1, \mathcal{C}_1) = (\mathcal{I}_2, \mathcal{O}_2, \mathcal{S}_2, \mathcal{C}_2)$ and if we substitute every signal s in Init_1 and $\mathcal{A}_1$ by $\varphi(s)$ we obtain Init_2 and $\mathcal{A}_2$.

Thus, of all the fblocks that are candidates for function abstraction, we only consider those fblocks in $\mathcal{D}_1$ that have an isomorphic counterpart in $\mathcal{D}_2$ (and vice-versa).

Definition 5.2. A *replicated fblock* is an fblock in $\mathcal{D}_1$ that has an isomorphic counterpart in $\mathcal{D}_2$.

[1]ATLAS stands for Automatic Term-Level Abstraction of Systems

For example, each ALU in Figure 2.18 is a replicated fblock.

In equivalence or refinement checking, replicated fblocks are easy to identify as instances of the same RTL module that appear in both designs, and this is how we identify them in this work. Note, however, that it is also possible for fblocks that are not instances of the same module to be isomorphic.

Given that we identify replicated fblocks as instances of the same module, the question then becomes one of selecting RTL modules whose instances generate candidate fblocks for abstraction. Currently, we make this selection based on heuristic rules, such as the size of the module in terms of the number of input, output and internal signals, or the presence of operators such as multiplication or XOR that are hard for formal verification engines (such as SAT solvers) to reason about efficiently. Note however, that this is purely an optimization step. One can identify any set of replicated fblocks as candidates for function abstraction.

To summarize, given designs $\mathcal{D}_1$ and $\mathcal{D}_2$ that are to be checked for equivalence or refinement, we can generate the set containing all replicated fblocks in those designs. This set can be partitioned into a collection of sets of fblocks $\mathcal{FS} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k\}$. Each set $\mathcal{F}_j$ comprises replicated fblocks that are isomorphic to each other. We term each $\mathcal{F}_j$ as an *equivalence class of the fblocks it contains*. Our ATLAS approach uses the same function abstraction for every fblock in $\mathcal{F}_j$. For example, in the design of Figure 2.18, the ALU modules are isomorphic to each other and together constitute one set $\mathcal{F}_j$. ATLAS will compute the same function abstraction, shown in Figure 2.13(c), for both fblocks.

In the following sections, we describe how ATLAS analyzes the sets in $\mathcal{FS}$ in two phases. In the first phase, *random simulation* is used to prune out replicated fblocks that likely cannot be abstracted with uninterpreted functions. Every fblock that survives the first phase is then *statically analyzed* in the second phase in order to compute conditions under which that fblock can be abstracted with an uninterpreted function. The resulting conditions are used to generate a term-level netlist for further formal verification.

5.2 Random Simulation

Given an equivalence class of functional blocks $\mathcal{F}$, we use random simulation to determine whether the fblocks it contains are considered for abstraction with an uninterpreted function.

We begin by introducing some notation. Let the cardinality of $\mathcal{F}$ be l. Let each fblock $f_i \in \mathcal{F}$ have m bit-vector output signals $\langle v_{i1}, \ldots, v_{im}\rangle$, and n input signals $\langle u_{i1}, \ldots, u_{in}\rangle$. Then, we term the tuple of corresponding output signals $\chi = (v_{1j}, v_{2j}, \ldots, v_{lj})$, for each $j = 1, 2, \ldots, m$, as a tuple of *isomorphic output signals*.

Definition 5.3. A *random function* $RF(v_1, \ldots, v_n)$ is a function that returns a randomly chosen output value for a particular set of input values.

A random function is similar to an uninterpreted function in that it is functionally consistent, i.e., when applied to equal arguments, it produces the same output. The only difference between a random function and an uninterpreted function is the context within which they are used. An uninterpreted function is used in formal verification, where the inputs and output values are symbolic. A random function is used within simulation, thus, the inputs and output values are concrete. A random function can be implemented using a random number generator and a hashtable. Anytime a random function is applied to a tuple of arguments, we check to see whether the random function has been applied to that tuple previously. If so, we retrieve and return the value from the hashtable. If not, we generate a random number, store it in a hashtable with the input tuple as the key, and return it. Table 5.1 illustrates the behavior of a random function. Notice that the value of RF_{ALU} is the same in cycles 3, 4, 7 and 9. This is due to the fact that the arguments are the same in cycles 3 and 9 as well as 4 and 7.

Time	arg_A	arg_B	op	ALU	RF_{ALU}
1	0	0	+	0	9
2	0	4	+	4	17
3	4	0	+	4	3
4	7	9	+	16	3
5	4	5	+	9	6
6	2	3	+	5	1
7	7	9	+	16	3
8	6	7	+	13	12
9	4	0	+	4	3
10	5	5	+	10	8

Table 5.1: **Illustration of a random function.** *Time* denotes the current time frame; arg_A, arg_B, and op are the inputs to the ALU module and the random function RF_{ALU}; ALU is the output of the precise ALU module; and, RF_{ALU} is the output of the random function. The rows shown in dark (light) gray have equal values for RF_{ALU} because the input arguments are the same between both function applications (i.e., RF_{ALU} is functionally consistent).

As mentioned in Section 2.2.2, it is only sound to abstract combinational fblocks or acyclic, sequential fblocks. Regardless of the type of fblock (i.e., combinational or sequential) we use random functions to determine whether or not to abstract the fblock.

Combinational fblocks are considered first. Given a tuple of isomorphic output signals $\chi = (v_{1j}, v_{2j}, \ldots, v_{lj})$, we create a *random function* RF_χ unique to χ that has n inputs (corresponding to input signals $\langle u_{i1}, \ldots, u_{in}\rangle$, for fblock f_i). For each fblock f_i, $i = 1, 2, \ldots, l$, we replace the assignment to the output signal v_{ij} with the random assignment $v_{ij} \leftarrow RF_\chi(u_{i1}, \ldots, u_{in})$. This substitution is performed for all output signals $j = 1, 2, \ldots, m$.

The resulting designs $\mathcal{D}_1$ and $\mathcal{D}_2$ are then verified through simulation. This process is repeated for T different random functions RF_χ. If the fraction of failing verification runs is greater than a threshold τ, then we drop the equivalence class $\mathcal{F}$ from further consideration. (The values of T and τ we used in experiments are given in Section 5.6.) Otherwise, we retain $\mathcal{F}$ for static analysis, as described in the following section.

Now we consider replacing sequential fblocks with random functions. The difference is that for each output signal v_{ij}, instead of replacing it with the random assignment $v_{ij} \leftarrow RF_\chi(u_{i1}, \ldots, u_{in})$, we must replace it with an assignment that takes into account the history of the inputs $\langle u_{i1}, \ldots, u_{in} \rangle$, for some bounded depth, as discussed in Section 2.2.2. Let U_{ik} denote the tuple of inputs $\langle u_{i1}, \ldots, u_{in} \rangle$ that occurred k cycles in the past. Thus, U_{i0} refers to the input tuple in the current cycle, U_{i1} refers to the input tuple 1 cycle ago, and so on. Recall that to soundly abstract a sequential fblock, we must account for a history of depth 2^n, where n is the maximum number of latches in any path in the sequential fblock being considered for abstraction. We replace the assignment to the output signal v_{ij}, with the random assignment $v_{ij} \leftarrow RF_{\chi}(U_{i0}, U_{i1}, \ldots, U_{ik})$ where $k = 2^n$.

5.3 Static Analysis

The goal of static analysis is to compute conditions under which fblocks can be abstracted with uninterpreted functions (UFs) without loss of *precision* – i.e., without generating spurious counterexamples. ATLAS performs this analysis by attempting to compute the opposite condition, under which the fblocks are not abstracted with UFs.

More specifically, for each tuple of isomorphic output signals χ of each equivalence class $\mathcal{F}$, we compute a Boolean condition under which the elements of χ *should not be abstracted* as uninterpreted functions of the inputs to their respective fblocks. We term these conditions as *interpretation conditions*, with the connotation that the fblocks are precisely interpreted if and only if these conditions are **true**. Thus, we perform conditional abstraction by replacing the original fblock with a circuit that chooses between the original implementation and the abstract representation. Figure 5.1 illustrates how the output signals of an fblock are conditionally abstracted. The original word-level circuit is shown in Figure 5.1(a) and the conditionally abstracted version with interpretation condition c is shown in Figure 5.1(b). The intuition behind replacing a word-level circuit with the circuit shown in Figure 5.1(b), we are introducing a case-split on the interpretation condition. Thus, anytime the interpretation condition c is **false**, there is no need to reason about the word-level circuit. The goal of this replacement is to prevent a decision procedure from reasoning about a complicated portion of a design unless it is absolutely necessary.

Clearly, **true** is a valid interpretation condition, but it is a trivial one and not very useful. It turns out that even *checking whether a given interpretation condition is precise is co-NP-hard.* We prove this by formalizing the problem as below:

> IntCondChk: Given word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, let f_1 and f_2 be fblocks in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, where f_1 and f_2 are isomorphic. Let c be a Boolean condition such that $c \not\equiv \mathbf{true}$. Let designs $\mathcal{T}_1$ and $\mathcal{T}_2$ result from conditionally abstracting f_1 and f_2 with an uninterpreted function UF only when condition c is **false**.
>
> Then, the IntCondChk problem is to decide whether, given $\langle \mathcal{D}_1, \mathcal{D}_2, f_1, f_2, c \rangle$, $\mathcal{D}_1$ is equivalent to $\mathcal{D}_2$ if and only if $\mathcal{T}_1$ is equivalent to $\mathcal{T}_2$.

Theorem 5.4. *Problem* IntCondChk *is co-NP-hard.*

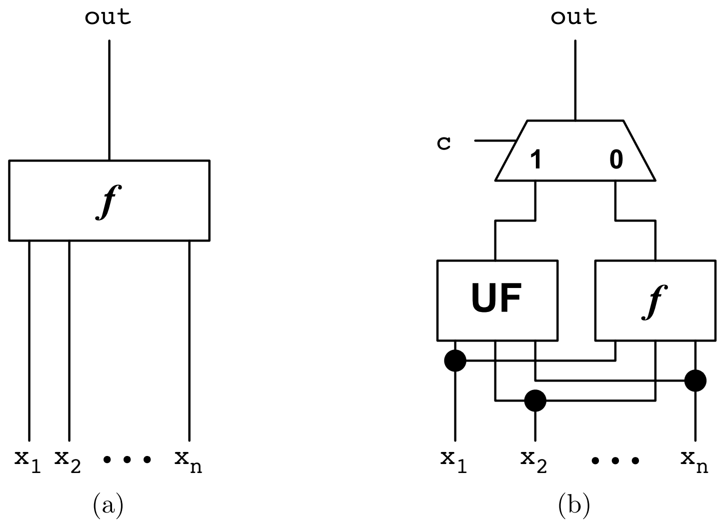


Figure 5.1: **Conditional abstraction** (a) Original word-level fblock f. (b) Conditionally abstracted version of f with interpretation condition c

Proof. The proof is by reduction from UNSAT – the Boolean unsatisfiability problem. We map an arbitrary Boolean formula f to a tuple $\langle \mathcal{D}_1, \mathcal{D}_2, f_1, f_2, c \rangle$, so that f is unsatisfiable if and only if $\mathcal{D}_1$ is equivalent to $\mathcal{D}_2$ if and only if $\mathcal{T}_1$ is equivalent to $\mathcal{T}_2$.

Consider the word-level circuit in Figure 5.2, where the $\mathcal{D}_1$ is the circuit rooted at the left-hand input of the equality node, and $\mathcal{D}_2$ is the circuit rooted at the right-hand input. Clearly, $\mathcal{D}_1$ is equivalent to $\mathcal{D}_2$. Let $c = \textbf{false}$, in other words, we want to know if unconditional abstraction is precise. Consider the multiplier blocks in $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. Since these blocks are isomorphic, we can consider replacing them with the same uninterpreted function. Note that, unless $f(x_1, x_2, \ldots, x_n)$ is equivalent to **false**, this abstraction can result in spurious counterexamples, since it is possible that $\mathit{UF}(2,5) \neq \mathit{UF}(1,10)$, whereas $\mathrm{MULT}(2,5) = \mathrm{MULT}(1,10)$ always. In other words, we answer 'yes' to this instance of IntCondChk if and only if $f(x_1, x_2, \ldots, x_n) \equiv \textbf{false}$, implying that IntCondChk is co-NP-hard.

□

Given this hardness result, ATLAS uses the following three-step procedure for verification by term-level abstraction:

1. Unconditionally abstract all isomorphic fblocks with the same uninterpreted function, for all equivalence classes of fblocks. Verify the resulting term-level designs. If the term-level verifier returns "VERIFIED", then return that result and terminate. However, if we get a counterexample, evaluate the counterexample on the word-level design to check if it is spurious. If non-spurious, return the counterexample, else go to Step 2.

2. Call Procedure ConditionalFuncAbstraction to conditionally abstract to the term-level. Again, verify the resulting term-level designs, performing exactly the same

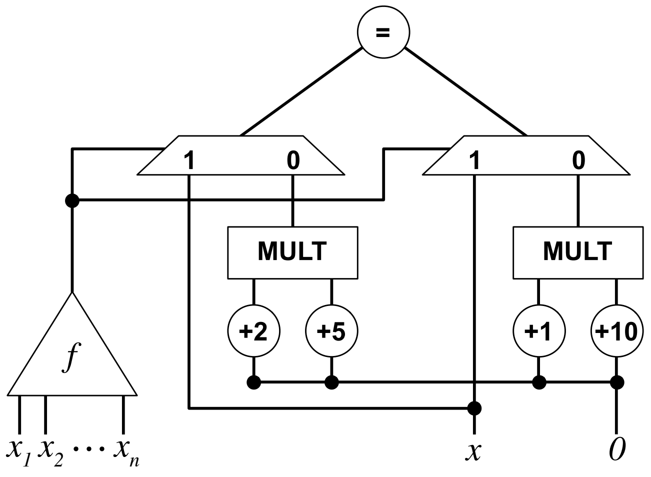


Figure 5.2: **Circuit for showing NP-hardness of IntCondChk.** f is any arbitrary Boolean function of $x_1, x_2, \ldots, x_n$.

checks as in Step 1 above: If the term-level verifier returns "VERIFIED", we return that result; otherwise, we return the counterexample only if it is non-spurious, going to Step 3 if it is spurious.

3. Invoke a word-level verifier on the original word-level designs.

The following theorem about ATLAS follows easily.

Theorem 5.5. ATLAS *is sound and complete.*

Proof. First, we give notation for the proof. Given word-level designs $\mathcal{D}_1$ and $\mathcal{D}_2$, let f_1 and f_2 be fblocks in $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively, where f_1 and f_2 are isomorphic. Let designs $\mathcal{T}_1$ and $\mathcal{T}_2$ result from conditionally abstracting f_1 and f_2 with an uninterpreted function *UF*. Note that f_1 and f_2 are two different instantiations of bit-vector function $f(x_1, \ldots, x_n)$. $Abs(c, x_1, \ldots, x_n)$ denotes the conditionally abstracted function that will replace $f(x_1, \ldots, x_n)$ in $\mathcal{D}_1$ and $\mathcal{D}_2$ to create $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and is defined as:

$$Abs(c, x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n) & \text{if } c \text{ is } \mathbf{true} \\ UF(x_1, \ldots, x_n) & \text{if } c \text{ is } \mathbf{false} \end{cases}$$

(Soundness) Soundness follows from the fact that ATLAS only attempts to verify over-approximate abstractions of the original designs. Anytime abstraction is performed within ATLAS, a precisely implemented fblock is replaced with an fblock that switches between the original, precise fblock and an uninterpreted function that contains more behaviors than the original fblock.

Notice that $f(x_1, \ldots, x_n)$ is a specific bit-vector function with the type $\mathcal{V}^n \rightarrow \mathcal{BV}$ and $UF(x_1, \ldots, x_n)$ is any bit-vector function with type $\mathcal{V}^n \rightarrow \mathcal{BV}$, including the specific bit-vector function f. There are two cases to consider. In the first case, if $c = \mathbf{true}$, then

$Abs(\mathbf{true}, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$ and $\mathcal{D}_1 = \mathcal{T}_1$ and $\mathcal{D}_2 = \mathcal{T}_2$ and therefore, $\mathcal{D}_1$ is equivalent to $\mathcal{D}_2$ if and only if $\mathcal{T}_1$ is equivalent to $\mathcal{T}_2$. In the other case, if $c = \mathbf{false}$, then $Abs(\mathbf{false}, x_1, \ldots, x_n) = UF(x_1, \ldots, x_n)$. Recall that $UF(x_1, \ldots, x_n)$ contains the behavior for all possible functions over $x_1, \ldots, x_n$, including the behavior specified by $f(x_1, \ldots, x_n)$. If $\mathcal{T}_1$ is equivalent to $\mathcal{T}_2$ when $c = \mathbf{false}$, then it means $\mathcal{T}_1$ is equivalent to $\mathcal{T}_2$ under all interpretations of $UF(x_1, \ldots, x_n)$. Under the interpretation $UF(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$, $\mathcal{T}_1 = \mathcal{D}_1$ and $\mathcal{T}_2 = \mathcal{D}_2$. Therefore, $\mathcal{D}_1 = \mathcal{D}_2$ and ATLAS is sound.

(Completeness) Completeness follows because ATLAS only outputs a counterexample if it is evaluated to be a counterexample on the original word-level design.

Anytime a counterexample arises when attempting to verify that $\mathcal{T}_1$ is equivalent to $\mathcal{T}_2$, ATLAS checks if the counterexample is spurious by simulating it on the original, word-level verification problem. If the counterexample occurs in the word-level problem, then it is a real counterexample. This counterexample is output and the procedure terminates. If the counterexample is spurious and does not occur in the word-level verification problem, the interpretation conditions are set to **true**, meaning that all modules are interpreted at all times. Thus, we now are checking the original word-level problem. Any counterexample found from this point on is real. Hence, ATLAS only outputs real counterexamples. Therefore, ATLAS is complete. □

For ease of presentation, the algorithms presented throughout this chapter are described with respect to combinational fblocks. However, the techniques can be generalized to sequential fblocks in a manner similar to that described in Section 5.2.

Algorithm 1 summarizes our static abstraction procedure. Procedure ConditionalFuncAbstraction takes two inputs. The first is the netlist $\mathcal{D}$ obtained by combining $\mathcal{D}_1$ and $\mathcal{D}_2$ to do equivalence or refinement checking. For equivalence checking, this is the standard *miter* circuit. A miter between two designs $\mathcal{D}_1$ and $\mathcal{D}_2$ is the circuit that evaluates to **true** if and only if the circuits are functionally different (i.e., when the outputs differ when the same inputs are applied to both circuits). For refinement checking, $\mathcal{D}$ is obtained by connecting inputs to $\mathcal{D}_1$ and $\mathcal{D}_2$ for use in symbolic simulation (e.g., a “flush” input to the pipeline for Burch-Dill style processor verification), as well as logic to compare the outputs of $\mathcal{D}_1$ and $\mathcal{D}_2$. The second input to ConditionalFuncAbstraction is the set of all equivalence classes of fblocks $\mathcal{FS}$. Given these inputs, ConditionalFuncAbstraction generates a rewritten netlist as output where some outputs of fblocks are conditionally rewritten as outputs of uninterpreted functions.

Algorithm 1 operates in two phases. In the first phase (lines 1-10), we identify outputs of fblocks that can be *unconditionally* abstracted with an uninterpreted function. This is performed by first computing, for every bit-vector output signal v in $\mathcal{FS}$, the equivalence class of signals $\mathcal{E}(v)$ that its value flows to or which it is compared to. Table 4.1 lists the rules for computing $\mathcal{E}(v)$. Suppose there is no signal in $\mathcal{E}(v)$ that is assigned or compared to a bit-vector constant, or is the input or output of a bit-vector arithmetic or relational operator other than equality. This implies that the value of v does not flow to any bit-vector operation, arithmetic or relational, and is never compared with a specific bit-vector constant. In such a scenario, it is possible to always abstract v as the output of an uninterpreted function. Procedure AbstractWithUF shown in Algorithm 2 performs this full function abstraction and is described later in this section.

The second phase of Algorithm 1, comprising lines 12-18, is responsible for performing conditional abstraction on the fblocks that were not unconditionally abstracted during the first phase (because those fblocks will have been removed in line 10). First, an interpretation condition c_{v_i} is associated with each bit-vector signal $v_i \in f_i$. The interpretation conditions, which are initialized to **false**, are computed by applying a set of rules to each signal $v_i \in f_i$. These rules are applied to the signals in an arbitrary order, however, the appropriate rule is applied to each signal $v_i \in f_i$ before moving onto the next iteration. This process stops when the interpretation conditions converge (i.e., there is no change to any interpretation condition c_{v_i} for all $v_i \in f_i$). Note that this process is guaranteed to converge because we only consider verification problems over a bounded number of cycles. Furthermore, it is possible to incorporate this process into an abstraction-refinement loop, by initially performing a limited number of iterations and increasing the number of iterations performed only when spurious counterexamples arise. At this point, each signal $v_i \in f_i$ has an interpretation condition c_{v_i} and conditional abstraction is performed. Finally, this procedure is repeated for each equivalence class of fblocks $\mathcal{F}_i \in \mathcal{FS}$. Recall that each $\mathcal{F}_i$ is associated with an abstraction and the set of all fblock equivalence classes $\mathcal{FS}$ corresponds to the set of all modules that we wish to abstract. Now we discuss in more detail the rules used to compute the interpretation conditions and how the interpretation conditions are used to perform conditional abstraction.

The interpretation condition rules are listed in Table 5.2. Most of the rules are intuitive, so we describe them only briefly. Consider rules 1,4, and 6: all of these involve a bit-vector operator or constant. Therefore, any signal involved in such an assignment is assigned an interpretation condition of **true**. For equality comparisons or combinational assignments (rules 2 and 3), both sides of the comparison or assignment must have the same interpretation condition. For a multiplexor assignment (rule 5), the condition under which an input of the multiplexor flows to its output is incorporated into the interpretation conditions. Rule 7, for a sequential assignment, makes use of special prev and next operators. The prev operator indicates that the condition is to be evaluated in the preceding cycle, whereas the next operator indicates that it must be evaluated in the following cycle. During symbolic simulation for term-level equivalence or refinement checking, these operators are translated to point to the conditions in the appropriate cycles. Rules 8 and 9 deal with memory reads and writes. Finally, rules 10 and 11 handle the case where we have some fblock outputs replaced with uninterpreted functions. In this case, we ensure that the arguments to uninterpreted functions (predicates) are interpreted consistently across function (predicate) applications.

As described above, the interpretation conditions are computed by applying the rules in Table 5.2 for a bounded number of iterations. After the interpretation conditions are computed, they are used to perform conditional function abstraction. Lines 15-16 of Algorithm 1 indicate the process: we first compute the disjunction of all conditions computed for output signals in an isomorphic tuple χ_j, and then use this disjunction oc_j within Procedure ConditionalAbstractWithUF to compute the new output assignment for each element of

χ_j as a conditional (ITE) expression. It is necessary to take the disjunction over the conditions computed for all output signals in χ_j so that each instance of the isomorphic fblock is interpreted (or abstracted) at the same time. The new design $\mathcal{D}'$ is the output of Procedure ConditionalFuncAbstraction. We describe the Procedures AbstractWithUF and ConditionalAbstractWithUF below.

Algorithm 1 Procedure ConditionalFuncAbstraction ($\mathcal{D}$, $\mathcal{FS}$): abstracting fblocks with uninterpreted functions, either wholly or partially.

```
Input: Combined netlist (miter) $\mathcal{D} := \langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\}\rangle$
Input: Equivalence classes of fblocks $\mathcal{FS} := \{\mathcal{F}_j \mid j = 1, \ldots, k\}$,
Output: Rewritten netlist (miter) $\mathcal{D}' := \langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}'_i \mid i = 1, \ldots, N\}\rangle$
 1: for each $\mathcal{F}_j \in \mathcal{FS}$ do
 2:   for each tuple of isomorphic output signals $\chi_j = (v_1, v_2, \ldots, v_{l_j})$, where $l_j = |\mathcal{F}_j|$,
      $v_i \in f_i$ for fblock $f_i \in \mathcal{F}_j$, $i = 1, 2, \ldots, l_j$ do
 3:     for each output signal $v_i \in \chi_j$ do
 4:       Compute equivalence class $\mathcal{E}(v_i)$ of $v_i$ by repeatedly applying rules in Table 4.1
          to all assignments in $\mathcal{D}$ except for those inside the fblock $f_i$
 5:       If $\mathcal{E}(v_i)$ contains a signal $u$ s.t. $u$ is assigned a bit-vector constant or is the input
          to or output of a bit-vector operator, mark $\chi_j$.
 6:     end for
 7:     If $\chi_j$ is unmarked { $\mathcal{F}_j \leftarrow$ AbstractWithUF ($\mathcal{F}_j$,$\chi_j$) }
 8:   end for
 9: end for
10: For all $\mathcal{F}_j \in \mathcal{FS}$, if all isomorphic output signal tuples $\chi_j$ are unmarked, delete $\mathcal{F}_j$ from
    $\mathcal{FS}$.
11: // Now compute conditions for partial abstraction with a UF
12: for each remaining $\mathcal{F}_j \in \mathcal{FS}$ do
13:   for each tuple of isomorphic output signals $\chi_j = (v_1, v_2, \ldots, v_{l_j})$, where $v_i \in f_i$ for
      fblock $f_i \in \mathcal{F}_j$, $i = 1, 2, \ldots, l_j$ do
14:     Compute interpretation conditions $\mathsf{c}_{v_i}$ for all $i$ by repeatedly applying rules in Ta-
        ble 5.2 to the netlist obtained by deleting all signals (and corresponding assign-
        ments) inside fblocks in $\mathcal{F}_j$. The rules are applied until the conditions do not
        change or up to a specified bounded number of iterations, whichever is smaller.
15:     Compute $\mathsf{oc}_j := \bigvee_{i=1}^{l_j} \mathsf{c}_{v_i}$.
16:     Perform partial function abstraction of $\mathcal{F}_j$ with $\mathsf{oc}_j$:
        $\mathcal{F}_j \leftarrow$ ConditionalAbstractWithUF ($\mathcal{F}_j$,$\chi_j$,$\mathsf{oc}_j$)
17:   end for
18: end for
```

Procedure AbstractWithUF, shown in Algorithm 2, replaces an fblock with a completely uninterpreted function. Procedure AbstractWithUF takes two inputs. The first is an equivalence class $\mathcal{F}$ of fblocks. The second is a tuple of isomorphic output signals $\chi = (v_1, v_2, \ldots, v_l)$ corresponding to the fblocks in $\mathcal{F}$. Procedure AbstractWithUF operates as follows. First, a fresh uninterpreted function UF_χ is created for the tuple of isomorphic outputs χ (line 1). Next, the assignment to each output signal $v_i \in \chi$ is replaced with the assignment $v_i \leftarrow UF_\chi(i_1, \ldots, i_{k_i})$ (lines 3-4). Finally, all of the assignments contained within the fblocks f_i's, are deleted (line 5). The deletion of assignments occurs in order from the outputs to the inputs, terminating at (but not deleting) the assignments to the input signals.

Procedure ConditionalAbstractWithUF, shown in Algorithm 3, replaces an fblock with a conditionally abstracted fblock as illustrated in Figure 5.1. Procedure ConditionalAbstractWithUF takes three inputs. The first two inputs are the same as Procedure AbstractWithUF. Namely, an equivalence class $\mathcal{F}$ of fblocks and a tuple of isomorphic output signals $\chi = (v_1, v_2, \ldots, v_l)$ corresponding to the f_i's of $\mathcal{F}$. The third input argument is an interpretation condition `oc` corresponding to the isomorphic output tuple χ. Procedure ConditionalAbstractWithUF operates as follows. First, similarly with Procedure AbstractWithUF, a fresh uninterpreted function UF_χ is created for the tuple of isomorphic outputs χ (line 1). Next, the assignment to each output signal $v_i \in \chi$ is replaced with the assignment $v_i \leftarrow ITE(\texttt{oc}, e, UF_\chi(i_1, \ldots, i_{k_i}))$ (lines 3-4). Note that it is not necessary to delete any assignments as is the case with Procedure AbstractWithUF. In fact, the assignments *must* be kept because they are used anytime `oc` is **true**.

Algorithm 2 Procedure AbstractWithUF ($\mathcal{F}$, χ): wholly abstract outputs in χ with uninterpreted functions.

```
Input: Equivalence class of functional blocks $\mathcal{F} = \{f_1, f_2, \ldots, f_l\}$
Input: Tuple of isomorphic output signals of $f_i$'s $\chi = (v_1, v_2, \ldots, v_l)$
Output: Updated functional blocks $\mathcal{F}'$.
1: Create a fresh uninterpreted function symbol $UF_\chi$.
2: for each output signal $v_i \in \chi$ do
3:     Let $(i_1, \ldots, i_{k_i})$ denote the input symbols to fblock $f_i$.
4:     Replace the assignment $v_i \leftarrow e$ in $f_i$ with the assignment $v_i \leftarrow UF_\chi(i_1, \ldots, i_{k_i})$.
5:     Transitively delete all assignments $u \leftarrow e$ or $u := e$ in $f_i$ where signal $u$ does not
       appear on the right-hand side of any assignment in $f_i$.
6:     Denote the resulting fblock by $f_i'$.
7: end for
8: return Updated equivalence class of fblocks $\mathcal{F}' = \{f_1', f_2', \ldots, f_l'\}$.
```

5.4 Illustrative Example

We illustrate the operation of our approach on the equivalence checking problem shown in Figure 5.3. Note that Figure 5.3 describes the same circuit as Figure 2.18, we duplicate it here for convenience. Note that all signals in this design have been given names from v_1 (out_A) to v_{17} (out_B).

Assume that the ALU modules have passed the first two steps in ATLAS: identifying replicated fblocks `A.ALU` and `B.ALU` and performing random simulation.

Rule No.	English Description	Form of Assignments	Rules for Updating Interpretation Condition
1.	Bit-vector constant	$v \leftarrow c$	$\mathsf{c}'_v := \mathbf{true}$
2.	Combinational copy	$v \leftarrow u$	$\mathsf{c}'_v := \mathsf{c}_v \vee \mathsf{c}_u$ $\mathsf{c}'_u := \mathsf{c}_v \vee \mathsf{c}_u$
3.	Equality comparison	$b \leftarrow v = u$	$\mathsf{c}'_v := \mathsf{c}_v \vee \mathsf{c}_u$ $\mathsf{c}'_u := \mathsf{c}_v \vee \mathsf{c}_u$
4.	Bit-vector relational operator	$b \leftarrow \mathsf{bvrel}(v_1, v_2, \ldots, v_k)$	$\mathsf{c}'_{v_i} := \mathbf{true}$ $\forall i = 1, 2, \ldots, k$
5.	Multiplexor assignment	$v \leftarrow ITE(b,\ v_1,\ v_2)$	$\mathsf{c}'_v := \mathsf{c}_v \vee (b \wedge \mathsf{c}_{v_1} \vee \neg b \wedge \mathsf{c}_{v_2})$ $\mathsf{c}'_{v_1} := \mathsf{c}_{v_1} \vee (b \wedge \mathsf{c}_v)$ $\mathsf{c}'_{v_2} := \mathsf{c}_{v_2} \vee (\neg b \wedge \mathsf{c}_v)$
6.	Bit-vector operator	$v \leftarrow \mathsf{bvop}(v_1, v_2, \ldots, v_k)$ $v \leftarrow \mathsf{bvmanip}(v_1, v_2, \ldots, v_k)$	$\mathsf{c}'_v := \mathbf{true}$ $\mathsf{c}'_{v_i} := \mathbf{true}\ \forall i = 1, 2, \ldots, k$
7.	Sequential assignment	$v := u$	$\mathsf{c}'_v := \mathsf{c}_v \vee \mathsf{prev}(\mathsf{c}_u)$ $\mathsf{c}'_u := \mathsf{c}_u \vee \mathsf{next}(\mathsf{c}_v)$
8.	Memory read	$v \leftarrow \mathbf{read}(M, u)$	$\mathsf{c}'_v := \mathsf{c}_v \vee \mathsf{c}_M$ $\mathsf{c}'_M := \mathsf{c}_v \vee \mathsf{c}_M$
9.	Memory write	$M := \mathbf{write}(M, v_a, v_d)$	$\mathsf{c}'_M := \mathsf{c}_M \vee \mathsf{prev}(\mathsf{c}_{v_d})$ $\mathsf{c}'_{v_d} := \mathsf{next}(\mathsf{c}_M) \vee \mathsf{c}_{v_d}$
10.	Uninterpreted function	$v_x \leftarrow UF(v_{x_1}, \ldots, v_{x_k})$ $v_y \leftarrow UF(v_{y_1}, \ldots, v_{y_k})$	$\mathsf{c}'_{v_x} := \mathsf{c}_{v_x},\ \mathsf{c}'_{v_y} := \mathsf{c}_{v_y}$ $\mathsf{c}'_{v_{x_i}} := \mathsf{c}_{v_{x_i}} \vee \mathsf{c}_{v_{y_i}} \forall i = 1 \ldots k$ $\mathsf{c}'_{v_{y_i}} := \mathsf{c}_{v_{x_i}} \vee \mathsf{c}_{v_{y_i}} \forall i = 1 \ldots k$
11.	Uninterpreted predicate	$b_x \leftarrow UP(v_{x_1}, \ldots, v_{x_k})$ $b_y \leftarrow UP(v_{y_1}, \ldots, v_{y_k})$	$\mathsf{c}'_{v_{x_i}} := \mathsf{c}_{v_{x_i}} \vee \mathsf{c}_{v_{y_i}} \forall i = 1 \ldots k$ $\mathsf{c}'_{v_{y_i}} := \mathsf{c}_{v_{x_i}} \vee \mathsf{c}_{v_{y_i}} \forall i = 1 \ldots k$

Table 5.2: **Rules for computing interpretation conditions.** Rules for computing the *interpretation condition* c_v for every bit-vector (or memory) signal v (or M) in a set of signals V. Every condition c_v initially starts out as **false**. c'_x denotes the updated value of c_x for a bit-vector or memory signal x.

Algorithm 3 Procedure ConditionalAbstractWithUF ($\mathcal{F}$, χ, oc): conditionally abstract outputs in χ with uninterpreted functions using condition oc.

```
Input: Equivalence class of functional blocks $\mathcal{F} = \{f_1, f_2, \ldots, f_l\}$
Input: Tuple of isomorphic output signals of $f_i$'s $\chi = (v_1, v_2, \ldots, v_l)$
Input: Boolean condition: $\mathsf{oc} := \bigvee_{i=1}^{l} \mathsf{c}_{v_i}$.
Output: Updated functional blocks $\mathcal{F}'$.
1: Create a fresh uninterpreted function symbol $UF_\chi$.
2: for each output signal $v_i \in \chi$ do
3:     Let $(i_1, \ldots, i_{k_i})$ denote the input symbols to fblock $f_i$.
4:     Replace the assignment $v_i \leftarrow e$ in $f_i$ with the assignment
       $v_i \leftarrow ITE(\mathsf{oc},\ e,\ UF_\chi(i_1, \ldots, i_{k_i}))$.
5:     Denote the resulting fblock by $f_i'$.
6: end for
7: return Updated equivalence class of fblocks $\mathcal{F}' = \{f_1', f_2', \ldots, f_l'\}$.
```

We describe how procedure ConditionalFuncAbstraction operates on this example. The first phase of ConditionalFuncAbstraction computes equivalence classes of the output signals v_1 and v_{17} of the two ALUs. We observe that

$$\begin{aligned}\mathcal{E}(v_1) &= \mathcal{E}(v_{17}) \\ &= \{v_1, v_2, v_4, v_5, v_{13}, v_{10}, v_9, v_{12}, v_{16}, v_{14}, v_{15}, v_{11}, v_{17}\}\end{aligned}$$

Clearly, since some of the above signals are outputs or inputs of bit-vector arithmetic operators such as + and bit-extraction, we cannot abstract the two ALUs unconditionally with an uninterpreted function.

Therefore, ConditionalFuncAbstraction performs the second phase: computing interpretation conditions for the signals in Designs A and B.

As stated in the caption of Table 5.2, all conditions are initialized to **false**.

Next, consider all signals that are inputs or outputs of bit-vector operators, or compared with a bit-vector constant (such as JMP). We apply Rules 1 and 6 to these signals, to get:

$$\begin{gathered}\mathsf{c}_{v_2} = \mathsf{c}_{v_5} = \mathsf{c}_{v_8} = \mathsf{c}_{v_6} = \mathsf{c}_{v_7} = \textbf{true} \text{ and} \\ \mathsf{c}_{v_{13}} = \mathsf{c}_{v_{10}} = \mathsf{c}_{v_{14}} = \mathsf{c}_{v_1} = \mathsf{c}_{v_{15}} = \textbf{true}\end{gathered}$$

Since we have the assignments $v_5 := v_4$ and $v_{13} := v_{12}$, we can apply Rule 7 to obtain

$$\mathsf{c}_{v_4} = (v_5) = \textbf{true} \text{ and } \mathsf{c}_{v_{12}} = (v_{13}) = \textbf{true}$$

Now, using Rule 5 for the multiplexor in Design A, we obtain

$$\mathsf{c}_{v_1} = \{(v_7 = \mathsf{JMP}) \wedge \mathsf{c}_{v_4}\} = (\texttt{A.instr}[19:16] = \mathsf{JMP})$$

Finally, using Rule 3 for the equality corresponding to `out_ok`, we conclude that $\mathsf{c}_{v_{17}} = \mathsf{c}_{v_1} = (\texttt{A.instr}[19:16] = \mathsf{JMP})$.

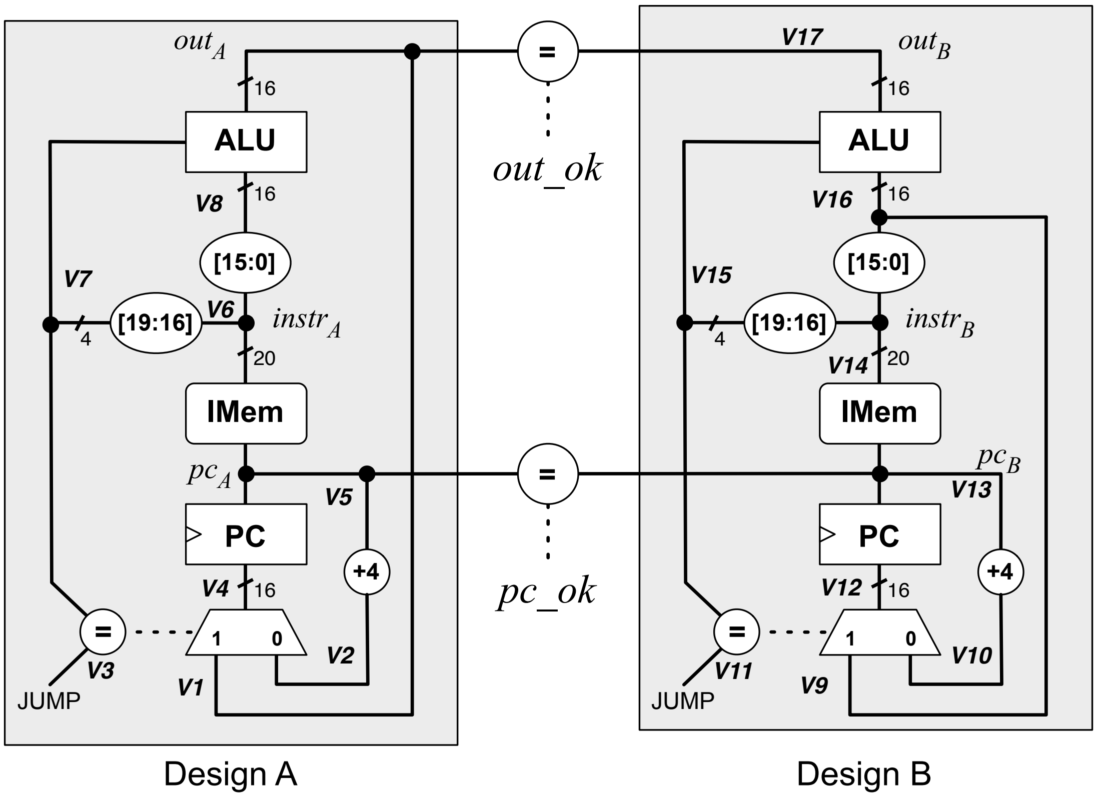


Figure 5.3: **Equivalence checking of two versions of a portion of a processor design.** Boolean signals are shown as dashed lines and bit-vector signals as solid lines.

At this point, the interpretation conditions have converged, causing the procedure to terminate. We can thus compute a partial abstraction of the ALUs using a fresh uninterpreted function symbol *UF* by employing the new assignments below:

$$\begin{aligned} v_1 &\leftarrow \mathit{ITE}(\texttt{A.instr}[19:16] = \mathsf{JMP},\ \mathit{ALU}(v_8),\ \mathit{UF}(v_8)) \\ v_{17} &\leftarrow \mathit{ITE}(\texttt{A.instr}[19:16] = \mathsf{JMP},\ \mathit{ALU}(v_{16}),\ \mathit{UF}(v_{16})) \end{aligned}$$

Note that *ALU* above refers to the original ALU as shown in Figure 2.13(a). The right-hand side expressions in the new assignments shown above are instances of the partially-abstracted ALU shown in Figure 2.13(c).

In summary, for our running example, ATLAS correctly computes the conditions under which the ALU can be abstracted with an uninterpreted function.

5.5 Benchmarks

In addition to the example described in Section 5.4, we performed experiments on four benchmarks: a simplified pipelined processor [20], the packet disassembler from the USB 2.0 function core [58], a power-gated calculator design [65], and the Y86 processor designs [26]. We describe these benchmarks here.

5.5.1 Pipelined Datapath (PIPE)

This example is based on the simple pipeline described in the UCLID userguide [20]. This processor consists of 3-stages: fetch, execute, and writeback. It supports 7 arithmetic instructions and has a 32x32-bit, dual-read, single-write register file. The design we use here differs from the one in the UCLID manual [20] only in that it does not use memory abstraction for the register file. We verify that the pipelined processor refines (i.e., is simulated by) a single-cycle, sequential version of the processor using Burch-Dill style correspondence checking [31]. The shared state variables that are checked for equality are the program counter (PC) and register file (RF). Excluding the top-level processor modules, there are 3 candidate modules for abstraction: PC update, RF, and arithmetic-logic unit (ALU). The PC update and ALU modules both pass the random simulation stage of ATLAS, however, we only abstract the ALU due to the small size of the PC update module. The RF module does not pass random simulation and, hence, we don't abstract it. By replacing the RF with a combinational random function, we lose the ability to store values, which causes random simulation to fail.

5.5.2 USB Controller (USB)

This example relies on a modified version of the packet disassembler in the USB 2.0 Function Core [58]. In the refined version, we removed the notion of a TOKEN packet and updated the state machine and other relevant logic accordingly. We performed bounded equivalence checking on the original and refined packet disassemblers by injecting packets on each cycle. The property checked was that the disassembler state, error condition state, and cyclic redundancy check (CRC) error signals were the same for each cycle. The two candidate modules for abstraction were the 16- and 5-bit CRC modules. Both passed random simulation, which is expected because neither influences the state machine control, however, only the 16-bit CRC module was abstracted because the 5-bit CRC module is not in the cone-of-influence of the property being checked.

5.5.3 Calculator (CALC)

This experiment is based on one of the calculator designs described in [65]. The design we use is the second in a series of calculator implementations, we refer to it as CALC. CALC is a pipelined implementation of a calculator that has 4 datapaths each with its own input and output port. CALC accepts accepts 4 instructions: add, subtract, shift left, and shift right. Each port can have up to 4 outstanding instructions. A two-bit tag is used to keep track of outstanding instructions. There are two main execution units shared between the datapaths in this design: the adder-subtractor unit (ASU) and the shifter unit.

For this experiment, we created a power-gated version of an existing calculator design, in a manner similar to that in [41]. In the power-gated version, the ASU is powered down (by fencing the outputs) whenever there are no add or subtract instructions in the add/subtract queue. We performed equivalence checking on the outputs of the two versions to make sure that the correct results come out in the same order, with the proper tags, and on the correct ports. For this design, there are only 2 modules which passed random simulation: the ASU and the shifter. There are many modules which didn't pass random simulation. An example is the priority module. The priority module takes the incoming commands and adds them to the appropriate queues and dispatches commands to the appropriate unit (ASU or shifter). The priority module is a sequential fblock containing cycles, therefore we do not consider it for abstraction.

5.5.4 Y86 Processor (Y86)

In this experiment, we verify two versions of the well-known Y86 processor model introduced by Bryant and O'Hallaron [30]. The Y86 processor is a pipelined CISC microprocessor styled after the Intel IA32 instruction set. While the Y86 is relatively small for a processor, it contains several realistic features, such as a dual-read, dual-write register file, separate data and instruction memories, branch prediction, hazard resolution, and an ALU that supports bit-vector arithmetic and logical instructions. There are several variants of the Y86 processor:

STD The base implementation. Hazards are resolved by a combination of forwarding and stalling. Branches are predicted as taken, with up to two instructions cancelled on misprediction.

FULL Implements two additional instructions.

STALL Uses stalling to resolve all hazards.

NT Conditional branches are predicted as not taken.

BTFNT Similar to NT, except that backward branches are predicted as taken, while forward branches are predicted as not taken.

LF An additional forwarding path is added, allowing some load/use hazards to be resolved by forwarding rather than stalling.

SW The register file has only one write port, requiring execution of the pop instruction to occur over two cycles: one to update the stack pointer and the other to read from memory.

The property we wish to prove on the Y86 variants is Burch-Dill style correspondence-checking [31]. In correspondence checking, a pipelined version of a processor is checked against a single-cycle version. The main goal of correspondence checking is to verify that the pipeline control logic allows all of the same behaviors that the instruction set architecture (ISA) supports. Each run of correspondence checking involves injecting an instruction into the pipeline and subsequently flushing the pipeline to allow the effects of the injected instruction to update the processor state. A single run of correspondence checking requires over a dozen cycles of symbolic simulation.

Each Y86 version has the same module hierarchy. The following modules are candidates for abstraction: register file (RF), condition code (CC), branch function (BCH), arithmetic-logic unit (ALU), instruction memory (IMEM), and data memory (DMEM). The RF module is ruled out as a candidate for abstraction during the random simulation stage due to a large number of failures during verification via simulation. This occurs because an uninterpreted function is unable to accurately model a mutable memory. We do not consider IMEM and DMEM for automatic abstraction because they are memories and we do not address automatic memory abstraction in this work. Instead, we manually model IMEM and DMEM with completely uninterpreted functions. The CC and BCH modules are also removed from consideration due to the relatively simple logic contained within them. Abstracting these modules is unlikely to yield substantial verification gains and may even hurt performance due to the overhead associated with uninterpreted functions. This leaves us with the ALU module.

5.6 Results

The hypothesis we test with our experiments is that performing automatic term-level abstraction before verification can yield substantial speedups over verifying the original word-level design. While we would have liked to compare with the Reveal or Vapor systems, they are not publicly available. Our own experience with performing fine-grained term-level abstraction as with Reveal/Vapor is that there are far too many spurious counterexamples generated to yield any improvements, especially given the recent advances in bit-vector SMT solvers.

Our experiments were performed by first extracting ATLAS netlist representations from the Verilog RTL. Random simulations were performed using the Icarus Verilog simulator [66]. We used $T = 1000$ random functions for each equivalence class of fblocks, selecting a class for function abstraction if at most $\tau = 50$ (5%) simulations failed. ATLAS translates both word-level and term-level netlists into UCLID format [20], before using UCLID's symbolic simulation engine to perform bounded equivalence checking or refinement (correspondence) checking of processor designs. Experiments were run on a Linux workstation with 64-bit 3.0 GHz Xeon processors and 2 GB of RAM.

Some characteristics of the benchmarks are given in the first six columns of Table 5.3. The size of the designs are described in terms of the numbers of latches as well as the number of signals in the word-level netlist (based on ATLAS' representation). In general, random simulation was very effective at pruning out fblocks that cannot be replaced with uninterpreted functions. For the PIPE, USB, and Y86 designs, only two fblocks survived the results of random simulation, both being instantiations of the same Verilog module (one in each circuit in the equivalence/refinement check). For the CALC, the ADD/SUB as well as the Shifter fblocks could be abstracted, again symmetrically on each side of the miter.

Once candidate fblocks are identified for abstraction, ATLAS generates word-level and term-level UCLID models using the approach outlined in Section 5.3. UCLID is used to perform symbolic simulation. For refinement checking of processor designs, the number of cycles of symbolic simulation is defined by the Burch-Dill approach [31] and based on the

Name	*Benchmark Characteristics*					*Performance Comparison*										
	L	N_{orig}	N_{fb}	N_{abs}	Abs	N	Word-Level (sec.)		ATLAS & Term-Level (sec.)					Speedup	
							SMT	Total	Iter	RSim	Static	SMT	Total	Total	SMT
PIPE	2233	1233	2	42	ALU (2)	9	171.09	171.22	0	3.71	0.20	1.86	5.77	29.7	92.0
USB	134	892	2	252	CRC16 (2)	15	0.29	0.38	0	3.52	0.06	0.11	3.69	0.1	2.6
						25	0.53	0.62	0	5.63	0.06	0.20	5.89	0.1	2.7
CALC	5539	2913	4	54	Add/Sub (2), Shifter (2)	15	11.82	12.64	0	8.43	0.88	2.40	11.71	1.1	4.9
						25	133.72	134.76	0	14.14	1.16	23.86	39.16	3.4	5.6
Y86-BTFNT	567	936	2	36	ALU (2)	13	1077.72	1077.84	1	5.20	0.09	1385.34	1390.63	0.7	0.7
Y86-FULL	567	961	2	36	ALU (2)	13	2166.66	2166.78	0	4.44	0.09	56.30	60.83	35.6	38.5
Y86-LF	567	931	2	36	ALU (2)	13	728.05	728.17	0	4.18	0.09	42.11	46.38	15.7	17.3
Y86-NT	567	928	2	36	ALU (2)	13	1736.66	1736.77	1	4.37	0.08	1350.95	1355.40	1.28	1.28
Y86-STD	567	923	2	36	ALU (2)	13	1239.00	1239.12	0	5.22	0.08	54.19	59.49	20.8	22.9

Table 5.3: **Performance Comparison and Benchmark Characteristics.** Column headings are as follows: L: Number of latches in original word-level netlist; N_{orig}: Number of signals in word-level netlist; N_{fb}: Number of fblocks selected by random simulation; N_{abs}: Total number of signals in the selected fblocks, Abs are the names of the RTL module abstracted (with number of instances); "Word-level" indicates columns for verification of original word-level model; "Term-level" indicates columns for verification of ATLAS-generated term-level model; N: Number of steps of symbolic simulation for equivalence/refinement checking; SMT indicates the time taken by the Boolector SMT solver; Iter is the number of iterations of interpretation condition computation; RSim is the runtime for random simulation; Static is the time for ATLAS' static analysis; Total indicates the total verifier time (for the word-level model, this includes SMT time, for the term-level model, this includes RSim, Static, and SMT times); "Speedup": the speedup of the term-level verification over the word-level verification tasks, for both SMT time and Total time.

pipeline depth. For equivalence checking tasks, we performed symbolic simulation for various numbers of cycles. Both verification tasks, at the end, generate a decision problem in a combination of logical theories. For word-level models, this problem is in the theory of finite-precision bit-vector arithmetic, possibly including the theory of arrays if memory abstraction is performed (as for Y86 benchmarks). For term-level models, the decision problem is in the combination of bit-vector arithmetic, uninterpreted functions, and arrays. We experimented with several SMT solvers for this combination of theories, including Boolector, MathSAT, and Yices, three of the top solvers in the 2008 and 2009 SMTCOMP competition [33]. We present our results for Boolector [22], the SMT solver that performs best for the word-level designs.

The experimental results are presented in the last 9 columns of Table 5.3. Consider the last two columns of the table. Here we present two ratios: "SMT" indicates the speedup of running Boolector on ATLAS output versus the original design. We observe that we get a speedup on all benchmarks, ranging from a factor of 2 to 92. However, when the running time for random simulations and static analysis is factored in ("Total"), we observe that ATLAS does worse on the USB design, and has a somewhat smaller speedup on the other designs. The main reason is the time spent in random simulation. We believe there is scope for optimizing the performance of the random simulator, as well as amortizing simulation time across different formal verification runs.

We also experimented with a purely SAT-based approach. Here the word-level problems are bit-blasted to a SAT problem. For term-level problems, UCLID first eliminates uninterpreted function applications using Ackermann's method as described in Section 3.4.1, and then the resulting word-level problem is bit-blasted to SAT. We experimented with several SAT engines, including MiniSAT [40, 38], PicoSat [12], and Precosat [47].

Table 5.4 reports the SAT problem sizes and run-times for a selected subset of generated SAT problems. The run-time of the best SAT solver is reported for each run. For the CALC example, the SAT solvers perform better on the original word-level model, which is understandable, since reasoning about addition, subtraction, and shifting is not particularly hard for SAT engines. Thus, by abstracting these operators, we complicate the verification problem by adding functional consistency constraints (as described in Section 3.4.2). For the PIPE and USB examples, however, term-level abstraction by ATLAS performs significantly better *even when the SAT problem size is much bigger*. This indicates the benefit of abstracting modules such as CRC16 which can have operators such as XORs that are hard for SAT engines. This data supports the intuition given in Section 5.3 behind why conditional abstraction is effective.

5.7 Related Work

While Vapor and Reveal [6, 4, 5], as described in Section 4.7, are related to the ATLAS approach described in this chapter, there are a few key differences.

Vapor and Reveal perform both data and function abstraction, whereas ATLAS focuses solely on function abstraction. Furthermore, instead of individually abstracting all bit-vector operators, ATLAS performs selective abstraction at the module level. Thus, instead

Name	N	Abs?	SAT Size		Run-time
			#Vars	#Clauses	(sec.)
PIPE	9	No	41911	122203	>3600
		Yes	45644	133084	29.86
USB	25	No	17667	51916	>3600
		Yes	159509	475057	68.74
CALC	25	No	351892	1039501	823.71
		Yes	753164	2234485	1771.66

Table 5.4: **Performance Comparison of SAT-based Verification.** N is the number of cycles symbolically simulated. "Abs?" indicates whether term-level abstraction via ATLAS was used or not.

of replacing individual operators with uninterpreted functions, we replace entire modules with uninterpreted functions. Another key difference is that ATLAS employs *conditional abstraction*. Thus, a verification model produced by ATLAS retains the original functionality, as well as the abstract functionality, and then chooses between the two based on the interpretation conditions computed via static analysis. The benefits of using the ATLAS approach are as follows:

1. The relative gain from using an uninterpreted function is typically greater, because each uninterpreted function is replacing many precise operators.

2. Uninterpreted functions require constraints to enforce functional consistency. These constraints grow quadratically with the number of applications of a particular uninterpreted function. Thus, it is important to use uninterpreted functions only when necessary, otherwise the functional consistency constraints are introducing unnecessary overhead.

3. Vapor and Reveal replace all operators with uninterpreted functions. However, certain operations, such as extract and concatenation, are not hard for bit-vector solvers to reason about. Thus, the use of uninterpreted functions will be more expensive due to the consistency constraints and, in the case of extract and concatenate, the uninterpreted function is essentially replacing a wire.

This work is the first to propose *conditional term-level abstraction* and demonstrate that it actually works. Furthermore, this work is the first to combine random simulation with static analysis to perform *automatic conditional function abstraction.* Potentially, if the statically-computed conditions generated by ATLAS make the problem size too large, one can fall back to a CEGAR approach.

We also note that the ATLAS approach presented herein could in principle be combined with bit-width reduction techniques (e.g. [49, 13, 16]) to perform combined function and data abstraction.

5.8 Summary

In this chapter we presented an automatic approach to function abstraction called ATLAS. ATLAS takes a word-level RTL design and creates a conditionally abstracted term-level design. ATLAS relies on a combination of random simulation and static analysis. Random simulation is used to identify what modules to abstract. Static analysis is used to compute conditions under which it is precise to abstract. We presented experimental evidence that ATLAS can produce easier-to-verify models, even when the underlying SAT or SMT problem is larger than the corresponding word-level model.

A binary distribution of ATLAS, UCLID models of the benchmarks used throughout this chapter, and the experimental data from which the results in Section 5.6 are based, can be obtained at `http://uclid.eecs.berkeley.edu/atlas`.

Chapter 6

Learning Conditional Abstractions

In this chapter we present CAL[1] , an alternative approach to automatic function abstraction. CAL is a layered approach based on a combination of random simulation, machine learning, and counterexample-guided abstraction-refinement (CEGAR). CAL is similar to ATLAS in the way it decides what blocks to abstract, however, it is different in the way in which interpretation conditions are computed. CAL uses machine learning [56] instead of static analysis to compute interpretation conditions.

A high-level overview of the CAL approach is as follows. Random simulation is used to determine candidate modules for abstraction. A verifier is invoked on a term-level model where the candidate modules are completely abstracted with uninterpreted functions. If spurious counterexamples arise, machine learning is used to compute interpretation conditions under which abstraction can be performed without loss of precision. This process is repeated until we arrive with a term-level model that is valid or a legitimate counterexample is found. Figure 6.1 illustrates the CAL approach.

6.1 Top-Level CAL Procedure

The top-level CAL procedure, VerifyAbs, is described in Algorithm 4. VerifyAbs takes two arguments, the design $\mathcal{D}$ being verified and the set of equivalence classes being abstracted $\mathcal{FS}_{\mathcal{A}}$. Initially, the interpretation conditions $\mathsf{c}_i \in \mathcal{IC}$ are set to **false** meaning that we start by unconditionally abstracting the fblocks in $\mathcal{D}$. The procedure CondAbs creates the abstracted term-level design $\mathcal{T}$ from three inputs: the word-level design $\mathcal{D}$, the set of equivalence classes to be abstracted $\mathcal{FS}_{\mathcal{A}}$, and the set of interpretation conditions $\mathcal{IC}$. Next, a term-level verifier is invoked on $\mathcal{T}$. If Verify $(\mathcal{T})$ returns Valid, we report that result and terminate. If a counterexample arises, the counterexample is evaluated on the word-level design. If the counterexample is non-spurious, VerifyAbs reports the counterexample and terminates, otherwise the counterexample is stored in $\mathcal{CE}$ and the abstraction condition learning procedure, LearnAbsConds $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{CE})$, is invoked.

We say that VerifyAbs is *sound* if it reports Valid if and only if $\mathcal{D}$ is correct. It is

[1]CAL stands for Conditional Abstraction through Learning

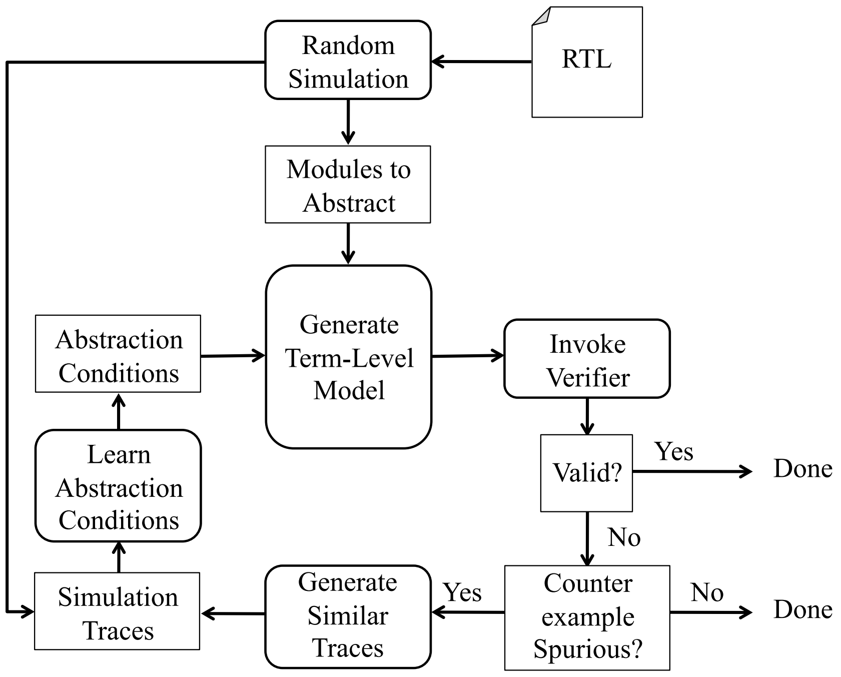


Figure 6.1: **The CAL approach** A CEGAR-based approach, CAL identifies candidate abstractions with random simulation and uses machine learning to refine the abstraction if necessary.

complete if any counterexample reported by it when it terminates is a true counterexample (i.e., not spurious). We have the following guarantee for the procedure VerifyAbs:

Theorem 6.1. *If* VerifyAbs *terminates, it is sound and complete.*

Proof. Any term-level abstraction is a sound abstraction of the original design, since any partially-interpreted function (for any interpretation condition) is a sound abstraction of the fblock it replaces. Thus VerifyAbs is sound. Moreover, VerifyAbs terminates with a counterexample only if it deems the counterexample to be non-spurious, by simulating it on the concrete design $\mathcal{D}$. Therefore VerifyAbs is complete. □

Note that VerifyAbs performs abstraction in a manner similar to the abstraction performed by ATLAS as described in Section 5.3. The difference is in the way interpretation conditions are computed and refined. Thus, assuming that VerifyAbs terminates, the proof of Theorem 5.5 can be used to prove Theorem 6.1, so we include only a proof sketch above.

In order to guarantee termination of VerifyAbs, we must impose certain constraints on the learning algorithm LearnAbsConds. This is formalized in the theorem below.

Theorem 6.2. *Suppose that the learning algorithm* LearnAbsConds *satisfies the following properties:*

(i) If c_i *denotes the interpretation condition for an fblock learned in iteration* i *of the* VerifyAbs *loop, then* $\mathsf{c}_i \implies \mathsf{c}_{i+1}$ *and* $\mathsf{c}_i \neq \mathsf{c}_{i+1}$*;*

(ii) The trivial interpretation condition **true** *belongs to the hypothesis space of* LearnAbsConds*, and*

(iii) The hypothesis space of LearnAbsConds *is finite.*

Then, VerifyAbs *will terminate and return either* Valid *or a non-spurious counterexample.*

Proof. Consider an arbitrary fblock that is a candidate for function abstraction. Let the sequence of interpretation conditions generated in successive iterations of the VerifyAbs loop be $\mathsf{c}_0 = \textbf{false}, \mathsf{c}_1, \mathsf{c}_2, \ldots$. By condition (i), $\mathsf{c}_0 \implies \mathsf{c}_1 \implies \mathsf{c}_2 \implies \ldots$ where $\mathsf{c}_i \neq \mathsf{c}_{i+1}$. Since no two elements of the sequence are equal, and the hypothesis space is finite, no element of the sequence can repeat. Thus, the sequence (for any fblock) forms a finite chain of implications. Moreover, since **true** belongs to the hypothesis space, in the extreme case, VerifyAbs can generate in its final iteration the term-level design $\mathcal{T}$ identical to the original design $\mathcal{D}$, which will yield termination with either Valid or a non-spurious counterexample. □

In practice, the conditions (i)-(iii) stated above can be implemented on top of any learning procedure. The most straightforward way is to set an upper bound on the number of iterations that LearnAbsConds can be invoked, after which the interpretation condition is set to **true**. Another option is to set c_{i+1} to $\mathsf{c}_i \vee d_{i+1}$ where d_{i+1} is the condition learned in the $i+1$-th iteration. Yet another option is to keep a log of the interpretation conditions generated, and if an interpretation condition is generated for a second time, the abstraction procedure is terminated by setting the interpretation condition to **true**. Many other heuristics are possible; we leave an exploration of these to future work.

6.2 Conditional Function Abstraction

Procedure CondAbs $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{IC})$ is responsible for creating a term-level design $\mathcal{T}$ from the original word-level design $\mathcal{D}$, the set of equivalence classes to be abstracted $\mathcal{FS}_{\mathcal{A}}$, and the set of interpretation conditions $\mathcal{IC}$. Algorithm 5 outlines the conditional abstraction procedure.

CondAbs operates by iterating through the equivalence classes in $\mathcal{FS}_{\mathcal{A}}$. A fresh uninterpreted function symbol UF_j is created for each tuple of isomorphic output signals χ_j associated with equivalence class $\mathcal{F}_i \in \mathcal{FS}_{\mathcal{A}}$. Each output signal $v_{ij} \in \chi_j$ is conditionally abstracted with UF_j. Figure 6.2 illustrates how the output signals of an fblock are conditionally abstracted. The original word-level circuit is shown in Figure 6.2(a) and the conditionally abstracted version with interpretation condition c is shown in Figure 6.2(b).

Algorithm 4 Procedure VerifyAbs $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}})$: Top-level CAL verification procedure.

```
Input: Combined word-level design (miter): $\mathcal{D} := \langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\}\rangle$
Input: Equivalence classes of fblocks: $\mathcal{FS}_{\mathcal{A}} := \{\mathcal{F}_j \mid j = 1, \ldots, k\}$
Output: Verification result: $\mathit{Result} \in \{\mathsf{Valid}, \mathsf{CounterExample}\}$
 1: Set $\mathsf{c}_i$ = false for all $\mathsf{c}_i \in \mathcal{IC}$.
 2: while true do
 3:   $\mathcal{T}$ = CondAbs $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{IC})$
 4:   $\mathit{Result}$ = Verify $(\mathcal{T})$
 5:   if $\mathit{Result}$ = Valid then
 6:     return Valid.
 7:   else
 8:     Store counterexample in $\mathcal{CE}$.
 9:     if $\mathcal{CE}$ is spurious then
10:       $\mathcal{IC} \leftarrow$LearnAbsConds $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{CE})$
11:     else
12:       return CounterExample.
13:     end if
14:   end if
15: end while
```

Algorithm 5 Procedure CondAbs $(\mathcal{D}, \mathcal{FS}_{\mathcal{A}}, \mathcal{IC})$: Create term-level design $\mathcal{T}$ from word-level design $\mathcal{D}$, the set of fblocks being abstracted $\mathcal{FS}_{\mathcal{A}}$, and the set of interpretation conditions $\mathcal{IC}$.

```
Input: Combined word-level design (miter): $\mathcal{D} := \langle \mathcal{I}, \mathcal{O}, \{\mathcal{N}_i \mid i = 1, \ldots, N\}\rangle$
Input: Equivalence classes of fblocks: $\mathcal{FS}_{\mathcal{A}} := \{\mathcal{F}_j \mid j = 1, \ldots, k\}$
Input: Set of interpretation conditions: $\mathcal{IC}$
Output: Term-level design: $\mathcal{T}$
 1: for each equivalence class $\mathcal{F}_i \in \mathcal{FS}_{\mathcal{A}}$ do
 2:   Let $\chi_j$ denote the isomorphic output tuple associated with the j-th output signal in
      fblock $f_i \in \mathcal{F}_i$
 3:   for all $j \in 1, ..., k$ do
 4:     Create a fresh uninterpreted function symbol $UF_j$.
 5:     Let $(i_1, ..., i_k)$ denote the input symbols to fblock $f_i$.
 6:     for each output signal $v_{ij} \in \chi_j$ do
 7:       Let $\mathsf{c}_{v_{ij}} \in \mathcal{IC}$ denote the interpretation condition associated with $v_{ij}$
 8:       Replace the assignment $v_{ij} \leftarrow e$ in $f_i$ with the assignment
          $v_{ij} \leftarrow \mathit{ITE}(\mathsf{c}_{v_{ij}}, e, UF_j(i_1, ..., i_k))$
 9:     end for
10:   end for
11: end for
12: return Term-level design $\mathcal{T}$
```

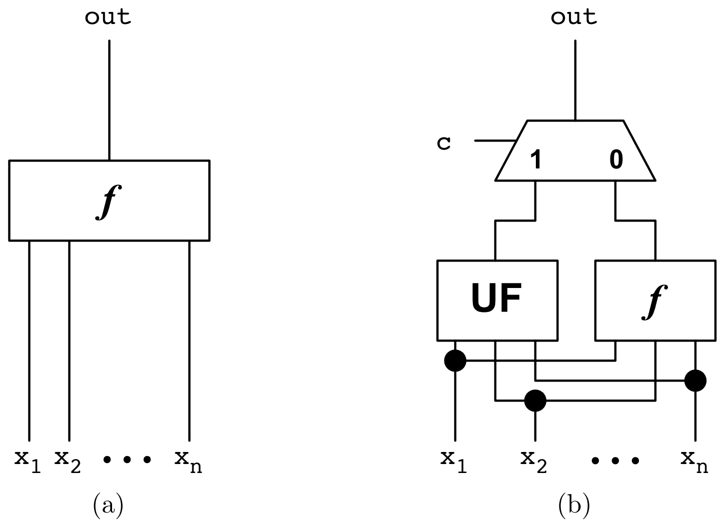


Figure 6.2: **Conditional abstraction** (a) Original word-level fblock f. (b) Conditionally abstracted version of f with interpretation condition c

6.3 Learning Conditional Abstractions

Spurious counterexamples arise due to imprecision introduced during abstraction. More specifically, when a spurious counterexample arises, it means that at least one fblock $f_i \in \mathcal{F}$ (where $\mathcal{F} \in \mathcal{FS_A}$) is being abstracted when it needs to be modeled precisely. In the context of our abstraction procedure VerifyAbs, if Verify $(\mathcal{T})$ returns a spurious counterexample $\mathcal{CE}$, then we must invoke the procedure LearnAbsConds $(\mathcal{D}, \mathcal{FS_A}, \mathcal{CE})$.

The LearnAbsConds procedure invokes a *decision tree learning* algorithm on traces generated by randomly mutating fblocks $f_i \in \mathcal{F}$ to generate "good" and "bad" traces. Good traces are those where the mutation does not lead to a property violation; the other traces are bad. The learning algorithm generates a classifier in the form of a decision tree to separate the good traces from the bad ones. The classifier is essentially a Boolean function over signals in the original word-level design. More information about decision tree learning can be found in Mitchell's textbook [56].

There are three main steps in the LearnAbsConds procedure:

1. Generate good and bad traces for the learning procedure;
2. Determine meaningful features that will help decision tree learning procedure compute high quality decision trees, and
3. Invoke a decision tree learning algorithm with the above features and traces.

The data input to the decision tree software is a set of tuples where one of the tuple elements is the target attribute and the remaining elements are features. In our context, a target attribute α is either Good or Bad. Our goal is to select features such that we can classify the set of all tuples where $\alpha =$ Bad based on the rules provided by the decision tree learner. It is very important to provide the decision tree learning software with quality input data and features, otherwise, the rules generated will not be of use. The data generation procedure is described in Section 6.4 and feature selection is described in Section 6.5.

6.4 Generating Data

In order to obtain high quality decision trees, we need to generate good and bad traces for the design being verified. Of course, whenever the procedure LearnAbsConds is called, we have a spurious counterexample stored in $\mathcal{CE}$. However, a single trace comprising only the counterexample is far from adequate, and will result in a trivial decision tree stating that α always equals Bad (i.e., the decision tree is simply **true**), since the learning algorithm with not have sufficient positive and negative examples to generalize from.

In order to produce a meaningful decision tree, we must provide the decision tree learner with both good and bad traces. We use random simulation to generate *witnesses* and *counterexamples* and describe these procedures in detail in Sections 6.4.1 and 6.4.2, respectively.

6.4.1 Generating Witnesses

We generate good traces, or *witnesses*, for the decision tree learner using a modified version of the random simulation procedure described in Section 5.1. Instead of simulating the abstract design when only a single fblock has been replaced with a random function, we replace all fblocks with their respective random functions at the same time and perform verification via simulation. Replacing all the fblocks to be abstracted with the respective random function is an important step, because the goal of generating witnesses is to generate many different witnesses. If the fblocks are not replaced with random functions, the witnesses will be similar and this will degrade the results of the decision tree learner.

After replacing each fblock to be abstracted with the corresponding random functions, we perform simulation by verification for N iterations, in the same manner that we did in Section 5.2. Note that N is chosen heuristically and we discuss typical values for N in Section 6.6. The initial state of design $\mathcal{D}$ is set randomly before each run of simulation. This usually results in simulation runs that pass. Recall that at this stage we only consider fblocks that produce failing runs in a small fraction of simulation runs. Now, instead of only logging the result of the simulation, we log the value of every signal in the design for every cycle of each passing simulation. It is up to the feature selection step, described in Section 6.5, to decide what signals are important. Let *Good* be the set of all witnesses produced in this step.

Note that in practice, if the set of features are known before this step, there is no need to log every signal. Instead, we log only the features that the decision tree learner will operate on. However, for small N, it is not much of a burden to log all signals, and doing this has the added benefit that in the case that the features are not sufficient, there is no need to rerun this process.

6.4.2 Generating Similar Counterexamples

There are several options that can be used to generate bad traces, or *counterexamples*. The counterexamples generated in this step are similar to the original spurious counterexample stored in $\mathcal{CE}$.

The first option is to use random simulation in a manner similar to that used to identify abstraction candidates. If more than one fblock has been abstracted, the counterexample $\mathcal{CE}$ can be the result of abstracting any individual fblock, or a combination of fblocks. Consider the situation where $\mathcal{CE}$ is the result of only a single abstraction. In this situation, we replace each fblock that has been abstracted with a random function in the word-level design, just as we did when identifying abstraction candidates. Next, verification via simulation is performed for N iterations, with a different random function for each iteration, just as in the procedure described in Section 5.2 (again, N is chosen heuristically). A main point of difference between generating similar counterexamples and generating witnesses is that in generating similar counterexamples, we set the initial state of design $\mathcal{D}$ to be consistent with the initial state in $\mathcal{CE}$, whereas we randomly set the initial state of design $\mathcal{D}$ when generating witnesses.

We log the values of every signal in the design for each failing simulation run. It is possible that none of the simulation runs fail, because the counterexample could be the result of abstracting a different fblock. In either case, we repeat this process for each fblock that is being abstracted. If random simulation for individual fblocks does not result in any failing simulation run, we must take into account combinations of fblocks, otherwise, we terminate the process of generating similar counterexamples. When it is necessary to take into account combinations of fblocks, there are several options. On one hand, trying every possible combination of fblocks could lead to an exponential number of iterations. On the other hand, trying the situation where every fblock is responsible for the counterexample could lead to interpretation conditions that are less precise.

Consider the case where only a small number of abstracted fblocks are responsible for the spurious counterexample and a large number of fblocks are being abstracted. By using all of the fblocks to determine the appropriate interpretation condition means that we will need to use the same interpretation condition for each fblock. Thus, we would be using an interpretation condition that only a small number of fblocks require, for a large number of fblocks. None of the examples we use in this work require interpretation conditions for more than a single fblock, so we leave the exploration of heuristics that determine how to choose interpretation conditions for combinations of fblocks for future work.

Alternative Approaches. The random simulation based approach is the most versatile, and likely the least computationally intensive, method of generating similar counterexamples. However, two alternative approaches exist. The first involves using the Verify procedure where the previously seen counterexamples are ruled out. The drawback to this approach is that using a formal verifier will likely be more computationally expensive than running a small number of simulations.

The second approach involves modifying the verification property and is only applicable in certain scenarios. For example, in bounded-model checking, it is possible to construct a property such that any counterexample violates the property in every stage of BMC, as opposed to only failing in a single stage. Not all equivalence or refinement checking problems lend themselves to this approach. However, when it is possible to use this option, it can reduce the number of traces required to create a quality decision tree. We give an example of this option in Section 6.6.3.

Regardless of how the bad traces are generated, we denote the set of all bad traces by *Bad*. Furthermore, we annotate each trace in *Bad* with the Bad attribute and each trace in *Good* with the Good attribute.

6.5 Choosing Features

The quality of the decision tree generated is highly dependent on the features used to generate the decision tree. We use two heuristics to identify features:

1. Include input signals to the fblock being abstracted.
2. Include signals encoding the "unit-of-work" being processed by the design, such as the instruction being executed.

Input signals. Suppose we wish to determine when fblock f must be interpreted. It is very likely that whether or not f must be interpreted is dependent on the inputs to f. So, if f has input signals $(i_1, i_2, ..., i_n)$ it is almost always the case that we would include the input arguments as features to the decision tree learner.

Unit-of-work signals. There are cases when the input arguments alone are not enough to generate a quality decision tree. In these cases, human insight can be provided by defining the unit-of-work being performed by the design. For example, in a microprocessor design, the unit-of-work is an instruction. Similarly, in a network-on-a-chip (NoC), the unit-of-work is a packet, where the relevant signals could include the source address, destination address, or possibly the type of packet being sent across the network. After signals corresponding to a unit-of-work are identified, it is easy to propagate this information to identify all signals directly derived from the original unit-of-work signals. For instance, in the case of a pipelined processor, the registers storing instructions in each stage of the pipeline are relevant signals to treat as features.

6.6 Experimental Results

We performed three case studies to evaluate CAL. Each of these case studies has also been verified using ATLAS. Additionally, each case study requires a non-trivial interpretation condition (i.e., an interpretation condition different from **false**). The first case study involves verifying the example shown in Figure 5.3. Next, we verify, via correspondence checking, two versions of the the Y86 microprocessor. Finally, we perform equivalence checking between a low-power multiplier and the corresponding non-power-aware version.

Our experiments were performed using the UCLID verification system [18]. MiniSAT [39, 40] was used as the SAT backend, while Boolector [22] was used as the SMT backend. Random simulation was performed using the Icarus Verilog [66] simulator. The interpretation conditions were learned using the C5.0 decision tree learner [61]. All experiments were run on a Linux workstation with 64-bit 3.0 GHz Xeon processors with 2 GB of RAM.

6.6.1 Processor Fragment

In this experiment, we perform equivalence checking between Design A and B shown in Figure 5.3. First, we initialize the designs to the same initial state and inject an arbitrary instruction. Then we check whether the designs are in the same state. The precise property that we wish to prove is that the ALU and PC outputs are the same for design A and B. Let $\texttt{out}_\texttt{A}$ and $\texttt{out}_\texttt{B}$ denote the ALU outputs and $\texttt{pc}_\texttt{A}$ and $\texttt{pc}_\texttt{B}$ denote the PC outputs for designs A and B, respectively. The property we prove is:

$$\texttt{out}_\texttt{A} = \texttt{out}_\texttt{B} \wedge \texttt{pc}_\texttt{A} = \texttt{pc}_\texttt{B}$$

Aside from the top-level modules, the design consists of only two modules, the instruction memory (IMEM) and the ALU. We do not consider the instruction memory for abstraction because we do not address automatic memory abstraction. The ALU passes the random simulation stage, so it is an abstraction candidate.

For this example, we have the benefit of knowing a priori the exact interpretation condition needed in order to precisely abstract the ALU module. A counterexample is generated during the first verification stage due to the unconditionally abstracted ALU. Due to the rather simple and contrived nature of this example, not many signals need to be considered as features for the decision tree learner. The features we use in this case are arguments to the ALU; the instruction and the data arguments. The interpretation condition learned from the trace data is $\texttt{op} = \mathsf{JMP}$ where $\texttt{op}$ is the top 4 bits of the instruction.

Interpretation	UCLID Runtime (sec)	
Condition	SAT	SMT
true	28.51	27.01
$\texttt{op} = \mathsf{JMP}$	**0.31**	**0.01**

Table 6.1: **Performance comparison for Processor Fragment.** UCLID runtime comparison for the processor fragment shown in Figure 5.3. The runtime associated with the model abstracted with CAL is shown in **bold**.

6.6.2 Y86 Processor

In this experiment, we verify two versions of the well-known Y86 processor model introduced by Bryant and O'Hallaron [30] and described in Section 5.5.4. The Y86 variants we consider in this section are NT and BTFNT. We focus on these versions in particular because they require non-trivial interpretation conditions (e.g., interpretation conditions not equal to **true** or **false**).

During the random simulation phase the ALU generates very few, if any, failures. In fact, in order to get a failure, it is usually necessary to simulate for 10,000–100,000 runs of correspondence checking; potentially several hundred thousand to a million individual cycles before obtaining an error. Of course we never run that many cycles of random simulation. Instead, we perform at most 500-1000 runs of correspondence checking and usually do not receive a violation. However, in the first iteration of VerifyAbs we obtain a spurious counterexample which requires the computation of non-trivial interpretation conditions. We use this counterexample to initialize the initial state of the processor models as discussed in Section 6.4.

Decision tree feature selection

In the case of both BTFNT and NT using only the arguments of the abstracted ALU is not sufficient to generate a useful decision tree. The ALU takes three arguments, the op-code *op* and two data arguments *a* and *b*. Closer inspection of the data provided to the decision tree learner reveals a problem. In almost every single case in both good and bad traces, the ALU *op* is equal to ALUADD and the *b* argument is equal to 0, in almost every cycle of correspondence checking. The underlying cause of this poor data stems from a perfectly reasonable design decision. Many of the Y86 instructions do not require the ALU to perform any operation. In these cases, the default values fed into the ALU are $op = \mathsf{ALUADD}$ and $b = 0$ and this condition holds in good and bad traces alike.

In this situation, the arguments to the ALU are not good features by themselves. Conceptually, the unit-of-work that we are performing in a pipelined processor is a sequence of instructions, specifically the instructions that are currently in the pipeline. When we include the instructions that are in the pipeline during the cycle in which the instruction is injected as described earlier in this section, we are able to obtain a much more high quality decision tree, or interpretation condition. In fact, the interpretation condition obtained when considering all instructions currently in the pipeline during the cycle in which the new instruction is injected is:

$$\mathsf{c} := \mathit{Instr}_E = \mathsf{JXX} \wedge \mathit{Instr}_M = \mathsf{RMMOV}$$

The interpretation condition c indicates that we need to interpret the ALU whenever there is a JUMP instruction (JXX) in the execute stage and there is a register-to-memory move in the memory stage. While this interpretation condition is an improvement over the original decision tree, it still leads to further spurious counterexamples. The reason is that we are now including too many features, some of which have no effect on whether the ALU needs to be interpreted. These additional features restrict the situations when the ALU is interpreted and this causes further spurious counterexamples. A logical step is to include as a feature only the instruction that is currently in the ALU. So, when we use the following features: Instr_E, op, a, and b we obtain the interpretation condition:

$$\mathsf{c}_{E,b} := \mathit{Instr}_E = \mathsf{JXX} \wedge b = 0$$

This is the best interpretation condition we can hope for. In fact, in previous attempts to manually abstract the ALU in the BTFNT version, we used:

$$\mathsf{c}_{Hand} := op = \mathsf{ALUADD} \wedge b = 0$$

When we compare the runtimes for verification of the Y86-BTFNT processor, we see that verifying BTFNT with the interpretation condition $\mathsf{c}_{E,b}$ outperforms the unabstracted version and the previously best known abstraction condition (c_{Hand}). Table 6.2 compares the UCLID runtimes for the Y86 BTFNT model with the different versions of the abstracted ALU.

Interpretation Condition	UCLID Runtime (sec)	
	SAT	SMT
true	> 1200	> 1200
c_{Hand}	133.03	105.34
$\mathsf{c}_{E,b}$	**101.10**	**65.52**

Table 6.2: **Performance comparison for Y86-BTFNT processor.** UCLID runtime comparison for Y86-BTFNT for different interpretation conditions. The runtime associated with the model abstracted with CAL is shown in **bold**.

Abstraction-refinement

The NT version of the Y86 processor requires an additional level of abstraction refinement. In general, requiring multiple iterations of abstraction refinement is not interesting by itself. However, it is interesting to see how the interpretation conditions change using this machine learning-based approach.

Attempting unconditional abstraction of the ALU in the NT version results in a spurious counterexample. The interpretation condition learned from the traces generated in this step is $\mathsf{c} := a = 0$. It is interesting that the same interpretation condition is generated regardless of whether we consider all of the instructions as features, or only the instruction in the same stage as the ALU. Not surprisingly, the second attempt at verification using the interpretation condition c results in another spurious counterexample. In this case, the interpretation condition learned is $\mathsf{c}_E := Instr_E = \mathsf{ALUADD}$, which states that we must interpret anytime an addition operation is present in the ALU. Similarly with the first iteration, the interpretation condition learned is the same regardless of whether use all of the instructions as features, or only the instruction in the execute stage. Verification is successful when c_E is used as the interpretation condition.

A performance comparison for the NT variant of the Y86 processor is shown in Table 6.3. Unlike the BTFNT case, the abstraction condition we learn for the NT model is not quite as precise as the previously best known interpretation condition, and the performance isn't as good. However, the runtimes for conditional abstraction, including the time spent in abstraction-refinement, are smaller than that of verifying the original word-level circuit. That is, the runtime when the interpretation condition is c_E is accounting for two runs of UCLID that produce a counterexample and an additional run when the property is proven

Valid. Note that the most precise abstraction condition is the same for both BTFNT and NT. The best performance on the NT version is obtained when the interpretation condition $\mathsf{c}_{BTFNT} := Instr_E = \mathsf{JXX} \wedge b = 0$ is used.

Interpretation Condition	UCLID Runtime (sec)	
	SAT	SMT
true	> 1200	> 1200
c_{Hand}	154.95	89.02
c_{E}	**191.34**	**187.64**
c_{BTFNT}	94.00	52.76

Table 6.3: **Performance comparison for Y86-NT processor.** UCLID runtime comparison for Y86-NT for different interpretation conditions. The runtime associated with the model abstracted with CAL is shown in **bold**.

The reason the interpretation condition for BTFNT differs from that of NT is because the root cause of the counterexamples are different. The counterexample generated for the BTFNT model arises because the branch target that would pass through the ALU unaltered, gets mangled when the ALU is abstracted. The counterexample generated for the NT model arises because the abstracted ALU incorrectly squashes a properly predicted branch.

6.6.3 Low-Power Multiplier

The next example we experimented with is a low-power multiplier design first introduced in [15]. Consider a design that has a multiplier which can be powered down when no multiply instructions are present to save power. Any time a multiply instruction is issued, the multiplier would have to be powered up in order to compute the multiplication. An optimization to this design is to send any multiply instructions in which at least one operand is a power of 2 to a shifter. Figure 6.3(b) illustrates this design. The `pow2` module takes inputs `a` and `b` and generates `cond` which is true if either operand is a power of 2. If one or more operand is a power of 2, `pow2` ensures that the proper shift amount is computed and sent to the appropriate shifter input. If neither operand is a power of two, `cond` is false, so the inputs to the shifter are don't cares. To safely use such a circuit, we must first verify that the optimized multiplier performs the same computation as a regular multiplier. We consider the unoptimized multiplier module for abstraction, because a) it is the only replicated fblock in the design and b) it passes the random simulation stage.

Note that in the implementation of the design in Figure 6.3(b) there are two signals `a_is_pow_2` and `b_is_pow_2` that are **true** when `a` or `b`, respectively, are powers of two. When performing unconditional abstraction on the miter constructed between the multiplier designs, a spurious counterexample is generated. This happens due to the fact that when either input argument is a power of two, then the output of the low-power multiplier is precise. In order for the verification to succeed, we must capture this behavior in an interpretation condition.

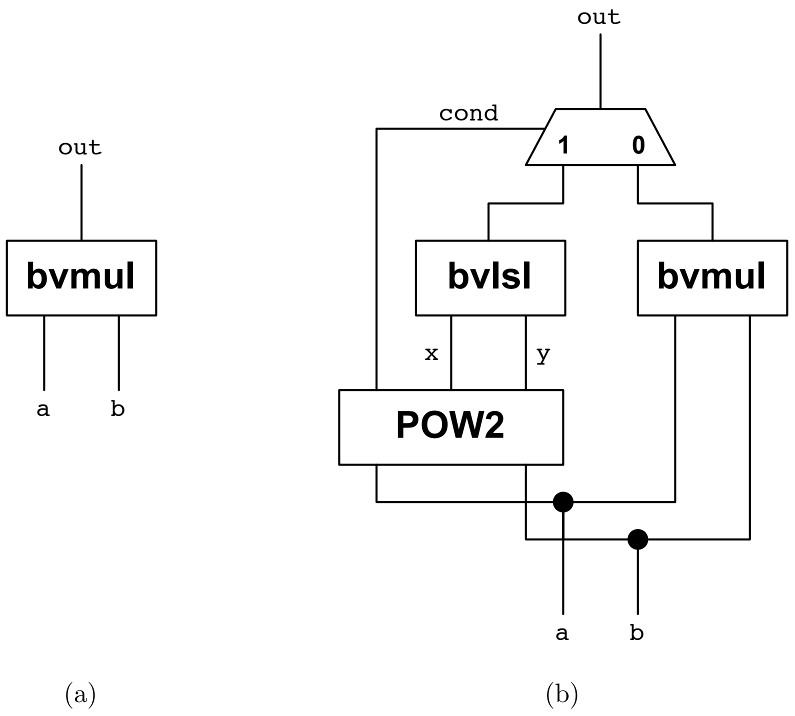


Figure 6.3: **Low-power design example.** (a) the original, word-level multiplier; (b) multiplier that uses a shifter when an operand is a power of 2.

It is possible to generate similar counterexamples using random simulation. However, in this case, it is also possible to use the alternative method of modifying the property. In order to do this, we generate multiple counterexamples by unrolling the circuit many times and constructing a property that is valid if and only if there is never a time frame in which the two circuits differ. A counterexample of this property will be a trace in which the two circuits have different results in every stage. We use this method to generate bad traces for the decision tree learner. To generate good traces, we simply randomly simulate the circuits for some number of cycles. The chance of one of the inputs being a power of two is so small that it is unlikely to occur in a small number of simulation runs. In addition to the arguments to the multiplier circuits, `a` and `b`, we consider `a_is_pow_2` and `b_is_pow_2` as features. The interpretation condition obtained due to this configuration of features and traces is $\mathtt{c}_{pow2} := \mathtt{a\_is\_pow\_2} \vee \mathtt{b\_is\_pow\_2}$

The performance comparison for the equivalence checking problem between a standard multiplier and the multiplier optimized for power savings is shown in Table 6.4. While the abstracted designs show a small speedup in most cases, it is by no means a performance victory. Instead, what this experiment shows is that we are able to learn interpretation conditions for another type of circuit and verification problem. Additionally, performing abstraction on this circuit doesn't cause a performance degradation.

BMC Depth	UCLID Runtime (sec)			
	SAT		SMT	
	No Abs	Abs	No Abs	Abs
1	2.81	**2.55**	1.27	**1.38**
2	12.56	**14.79**	2.80	**2.63**
5	67.43	**22.45**	8.23	**8.16**
10	216.75	**202.25**	21.18	**22.00**

Table 6.4: **Performance comparison for low-power multiplier.** UCLID runtime comparison for equivalence checking between multiplier and low-power multiplier. The runtime associated with the model abstracted with CAL is shown in **bold**.

6.6.4 Comparison with ATLAS

ATLAS and CAL compute the same interpretation conditions for the processor fragment described in Section 6.6.1 and the low-power multiplier described in Section 6.6.3. Thus, the only interesting comparison with regard to the interpretation conditions is for the Y86 design.

ATLAS is able to verify both BTFNT and NT Y86 versions with one caveat—the multiplication operator was removed from the ALU to create a more tractable verification problem. When multiplication is present inside the ALU, the ATLAS approach cannot verify BTFNT or NT in under 20 minutes. In the case where the multiplication operator is removed, the interpretation conditions generated by ATLAS for both BTFNT and NT are quite large, even though the procedure to generate the conditions is iterated very few times. Running this procedure for more iterations leads to exponential growth of the interpretation condition expressions. Upon closer inspection of the interpretation conditions, it turns out that the expressions simplify to **true** (i.e., no abstraction takes place). In this case, ATLAS actually takes longer to verify BTFNT as shown in [17]. This behavior highlights the main drawback of ATLAS. The static analysis procedure blindly takes into account the structure of the design, giving equal importance to every signal. In reality, bugs stem from very specific situations where only small fragments of the overall design contribute to the buggy behavior. This was the inspiration behind the using machine learning to compute interpretation conditions. Not only is CAL able to verify the BTFNT and NT Y86 versions when multiplication *is* included in the ALU, but it does so with an order of magnitude speedup over the unabstracted version.

6.6.5 Remarks

We have mainly focused thus far on the runtime taken by UCLID. The remaining runtime taken by the other components of the CAL procedure is, in comparison, negligible. First, the runtime of the decision tree learner is less than 0.1 seconds in every case. Second, the simulation time is quite small. For instance, simulating 1000 correspondence checking runs for the Y86 model takes less than 5 seconds. Whereas we are unable to verify the original word-level Y86 designs within 20 minutes, so the CAL runtime is negligible. Note that the simulation runtime for the low-power multiplier example is even smaller than that of the Y86 design. For instance, it takes less than 0.2 seconds to simulate the low-power multiplier for 1000 cycles. The number of good and bad traces required to produce a quality decision tree for the processor fragment example in Section 6.6.1 and the low-power multiplier example in Section 6.6.3 is 5 (10 total). For the Y86 examples, the number of good and bad traces was 50 (100 total). Thus, in every example, it takes only a fraction of a second to generate enough data for the machine learning algorithm to be able to produce useful results.

A key point to note is that while the entire CAL procedure can be automated, human insight can be invaluable in speeding up verification. For instance, if a designer or verification engineer could mark the most important signals in a design, we could give those signals priority when choosing features to give to the decision tree learner. In the context of the Y86 examples, if the designer would specify up front that the instruction code signals for each stage were important, we could fully automate the examples shown in this paper. Noticing that the instruction codes are important signals in a processor design does not require any leap of faith. It is obvious that they are important—they directly affect the operation of the design being verified!

6.7 Related Work

CAL shares many similarities with ATLAS. Thus, work related to ATLAS as discussed in Section 5.7 is relevant here as well; however, we refrain from duplicating it here.

ATLAS is the first technique to exploit the module structure specified by the designer by using random simulation to determine the module instantiations that are suitable for abstraction with uninterpreted functions. CAL uses this same technique to identify abstraction candidates. The main difference between CAL and ATLAS is the way in which interpretation conditions are computed. Instead of using static analysis to compute interpretation conditions, as described in Section 5.3, CAL uses a dynamic approach based on machine learning. The benefit of this machine learning based approach is that the conditions learned are actually causing spurious counterexamples. To the best of our knowledge, CAL is the first work to use machine learning to dynamically compute conditions under which it is precise to abstract. As is the case with ATLAS, the CAL approach could be combined with bit-width reduction techniques (e.g. [49, 13]) to perform combined function and data abstraction.

To our knowledge, Clarke, Gupta *et al.* [32, 45] were the first to use machine learning to compute abstractions for model checking. Our work is similar in spirit to theirs. One difference is that we generate term-level abstract models for SMT-based verification, whereas their work focuses on bit-level model checking and localization abstraction. Consequently, the learned concept is different: we learn Boolean interpretation conditions while they learn sets of variables to make visible. Additionally, our use of machine learning is more direct — e.g., while Clarke *et al.* [32] also use decision tree learning, they only indirectly use the learned decision tree (the make visible all variables branched upon in the tree), whereas we use the Boolean function corresponding to the entire tree as the learned interpretation condition.

6.8 Summary

In this chapter, we present CAL, an automatic function abstraction technique. As is the case with ATLAS, CAL takes a word-level RTL design and creates a conditionally abstracted term-level design. CAL relies on a combination of random simulation, machine learning, and counterexample-guided abstraction-refinement. The random simulation-based technique presented in Section 5.2 is used to identify abstraction candidates. CAL computes abstraction conditions by integrating machine learning and counterexample-guided abstraction-refinement. We have evaluated the effectiveness and efficiency of CAL on equivalence and refinement checking problems involving pipelined processors and low-power designs. Furthermore, we were able to learn abstraction conditions that were more precise than previously known hand-crafted abstraction conditions.

A binary distribution of CAL is unavailable at the time of this writing. However, the experimental data from which the results are based can be found at `http://uclid.eecs.berkeley.edu/cal`.

Chapter 7

Conclusions and Future Work

This chapter summarizes the main theoretical results and suggests avenues of future work.

7.1 Summary

Automatic abstraction can help increase the performance and capacity of formal verification tools. This thesis has presented automatic data and function abstraction techniques that have been used to successfully verify designs with realistic characteristics.

The data abstraction technique, V2UCL, presented in Chapter 4 exploits the small domain property of the logic of equality with uninterpreted functions. By re-encoding portions of a bit-vector design in EUF, we are able to reduce the bit-width of certain datapaths within the original design. We have presented experimental evidence that shows that the data abstraction technique presented herein leads to smaller, easier-to-verify models.

The notion of using random functions in place of uninterpreted functions in simulation is a novel idea. Identifying abstraction candidates using simulation while exploiting the module structure given by the designer is a new and useful technique. Automating the identification of abstraction candidates is crucial if automatic abstraction is to be used in industrial-quality tools.

The function abstraction techniques presented in Chapters 5 and 6 present two methods of computing conditions under which it is precise to abstract. The static analysis-based procedure, ATLAS, discussed in Chapter 5 can yield accurate interpretation conditions in some cases. Another technique, CAL, described in Chapter 6 is based on machine learning. We use machine learning to determine conditions that are shared amongst the spurious counterexamples. These conditions can then be used to compute abstraction conditions. We've shown that CAL can be used to compute high quality interpretation conditions leading to easier to verify models.

7.2 Future Work

In this section we present avenues for future work.

7.2.1 Combining Data and Function Abstraction

Data abstraction is often limited by bit-vector operators present in the datapaths being abstracted. Applying function abstraction to these operators could lead to further data abstraction. This type of abstraction is more fine grained than the abstraction procedures described in Chapters 5 and 6. Combining function and data abstraction presents new challenges in automatic abstraction. In the data abstraction technique presented in Chapter 4, portions of the circuit are abstracted to terms unconditionally. If this abstraction were to be partial, we would need to compute interpretation conditions similar in nature to the conditions we use for function abstraction, except this time, it will be activating or deactivating entire portions of a circuit, instead of just an uninterpreted function.

7.2.2 Constraining the Progress of Learning Algorithms

As discussed in Section 6.1, in order for CAL to terminate, constraints must be imposed on the progress of the learning algorithm. While the techniques we suggest to enforce termination are correct, there is room for improvement.

In first option, a bound is imposed on the number of iterations. Once the bound is reached, the interpretation conditions are set to **true**. This is the most straightforward technique, but it does not prevent the situation where the learning algorithm gets into a cycle. For example, the learning algorithm could alternate between learning c_1 and c_2. In this case, the bound will enforce termination, but only the first two iterations of refinment will be useful, the remaining iterations will not discover any new information.

The next option sets the interpretation condition c_{i+1} to $\mathsf{c}_i \vee \mathsf{d}_{i+1}$, where d_{i+1} is the condition learned on the $i+1$-th iteration. The idea behind this option is to avoid cycles by incorporating everything that has been learned into the current condition. This option can result in pathological behavior where an exponential number of refinement iterations are required before termination. In the worst case, the word-level design must be modeled precisely and all the time spent learning conditions would be wasted.

The last option we suggested in Section 6.1 is to keep a log of all the conditions learned. If any condition is repeated, set the condition to **true**. Recall that setting the condition to **true** means that the module in question is represented precisely. Consider the example discussed above where the learning algorithm cycles between learning c_1 and c_2. On the third iteration (i.e., the second time we learn c_1), we would set the condition to be **true**. However, it is possible that the actual condition is $\mathsf{c}_1 \vee \mathsf{c}_2$. Thus, we would give up, while being very close to learning the desired solution.

The ideal method of constraining the progress of the conditions being learned will find the balance between requiring few iterations of learning and finding a interpretation condition that is relatively precise. Precise in this context means that the interpretation condition evaluates to **true** only when necessary. The options listed above are limited in the sense that they are overly conservative or aggressive in the way in which they constrain progress. It is likely that a combination of the above heuristics can be more effective than any individual heuristic. One possible combination of the above heuristics is as follows. The learning algorithm can operate unconstrained, but if a particular condition is repeated, then take the disjunction of the current condition and every other condition that occured since the last occurance of the current condition. If no condition is repeated after some number of cycles, then either set the condition to **true** or take the disjunction of all conditions seen thus far.

It is also possible that many iterations of refinement are required do to a poor feature selection. Thus, being able to add or remove features on the fly could be helpful in learning conditions more efficiently. This leads into our final suggestion for possible directions of future work.

7.2.3 Automatic Feature Selection

One of the challenges in verifying the Y86 design was in selecting the features necessary to obtain a high quality decision tree. Thus, automatic methods for choosing the features to use in a decision tree learner are desirable. Techniques such as Bayesian multiple instance learning [62] can be used to identify a subset of features that are relevant. A similar technique could be incorporated into the the generation of witnesses and counterexamples. After logging every signal in the design, we could then apply, for example, a Bayesian multiple instance learning technique to identify what features will lead to a high quality decision tree.

Bibliography

[1] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[2] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 2009.

[3] W. Ackermann. *Solvable Cases of the Decision Problem.* North-Holland, Amsterdam, 1954.

[4] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of ASP-DAC*, pages 19–24, 2006.

[5] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. CEGAR-based formal hardware verification: A case study. Technical Report CSE-TR-531-07, University of Michigan, May 2007.

[6] Zaher S. Andraus and Karem A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41st Design Automation Conference*, pages 218–223, 2004.

[7] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[8] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[9] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[11] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish

Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[12] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

[13] Per Bjesse. A practical approach to word level model checking of industrial netlists. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 446–458, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.

[15] Bryan Brady. Low-power verification with term-level abstraction. In *Proceedings of TECHCON 2010*, September 2010.

[16] Bryan Brady, Randal Bryant, and Sanjit A. Seshia. Abstracting RTL designs to the term level. Technical Report UCB/EECS-2008-136, EECS Department, University of California, Berkeley, Oct 2008.

[17] Bryan Brady, Randal E. Bryant, Sanjit A. Seshia, and John W. O'Leary. ATLAS: automatic term-level abstraction of RTL designs. In *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2010. To appear.

[18] Bryan Brady and Sanjit A. Seshia. The UCLID Page. http://uclid.eecs.berkeley.edu.

[19] Bryan Brady and Sanjit A. Seshia. Learning conditional abstractions. Technical Report UCB/EECS-2011-24, EECS Department, University of California, Berkeley, Apr 2011.

[20] Bryan Brady, Sanjit A. Seshia, Shuvendu K. Lahiri, and Randal E. Bryant. *A User's Guide to UCLID Version 3.0*, October 2008.

[21] Robert Brummayer. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. PhD thesis, Johannes Kepler University of Linz, 2009.

[22] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT Solver. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.

[25] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[26] Randal E. Bryant. Term-level verification of a pipelined CISC microprocessor. Technical Report CMU-CS-05-195, Computer Science Department, Carnegie Mellon University, 2005.

[27] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[28] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. An abstraction-based decision procedure for bit-vector arithmetic. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(2):95–104, 2009.

[29] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, LNCS 2404, pages 78–92, July 2002.

[30] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2002. Website: http://csapp.cs.cmu.edu.

[31] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV)*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.

[32] Edmund M. Clarke, Anubhav Gupta, James H. Kukula, and Ofer Strichman. SAT based abstraction-refinement using ilp and machine learning techniques. In *Proc. Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279, 2002.

[33] SMT Competition. http://www.smtcomp.org/.

[34] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, July 1962.

[35] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, July 1960.

[36] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[37] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[38] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*, pages 61–75, June 2005.

[39] Niklas Eén and Niklas Sörensson. The MiniSAT Page. http://minisat.se.

[40] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

[41] Cindy Eisner, Amir Nahir, and Karen Yorav. Functional verification of power gated designs by compositional reasoning. In *CAV*, pages 433–445, 2008.

[42] Jeffery S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality.* PhD thesis, University of California, Berkeley, 2002.

[43] Anders Franzén. *Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT.* PhD thesis, University of Trento, 2010.

[44] Alberto Griggio. *An Effective SMT Engine for Formal Verification.* PhD thesis, University of Trento, 2009.

[45] Anubhav Gupta and Edmund M. Clarke. Reconsidering CEGAR: Learning good abstractions without refinement. In *Proc. International Conference on Computer Design (ICCD)*, pages 591–598, 2005.

[46] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[47] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 129–144, 2010.

[48] Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.

[49] Peer Johannesen. BOOSTER: Speeding up RTL property checking of digital designs through word-level abstraction. In *Computer Aided Verification*, 2001.

[50] Peer Johannesen. *Speeding up hardware verification by automated data path scaling.* PhD thesis, Christian-Albrechts-Universität zu Kiel, 2002.

[51] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[52] Shuvendu K. Lahiri and Randal E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 341–354, 2003.

[53] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID decision procedure. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV)*, pages 475–478, July 2004.

[54] P. Manolios and S. K. Srinivasan. Refinement maps for efficient verification of processor models. In *Design, Automation, and Test in Europe (DATE)*, pages 1304–1309, 2005.

[55] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[56] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[57] 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*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[58] Opencores.org. USB controller. http://www.opencores.org/project,usb.

[59] Li-Shiuan Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks.* PhD thesis, Stanford University, August 2001.

[60] Amir Pnueli, Yoav Rodeh, Ofer Strichmann, and Michael Siegel. The small model property: how small can it be? *Information and Computation*, 178:279–293, October 2002.

[61] Ross Quinlan. Rulequest research. http://www.rulequest.com.

[62] Vikas C. Raykar, Balaji Krishnapuram, Jinbo Bi, Murat Dundar, and R. Bharat Rao. Bayesian multiple instance learning: automatic feature selection and inductive transfer. In *in Proceedings of the 25th International Conference on Machine learning*, pages 808–815. ACM, 2008.

[63] Sanjit A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification.* PhD thesis, Carnegie Mellon University, May 2005.

[64] Fabio Somenzi. CUDD: CU Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/.

[65] Bruce Wile, John Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, 2005.

[66] Stephen Williams. Icarus Verilog. `http://www.icarus.com/eda/verilog`.

[67] Lintao Zhang. *Searching for Truth: Techniques for Satisfiability of Boolean Formulas*. PhD thesis, Princeton University, 2003.