Unbounded Scalable Hardware Verification

by

Suho Lee

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2016

Doctoral Committee:

Professor Karem A. Sakallah, Chair
Professor Robert Dick
Professor John P. Hayes
Professor Thomas F. Wenisch

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF ABBREVIATIONS

ACEXT abstract counterexample trace

Averroes abstract verification of reachability of electronic systems

BDD binary decision diagram

BMC bounded model checking

CAL conditional abstraction through learning

CEGAR counterexample-guided abstraction and refinement

CEXT counterexample trace

CLU counter arithmetic with lambda expressions and uninterpreted functions

CTI counterexample to induction

DAG directed acyclic graph

EMC extended model checker

EUF equality with uninterpreted functions

IC3 incremental construction of inductive clauses for indubitable correctness

MUS minimal unsatisfiable subset

PDR property-directed reachability

RAM random access memory

SAT Boolean satisfiability problem

SMT satisfiability modulo theories

UCLID uninterpreted functions, counter arithmetic and lambda expressions for infinite domains

UF uninterpreted function

UP uninterpreted predicate

ABSTRACT

Unbounded Scalable Hardware Verification

by

Suho Lee

Model checking is a formal verification method that has been successfully applied to real-world hardware and software designs. Model checking tools, however, encounter the so-called state-explosion problem, since the size of the state spaces of such designs is exponential in the number of their state elements. In this thesis, we address this problem by exploiting the power of two complementary approaches: (a) counterexample-guided abstraction and refinement (CEGAR) of the design's datapath; and (b) the recently-introduced incremental induction algorithms for approximate reachability. These approaches are well-suited for the verification of control-centric properties in hardware designs consisting of wide datapaths and complex control logic. They also handle most complex design errors in typical hardware designs. Datapath abstraction prunes irrelevant bit-level details of datapath elements, thus greatly reducing the size of the state space that must be analyzed and allowing the verification to be focused on the control logic, where most errors originate. The induction-based approximate reachability algorithms offer the potential of significantly reducing the number of iterations needed to prove/disprove given properties by avoiding the implicit or explicit enumeration of reachable states. Our implemen-

tation of this verification framework, which we call the Averroes system, extends the approximate reachability algorithms at the bit level to first-order logic with equality and uninterpreted functions. To facilitate this extension, we formally define the solution space and state space of the abstract transition system produced by datapath abstraction. In addition, we develop an efficient way to represent sets of abstract solutions involving present- and next-states and a systematic way to project such solutions onto the space of just the present-state variables. To further increase the scalability of the Averroes verification system, we introduce the notion of structural abstraction, which extends datapath abstraction with two optimizations for better classification of state variables as either datapath or control, and with efficient memory abstraction techniques. We demonstrate the scalability of this approach by showing that Averroes significantly outperforms bit-level verification on a number of industrial benchmarks.

CHAPTER I

Introduction

The goal of our research is to explore a scalable formal verification methodology for complex hardware systems. To achieve this goal, we exploit the power of two complementary approaches: counterexample-guided abstraction and refinement (CEGAR) [1] and the IC3 [2] and PDR [3] approximate reachability algorithms[1]. This framework is suitable for the verification of control-centric properties in hardware designs that contain wide datapath elements and complex control logic. This covers a wide range of hardware designs from general-purpose microprocessors to special-purpose embedded controllers and accelerators. In addition, most complicated properties in hardware designs are control-centric because design errors typically reside in control logic. In fact, the correctness of datapath components can usually be verified independently.

Datapath abstraction is especially effective for the verification of control-centric properties in hardware designs including relatively small amounts of control state variables and orders-of-magnitude larger numbers of datapath state variables. It prunes irrelevant bit-level details in the large state space of datapath elements and focuses on the much smaller state space of control logic. Our framework applies the approximate reachability algorithms to this much smaller state space and can be

[1]IC3 stands for incremental construction of inductive clauses for indubitable correctness, and PDR stands for property-directed reachability

significantly more scalable than verification at the bit level.

Our implementation of this verification framework, which we call the Averroes system[2] for sequential verification [4], is premised on the conjecture that the complexity of sequential verification can be reduced significantly by a) abstracting away irrelevant datapath "state" that basically clutters reachability computation without providing any useful guidance for its convergence, and b) performing approximate reachability on this abstracted state space. The approach can be viewed as a "layering" of two CEGAR loops: an inner loop that performs approximate reachability on the datapath-abstracted state space, and an outer datapath refinement loop that tightens the abstraction based on the spurious counterexamples generated by the inner loop. The empirical evaluation of this approach shows that it significantly outperforms bit-level verification on a set of industrial RTL benchmarks and suggests that the combination of datapath abstraction and approximate reachability makes it possible to perform automatic unbounded scalable verification on real-world industrial benchmarks.

1.1 Preliminaries

In this section, we provide some necessary definitions and notations. We follow common definitions and notations that are widely used in the literature with one exception: we define the notions of *abstract state* and *abstract state space* for the transition system model obtained by abstracting away datapath components and signals.

1.1.1 Partition

A *set* $A = \{a_1, a_2, \cdots, a_n\}$ is a collection of distinct elements $a_1, a_2, \cdots, a_n$, and a set A is a *subset* of a set B if and only if every element in A is also an element of

[2]Averroes stands for abstract verification of reachability of electronic systems.

B. A *partition* of a set A is a grouping of the elements in A into non-empty disjoint subsets so that every element in A is in one of the subsets. The subsets of a partition are called the *cells* or *blocks* of the partition. A partition is represented by a set of its cells like $\{\{a_1\}, \{a_2, a_3\}, \cdots, \{a_n\}\}$. The *nth Bell number* [5], B_n, is the total number of partitions we can create from a set of n elements.

1.1.2 Directed Graph

A *directed graph* is a tuple $< V, E >$ where V is a set of *vertices* or *nodes* and E is a set of ordered pairs of vertices, called *edges* or *arrows*. For each edge $(u, v) \in E$, u is a *direct predecessor* of v, and v is a *direct successor* of u. If there exists a path from x to y for any $x, y \in V$, x is a *predecessor* of y, and y is a *successor* of x. A *source* is a vertex that has no predecessors, and a *sink* is a vertex that has no successors. A directed graph that has no directed cycles is called a *directed acyclic graph* (DAG).

1.1.3 SAT and SMT

A Boolean domain B is a set that consists of exactly two elements, $true$ and $false$. A *Boolean function* with m inputs is a function of the form $f : B^m \rightarrow B$. A *literal* is the input variable of a Boolean function or its negation. A *cube* is a conjunction of literals, and a *clause* is a disjunction of literals. The *conjunctive normal form* (CNF) formula is a conjunction of clauses, and it represents a Boolean function. A Boolean function is *satisfiable* if and only if there exists a solution, a Boolean assignment to its inputs, that makes the formula evaluate to $true$. The *Boolean satisfiability problem*, often abbreviated as *SAT*, is the problem of determining whether or not a Boolean formula is satisfiable. A *SAT solver*, such as GRASP [6], Chaff [7], or MiniSat [8], solves the problem and returns a solution if the Boolean formula is satisfiable. If the formula is unsatisfiable, a *minimal unsatisfiable subset* (MUS) can be derived. An MUS is a subset of the clauses of a CNF formula such that (1) a conjunction of

```
formula ::= ⊥ | ⊤ | literal
      | ¬ formula
      | propositional-connective (formula+)

propositional-connective ::= ∧ | ∨ | → | ↔

literal ::= propositional-variable | ¬ propositional-variable
```

Figure 1.1: Abstract Syntax of Propositional Logic

the MUS is still unsatisfiable, and (2) it becomes satisfiable if any of the clauses are eliminated.

A *proposition* is a declarative sentence that is either *true* or $false$. *Propositional logic* is a mathematical model for reasoning about the truth or falsehood of propositions, and a *propositional logic formula* is defined by the abstract syntax in Fig. 1.1. *First-order logic* (FOL) extends propositional logic with quantifiers and non-logical symbols such as function and predicate symbols for more expressiveness (a more detailed description can be found in [9]). A *satisfiability modulo theories* (SMT) *formula* [10] is a logical formula in first-order logic where the interpretation of some function and predicate symbols is constrained by background theories such as Linear Arithmetic, bit vectors, and so on. An SMT formula is satisfiable if there exists a solution, which is an interpretation for the variables, function symbols, and predicate symbols that makes the formula evaluate to *true*. Analogous to SAT, an *SMT problem* is the problem of determining whether or not an SMT formula is satisfiable, and an *SMT solver*, such as Z3 [11] or Yices [12], solves the problem and returns a solution if the formula is satisfiable.

1.1.4 Sequential Circuits

The *combinational circuit* is a circuit whose output depends on only the present values of its inputs. The circuit is used to implement a Boolean function. A *sequen-*

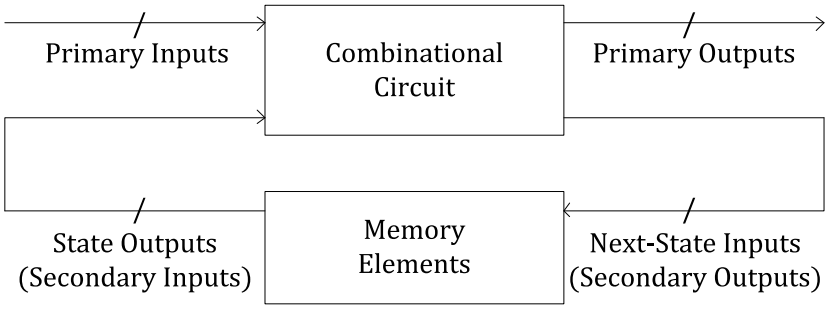


Figure 1.2: Block Diagram of a Sequential Circuit

tial circuit is a circuit whose output depends on the present values of its inputs as well as the history of the inputs. The history of the inputs is captured by *memory elements* storing binary information that defines the current state of the system. As shown in Fig. 1.2, a sequential circuit consists of a combinational circuit and memory elements. The inputs and outputs of the circuit are called *primary inputs* and *primary outputs*, and the inputs and outputs of the memory element are called *next-state inputs* (or *secondary outputs*) and *state outputs* (or *secondary inputs*) respectively. A combinational circuit computing the input of a memory element from the primary and secondary inputs is called the *next-state circuit* of that memory element. The next-state circuit basically computes next state as a function of current state and inputs. We assume that a sequential circuit is described at the *register transfer level* (RTL). The RTL description describes a circuit in terms of the flow of data from inputs or registers to outputs or registers through combinational logic blocks, and it can be written in hardware description languages (HDLs) such as Verilog-HDL [13] and VHDL [14]. An RTL HDL represents combinational logic blocks compactly using the following operators that can be classified into ten types:

Arithmetic Operators	+ - *
Relational Operators	< > <= >=
Equality Operators	= !=
Logical Operators	! && ||
Bit-wise Operators	~ & | ^ ~^
Reduction Operators	& ~& | ~| ^ ~^
Shift Operators	<< >> <<< >>>
Conditional Operators	?: (≡ ite)
Concatenation Operator	{}
Extraction operator	[]

This classification is based on the Verilog 2005 language reference manual [13][3].

In the remainder of this section, we provide a mathematical model of a sequential circuit. We consider a sequential circuit with n registers and $m > n$ combinational blocks. The circuit variables are classified as follows:

- Independent:
  - Primary Inputs: $\mathbf{y} = \langle y_1, \cdots, y_k \rangle$
  - Present-State Variables: $\mathbf{x} = \langle x_1, \cdots, x_n \rangle$
- Dependent:
  - Combinational Block Outputs: $\mathbf{w} = \langle w_1, \cdots, w_n, w_{n+1}, \cdots, w_m \rangle$
  - Next-State Variables: $\mathbf{x}^+ = \langle x_1^+, \cdots, x_n^+ \rangle$

The bit width of a variable $v \in \{\mathbf{w}, \mathbf{x}, \mathbf{y}\}$ is denoted $|v|$. v is a single-bit if $|v| = 1$; otherwise v is a bit vector. Each of the m combinational blocks represents a (vector) combinational function:

$$w_i = f_i(\mathbf{w}, \mathbf{x}, \mathbf{y}), \ i = 1, \cdots, m$$

[3]Verilog refers to Extraction operators as bit-selects or part-selects depending on the bit width of their outputs. Non-synthesizable operators and the replication operator ($\{n\{\}\}$), which can be replaced by a concatenation operator, are excluded.

We assume the outputs of the first n combinational blocks are connected to the inputs of the n registers yielding the next-state equations:

$$x_i^+ = w_i = f_i\left(\mathbf{w}, \mathbf{x}, \mathbf{y}\right), \; i = 1, \cdots, n$$

The support of a function $f_i\left(\mathbf{w}, \mathbf{x}, \mathbf{y}\right)$, denoted as $\text{supp}(f_i)$, are those variables $v \in \{\mathbf{w}, \mathbf{x}, \mathbf{y}\}$ that f_i depends on *directly*. They are also referred to as the function's *local* variables.

The combinational part of the sequential circuit can be viewed as a directed acyclic graph G whose vertices correspond to the set of variables $v \in \{\mathbf{w}, \mathbf{x}, \mathbf{y}\}$ and whose edges are defined by:

$$E = \bigcup_{i=1}^{m} \{\left(v, w_i\right) | \, v \in \text{supp}\left(f_i\right)\}$$

A path exists between vertex u and vertex w_i if there is a sequence of vertices $v_1, v_2, \cdots, v_l$ such that $(u, v_1) \in E$, $(v_l, w_i) \in E$, and $(v_j, v_{j+1}) \in E$ for $j = 1, \cdots, l-1$. This is captured by the predicate $\text{path}(u, w_i)$ which evaluates to true if there is a path between u and w_i and to false otherwise. The cone-of-influence of vertex w_i, denoted $\text{COI}(w_i)$, is the set of vertices defined by:

$$\text{COI}(w_i) = \{w_i\} \cup \{u \in \{\mathbf{w}, \mathbf{x}, \mathbf{y}\} \, | \, \text{path}(u, w_i)\}$$

The arguments of function f_i (as opposed to its support) are those independent variables on which it (indirectly) depends:

$$\arg\left(f_i\right) = \{\, u \in \{\mathbf{x}, \mathbf{y}\} | \, \text{path}\left(u, w_i\right)\}$$

They are also referred to as the function's *global* variables. We will refer to $f_i(\mathbf{w}, \mathbf{x}, \mathbf{y})$

as the *local* function of w_i. The *global* function of w_i is now defined by:

$$w_i = \bigwedge_{j \in \mathrm{COI}(w_i)} \left(w_j = f_j\left(\mathbf{w}, \mathbf{x}, \mathbf{y}\right)\right)$$

By repeatedly replacing occurences of (dependent) variables in the above expression with their local functions, w_i can always be expressed as a nested expression in terms of $\arg(f_i)$.

Given a predicate $Q(\mathbf{w}, \mathbf{x}, \mathbf{y})$ on current state and inputs, its evaluation on next state and inputs is accomplished by a syntactic substitution:

$$Q\left(\mathbf{w}^+, \mathbf{x}^+, \mathbf{y}^+\right) = Q\left(\mathbf{w}^+ \backslash \mathbf{w}, \mathbf{x}^+ \backslash \mathbf{x}, \mathbf{y}^+ \backslash \mathbf{y}\right)$$

The transition relation for the circuit can now be expressed in (generalized) conjunctive nomal form as follows:

$$T\left(\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{x}^+\right) = \bigwedge_{i=1}^{m} \left(w_i = f_i\left(\mathbf{w}, \mathbf{x}, \mathbf{y}\right)\right) \wedge \bigwedge_{i=1}^{n} \left(x_i^+ = w_i\right)$$

1.1.5 Finite Transition Systems

Our concern is to determine if a given sequential circuit satisfies a specified safety property. We model a sequential circuit as a *finite transition system*. A finite transition system is a tuple $< \mathbf{x}, \mathbf{y}, T, I >$ where $\mathbf{x}$ is a finite set of present-state variables, $\mathbf{y}$ is a finite set of primary inputs, $T\left(\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{x}^+\right)$ is a transition relation as defined in Section 1.1.4, and $I(\mathbf{x})$ is an initial condition that represents a finite set of initial states. In this finite transition system, we verify a *safety property*, which asserts that something bad will not happen (a formal definition can be found in [15]). The safety property, P, can be viewed as the output of the m^{th} (single-bit) combinational block

allowing it to be expressed as follows:

$$P(\mathbf{w}, \mathbf{x}, \mathbf{y}) = \bigwedge_{j \in \mathrm{COI}(w_m)} (w_j = f_j(\mathbf{w}, \mathbf{x}, \mathbf{y}))$$

A *state* in a transition system is identified as a Boolean assignment to state variables. We refer to a state at the bit level as a *concrete state* to distinguish it from the notion of an *abstract state* to be defined in Section 1.1.6. We define a *state literal* as a state variable or its negation, a *state cube* as a conjunction of state literals, a *state minterm* as a state cube involving all state variables, and a *state clause* as the negation of a state cube. Informally, we will refer to the states that satisfy (resp. violate) P as good (resp. bad or error) states. We denote by $R(\mathbf{x})$ the system's set of reachable states, i.e., those states that can be reached from I in one or more transitions. A *trace* Π is a state sequence $\langle s_0(\mathbf{x}), s_1(\mathbf{x}), \cdots, s_{k-1}(\mathbf{x})\rangle$ such that each s_i is a set of states, $s_0(\mathbf{x}) \in I$, and $s_i(\mathbf{x}) \wedge T \rightarrow s_{i+1}(\mathbf{x}^+)$ holds for $0 \leq i \leq k-2$. The length of a trace with k state sets is $k-1$. An empty trace is one whose state sequence (as a set) is empty; its length is undefined. The *sequential depth* [16] of a finite transition system is the maximum length of a trace that visits each state only once. Obviously, the maximum sequential depth of a finite transition system is 2^n where n is the number of state bits in the system. Our interest is to determine if all reachable states of a given system satisfy P. If so, P is an *invariant* of the system. Otherwise, there exists a trace whose last states violate P. Such a trace is called a *counterexample trace* (CEXT). The verification task can now be stated as follows: prove that all states in R are good or derive a counterexample trace that starts in I and ends in $\neg P$.

One way of determining if P is an invariant is to examine every possible trace in the system. However, this is not scalable because the number of possible traces is exponential in the number of state bits. *Induction* is a more scalable way to achieve

the goal [17]. P is *inductive* if

- **base case**: the initial states are good: $I \rightarrow P$.
- **inductive step**: P is *closed under* T:

  $P \wedge T \rightarrow P^+$, where P^+ is shorthand for $P(\mathbf{w}^+, \mathbf{x}^+, \mathbf{y}^+)$

In the above expressions, a formula M *implies* a formula N or $M \rightarrow N$ if every solution of M makes N evaluate to *true*. $M \rightarrow N$ holds if and only if $M \wedge \neg N$ is unsatisfiable. When the induction step check, $P \wedge T \wedge \neg P^+$, is satisfiable, a solution returned by a solver is a *counterexample to induction* (CTI). We call the induction step check a CTI check. In many cases where P is an invariant, P is often not inductive. Then, we can verify the property by deriving a *strengthening assertion*, $A(\mathbf{x})$, such that $A \wedge P$ is inductive [18]. If such a strengthening assertion exists, we can conclude that P is an invariant. $P \wedge A$ is called an *inductive invariant*.

1.1.6 Datapath Abstraction

Datapath components in hardware designs are regular, so they can be easily verified in isolation using customized algorithms. Therefore, we can abstract datapath components by assuming they are correct. This type of abstraction, called datapath abstraction, yields a much smaller system that can be more easily handled by verification tools. One way to abstract the datapath elements of a design is to replace them with uninterpreted terms and predicates [19]. The logic of equality with uninterpreted functions (EUF) proposed in [20] provides a very simple but powerful way to represent the abstracted design. Fig. 1.3 is the abstract syntax of EUF logic. During datapath abstraction, we replace a datapath operator with an uninterpreted function (UF) or an uninterpreted predicate (UP) depending on the bit width of its output (uninterpreted predicate if the output bit width is one, and uninterpreted function otherwise).

```
formula ::= ite ( formula, formula, formula )
      | propositional-connective (formula+)
      | literal | true | false

function ::= function-symbol (argument*)

predicate ::= predicate-symbol (argument*)

argument ::= formula | term

term ::= ite ( formula, term, term )
      | function
      | term-variable

literal ::= propositional-variable
      | ¬ propositional-variable
      | predicate
      | ¬ predicate
      | (term = term)
      | (term ≠ term)

propositional-connective ::= ¬ | ∧ | ∨ | → | ↔
```

Figure 1.3: Abstract Syntax of EUF Logic

The arity of a function or predicate is the number of its arguments. For example, a zero-arity function is a function with no arguments. In the abstract syntax, a term variable or an uninterpreted term is a zero-arity uninterpreted function, and a propositional variable is a zero-arity uninterpreted predicate. We classify term- and propositional variables into state variables, input variables, and constants according to their bit-level counterparts. We distinguish the abstract-level state and input variables from their bit-level counterparts by putting a circumflex accent over the symbols like $\hat{x}$. We denote the uninterpreted term of a constant as follows:

K{associated value}_{bit width of the constant}

For example, an abstract version of a three-bit constant 1 is denoted by $K1\_3$. We often denote the maximum value as MAX, so $KMAX\_3$ represents an abstract version of a three-bit constant 7. In this thesis, we omit the suffixes representing bit widths

Type	Symbol	Prefix
Arithmetic	-	Minus
	+	Add
	-	Sub
	*	Mult
	/	Div
	%	Mod
Relational	>	Gt
	<	Lt
	>=	GtEq
	<=	LtEq
Bit-wise	&	BitWiseAnd
	|	BitWiseOr
	~	BitWiseNot
	^	BitWiseXor
	~^	BitWiseXNor
	~|	BitWiseNor
	~&	BitWiseNand
Reduction	&	ReductionAnd
	|	RedcutionOr
	^	ReductionXor
	~^	ReductionXNor
	~&	ReductionNand
	~|	ReductionNor
Shift	<<<	AShiftL
	>>>	AShiftR
	<<	ShiftL
	>>	ShiftR
Concatenation	{ }	Concat
Extraction	[ ]	Ex

Figure 1.4: Prefix of Datapath Operator

for the sake of simplicity except where confusion could result.

A function symbol in the abstract syntax is the label of an uninterpreted function, and a predicate symbol is that of an uninterpreted predicate. The notation for function and predicate symbols for all datapath operators except extraction operators follow the pattern below:

{Prefix}_{bit width of the first input}_$\cdots$_{bit width of the last input}_{bit width of the output}

Fig. 1.4 lists the prefix of each datapath operator. For example, the function symbol

of a concatenation operator concatenating three 8-bit signals is $Concat\_8\_8\_8\_24$. In the case of extraction operators, we use a slightly different naming pattern:

Ex_{index of the most significant bit}_{index of the least significant bit}_{bit width of the input}

For example, the function symbol of an extraction operator extracting the first four bits from an 8-bit signal is $Ex\_3\_0\_8$

Datapath abstraction transforms the bit-level representations of T, I, and P to formulas in EUF which can be analyzed using an SMT solver. The solution space of an EUF formula is the Cartesian product of bit assignments to every predicate and literal and a partition of the set of terms in the formula, so the number of possible solutions is the product of the nth Bell number B_n and 2^m where n is the number of terms and m is the number of predicates and literals.

At the abstract- or EUF-level, an *abstract state* is identified as a Boolean assignment to single-bit state variables and uninterpreted predicates involving state variables and constants, and a partition among the terms involving state variables and constants. An abstract state can map to zero, one, or multiple concrete states. If it maps to zero concrete states, we refer to the abstract state as an *infeasible abstract state*. We refer to the other abstract states as *feasible abstract states*. Similarly, a transition between two abstract states is *feasible* if a bit-level counterpart exists. Otherwise, it is *infeasible*.

We define a *state term* as an uninterpreted term that involves state variables and possibly constants, a *state predicate* as an uninterpreted predicate of state terms, and a state literal as 1) a single-bit state variable or its negation, 2) a state predicate or its negation, or 3) an equality or dis-equality between state terms. The definitions of a *state cube* and *state clause* are the same as the ones at the bit level. A *state minterm* is a state cube involving all current-state variables, and we can derive this by collecting the state literals of all current state variables in a solution.

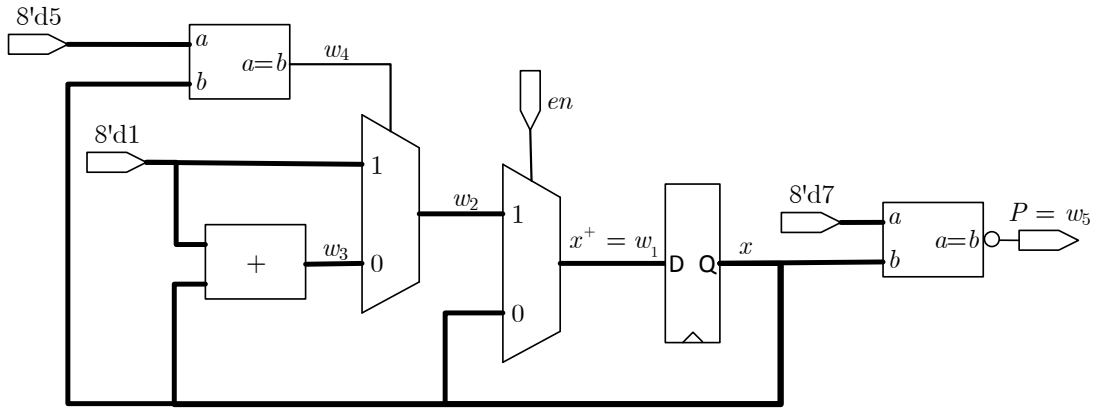


Figure 1.5: Circuit Representation of a Simple Sequential Circuit

1.1.7 Abstract State Space

Fig. 1.5 shows the circuit of an example sequential circuit that serves to visualize the abstract state space. The circuit is basically an 8-bit counter. The 8-bit register x is initialized to 1 and incremented by 1 if the primary input, en, is $true$. The counter is initialized back to 1 when it reaches 5. The safety property to verify is that x is not equivalent to 7. This property is always $true$ because x cannot be larger than 5. The concrete T, I, P equations for this example are as follows:

$$T(\mathbf{w}, \mathbf{x}, en, \mathbf{x}^+) = (w_1 = en\,?\,w_2 : x) \wedge (w_2 = w_4\,?\,8\text{'d1} : w_3) \wedge (w_3 = x + 8\text{'d1}) \wedge (w_4 = (x == 8\text{'d5})) \wedge (x^+ = w_1)$$

$$I(\mathbf{x}) = (x = 8\text{'d1})$$

$$P(\mathbf{w}, \mathbf{x}, en) = w_5 \wedge (w_5 = \neg(x == 8\text{'d7}))$$

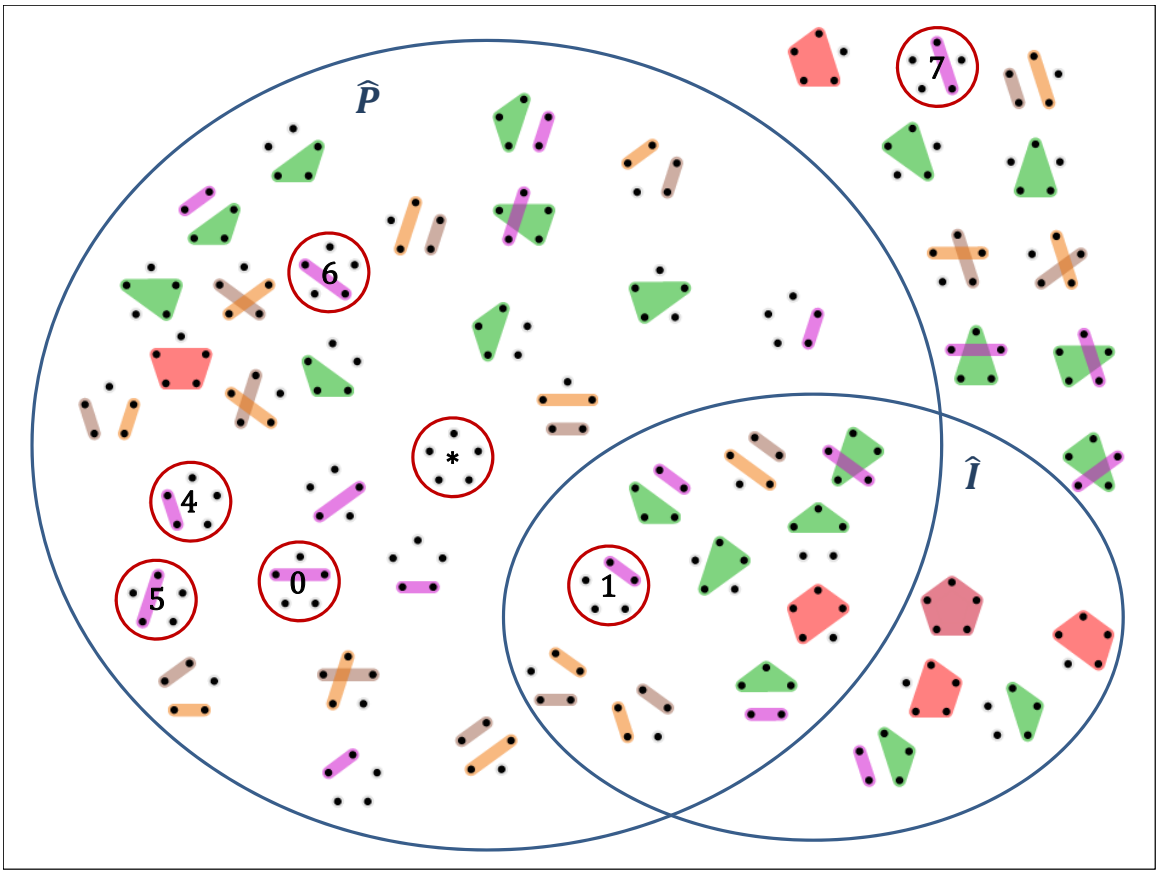


Figure 1.6: Abstract State Space

After applying datapath abstraction, we obtain the following EUF formulas:

$$\hat{T}(\hat{\mathbf{w}}, \hat{\mathbf{x}}, en, \hat{\mathbf{x}}^+) = (\hat{w}_1 = en\,?\,\hat{w}_2 : \hat{x}) \wedge (\hat{w}_2 = w_4\,?\,8\text{'d1} : \hat{w}_3) \wedge (\hat{w}_3 = \mathrm{ADD}(\hat{x}, \mathrm{K1})) \wedge (w_4 = (\hat{x} == \mathrm{K5})) \wedge (\hat{x}^+ = \hat{w}_1)$$

$$\hat{I}(\hat{\mathbf{x}}) = (\hat{x} = \mathrm{K1})$$

$$\hat{P}(\hat{\mathbf{w}}, \hat{\mathbf{x}}, en) = w_5 \wedge (w_5 = \neg(\hat{x} == \mathrm{K7}))$$

In these formulas, no single-bit state variable or uninterpreted predicate exists. They only contain five state terms: $\hat{x}$, $K1$, $K5$, $K7$, and $ADD(\hat{x}, K1)$. Therefore, a single abstract state in the system corresponds to a partition among these five terms, and the number of abstract states is the fifth Bell number, 52.

At the bit level, a concrete state corresponds to a complete bit assignment to all

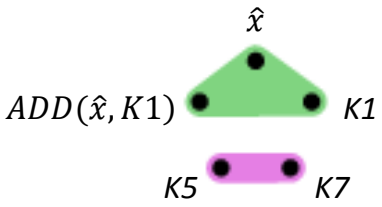


Figure 1.7: Graphical Representation of Abstract State

8 bits in x. As x is an 8-bit register, there are 2^8 concrete states in the system. We represent each concrete state by the decimal value of x. For example, 1 represents the concrete state $\bar{x}_7\bar{x}_6\bar{x}_5\bar{x}_4\bar{x}_3\bar{x}_2\bar{x}_1x_0$.

Fig. 1.6 shows the abstract state space of the system. Each abstract state in the diagram is represented by 5 dots with colored figures. Fig. 1.7 explains how we represent a partition[4]. Each dot represents a term in the partition, and the dots covered by the same figure are the elements in the same block. Therefore, the abstract state represented in Fig. 1.7 is: $\{\{\hat{x}, ADD(\hat{x}, K1), K1\}, \{K5, K7\}\}$.

Fig. 1.6 shows how the 52 abstract states in the system are classified into $\hat{I}$ and $\hat{P}$. The 7 states enclosed by red circles are feasible abstract states. The other 45 states are infeasible. Among the feasible states, 6 states map to 1 concrete state each. The number inside the red circle represents the concrete state. The other feasible abstract state maps to the remaining 250 concrete states, so we put an asterisk in the red circle.

As can be seen in the abstract state space, most of the abstract states are infeasible. They are automatically eliminated during datapath refinement, if necessary. Concrete states are grouped into a much smaller number of feasible abstract states. This grouping introduced by datapath abstraction allows for a simple representation of overapproximated reachable states, which improves the performance of the reachability computation.

[4]We obtained this graphical representation of partitions from Wikimedia Commons at `http://commons.wikimedia.org/wiki/File:Set_partitions_5;_circles.svg`.

```
`define W 2
`define MAX `W'b11
module example(CLK);
  input wire CLK;
  reg [`W-1:0] x1, x2;
  initial begin
    x1 = `W'd0;
    x2 = `W'd0;
  end
  always @(posedge CLK) begin
    x1 <= (x1 < x2)? x1 :
         ((x2==x1)|| x1!=`MAX))? x1+`W'd1):x2;
    x2 <= (x2==x1)? (x2+`W'd1) :
         ((x1<x2)||(x1!=`MAX))? x2 : x1;
  end
  wire P = !(x1 < x2);
endmodule
```

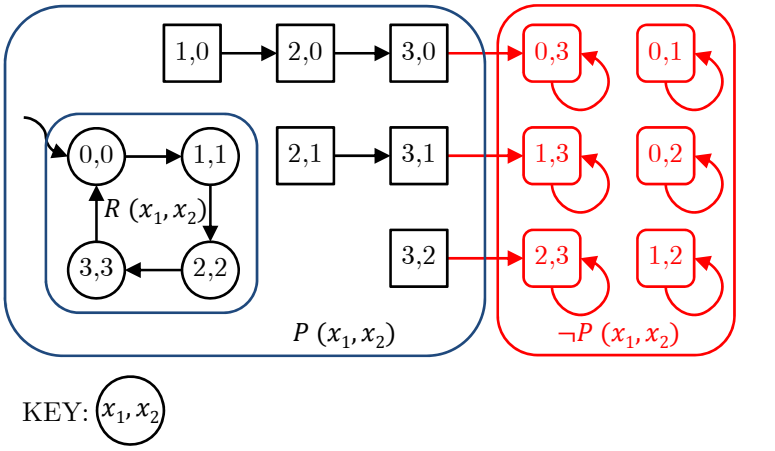


Figure 1.8: Verilog Description and Corresponding STG of an Example Sequential Circuit With a Specified Safety Property

1.2 Motivating Example

Fig. 1.8 gives the Verilog description and corresponding state transition graph (STG) of an example sequential circuit, shown in Fig. 1.9. The state variables are 2-bit unsigned integers $x_1 = x_{11}x_{10}$ and $x_2 = x_{21}x_{20}$ and their values are used to label the states (x_1 followed by x_2) in the STG. The good states are represented by circles (reachable states) and squares (unreachable states); squares with rounded corners correspond to bad states. Note that the circuit's sequential depth is exponential in

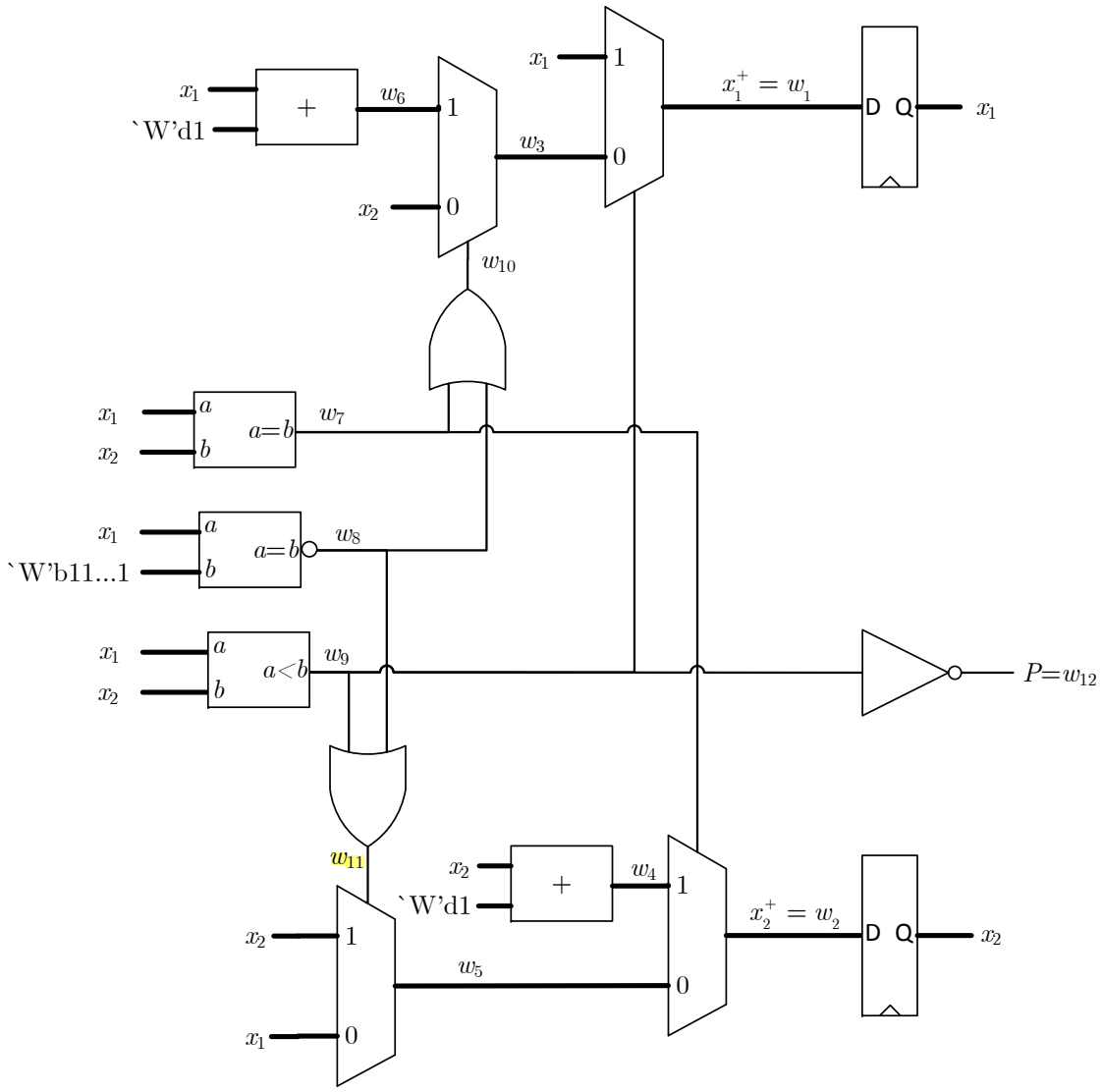


Figure 1.9: Circuit Representation of the Example Sequential Circuit in Fig. 1.8

the bit width W: $2^W = 2^2 = 4$. The circuit clearly satisfies the specified property $P(x_1, x_2) = \neg(x_1 < x_2)$ since, as can be seen from the STG, the reachable states satisfy $R(x_1, x_2) = (x_1 = x_2)$. From the Verilog description, we can derive T, I, and P as the following formulas[5]:

$$
\begin{aligned}
T(\mathbf{w}, \mathbf{x}, \mathbf{x}^+) = \; & (w_1 = w_9\,?\,x_1 : w_3) \wedge (w_2 = w_7\,?\,w_4 : w_5) \wedge (w_3 = w_{10}\,?\,w_6 : x_2) \wedge \\
& (w_4 = x_2 + \text{`W'd1}) \wedge (w_5 = w_{11}\,?\,x_2 : x_1) \wedge (w_6 = x_1 + \text{`W'd1}) \wedge \\
& (w_7 = (x_1 == x_2)) \wedge (w_8 = \neg\,(x_1 == \text{`W'b11...1})) \wedge \\
& (w_9 = (x_1 < x_2)) \wedge (w_{10} = w_7 \vee w_8) \wedge (w_{11} = w_8 \vee w_9) \wedge \\
& \left(x_1^+ = w_1\right) \wedge \left(x_2^+ = w_2\right)
\end{aligned}
$$

$$I(\mathbf{x}) = (x_1 = \text{`W'd0}) \wedge (x_2 = \text{`W'd0})$$

$$P(\mathbf{w}, \mathbf{x}) = (w_{12}) \wedge (w_{12} = \neg w_9) \wedge (w_9 = (x_1 < x_2))$$

Induction-based approaches such as IC3 and PDR follow the procedure illustrated in Fig. 1.10. First, they check a base case query, $I \wedge \neg P$. If it is satisfiable, a 0-step counterexample trace is returned. Otherwise, the induction step begins. Whenever a counterexample to induction (CTI) is found from the induction step query, $P \wedge T \wedge \neg P^+$ where P^+ is shorthand for $P(\mathbf{x}^+)$, the CTI is checked to see if it is reachable from initial states. If reachable, a counterexample trace is returned. If not, *refinement clauses* are derived to refute the CTI, and P is updated accordingly. This procedure iterates until no CTI is found, i.e., the updated P becomes an inductive invariant proving that P holds. The detailed procedure will be explained in Section 2.4.

When IC3 is run on the example, it makes two CTI checks and derives three refinement clauses: $\neg x_{11}$, $(\neg x_{10} \vee x_{20})$, and $(\neg x_{11} \vee x_{21})$. From the refinement clauses, it derives a strengthening assertion, $(\neg x_{10} \vee x_{20}) \wedge (\neg x_{11} \vee x_{21})$, which, together with P, basically says that x_1 is equal to x_2. As can be seen in Fig. 1.8, the concrete states

[5]Even though the formula is expressed in terms of the 2-bit state variables, x_1 and x_2, we view it as a bit-level formula with the understanding that RTL operations such as $x_1 < x_2$ can be synthesized to their bit-level implementations.

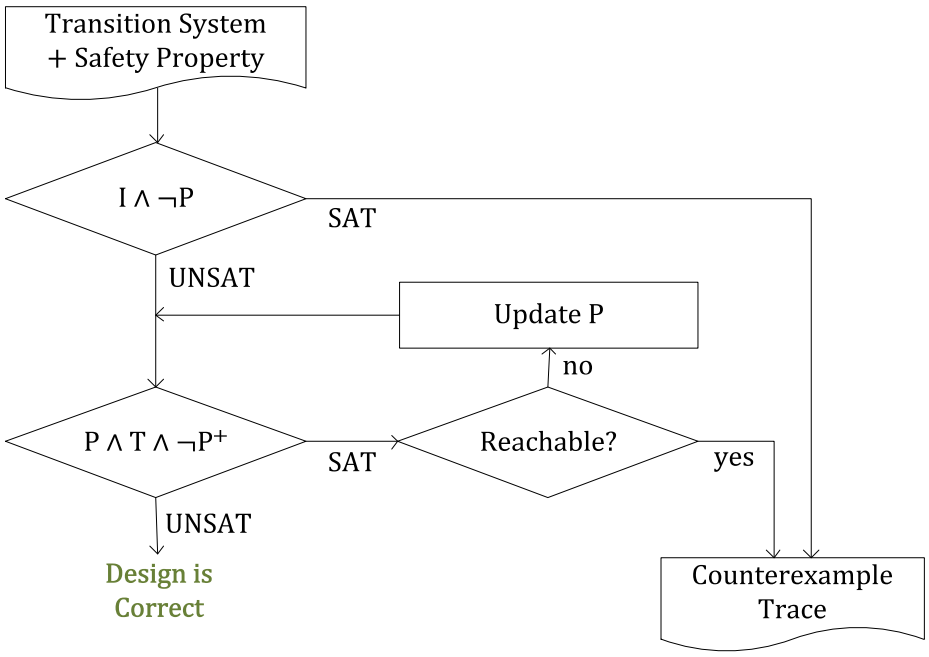


Figure 1.10: Procedure for Induction-Based Verification

$(0,0)$, $(1,1)$, $(2,2)$, and $(3,3)$, represented by $(\neg x_{10} \vee x_{20}) \wedge (\neg x_{11} \vee x_{21}) \wedge P$, i.e., a conjunction of P and a strengthening assertion, are closed under T.

The induction-based approaches at the bit level, however, do not scale as the bit width of the state variables in the example grows. Table 1.1 illustrates this problem. The table lists runtimes with a time-out of 1500 seconds, the number of CTI checks, and the number of refinement clauses for IC3 and PDR as the bit width is increased from 2 to 64. Every performance metric in the table increases exponentially as the bit width increases, and both IC3 and PDR time out when the bit width reaches 16. This is because of irrelevant bit-level details that bit-level approaches have to deal with. As can be seen in Fig. 1.8 and Fig. 1.11, the number of states and state transitions increases from 16 to 64 as the bit width increases from 2 to 3. This results in the exponentially larger number of CTIs and much more complicated transition relation leading to an exponentially longer runtime.

We wondered what would happen if we were to hide the irrelevant bit-level details.

Table 1.1: IC3 v. PDR on the Example Circuit for Different Bit Widths

Bit Width	Sequential Depth	Runtime, sec		CTI-Checks		Refinement-Clauses	
		IC3	PDR	IC3	PDR	IC3	PDR
2	4.00E+00	0.02	0.02	2	6	3	12
4	1.60E+01	0.07	0.05	16	71	84	114
8	2.56E+02	59.59	3.82	293	4782	31527	6503
16	6.55E+04	T.O.	T.O.	402	299511	179776	327581
32	4.29E+09	T.O.	T.O.	200	313973	28018	327958
64	1.84E+19	T.O.	T.O.	241	244916	11470	259737

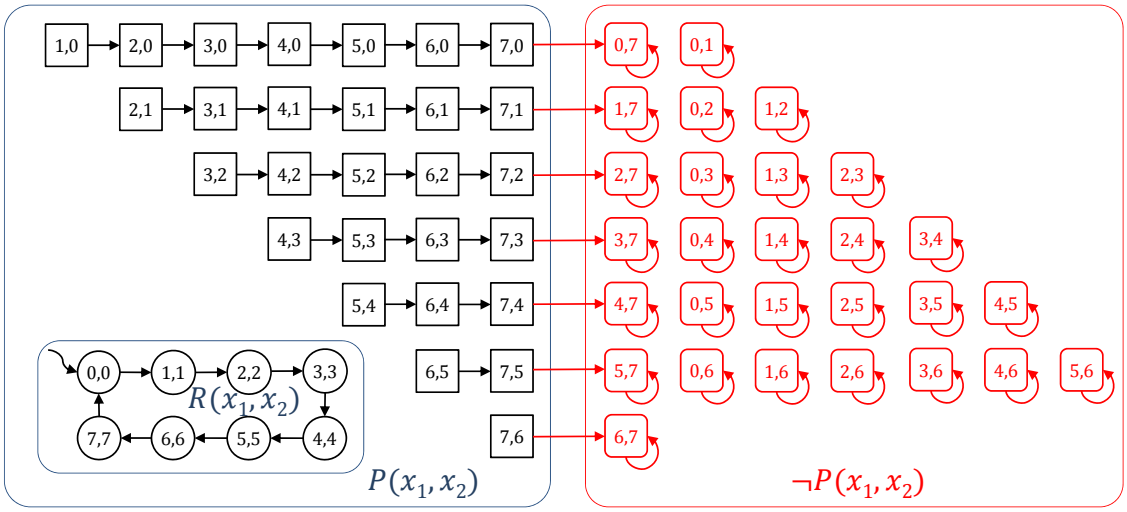


Figure 1.11: STG of the Example Circuit With the Bit Width of Three

This is the motivation of datapath abstraction. Datapath abstraction hides irrelevant bit-level details by abstracting complicated datapath operators and results in much simpler formulas to handle during the reachability computation. In addition, it allows us to derive a simple but powerful strengthening assertion regardless of the bit width of the state variables. Fig. 1.12 provides a high-level block diagram for the induction-based verification with datapath abstraction. The function blocks in the red-dashed box are the abstract-level counterparts of the bit-level ones for induction-based verification described in Fig. 1.10. The new procedure applies datapath abstraction before running the induction-based verifier. If the verifier returns an *abstract counterexample trace* (ACEXT), it checks to see if the abstract trace is feasible at the bit level. If

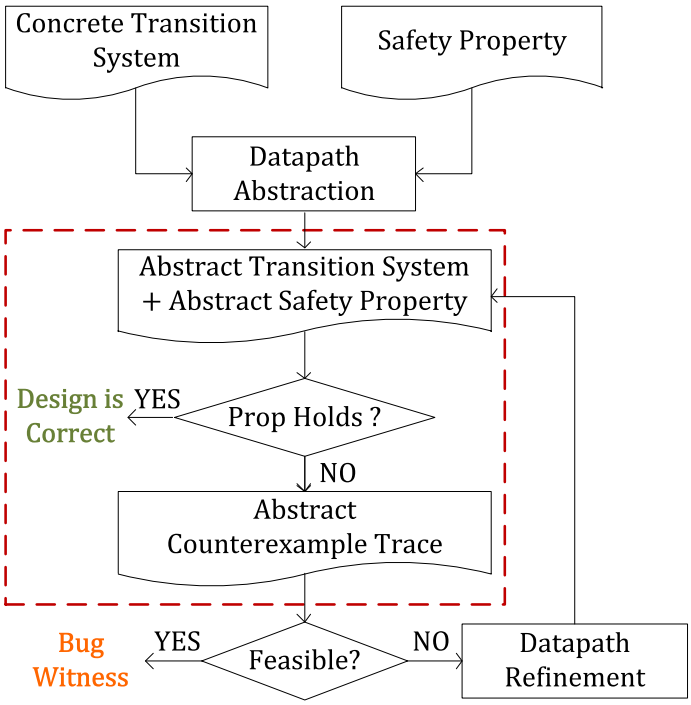


Figure 1.12: Induction-Based Verification With Datapath Abstraction

feasible, a bit-level counterexample trace is returned. If not, datapath refinement is triggered to refute the spurious trace, and the process is repeated.

Datapath abstraction creates the following uninterpreted variables, predicates, and functions from the corresponding bit-level equivalents:

`reg [`W-1:0]`x_1,x_2;		$\hat{x}_1, \hat{x}_2$
`W'd0	**DP-Abstract**	*K0*
`W'd1		*K1*
`W'b11...1		*KMAX*
x_1 < x_2	**DP-Concretize**	*LT*($\hat{x}_1, \hat{x}_2$)
x_1 + `W'd1		*ADD*($\hat{x}_1$,*K1*)
x_2 + `W'd1		*ADD*($\hat{x}_2$,*K1*)

Note that this abstraction is reversible; we just need to maintain the correspondence between the abstract entities and their bit-level counterparts. The abstract transition

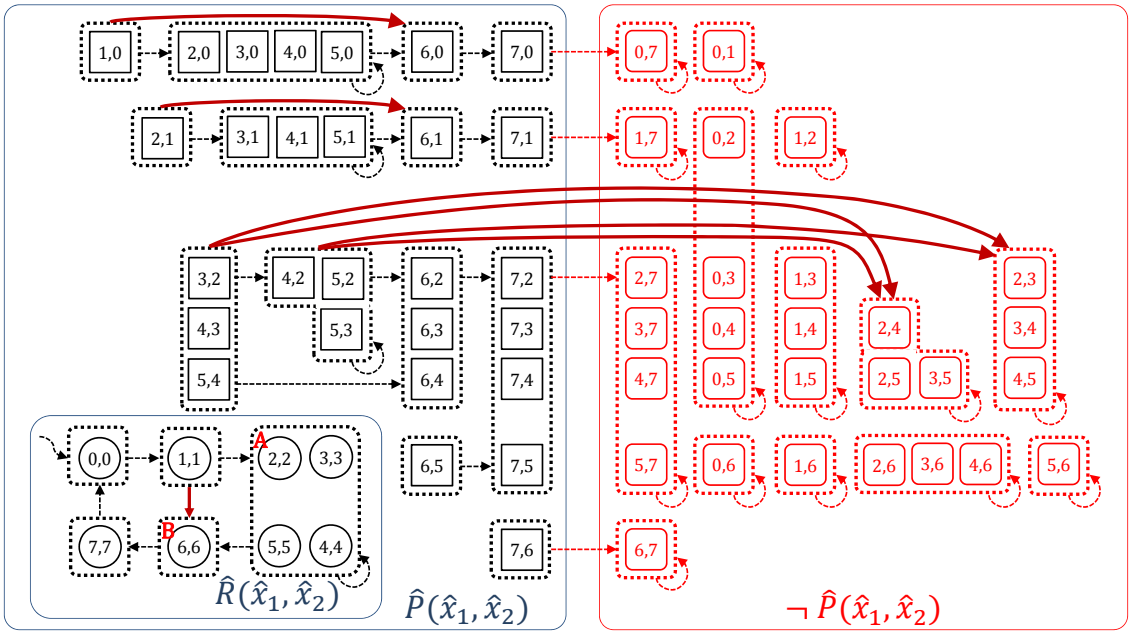


Figure 1.13: STG of the Example Circuit at the Abstract Level

relation $\hat{T}$, initial state $\hat{I}$, and safety property $\hat{P}$ of the example design are:

$$\begin{aligned}\hat{T}\left(\hat{\mathbf{w}}, \hat{\mathbf{x}}, \hat{\mathbf{x}}^{+}\right) = \quad & (\hat{w}_1 = w_9\,?\,\hat{x}_1 : \hat{w}_3) \wedge (\hat{w}_2 = w_7\,?\,\hat{w}_4 : \hat{w}_5) \wedge (\hat{w}_3 = w_{10}\,?\,\hat{w}_6 : \hat{x}_2) \wedge \\ & (\hat{w}_4 = \mathrm{ADD}\left(\hat{x}_2, \mathrm{K1}\right)) \wedge (\hat{w}_5 = w_{11}\,?\,\hat{x}_2 : \hat{x}_1) \wedge (\hat{w}_6 = \mathrm{ADD}\left(\hat{x}_1, \mathrm{K1}\right)) \wedge \\ & (w_7 = (\hat{x}_1 == \hat{x}_2)) \wedge (w_8 = \neg\left(\hat{x}_1 == \mathrm{KMAX}\right)) \wedge \\ & (w_9 = \mathrm{LT}\left(\hat{x}_1, \hat{x}_2\right)) \wedge (w_{10} = w_7 \vee w_8) \wedge (w_{11} = w_8 \vee w_9) \wedge \\ & \left(\hat{x}_1^{+} = \hat{w}_1\right) \wedge \left(\hat{x}_2^{+} = \hat{w}_2\right)\end{aligned}$$

$$\hat{I}\left(\hat{\mathbf{x}}\right) = (\hat{x}_1 = \mathrm{K0}) \wedge (\hat{x}_2 = \mathrm{K0})$$

$$\hat{P}\left(\hat{\mathbf{w}}, \hat{\mathbf{x}}\right) = (w_{12}) \wedge (w_{12} = \neg w_9) \wedge (w_9 = \mathrm{LT}\left(\hat{x}_1, \hat{x}_2\right))$$

We denote the abstract version of variables by putting a circumflex accent over the symbols like $\hat{x}_1$. In the abstracted formulas, K0, K1, KMAX, $\hat{x}_1$, $\hat{x}_2$, $\hat{x}_1^{+}$, and $\hat{x}_2^{+}$ are uninterpreted variables; ADD is an uninterpreted function; and LT is an uninterpreted predicate.

To see the effect of datapath abstraction on the STG, we derived the STG at the

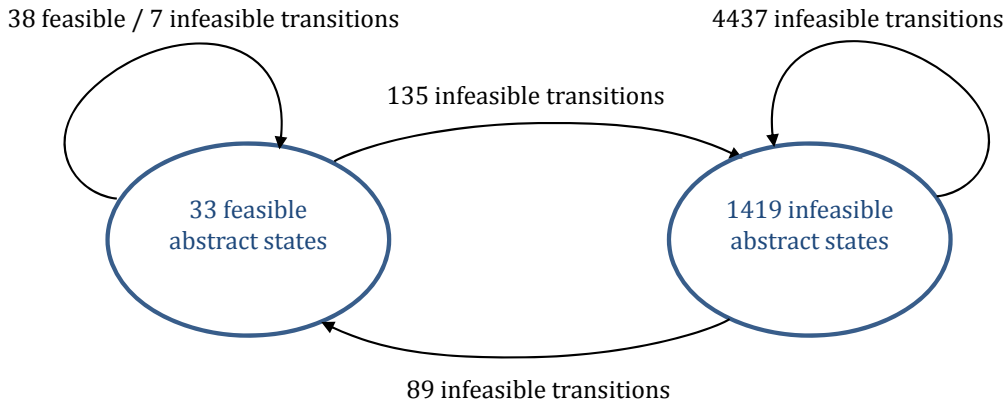


Figure 1.14: Number of Abstract States and Transitions of the Example Circuit

abstract level, as shown in Fig. 1.13. We obtained this graph by running a feasibility check on every abstract state and transition. For each abstract state minterm, $\hat{m}$, we checked to see whether it is satisfiable at the abstract level (i.e., consistent). If $\hat{m}$ is unsatisfiable, it cannot be a feasible abstract state (i.e., it cannot be satisfiable at the bit level), because the bit-level counterpart of an unsatisfiable abstract formula is unsatisfiable in a sound abstraction. If $\hat{m}$ is satisfiable, we checked the satisfiability of m with a bit-level solver, where m is the bit-level counterpart of $\hat{m}$. If m is satisfiable (i.e., it is feasible), we collected the set of concrete states mapped to the abstract state by running a bit-level solver with the query, $m \wedge s$, for each concrete state minterm s. If this query is satisfiable, we conclude that s maps to $\hat{m}$. Once all the feasible abstract states are collected, we computed the feasible abstract transitions among them by checking the satisfiability of $\hat{m}_1 \wedge \hat{T} \wedge \hat{m}_2^+$ for each pair of abstract state minterms, $\hat{m}_1$ and $\hat{m}_2$. If satisfiable at the bit level, we concluded that the abstract transition from $\hat{m}_1$ to $\hat{m}_2$ is feasible. We conducted this bit-level check only when the formula was satisfiable at the abstract level (i.e, consistent), just as we did for abstract states.

Now, consider how datapath abstraction affects the state transition graph. Datapath abstraction produces two major changes in the bit-level STG. First, some concrete

states are merged into a single abstract state. As can be seen in Fig. 1.13, each abstract state, which is enclosed by a black dotted line, contains one or more concrete states. This can lead to a *non-deterministic* behavior when the concrete states in a single abstract state have different next states under the same input condition. For example, the next abstract state of the abstract state A (that corresponds to the concrete states $(2,2)$, $(3,3)$, $(4,4)$, and $(5,5)$) is not uniquely defined. A can go to either A or B (which corresponds to $(6,6)$) under the same input condition (this example circuit has no input signal, so the input condition is *true*). This non-deterministic behavior occurs, because A contains two concrete states, $(2,2)$ and $(5,5)$, whose next states are $(3,3)$ and $(6,6)$, i.e., different. The transitions, $(2,2) \rightsquigarrow (3,3)$ and $(5,5) \rightsquigarrow (6,6)$, are captured by $A \rightsquigarrow A$ and $A \rightsquigarrow B$ respectively at the abstract level, which results in a non-deterministic behavior. Second, datapath abstraction introduces infeasible abstract states and abstract state transitions. For example, the red solid lines in the STG are infeasible abstract state transitions. Fig. 1.13 displays only the feasible abstract states and the transitions among them, but the entire abstract STG contains many more infeasible ones. Fig. 1.14 summarizes how many abstract states and transitions exist in the entire abstract STG. There are 1419 infeasible abstract states and 4668 infeasible transitions, 4437 of which are transitions among infeasible abstract states. Most of these infeasible states and transitions are not considered during the reachability computation at the abstract level.

When an induction-based verifier is run on the abstract formulas, it returns the 0-step abstract counterexample trace:

$(\hat{x}_1 = K0) \wedge (\hat{x}_2 = K0) \wedge (LT(\hat{x}_1, \hat{x}_2))$

since it does not know the semantics of the abstract constant $K0$ and the abstract predicate LT. However, upon concretization and bit-level feasibility checking, we can conclude that this counterexample is spurious and derive the following *datapath lemma*:

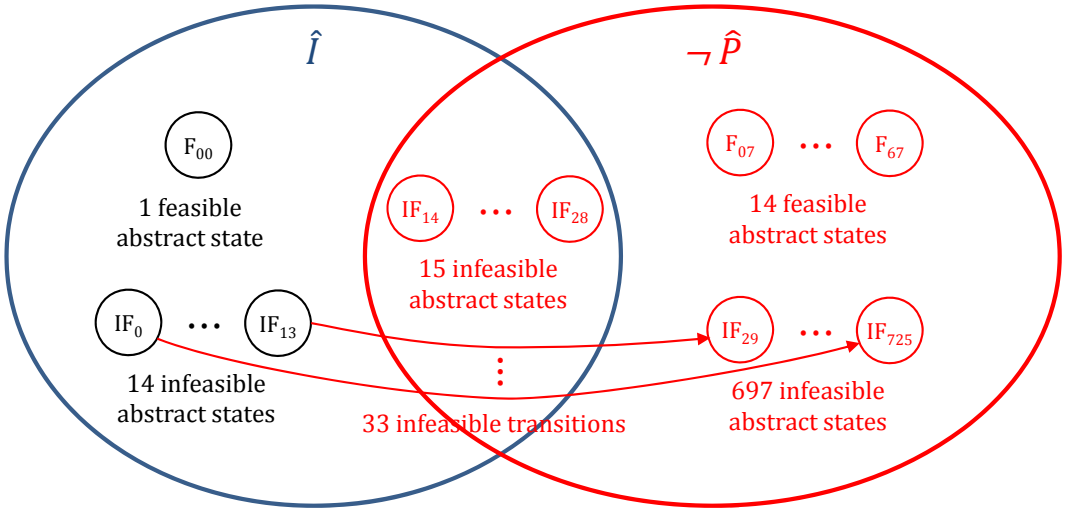


Figure 1.15: Close-up View of Abstract STG near I

$\delta_1 = \neg LT(\hat{x}_1, K0)$

to rule it out. The detailed procedure for deriving datapath lemmas will be explained in Section 3.3. The second run of the verifier returns a 1-step abstract counterexample trace, which is also found to be infeasible and is refuted by the datapath lemma:

$\delta_2 = \neg\big[(\hat{x}_1 = \hat{x}_2) \wedge LT(ADD(\hat{x}_1, K1), ADD(\hat{x}_2, K1))\big]$.

This lemma is a constraint that relates the uninterpreted LT predicate and the uninterpreted ADD function: in words, it states that incrementing two equal values cannot yield results in which one is less than the other. The effect of datapath refinements can be seen in Fig. 1.15. The STG displays all feasible and infeasible abstract states in $\hat{I}$ and $\neg\hat{P}$. The first datapath lemma removes the fifteen infeasible abstract states in the intersection of $\hat{I}$ and $\neg\hat{P}$, and the second datapath lemma removes thirty three infeasible transitions from infeasible abstract states in $\hat{I}$ to the ones in $\neg\hat{P}$. The third, and final run, of the verifier returns an empty trace after eliminating one CTI and generating a strengthening assertion: $(\hat{x}_2 = \hat{x}_1)$. Thus, after eliminating 0- and 1-step counterexamples with two datapath lemmas, the induction-based verifier is able to prove the property in just one CTI check *regardless of the bit width of the*

state variables.

This example illustrates two advantages of applying datapath abstraction to the induction-based approaches. First, datapath abstraction allows us to handle much simpler formula during the reachability computation by hiding unnecessary details of complex datapath operators. To verify control-centric properties, we do not need to know all the characteristics of datapath operators in a design. A small number of their characteristics is enough for the verification. This is why our approach works well. In our example, we were able to finish the proof with only two simple constraints on ADD and LT. Second, our approach takes advantage of the richer syntax of EUF logic in comparison to propositional logic at the bit level. In particular, we can represent a relationship among multi-bit registers in a formula that does not depend on their bit widths. In our example, we represented a strengthening assertion at the abstract level with a very simple EUF formula, $(\hat{x}_2 = \hat{x}_1)$, independent of the bit width, W. The bit-level counterpart of the formula would have required $(2 \times W)$ literals and clauses.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter II reviews previous work relevant to our research. Chapter III explains the Averroes system, which provides a scalable unbounded hardware verification framework. Chapter IV presents our abstraction methodology, which we call structural abstraction, and our datapath refinement procedure follows in Chapter V. We then provide the experimental results of Averroes in Chapter VI. Finally, Chapter VII discusses our conclusions and future work.

CHAPTER II

Previous Work

In this chapter, we review previous work that is related to our research. We then explain the IC3/PDR approximate reachability algorithms and the automatic datapath refinement algorithm in detail in the subsequent sections, because they are closely linked with our research.

2.1 Model Checking

Model checking (MC) [21, 22] is a technique that determines whether a finite transition system, M, satisfies a given specification, or a property, P. If M does not satisfy P, the technique produces a counterexample trace demonstrating how M violates P. Solving this problem involves a reachability computation in the state space of the transition system. This computation often leads to the so-called state-explosion problem [23, 24], because the number of states in the state space is exponential in the number of state elements. Since the first model checker based on explicit model checking, extended model checker (EMC) [25, 26, 27], was released in the early 1980s, most of the research in MC has focused on attacking this problem. One of the earliest major advances was symbolic model checking [28, 29, 30]. It represents a set of reachable states with binary decision diagrams (BDDs) and performs fixed-point algorithms for verifying temporal properties. Another major advance, bounded model

checking (BMC) [31, 32] came from the significant improvement of modern conflict-driven clause-learning SAT solvers [6, 7, 33]. BMC proves the existence of a k- or fewer-step counterexample trace by checking the satisfiability of a propositional formula with a SAT solver. The propositional formula is generated by unrolling the transition relation k times. BMC can efficiently disprove temporal properties, but it cannot prove those properties unless the sequential depth (the smallest number of transitions that can reach all the reachable states) is known, which is not common. Interpolation-based model checking [34] solves this problem by detecting a fixed point. It produces Craig interpolants from the UNSAT query of BMC to obtain an overapproximation of the set of reachable states. If the overapproximation reaches a fixed point, the property is proved to hold. Otherwise, it continues BMC with an increased bound. The procedure terminates if either a counterexample trace or a fixed point is found.

2.2 Reachability by Incremental Induction

The recently-introduced IC3 algorithm [2, 17] and its re-implementation in PDR [3] represent a major milestone for scalable model checking. Both can be described as SAT-based induction methods and both share some features of the earlier attempts at using induction [35, 36]. In particular, assuming that a given safety property P holds but is not inductive (i.e., is not closed under the transition relation), induction methods can be viewed as ways of performing *approximate reachability* with the goal of finding an assertion that strengthens (i.e., restricts) P so that it becomes an *inductive invariant* [17]. Alternatively, such methods can be seen as an application of counterexample-guided abstraction refinement (CEGAR) [1, 37, 38] whereby overapproximations of the reachable states are *refined* iteratively until enough unreachable states have been eliminated to prove that P does in fact hold or to produce a counterexample trace. Eliminating the need to compute exact reachability makes it

possible for induction methods to converge in a number of iterations that can be much smaller than the sequential depth of the transition relation. Additionally, induction methods can be applied without having to unroll the transition relation, which allows them to have better scalability than the earlier memory-intensive BDD or BMC approaches.

Several extensions of the IC3/PDR approach have already been proposed. Vizel et al. [39] layered an abstraction refinement framework based on localization reduction[1] on top of IC3 to significantly scale performance (over just IC3) on a set of large industrial benchmarks. Baumgartner et al. [40] utilized the invariants collected during an incomplete IC3 run to improve the abstract quality of localization reduction. Some approaches extended IC3/PDR to first-order logic. Hoder et al. [41] described an extension to PDR that enables reasoning about nonlinear predicate transformers and linear real arithmetic. Cimatti et al. [42, 43] extended IC3 to the theories of linear rational arithmetic, linear integer arithmetic, and fixed-size bit vectors by integrating implicit predicate abstraction. Welp et al. [44] generalized PDR to the quantifier free bit-vector theory. For the generalization, they proposed integer cubes and polytopes as a means to represent state cubes and applied interval simulation for cube enlargement.

2.3 Various Approaches to Abstraction

Four types of abstraction have been proposed to improve the scalability of model checking algorithms. *Predicate abstraction* [45, 46] identifies a set of predicates on data variables that are necessary for proving the property of interest. It then represents the transition relation with the predicates replaced by fresh Boolean variables, so the resulting abstract system does not include the original data variables, yielding a much simpler formula to verify. If the initial abstraction leads to spurious counterex-

[1]Localization reduction will be explained in Section 2.3.

amples, they are refuted by adding more predicates to the abstract system. Predicate abstraction was originally used in software verification [47, 48], but Jain et al. [49] applied it to hardware verification.

The second type of abstraction is *localization reduction* [37, 50, 51, 52]. It creates the abstract model by replacing some state variables or internal variables in the transition relation with unrestricted primary inputs. This substitution eliminates the logic attached to the variables, producing a smaller transition relation. If this abstraction causes spurious counterexamples, we can recover some of the variables causing the counterexamples.

The third type of abstraction is *bit width reduction* [53, 54, 55, 56]. This reduction technique separates datapaths from control logic and then reduces the widths of datapaths to lower bounds. The lower bounds, computed by a static analysis, guarantee that the properties being checked are preserved. Thus, the properties can be verified by running existing model checkers with the reduced system. This approach does not lead to any spurious counterexample that causes a refinement process, and it can be easily combined with any bit-level model checking algorithm. However, its use is quite limited; we can apply this method only when reducible datapaths exist.

The last type of abstraction is *datapath abstraction* [20, 57, 58, 59]. It classifies circuit components into datapath and control logic and abstracts datapath components by replacing them with uninterpreted functions or predicates in EUF[2] logic[20]. In the remainder of this section, we will discuss this type of abstraction. Burch and Dill [20] pioneered this type of abstraction to efficiently verify the control logic of pipelined microprocessors. Bryant et al. [57] introduced the CLU[3] logic, which extended EUF logic with counting logic expressions and lambda expressions. This extension allowed efficient memory modeling, and they built the UCLID[4] verification system [57, 60]

[2]EUF stands for equality with uninterpreted functions.

[3]CLU stands for counter arithmetic with lambda expressions and uninterpreted functions.

[4]UCLID stands for uninterpreted functions, counter arithmetic and lambda expressions for infinite domains. UCLID is also the name of the modeling language used in the system.

based on this CLU logic.

Counterexample-guided abstraction and refinement(CEGAR) [1] provides an automated procedure for datapath abstraction. Andraus et al. demonstrated a fully automatic CEGAR-based verification system called Reveal [58, 61, 62]. From the Verilog HDL description of a hardware design, Reveal automatically abstracts datapaths and produces EUF or CLU logic formulas. These formulas are passed to an SMT solver for word-level reasoning. If the word-level reasoning returns spurious counterexamples, Reveal applies automatic refinement based on localization, generalization and minimal unsatisfiable subsets extraction. The refinement process derives lemmas that tighten the initial rough abstraction and refute the spurious counterexamples.

Reveal applies abstraction on datapath operators such as addition and subtraction, but Brady et al. later demonstrated that we can abstract a bigger component consisting of many datapath operators as an uninterpreted function or predicate. In their abstraction technique, called CAL[5] [59, 63], they applied abstraction on module instantiations. CAL applies random simulation to collect candidate module instantiations for abstraction. They are abstracted as uninterpreted functions or predicates. If spurious counterexamples are found from the resulting abstract formula, CAL refines the initial abstraction by adding a multiplexer that selects either one of the original bit-level module instantiations or its abstracted version. The multiplexer is controlled by a conjunction of conditions. That is, the conditions specify when a model checker should use the precise model instead of an abstracted one to refute the spurious counterexamples. CAL uses machine learning (a decision tree learning algorithm [64]) to derive those conditions from simulation traces.

CAL's coarse-grained abstraction leads to a simple abstract formula at the beginning. However, the abstract formula can become bigger than the original bit-level

[5]CAL stands for conditional abstraction through learning

formula, because CAL brings back the original bit-level module instantiations during refinement. In addition, the condition of the multiplexer is a constraint on many local variables involved in the module instantiation to be refined. Thus, the condition can become big and complicated. Reveal's fine-grained abstraction, on the other hand, introduces a different kind of uninterpreted function or predicate for each datapath operation, generating a more complicated abstract formula. However, the abstract formula can be efficiently and effectively refined by a simple word-level lemma that is highly localized.

In datapath abstraction, memories are often modeled in a special way to avoid trivial spurious counterexamples caused by inconsistent memory read and write operations. That is, we need to model a memory so that what we read from a certain memory address is the same as the latest data we wrote to the address. To achieve this goal, Burch and Dill modeled a memory with two interpreted functions, *read* and *write* [20], and Bryant et al. modeled a memory with lambda expressions [57]. Unlike these approaches, Bjesse [65] proposed a memory abstraction technique that replaces a memory with a set of *representative slots*. His approach follows a CEGAR framework. It first replaces a memory with one represented slot representing one memory location. If this abstraction causes spurious counterexamples, it adds more slots based on the simulation-based analysis of the counterexamples. Because his approach performs only word-level netlist-to-netlist transformation for memory abstraction, we can easily apply datapath abstraction techniques to the transformed netlist. For example, we can replace datapath components in the represented slots with uninterpreted functions.

2.4 IC3 and PDR Approximate Reachability Algorithms

As defined in Section 1.1.5, our verification task is to determine whether or not a specified safety property P is always *true* on a finite transition system represented

```
1.  trace Reach-CEGAR(T, I, P){
2.      F0 = I;
3.      if (F0 ∧ !P)
4.          then return CEXT;// len(CEXT)=0
5.      if (F0 ∧ T ∧ !P+)
6.          then return CEXT;// len(CEXT)=1
7.      k = 1;
8.      Fk = P;
9.      while (true){
10.        Fk+1 = P;
11.        while(Fk ∧ T ∧ !P+){// CTI
12.            if Reachable(CTI, I)
13.                then return CEXT;// len(CEXT) ≥ k+1
14.                else Refine(1, k+1);
15.        }
16.        if (Fi = Fi-1 for some 2 ≤ I ≤ k+1)
17.            then return empty trace;// P holds
18.        k++;
19.    }
20. }
```

Figure 2.1: High-Level Pseudo Code for CEGAR-Based Reachability

by a transition relation T and an initial state I. The IC3/PDR approach solves this verification problem by induction. Using Bradley's terminology [17], these algorithms consist of two main steps:

- **Initiation**: prove that the initial states are good: $I \rightarrow P$.
- **Consecution**: derive a *strengthening* assertion $A(\mathbf{x})$ such that $A \wedge P \wedge T \rightarrow A^+ \wedge P^+$.

For our purposes, we find it useful to view the IC3/PDR approach as a clever application of CEGAR whereby a series of reachability overapproximations are systematically refined based on CTIs until either (i) a feasible state sequence from the initial state to an error state (a CEXT) is found or (ii) the refinements become sufficient to render the property being checked inductive, i.e., an overapproximation of the reachable

states that satisfies the property is found. A sketch of this approach, loosely mimicking IC3, is given in Fig. 2.1. The procedure, which we call **Reach-CEGAR**, takes as input T, I, and P, and returns a CEXT. An empty trace indicates that P holds; otherwise the returned CEXT demonstrates how P is violated.

Reach-CEGAR maintains an array of *frontiers* $F_0, F_1, \cdots, F_k, \cdots$ such that $F_0 = I$ and $F_j, j > 0$ is an overapproximation of what is reachable after j steps from I. After checking for 0- and 1-step CEXTs (lines 2 to 6), **Reach-CEGAR** enters its main loop (lines 9 to 19). At iteration $k > 0$, the goal is to check for the existence of CTIs that correspond to a CEXT whose length is at least $k + 1$. Each satisfying assignment to the current-state variables in the query on line 11 is a CTI that is checked to determine if it is reachable from I (line 12). If unreachable, the CTI is used to tighten the approximations of frontiers 1 to $k+1$ (line 14) by constraining them with appropriate *refinement clauses*. This process continues until either a reachable CTI is found (line 13) or all CTIs from the current frontier have been ruled out as unreachable. At that point, **Reach-CEGAR** checks for convergence (line 16) which is indicated when two frontier approximations become equal. If converged, **Reach-CEGAR** returns an empty trace signaling that P holds (line 17). Otherwise, it increments the iteration counter (line 18) and proceeds to check for the existence of CTIs that correspond to longer CEXTs.

This sketch hides many details that are critical to the performance of the algorithm. Specifically, in IC3/PDR **Reachable** and **Refine** are not separate procedures. Instead, the reachability check implied by **Reachable** is decomposed into a collection of 1-step backward reachability checks that are queued and processed in some order. Each such check may spawn further checks and/or yield one or more refinements that are propagated backward and forward to tighten the frontier approximations. The checks and attendant refinements, which are performed through appropriate calls to an incremental SAT solver, are closely choreographed to improve the quality of the

derived refinement clauses and speed up convergence. Different implementations will thus yield different refinements that can lead to drastically different performance.

There is, however, a critical detail in the implementation of **Reach-CEGAR** that deserves mention. Let ρ_j denote the CNF formula corresponding to the refinement clauses associated with frame j. With a slight abuse of notation, we will also view ρ_j as a set of clauses. At the beginning of each major iteration k, **Reach-CEGAR** insures that the sets of refinement clauses are distinct and subsumption-free, i.e., $\omega \nrightarrow \upsilon$ where $\omega \in \rho_j$ and $\upsilon \in \rho_i$ for $i \leqslant j$. The frontier overapproximations can now be expressed as:

$$F_j = P \wedge \bigwedge_{i=j}^{k+1} \rho_i, j \in [1, k+1]$$

which in turn implies that $F_1 \rightarrow F_2 \rightarrow \cdots \rightarrow F_{k+1}$, and reduces the convergence check on line 16 to checking that the set of refinement clauses at some frame j has become empty ($\rho_j = 1$). At that point, the refinement clauses at the last frontier serve as a strengthening assertion that helps prove the property: $\rho_{k+1} \wedge P \wedge T \rightarrow \rho_{k+1}^{+} \wedge P^{+}$

2.5 Feasibility Check and Datapath Refinement for Combinational Circuits

In a datapath abstraction and refinement framework, we conduct a feasibility check on an abstract counterexample and refine the abstraction if it is spurious. Suppose that datapath abstraction was applied to a problem verifying a safety property of a combinational circuit, and we found an abstract counterexample from the abstract formula representing the property. Then, one easy way to check the feasibility of the counterexample is to convert the entire abstract formula and the counterexample into a bit-level counterpart. If the bit-level formula is unsatisfiable, i.e., the abstract counterexample is spurious, we can refute the spurious counterexample by adding the negation of the counterexample to the original abstract formula. This kind of formula

that refines or tightens the initial abstraction is referred to as a *datapath lemma*. This simple approach has two problems. First, the converted bit-level formula for a feasibility check can be more complicated than the bit-level formula checking the property directly. In this case, it is better not to apply datapath abstraction. Second, the derived datapath lemma refutes only one spurious counterexample. In practice, this is not enough: this often results in too many refinement iterations.

The Reveal verification system provides an automatic procedure for efficiently deriving a set of powerful datapath lemmas. Reveal (i) conducts the feasibility check on a *solution cube*[6], which is much smaller than the original formula, and (ii) derives multiple strong datapath lemmas from a single refinement process. A solution cube, $\hat{C}$, is defined by a given satisfiable abstract formula, $\hat{F}$, and its solution, $\hat{S}$. $\hat{C}$ is an abstract formula implied by $\hat{S}$ and that implies $\hat{F}$. To derive a solution cube, the circuit representation of $\hat{F}$ is utilized. Reveal traverses the cone-of-influence of the property signal in the circuit and collects EUF literals, which affect the evaluation of the property signal. A solution cube, $\hat{C}$, is a conjunction of the collected literals. $\hat{C}$ then is concretized and passed to a bit-level solver. If it is satisfiable, a bit-level counterexample is returned. Otherwise, we can derive a datapath lemma, $\neg\hat{C}$, to refute the spurious counterexample. Reveal derives a set of more powerful datapath lemmas by extracting minimal unsatisfiable subsets (MUSes) from the concretized solution cube. The datapath lemmas derived from the minimal unsatisfiable subsets refute a much larger number of spurious counterexamples than the original datapath lemma, $\neg\hat{C}$.

[6]They call this a *violation*

CHAPTER III

Approximate Reachability at the Abstract Level

This chapter describes our scalable verification system, which combines (a) counter-example-guided datapath abstraction and refinement, and (b) induction-based approximate reachability computation. This chapter consists of two parts. Sections 3.1, 3.2, and 3.3 explain the two most complex modules in Averroes, and Section 3.4 describes the entire procedure in which Averroes integrates the two modules.

The first module is explained in Sections 3.1 and 3.2. These sections go into depth because this is a new approach. During the abstract-level reachability computation, Averroes incrementally refines the overapproximations of reachable states by eliminating a set of abstract states that can reach bad states. The set of abstract states is represented by an abstract state cube, which is a conjunction of equalities and disequalities among state terms. The most important and challenging task of the abstract-level reachability computation is to compute the abstract state cube from the solution of an abstract formula in EUF logic. The state cube must represent as many states as possible in order to reduce the number of the refinement iterations. In Sections 3.1 and 3.2, we describe a systematic and efficient way to compute this kind of abstract state cube. Section 3.1 defines this problem theoretically and explains our solution to this problem. Section 3.2 describes the entire procedure of deriving an

abstract state cube from a query made by Averroes.

The second module is explained in Section 3.3. Once the abstract-level reachability computation returns an abstract counterexample trace, we need to check the bit-level feasibility of the trace and perform datapath refinement if necessary. To conduct these tasks, we extend the procedure explained in Section 2.5 to sequential circuits. In Section 3.3, we explain the theoretical problems we encountered while developing the extended procedure and our solutions to these problems.

3.1 Partition Spaces

Datapath abstraction creates an abstract state space in which uninterpreted terms are used to encode abstract datapath states, and uninterpreted functions and predicates are used to approximate the combinational blocks in the design. Reachability computations in such an abstract state space require checking the satisfiability of a sequence of first-order logic queries. In this section we describe the projection procedure needed to reason about such queries. We begin by considering partition spaces defined over zero-arity terms. Next we consider partition spaces involving uninterpreted functions, followed by spaces containing uninterpreted predicates. We conclude by proving the correctness of our projection procedure.

3.1.1 Partial Order of Partition Spaces

Consider a set of n uninterpreted zero-arity terms $\{t_1, t_2, \cdots, t_n\}$ and let $\mathbb{P}^{(n)}(t_1, t_2, \ldots, t_n)$ denote the space of partitions defined on these terms. For example,

$\mathbb{P}^{(1)}(t_1) = \{t_1\}$

$\mathbb{P}^{(2)}(t_1, t_2) = \{\{t_1|t_2\}, \{t_1, t_2\}\}$

$\mathbb{P}^{(3)}(t_1, t_2, t_3) = \{\{t_1|t_2|t_3\}, \{t_1, t_3|t_2\}, \{t_1|t_2, t_3\}, \{t_1, t_2|t_3\}, \{t_1, t_2, t_3\}\}$,[1]

[1]This is not standard notation, but is sufficient for our purposes. For example $\{t_1, t_2|t_3\}$ should be interpreted to mean $\{\{t_1, t_2\}, \{t_3\}\}$ in standard set notation.

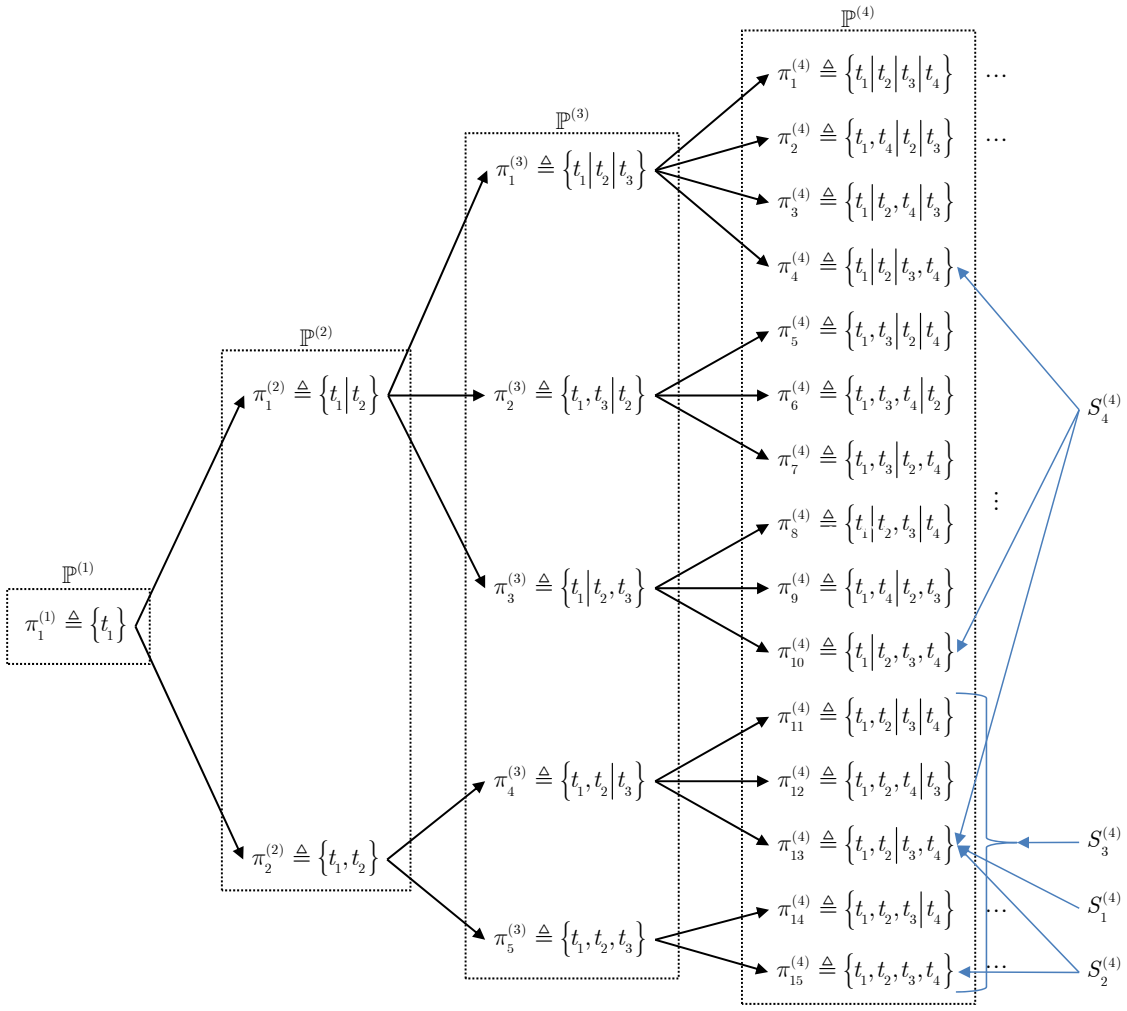


Figure 3.1: Partial Order of Partitions

etc. The size of $\mathbb{P}^{(n)}$ (the number of partitions it has) is the n^{th} Bell number B_n [5]. For ease of reference, the partitions of $\mathbb{P}^{(n)}$ will be labeled $\pi_1^{(n)}, \pi_2^{(n)}, \cdots, \pi_{B_n}^{(n)}$ based on a numbering scheme to be described shortly.

As shown in Fig. 3.1, the partitions of $\mathbb{P}^{(1)}, \mathbb{P}^{(2)}, \cdots, \mathbb{P}^{(n)}$ form a partial order whereby $\pi_i^{(n-1)} \leq \pi_j^{(n)}$ if and only if for all pairs of terms, if t_1 and t_2 are in the same cell (different cells) of $\pi_i^{(n-1)}$ then they are also in the same cell (different cells) of $\pi_j^{(n)}$. For example, $\pi_2^{(3)} \leq \pi_7^{(4)}$, because for all three terms in $\pi_2^{(3)}$, $\pi_7^{(4)}$ also contains t_1 and t_3 in the same cell, and t_2 and the other two terms in different cells. Given two partitions $\pi_i^{(n-1)}$ and $\pi_j^{(n)}$ such that $\pi_i^{(n-1)} \leq \pi_j^{(n)}$, we will say that $\pi_j^{(n)}$ is a

refinement of $\pi_i^{(n-1)}$ and that $\pi_i^{(n-1)}$ is *compatible* with $\pi_j^{(n)}$. Noting further that this partial order is a directed tree, each partition, execpt for the root partition, has a unique predecessor. Thus, $\pi_i^{(n-1)} \leq \pi_j^{(n)}$ is equivalent to $pred(\pi_j^{(n)}) = \pi_i^{(n-1)}$ where *pred* denotes the predessor function.

Given a set of partitions $S^{(n)} \subseteq \mathbb{P}^{(n)}$, their *projection* on $\mathbb{P}^{(n-1)}$ is the set of predecessor partitions according to the partial order. Specifically,

$$S^{(n)}(t_1, t_2, \cdots, t_n) \downarrow \mathbb{P}^{(n-1)} = \bigcup_{\pi^{(n)} \in S^{(n)}} pred\left(\pi^{(n)}\right) \tag{3.1}$$

where we use $\downarrow$ as the projection operator. This operation can be applied recursively to obtain the projection of $S^{(n)}$ on $\mathbb{P}^{(n-m)}$ for any $m \geq 1$:

$$S^{(n)}(t_1, t_2, \cdots, t_n) \downarrow \mathbb{P}^{(n-m)} = \left(\left(\left(S^{(n)} \downarrow \mathbb{P}^{(n-1)}\right) \downarrow \mathbb{P}^{(n-2)}\right) \cdots\right) \downarrow \mathbb{P}^{(n-m)} \tag{3.2}$$

The projection operation can also be written in the following equivalent form

$$S^{(n)}(t_1, t_2, \cdots, t_n) \downarrow \mathbb{P}^{(n-m)} = S^{(n)}(t_1, t_2, \cdots, t_n) \downarrow \{t_1, t_2, \cdots, t_{n-m}\} \tag{3.3}$$

which should be interpreted as either a *restriction* of the set $S^{(n)}$ to the first $(n-m)$ terms or as a *generalization* through elimination of the last m terms. To illustrate projection, consider the following four partition sets on 4 terms and their projections on $\{t_1, t_2, t_3\}$ (see Fig. 3.1):

$$\begin{array}{ll}
S_1^{(4)} = \left\{\pi_{13}^{(4)}\right\}: & S_1^{(4)} \downarrow \{t_1, t_2, t_3\} = \left\{\pi_4^{(3)}\right\} \\
S_2^{(4)} = \left\{\pi_{13}^{(4)}, \pi_{15}^{(4)}\right\}: & S_2^{(4)} \downarrow \{t_1, t_2, t_3\} = \left\{\pi_4^{(3)}, \pi_5^{(3)}\right\} \\
S_3^{(4)} = \left\{\pi_{11}^{(4)}, \pi_{12}^{(4)}\pi_{13}^{(4)}, \pi_{14}^{(4)}, \pi_{15}^{(4)}\right\}: & S_3^{(4)} \downarrow \{t_1, t_2, t_3\} = \left\{\pi_4^{(3)}, \pi_5^{(3)}\right\} \\
S_4^{(4)} = \left\{\pi_4^{(4)}, \pi_{10}^{(4)}\pi_{13}^{(4)}\right\}: & S_4^{(4)} \downarrow \{t_1, t_2, t_3\} = \left\{\pi_1^{(3)}, \pi_3^{(3)}, \pi_4^{(3)}\right\}
\end{array} \tag{3.4}$$

The partition numbering scheme used in Fig. 3.1 is based on a given, arbitrary,

ordering of the terms. Let $t_1 < t_2 < \cdots < t_{n-1} < t_n$ and assume that the partitions of $\mathbb{P}^{(1)}, \mathbb{P}^{(2)}, \cdots, \mathbb{P}^{(n-1)}$ have already been numbered. In particular, let $\pi_1^{(n-1)}, \pi_2^{(n-1)}, \cdots, \pi_{B_{n-1}}^{(n-1)}$ be the numbered partitions of $\mathbb{P}^{(n-1)}$. The partitions of $\mathbb{P}^{(n)}$ are now constructed and numbered sequentially by considering all possible ways of adding t_n to each partition of $\mathbb{P}^{(n-1)}$ starting from $\pi_1^{(n-1)}$ and ending in $\pi_{B_{n-1}}^{(n-1)}$. For example, the five partitions of $\mathbb{P}^{(3)}$ are obtained from the two partitions of $\mathbb{P}^{(2)}$ as follows:

1. From $\pi_1^{(2)}$ which has 2 cells we generate the following three cells of $\mathbb{P}^{(3)}$:
   (a) $\pi_1^{(3)}$ by adding t_3 as a new third cell
   (b) $\pi_2^{(3)}$ by adding t_3 to the first cell
   (c) $\pi_3^{(3)}$ by adding t_3 to the second cell
2. From $\pi_2^{(2)}$ which has 1 cell we generate the following two cells of $\mathbb{P}^{(3)}$:
   (a) $\pi_4^{(3)}$ by adding t_3 as a new second cell
   (b) $\pi_5^{(3)}$ by adding t_3 to the first (and only) cell

It is important to note that this numbering scheme is meant only as a way to easily refer to any partition or set of partitions in $\mathbb{P}^{(n)}$ and does not imply any restriction on the definition of the projection operator. In particular, given a set of partitions on $\{t_1, t_2, \cdots, t_n\}$ its projection on any subset of $T \subseteq \{t_1, t_2, \cdots, t_n\}$ can be obtained using (3.2) or (3.3) by simply re-ordering the terms so that the terms in T occur before the other terms.

3.1.2 Implicit Encoding of Partition Sets

Reasoning about partition sets by explicit enumeration is clearly unscalable. In this section we show how partition sets can be encoded implicitly using logical formulas that involve equality and dis-equality between terms. To facilitate the description

of this encoding, we introduce a set of $\frac{n(n-1)}{2}$ binary encoding variables

$$e_{ij} \triangleq (t_i = t_j), \quad i, j \in \{1, 2, \cdots, n\}, \text{ and } i < j$$

and a corresponding set of literals

$$E = \{\dot{e}_{12}, \dot{e}_{13}, \cdots, \dot{e}_{n-1,n}\}$$

where $\dot{e}_{ij}$ is a literal that stands for either e_{ij} or $\neg e_{ij}$. These encoding varaibles must satisfy the transitivity of equality, namely,

$$e_{ij} \wedge e_{jk} \rightarrow e_{ik}, \quad i < j < k \tag{3.5}$$

which states that if t_i and t_k are both equal to t_j, then they must also be equal to each other. This constraint can also be written in the following clausal form

$$(\neg e_{ij} \vee \neg e_{jk} \vee e_{ik}), \quad i < j < k \tag{3.6}$$

allowing further implications to be inferred. For example, if t_i is equal to t_j but not to t_k, then t_j and t_k cannot be equal. In general, wherever any two literals in (3.6) are false, the third literal must be true to satisfy the transitivity of equality.

Partition Minterms: Any single partition can be completely specified by a conjunction of $\frac{n(n-1)}{2}$ distinct literals from E that does not include opposing literals and that does not violate the transitivity of equality. We refer to such conjunctions as *partition minterms* and use the notation $\mu_i^{(n)}$ to refer to the minterm of partition $\pi_i^{(n)}$[2]. The minterms for $\mathbb{P}^{(2)}$ and $\mathbb{P}^{(3)}$ are shown in Fig. 3.2. Each such minterm is

[2]The minterm nomenclature is meant to conjure up the notion of minterms in the n-dimensional Boolean space. We must note, however, that the literals used to encode partition minterms cannot take all $2^{\frac{n(n-1)}{2}}$ possible valuations of $\{0, 1\}$ because of the transitivity of equality.

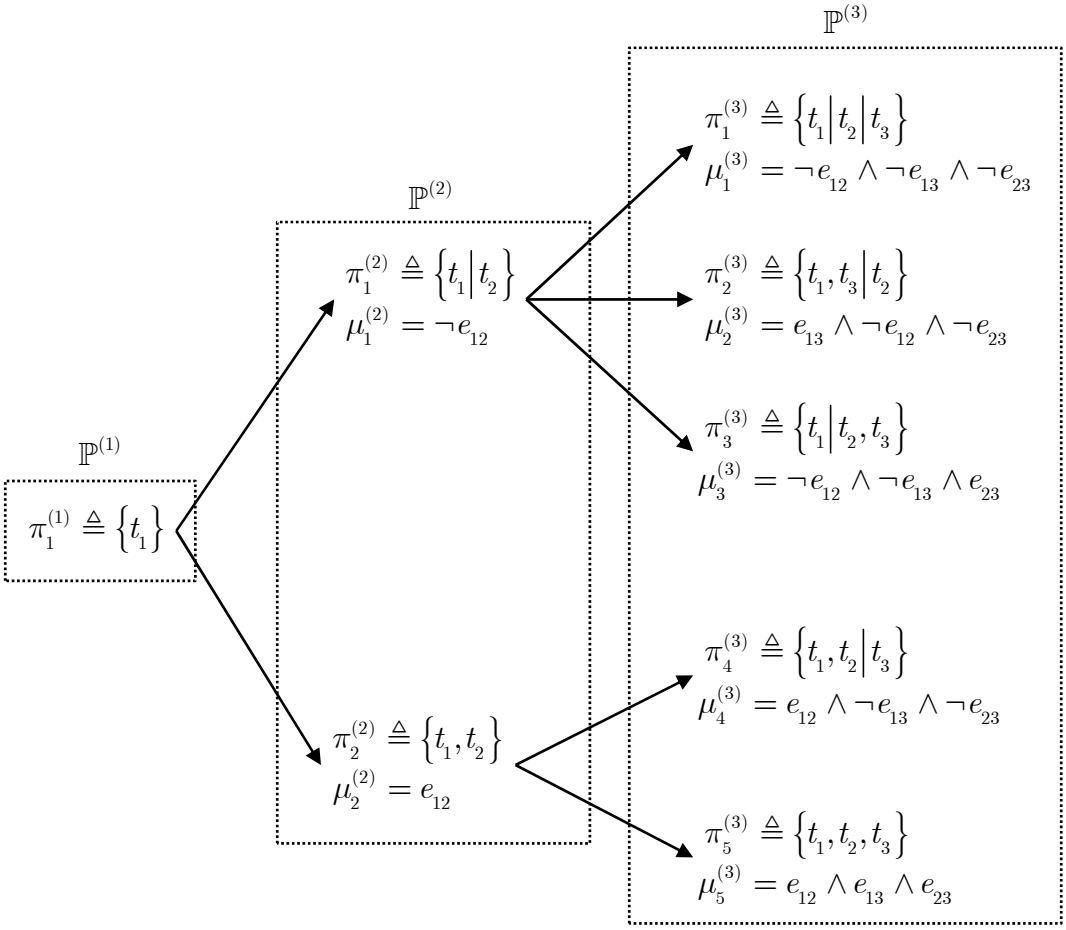


Figure 3.2: Partition Minterms

unique (modulo commutativity and associativity) but is not minimal. For example, $\mu_2^{(3)}$ can be simplified to either $(e_{13} \wedge \neg e_{12})$ or $(e_{13} \wedge \neg e_{23})$ because the removed conjunct is implied, by the transitivity of equality, from the two that are kept. On the other hand, these minimal forms are obviously non-unique.

Partition Cubes: A partition cube is a conjunction of up to $\frac{n(n-1)}{2}$ distinct literals from E that does not include opposing literals and that does not violate the transitivity of equality. A partition cube is an implicit representation of a set of partitions. A partition minterm is a partition cube that includes all $\frac{n(n-1)}{2}$ literals and *covers* a single partition. Larger partition cubes cover more than one partition. For example, the partition cube (e_{12}) covers $\{\pi_2^{(2)}\}$ in $\mathbb{P}^{(2)}$, $\{\pi_4^{(3)}, \pi_5^{(3)}\}$ in $\mathbb{P}^{(3)}$, and

$\{\pi_{11}^{(4)}, \pi_{12}^{(4)}, \pi_{13}^{(4)}, \pi_{14}^{(4)}, \pi_{15}^{(4)}\}$ in $\mathbb{P}^{(4)}$[3].

Subsets of E that include neither opposing literals nor literal combinations that violate the transitivity of equality will be referred to as *consistent*. A partition cube can now be viewed as a consistent subset of E and interpreted as the logical conjunction of the literals in that subset. Notationally, we will use $C \subseteq E$ to capture the literals of a partition cube and $\varphi(C) = \bigwedge_{\dot{e}_{ij} \in C} \dot{e}_{ij}$ to express it as a logical constraint on these literals. To simplify notation, we will also assume that we are operating in the B_n-dimensional partition space $\mathbb{P}^{(n)}$ and drop the (n) superscript notation.

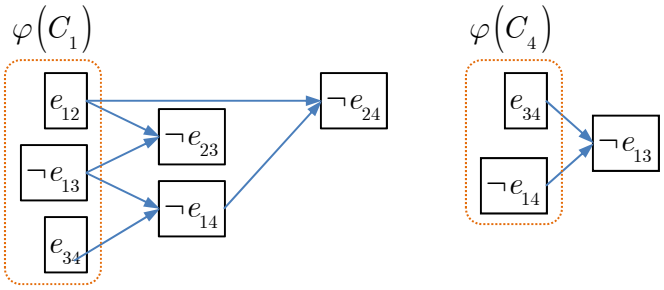
Partition sets $S_1^{(4)}, S_2^{(4)}, S_3^{(4)}$, and $S_4^{(4)}$ in (3.4) can now be implicitly encoded by the following partition cubes:

$$\begin{aligned} C_1 &= \{e_{12}, e_{34}, \neg e_{13}\} & \varphi(C_1) &= e_{12} \wedge e_{34} \wedge \neg e_{13} \\ C_2 &= \{e_{12}, e_{34}\} & \varphi(C_2) &= e_{12} \wedge e_{34} \\ C_3 &= \{e_{12}\} & \varphi(C_3) &= e_{12} \\ C_4 &= \{\neg e_{14}, e_{34}\} & \varphi(C_4) &= \neg e_{14} \wedge e_{34} \end{aligned} \tag{3.7}$$

as can be readily verified by reference to Fig. 3.1.

A partition cube is *complete* if it contains all the literals implied by the transitivity of equality. In (3.7), only C_2 and C_3 are complete, whereas C_1 and C_4 are incomplete. The missing literals in an incomplete partition cube can be derived in a manner similar to unit propagation in satisfiability solvers [6]. For example, C_1 and C_4 can be *completed* by the following implication sequences:

[3]As an aside, for $i \neq j$, the number of partitions covered by the single-literal cube e_{ij} in $\mathbb{P}^{(n)}$ is B_{n-1}, and the number covered by $\neg e_{ij}$ is $B_n - B_{n-1}$.



We will use C^* to indicate the complete cube induced by partition cube C.

Implicit Projection: The projection operator on explicit representations of partition sets corresponds to cofactoring the implicit representations of these sets. Recall that the positive cofactor of an n-variable Boolean function $f(x_1, \cdots, x_i, \cdots, x_n)$ with respect to x_i, usually denoted as f_{x_i} or $f|_{x_i}$, is the $(n-1)$-variable Boolean function obtained by setting x_i to 1, i.e., $f(x_1, \cdots, 1, \cdots, x_n)$ [66]. Similarly, the negative cofactor of f with respect to x_i, denoted as $f_{\neg x_i}$ or $f|_{\neg x_i}$ is $f(x_1, \cdots, 0, \cdots, x_n)$. This definition extends naturally to any set of literals. Thus, $f|_{\{x_i, \neg x_j\}}$ is obtained by setting x_i to 1 and x_j to 0. When f is a conjunction of literals (a cube), cofactoring with respect to a subset of those literals is simply done by dropping them from the cube. For example, $(x_1 \wedge \neg x_3 \wedge x_4 \wedge \neg x_6)_{\{\neg x_3, x_4\}} = x_1 \wedge \neg x_6$.

We can now define the implicit projection of a partition cube C on a set of terms $S \subset \{t_1, t_2, \cdots, t_n\}$ by the formula:

$$\varphi(C) \downarrow S = \varphi(C)\,|_{\{\dot{e}_{ij} \in C \,|\, t_i \notin S \vee t_j \notin S\}} \tag{3.8}$$

which simply says that the projection is accomplished by dropping all literals that do not exclusively refer to the terms in the set S. Applying this to the partition cubes

in (3.7) yields the following projections:

$$\begin{aligned}
&\varphi(C_1)\downarrow\{t_1,t_2,t_3\} = (e_{12}\wedge e_{34}\wedge\neg e_{13})\downarrow\{t_1,t_2,t_3\} = (e_{12}\wedge e_{34}\wedge\neg e_{13})\,|_{e_{34}} = (e_{12}\wedge\neg e_{13})\\
&\varphi(C_2)\downarrow\{t_1,t_2,t_3\} = (e_{12}\wedge e_{34})\downarrow\{t_1,t_2,t_3\} = (e_{12}\wedge e_{34})\,|_{e_{34}} = e_{12}\\
&\varphi(C_3)\downarrow\{t_1,t_2,t_3\} = e_{12}\downarrow\{t_1,t_2,t_3\} = e_{12}|_{e_{34}} = e_{12}\\
&\varphi(C_4)\downarrow\{t_1,t_2,t_3\} = (\neg e_{14}\wedge e_{34})\downarrow\{t_1,t_2,t_3\} = (\neg e_{14}\wedge e_{34})\,|_{\{\neg e_{14},e_{34}\}} = 1
\end{aligned} \tag{3.9}$$

Comparing (3.9) and (3.4) we can verify that the implicit projections match their explicit counterparts except for the projection of $\varphi(C_4)$. To match $\{\pi_1^{(3)},\pi_3^{(3)},\pi_4^{(3)}\}$ which is the explicit projection of $S_4^{(4)}$, the correct implicit projection should be $\neg e_{13}$ and not 1! However, the literal $\neg e_{13}$ is not present in C_4 even though it is implied by the transitivity of equality. This immediately suggests that cofactoring should be applied *after* all relevant literals implied by the transitivity of equality are included in the representation of the partition cube. In particular, the correct projection is guaranteed to be obtained from a complete cube. Because $C_4^* = C_4\cup\{\neg e_{13}\}$, the correct projection of C_4 is now obtained as follows:

$$\begin{aligned}
\varphi(C_4^*)\downarrow\{t_1,t_2,t_3\} &= (\neg e_{14}\wedge e_{34}\wedge\neg e_{13})\downarrow\{t_1,t_2,t_3\}\\
&= (\neg e_{14}\wedge e_{34}\wedge\neg e_{13})\,|_{\{\neg e_{14},e_{34}\}}\\
&= \neg e_{13}
\end{aligned} \tag{3.10}$$

We should note, though, that cofactoring a complete cube is a sufficient but not necessary condition to produce the correct projection. For example, the correct projection was obtained from the incomplete cube C_1. This suggests that we extend the definition of completeness to completeness relative to a given set of, instead of all, terms. A partition cube is *complete relative to the set of terms* S if it includes all literals involving terms in S. Given a partition cube C and a set of terms S, the complete cube induced by C relative to S will be denoted as C^{*S}. Computing C^{*S}

can be viewed as an optimization of the unit propagation procedure used to produce C^* by limiting the implications to those that only yield literals involving terms in S.

To summarize, given a partition cube C and a projection set $S \subset \{t_1, t_2, \cdots, t_n\}$, the implicit projection procedure is a 2-step process:

1. Compute C^{*S} by a restricted form of unit propagation from C to produce literals "in" S.

2. Cofactor $\varphi(C^{*S})$ with respect to those literals "not in" S.

3.1.3 Partition Spaces Involving Uninterpreted Functions

So far we have considered partition spaces involving zero-arity terms. We now consider partition spaces on n terms such that some of these terms have arity greater than 0, namely uninterpreted functions (UFs) of one or more arguments. To keep the notation simple, we will still refer to the terms as t_i for $i \in \{1, 2, \cdots, n\}$ and indicate that a particular term is a UF by associtaing it with the definition of that UF. For example, in the partition space defined over the 4 terms $\{t_1, t_2, f(t_1), f(t_2)\}$, we will have $t_3 = f(t_1)$ and $t_4 = f(t_2)$. The presence of UF terms imposes constraints on the partition space that render certain partitions impossible because they violate *functional consistency*. Continuing with this example, the functional consistency constraint is $(t_1 = t_2) \rightarrow (f(t_1) = f(t_2))$ which can be expressed in terms of the e variables as:

$$e_{12} \rightarrow e_{34} \tag{3.11}$$

Negation of this constraint identifies those impossible partitions that violate functional consistency. Specifically, $e_{12} \wedge \neg e_{34}$ corresponds to the three partitions $\pi_{11}^{(4)}$, $\pi_{12}^{(4)}$, and $\pi_{14}^{(4)}$ (see Fig. 3.3). Other than this, the machinery we developed in Section 3.1.2, augmented with a few extensions, applies to partition spaces containing UFs.

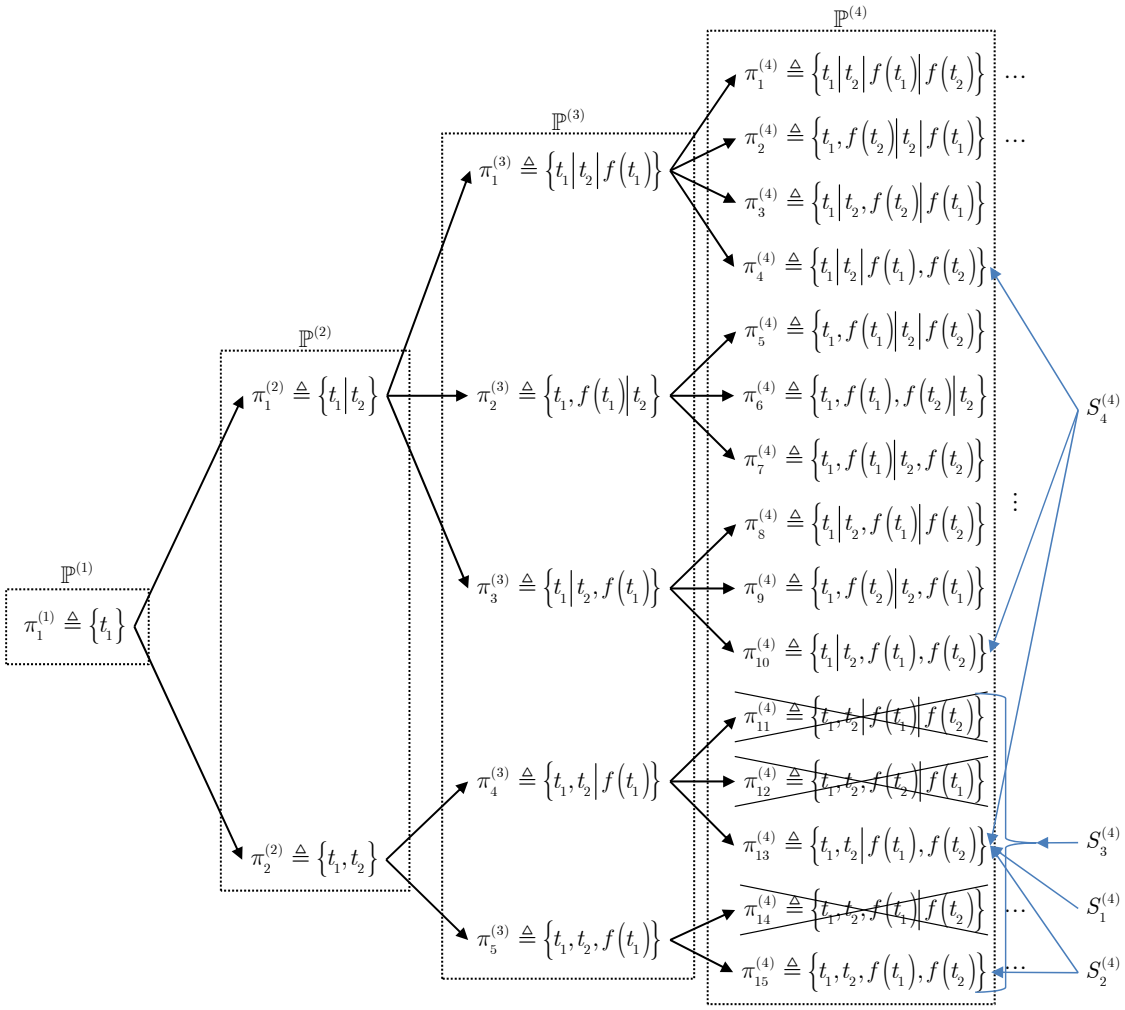


Figure 3.3: Partition Space With Impossible Partitions

In particular, we extend the definition of a consistent subset of literals as follows. A subset $C \subseteq E$ is consistent if:

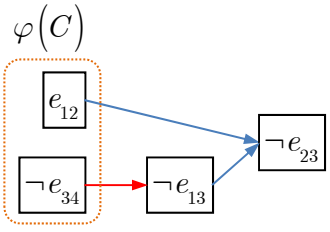
- It does not include opposing literals
- It does not include literals that violate the transitivity of equality
- It does not include literals that violate functional consistency

Partition cubes continue to be defined as consistent subsets of E and interpreted as a conjunction of the literals in that subset.

Next, the notion of a complete cube is extended to include not only the literals implied by the transitivity of equality but also those literals implied by functional consistency. The unit propagation procedure now involves both types of implication which can be interleaved to yield the desired implications. To illustrate, consider the partition space defined over $\{t_1, t_2, t_3 \triangleq f(t_1, t_2), t_4 \triangleq f(f(t_1, t_2), t_2)\}$. The functional consistency constraint can be expressed in the following equivalent ways:

$$\begin{gathered} t_1 = f\left(t_1, t_2\right) \rightarrow f\left(t_1, t_2\right) = f\left(f\left(t_1, t_2\right), t_2\right) \\ e_{13} \rightarrow e_{34} \\ \left(\neg e_{13} \vee e_{34}\right) \end{gathered} \tag{3.12}$$

Given the partition cube $C = \{e_{12}, \neg e_{34}\}$, its completion C^* is obtained by the implication sequence:



yielding

$$\varphi(C^*) = e_{12} \wedge \neg e_{13} \wedge \neg e_{23} \wedge \neg e_{34} \tag{3.13}$$

where $\neg e_{13}$ is implied by functional consistency and $\neg e_{23}$ is implied by the transitivity of equality. A variety of projections of C^* can now be easily computed:

$$\begin{aligned}
&(e_{12} \wedge \neg e_{13} \wedge \neg e_{23} \wedge \neg e_{34}) \downarrow \{t_1, t_3\} = \neg e_{13} = \neg (t_1 = f(t_1, t_2)) \\
&(e_{12} \wedge \neg e_{13} \wedge \neg e_{23} \wedge \neg e_{34}) \downarrow \{t_2, t_3\} = \neg e_{23} = \neg (t_2 = f(t_1, t_2)) \\
&(e_{12} \wedge \neg e_{13} \wedge \neg e_{23} \wedge \neg e_{34}) \downarrow \{t_1, t_2, t_3\} = e_{12} \wedge \neg e_{13} \wedge \neg e_{23} = (t_1 = t_2) \wedge \neg (t_1 = f(t_1, t_2)) \\
&(e_{12} \wedge \neg e_{13} \wedge \neg e_{23} \wedge \neg e_{34}) \downarrow \{t_3, t_4\} = \neg e_{34} = \neg (f(t_1, t_2) = f(f(t_1, t_2), t_2))
\end{aligned} \tag{3.14}$$

Finally, the notion of relative completeness is extended to account for functional consistency, which is a straightforward operation.

With these extenstions, the 2-step projection procedure described for zero-arity terms is applicable to partition spaces containing UFs.

3.1.4 Partition Spaces Involving Uninterpreted Predicates

Next, we consider partition spaces of n terms of zero- or higher-arity as well as r uninterpreted predicates (UPs) of one or more arguments. Adding r UP literals induces 2^r versions of each consistent partition. We represent each version as a conjunction of UP literals and a partition among terms. In the partition space defined over 4 terms $\{t_1, t_2, f(t_1), f(t_2)\}$, for example, the addition of two UP literals, $p(t_1)$ and $p(t_2)$, will induce four versions of each consistent partition. One of the consistent partitions, $\{t_1, t_2 \,|\, f(t_1), f(t_2)\}$, now has four different versions according to the evaluations of the two UP literals as follows:

$$\begin{aligned}
&\neg p(t_1) \wedge \neg p(t_2) \wedge \{t_1, t_2 \,|\, f(t_1), f(t_2)\} \\
&\neg p(t_1) \wedge p(t_2) \wedge \{t_1, t_2 \,|\, f(t_1), f(t_2)\} \\
&p(t_1) \wedge \neg p(t_2) \wedge \{t_1, t_2 \,|\, f(t_1), f(t_2)\} \\
&p(t_1) \wedge p(t_2) \wedge \{t_1, t_2 \,|\, f(t_1), f(t_2)\}
\end{aligned} \tag{3.15}$$

This extended space is defined on a set of terms and predicates:

$\{t_1, t_2, ..., t_n, p_1, p_1, ..., p_r\}$

so we extend $\mathbb{P}^{(n)}$ to $\mathbb{P}^{(n,r)}$, to denote the space of partitions defined on these terms and predicates. The size of the partition space defined by $\mathbb{P}^{(n,r)}$ is $h = 2^r \times B_n$. We represent each partition minterm in this partition space, $\mu_i^{(n,r)}$ where $1 \leq i \leq h$, as a conjunction of the literals in the following extended E:

$$E = \{\dot{e}_{12}, \dot{e}_{13}, ..., \dot{e}_{n-1,n}, \dot{p}_1, \dot{p}_2, ..., \dot{p}_r\} \tag{3.16}$$

where $\dot{p}_i$ is a UP literal that stands for either p_i or $\neg p_i$. That is, $\mu_a^{(n,r)} = \bigwedge_{\dot{e} \in E} \dot{e}$. A partition cube, $\varphi(C)$ is a conjunction of the literals in C where $C \subseteq E$. That is, $\varphi(C) = \bigwedge_{\dot{e} \in C} \dot{e}$. Similar to the case of UF terms, functional consistency constraints on UPs and their arguments make some of the partitions impossible. In this example, the functional constraint is $(t_1 = t_2) \rightarrow (p(t_1) = p(t_2))$, which can be expressed with the literals in E as:

$$e_{12} \rightarrow (p_1 = p_2) \tag{3.17}$$

,where p_1 and p_2 stands for $p(t_1)$ and $p(t_2)$ respectively. This constraint makes the second and the third partitions in (3.15) impossible.

With the presence of UP literals, the notion of a partial order is extended as follows: $\pi_i^{(m,q)} \leq \pi_j^{(n,r)}$ if and only if (1) every pair of terms in the same cell (different cells) of $\pi_i^{(m,q)}$ are in the same cell (different cells) of $\pi_j^{(n,r)}$, and (2) a conjunction of the UP literals in $\pi_j^{(n,r)}$ implies that of the UP literals in $\pi_i^{(m,q)}$. The projection operation is also restated as follows:

$$\begin{aligned} & S^{(n,r)}(t_1, t_2, \cdots, t_n, p_1, p_2, \cdots, p_r) \downarrow \mathbb{P}^{(m,q)} \\ & = S^{(n,r)}(t_1, t_2, \cdots, t_n, p_1, p_2, \cdots, p_r) \downarrow \{t_1, t_2, \cdots, t_m, p_1, p_2, \cdots, p_q\} \\ & = \bigcup_{\pi^{(n,r)} \in S^{(n,r)}} pred(\pi^{(n,r)}) \end{aligned} \tag{3.18}$$

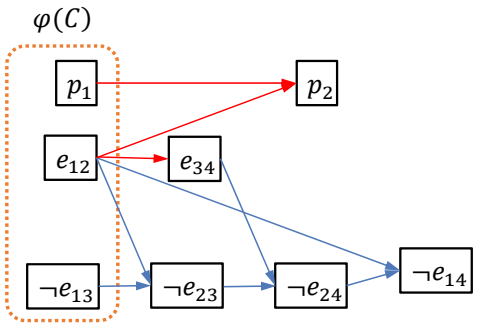
In addition, the implicit projection of a partition cube, $\varphi(C)$, on a set of terms and predicates $S \subset \{t_1, t_2, \cdots, t_m, p_1, p_2, \cdots, p_q\}$ is altered to:

$$\varphi(C) \downarrow S = \varphi(C)|_{\{\dot{e}_{ij} \in C \mid t_i \notin S \vee t_j \in S\} \cup \{\dot{p}_i \in C \mid p_i \notin S\}} \tag{3.19}$$

The procedure of unit propagation, and the definitions of a consistent subset, a complete cube, and a complete cube relative to a certain set are almost the same as those explained in Section 3.1.3; the only difference is that UPs as well as UFs must be taken into account when we consider functional consistency. In our example consisting of two UPs ($p(t_1)$ and $p(t_2)$) and 4 terms ($\{t_1, t_2, f(t_1), f(t_2)\}$), the following two functional consistency constraints need to be considered:

$$\begin{aligned} &(t_1 = t_2) \rightarrow (p(t_1) = p(t_2)) \triangleq e_{12} \rightarrow (p_1 = p_2) \triangleq \neg e_{12} \vee (p_1 \wedge p_2) \vee (\neg p_1 \wedge \neg p_2) \\ &(t_1 = t_2) \rightarrow (f(t_1) = f(t_2)) \triangleq e_{12} \rightarrow e_{34} \triangleq \neg e_{12} \vee e_{34} \end{aligned} \tag{3.20}$$

If we take the set of literals of a partition cube, $C = \{p_1, e_{12}, \neg e_{13}\}$, we can compute its completion, C^* by the following implication sequence:



yielding

$$\varphi(C^*) = e_{12} \wedge \neg e_{13} \wedge \neg e_{14} \wedge \neg e_{23} \wedge \neg e_{24} \wedge e_{34} \wedge p_1 \wedge p_2 \tag{3.21}$$

where e_{34} and p_2 are implied by functional consistency, and $\neg e_{14}$, $\neg e_{23}$, and $\neg e_{24}$ are implied by the transitivity of equality. We can compute the projection of $\varphi(C)$ on a

partition space from $\varphi(C^*)$. Here are some examples:

$$(e_{12} \wedge \neg e_{13} \wedge \neg e_{14} \wedge \neg e_{23} \wedge \neg e_{24} \wedge e_{34} \wedge p_1 \wedge p_2) \downarrow \{t_1, t_2\} = (e_{12})$$
$$(e_{12} \wedge \neg e_{13} \wedge \neg e_{14} \wedge \neg e_{23} \wedge \neg e_{24} \wedge e_{34} \wedge p_1 \wedge p_2) \downarrow \{t_1, t_2, p_1\} = (e_{12} \wedge p_1)$$
$$(e_{12} \wedge \neg e_{13} \wedge \neg e_{14} \wedge \neg e_{23} \wedge \neg e_{24} \wedge e_{34} \wedge p_1 \wedge p_2) \downarrow \{p_1, p_2\} = (p_1 \wedge p_2)$$
$$(e_{12} \wedge \neg e_{13} \wedge \neg e_{14} \wedge \neg e_{23} \wedge \neg e_{24} \wedge e_{34} \wedge p_1 \wedge p_2) \downarrow \{t_2, t_4, p_1, p_2\} = (\neg e_{24} \wedge p_1 \wedge p_2) \tag{3.22}$$

With these extensions to accommodate UPs, we can apply the 2-step projection procedure described in Section 3.1.3 to the projection of partition spaces containing both UFs and UPs.

3.1.5 Proof of Correctness

We want to show that our implicit projection procedure from $\mathbb{P}^{(n,r)}$ to $\mathbb{P}^{(m,q)}$ is correct. Without loss of generality, we can assume that
$\mathbb{P}^{(n,r)} = \{t_1, t_2, \cdots, t_m, \cdots, t_n, p_1, p_2, \cdots, p_q, \cdots, p_r\}$ and
$\mathbb{P}^{(m,q)} = \{t_1, t_2, \cdots, t_m, p_1, p_2, \cdots, p_q\}$.

We start by defining some expressions. For some a and b with $1 \leq a \leq 2^r \times B_n$ and $1 \leq b \leq 2^q \times B_m$, let $\mu_a^{(n,r)}$ and $\mu_b^{(m,q)}$ be partition minterms of $\pi_a^{(n,r)}$ and $\pi_b^{(m,q)}$ respectively, where $Pred(\pi_a^{(n,r)}) = \pi_b^{(m,q)}$. That is, $\mu_a^{(n,r)} \downarrow \mathbb{P}^{(m,q)} = \mu_b^{(m,q)}$. A partition minterm is a conjunction of literals, so we can represent $\mu_a^{(n,r)}$ and $\mu_b^{(m,q)}$ as:

$\mu_a^{(n,r)} = \bigwedge_{\dot{e} \in E_a^{(n,r)}} \dot{e}$, where $E_a^{(n,r)} = \{\dot{e}_{12}, \dot{e}_{13}, \cdots, \dot{e}_{n-1,n}, \dot{p}_1, \dot{p}_2, \cdots, \dot{p}_r\}$.
$\mu_b^{(m,q)} = \bigwedge_{\dot{e} \in E_b^{(m,q)}} \dot{e}$, where $E_b^{(m,q)} = \{\dot{e}_{12}, \dot{e}_{13}, \cdots, \dot{e}_{m-1,m}, \dot{p}_1, \dot{p}_2, \cdots, \dot{p}_q\}$.

Let $\varphi(C)$ be the partition cube of $S^{(n,r)}$ where $S^{(n,r)} = \{\pi_1^{(n,r)}, \pi_2^{(n,r)}, \cdots, \pi_h^{(n,r)}\}$. Then, $\varphi(C) = \bigvee_{1 \leq k \leq h} \mu_k^{(n,r)}$ where $\mu_k^{(n,r)}$ represents $\pi_k^{(n,r)}$. $\varphi(C^{*\mathbb{P}^{(m,q)}})$ is the complete cube induced by $\varphi(C)$ relative to the set of terms and predicates, $\mathbb{P}^{(m,q)}$.

Now, we can restate what we are seeking to prove as follows:

$$\varphi(C) \downarrow \mathbb{P}^{(m,q)} = \bigvee_{1 \le k \le h} \mu_k^{(n,r)} \downarrow \mathbb{P}^{(m,q)} = \varphi(C^{*\mathbb{P}^{(m,q)}})|_{\{\dot{e} \in E_\varphi\}}$$

$$\text{where } E_\varphi = \{\dot{e}_{ij} \in C^{*\mathbb{P}^{(m,q)}} | \, t_i \notin \mathbb{P}^{(m,q)} \vee t_j \notin \mathbb{P}^{(m,q)}\} \cup \{\dot{p}_i \in C^{*\mathbb{P}^{(m,q)}} | \, p_i \notin \mathbb{P}^{(m,q)}\}.$$

Our proof consists of two steps. First, we prove that $\mu_a^{(n,r)} \downarrow \mathbb{P}^{(m,q)} = \mu_b^{(m,q)}$ if and only if $E_b^{(m,q)} \subseteq E_a^{(n,r)}$ (or $\mu_a^{(n,r)} \rightarrow \dot{e}$ for all $\dot{e} \in E_b^{(m,q)}$).

If part: $E_b^{(m,q)} \subseteq E_a^{(n,r)}$, so $\dot{e}_{ij} \in E_a^{(n,r)}$ for all $\dot{e}_{ij} \in E_b^{(m,q)}$ where $1 \le i < j \le m$. Thus, for all pairs of terms, t_i and t_j in the same cell (different cells) of $\pi_b^{(m,q)}$ are also in the same cell (different cells) of $\pi_a^{(n,r)}$. In addition, $\dot{p}_k \in E_a^{(n,r)}$ for all $\dot{p}_k \in E_b^{(m,q)}$ where $1 \le k \le q$, so a conjunction of the UP literals in $\pi_a^{(n,r)}$ implies that of the UP literals in $\pi_b^{(m,q)}$. Therefore, $\mu_a^{(n,r)} \downarrow \mathbb{P}^{(m,q)} = \mu_b^{(m,q)}$.

Only if part: $\mu_a^{(n,r)} \downarrow \mathbb{P}^{(m,q)} = \mu_b^{(m,q)}$, so for all pairs of terms, t_i and t_j in the same cell (different cells) of $\pi_b^{(m,q)}$ are in the same cell (different cells) of $\pi_a^{(n,r)}$, where $1 \le i < j \le m$, and a conjunction of the UP literals in $\pi_a^{(n,r)}$ implies that of the UP literals in $\pi_b^{(m,q)}$ by definition. Thus, $\dot{e} \in E_a^{(n,r)}$ for all $\dot{e} \in E_b^{(m,q)}$, and $E_b^{(m,q)} \subseteq E_a^{(n,r)}$.

Second, we prove that $\varphi(C) \downarrow \mathbb{P}^{(m,q)} = \varphi(C^{*\mathbb{P}^{(m,q)}})|_{\{\dot{e} \in E\varphi\}}$. As defined at the beginning, $\varphi(C) = \bigvee_{1 \le k \le h} \mu_k^{(n,r)}$. Let $\mu_k^{(n,r)} \downarrow \mathbb{P}^{(m,q)} = \mu_k^{(m,q)}$ where $1 \le k \le h$. Let $\mu_1^{(n,r)} = \bigwedge_{\dot{e} \in E_1^{(n,r)}} \dot{e}$, where $E_1^{(n,r)} = \{\dot{e}_{12}, \dot{e}_{13}, \cdots, \dot{e}_{n-1,n}, \dot{p}_1, \dot{p}_2, \cdots, \dot{p}_r\}$. Let $\mu_1^{(m,q)} = \bigwedge_{\dot{e} \in E_1^{(m,q)}} \dot{e}$, where $E_1^{(m,q)} = \{\dot{e}_{12}, \dot{e}_{13}, \cdots, \dot{e}_{m-1,m}, \dot{p}_1, \dot{p}_2, \cdots, \dot{p}_q\}$.

$\varphi(C)$ is a conjunction of the literals in C, and $\mu_1^{(n,r)}$ implies $\varphi(C)$. Thus, $C \subseteq E_1^{(n,r)}$. In addition, $E_1^{(m,q)} \subseteq E_1^{(n,r)}$ by the first proof, so $C - (E_1^{(n,r)} - E_1^{(m,q)}) \subseteq E_1^{(m,q)}$. That is, if $\varphi(C) \rightarrow \dot{e}_{ij}$ where $1 \le i < j \le m$, then $\dot{e}_{ij} \in E_1^{(m,q)}$; and if $\varphi(C) \rightarrow \dot{p}_i$ where $1 \le i \le q$, then $\dot{p}_i \in E_1^{(m,q)}$.

For some $\dot{e} \in E_1^{(m,q)}$, $\mu_k^{(m,q)} \rightarrow \dot{e}$ if and only if $\mu_k^{(n,r)} \rightarrow \dot{e}$ by the first proof. Thus, for some $\dot{e} \in E_1^{(m,q)}$, $(\bigvee_{1 \le k \le h} \mu_k^{(m,q)}) \rightarrow \dot{e}$ if and only if $(\bigvee_{1 \le k \le h} \mu_k^{(n,r)}) \rightarrow \dot{e}$. Let

$\Omega = \{\dot{e} \,|\, \dot{e} \in E_1^{(m,q)}, \varphi(C) \to \dot{e}\}$. Then, $(\bigvee_{1 \le k \le h} \mu_k^{(n,r)}) \to \dot{e}$, which is $\varphi(C) \to \dot{e}$, can be restated as $\varphi(\Omega) \to \dot{e}$, and

$$\varphi(C) \downarrow \mathbb{P}^{(m,q)} = \bigvee_{1 \le k \le h} \mu_k^{(n,r)} \downarrow \mathbb{P}^{(m,q)} = \bigvee_{1 \le k \le h} \mu_k^{(m,q)} = \varphi(\Omega).$$

Recall that $E_\varphi = \{\dot{e}_{ij} \in C^{*\mathbb{P}^{(m,q)}} | \, t_i \notin \mathbb{P}^{(m,q)} \vee t_j \notin \mathbb{P}^{(m,q)}\} \cup \{\dot{p}_i \in C^{*\mathbb{P}^{(m,q)}} | \, p_i \notin \mathbb{P}^{(m,q)}\}$. $\Omega = C^{*\mathbb{P}^{(m,q)}} - E_\varphi$ by the definition of relative completeness, so $\varphi(\Omega) = \varphi(C^{*\mathbb{P}^{(m,q)}})|_{\{\dot{e} \in E\varphi\}}$. Therefore,

$$\varphi(C) \downarrow \mathbb{P}^{(m,q)} = \bigvee_{1 \le k \le h} \mu_k^{(n,r)} \downarrow \mathbb{P}^{(m,q)} = \varphi(\Omega) = \varphi(C^{*\mathbb{P}^{(m,q)}})|_{\{\dot{e} \in E\varphi\}}. \qquad \square$$

3.2 Abstract State Cube

In the IC3 and PDR approximate reachability algorithms, the size of the state cube derived during the reachability computation affects the overall performance significantly because it determines the size of the formula passed to a solver. In addition, the number of reachability refinements on the frontiers can be reduced exponentially if we derive a large state cube involving a small number of state literals.

At the bit level, a ternary-simulation-based approach [3] provides an efficient and effective way to enlarge a state minterm into a state cube. Suppose that a SAT solver returns a solution from the following 1-step query:

$$F_i(\mathbf{x}) \wedge T(\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{x}^+) \wedge N(\mathbf{w}^+, \mathbf{x}^+) \tag{3.23}$$

where F_i is the frontier of the ith frame, T is the transition relation, and N^+ is a state cube at the $(i+1)^{st}$ frame. This means that a state in F_i can reach one of the states in N in one transition. We need to collect a set of states including the state that can reach N. The ternary-simulation-based approach applies the values of register outputs in the solution to the original sequential circuit. It then changes one of the register outputs to X and checks to see whether this value propagates to the register inputs. If X does not appear in any of the register inputs, we can eliminate

the corresponding state variable from the state minterm derived from the solution. Otherwise, we keep the state variable. We repeat this process for all of the register outputs to obtain a state cube.

At the abstract level, however, we cannot apply this method because the solution of an EUF formula does not assign specific values to state variables. It specifies only the equality or disequality relationships among state terms. We explain this problem with our motivating example in Fig. 1.8. After deriving two datapath lemmas, we use the following query to identify a counterexample-to-induction at the first frame. We omit the two datapath lemmas for simplicity.

$$\begin{aligned}\varphi(\hat{\mathbf{w}}, \hat{\mathbf{x}}, \hat{\mathbf{x}}^+) &\equiv \hat{P}(\hat{\mathbf{w}}, \hat{\mathbf{x}}) \wedge \hat{T}(\hat{\mathbf{w}}, \hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \neg \hat{P}(\hat{\mathbf{w}}^+, \hat{\mathbf{x}}^+) \\ &= (w_{12}) \wedge (w_{12} = \neg w_9) \wedge (w_9 = \mathrm{LT}(\hat{x}_1, \hat{x}_2)) \wedge \hat{T}(\hat{\mathbf{w}}, \hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \\ &\quad (\neg w_{12}^+) \wedge (w_{12}^+ = \neg w_9^+) \wedge (w_9^+ = \mathrm{LT}(\hat{x}_1^+, \hat{x}_2^+))\end{aligned} \tag{3.24}$$

The query is satisfiable, and an SMT solver returns a solution as follows:

$$\begin{aligned}&\neg w_{12}^+ \wedge w_9^+ \wedge w_{11}^+ \wedge \mathrm{LT}(\hat{x}_1^+, \hat{x}_2^+) \wedge w_{12} \wedge \neg w_9 \wedge \neg w_7 \wedge w_8 \wedge w_{10} \wedge w_{11} \wedge \neg \mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge \\ &\{\hat{x}_1 \,|\, \mathrm{K1} \,|\, \mathrm{ADD}(\hat{x}_2, \mathrm{K1}), \hat{w}_4 \,|\, \mathrm{KMAX} \,|\, \hat{x}_2^+, \hat{w}_2, \hat{w}_5, \hat{x}_2 \,|\, \hat{x}_1^+, \hat{w}_1, \hat{w}_3, \hat{w}_6, \mathrm{ADD}(\hat{x}_1, \mathrm{K1}) \,|\, \mathrm{K0}\}\end{aligned} \tag{3.25}$$

The solution consists of a conjunction of bit assignments and a partition of terms. Thus, we cannot apply the ternary-simulation-based approach, which requires the bit assignments of all the state variables. During the reachability analysis, we need to compute a state cube from this solution, so we project this solution to the abstract state space of the system. This projection can be done by simply dropping terms and literals involving primary-input variables and dependent variables (next-state variables and local variables), yielding:

$$\neg \mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge \{\hat{x}_1 \,|\, \mathrm{K1} \,|\, \mathrm{ADD}(\hat{x}_2, \mathrm{K1}) \,|\, \mathrm{KMAX} \,|\, \hat{x}_2 \,|\, \mathrm{ADD}(\hat{x}_1, \mathrm{K1}) \,|\, \mathrm{K0}\}. \tag{3.26}$$

We then need to encode this solution information as an EUF formula. As discussed in Section 3.1, we can convert the partition into a conjunction of equalities and disequalities among every pair of terms. To derive a more compact formula, we can use the following procedure for the conversion:

1. Pick a cell representative from each cell in the partition.
2. Make every term in a cell equal to the cell representative.
3. Make every cell representative not equal to the cell representatives of other cells.

By applying this procedure, we can derive a state minterm. The state minterm is a conjunction of the bit assignments in the projected solution and the equalities and disequalities we have obtained:

$$\begin{aligned}
&\hat{C}_0(\hat{x}_1, \hat{x}_2) = \neg \mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge (\hat{x}_1 \neq \mathrm{K1}) \wedge \\
&(\hat{x}_1 \neq \mathrm{ADD}(\hat{x}_2, \mathrm{K1})) \wedge (\hat{x}_1 \neq \mathrm{KMAX}) \wedge (\hat{x}_1 \neq \hat{x}_2) \wedge (\hat{x}_1 \neq \mathrm{ADD}(\hat{x}_1, \mathrm{K1})) \wedge (\hat{x}_1 \neq \mathrm{K0}) \wedge \\
&(\mathrm{K1} \neq \mathrm{ADD}(\hat{x}_2, \mathrm{K1})) \wedge (\mathrm{K1} \neq \mathrm{KMAX}) \wedge (\mathrm{K1} \neq \hat{x}_2) \wedge (\mathrm{K1} \neq \mathrm{ADD}(\hat{x}_1, \mathrm{K1})) \wedge (\mathrm{K1} \neq \mathrm{K0}) \wedge \\
&(\mathrm{ADD}(\hat{x}_2, \mathrm{K1}) \neq \mathrm{KMAX}) \wedge (\mathrm{ADD}(\hat{x}_2, \mathrm{K1}) \neq \hat{x}_2) \wedge (\mathrm{ADD}(\hat{x}_2, \mathrm{K1}) \neq \mathrm{ADD}(\hat{x}_1, \mathrm{K1})) \wedge \\
&(\mathrm{ADD}(\hat{x}_2, \mathrm{K1}) \neq \mathrm{K0}) \wedge (\mathrm{KMAX} \neq \hat{x}_2) \wedge (\mathrm{KMAX} \neq \mathrm{ADD}(\hat{x}_1, \mathrm{K1})) \wedge (\mathrm{KMAX} \neq \mathrm{K0}) \wedge \\
&(\hat{x}_2 \neq \mathrm{ADD}(\hat{x}_1, \mathrm{K1})) \wedge (\hat{x}_2 \neq \mathrm{K0}) \wedge (\mathrm{ADD}(\hat{x}_1, \mathrm{K1}) \neq \mathrm{K0})
\end{aligned} \tag{3.27}$$

This state minterm represents a single abstract state. This kind of minterm expression easily becomes too big and complicated to be successfully handled by a solver. Thus, we need to enlarge this abstract state minterm for more compact representation. One easy way to enlarge it is to utilize the structural information of the circuit.

This approach, however, cannot be applied after the projection, because some solution information gets lost during the projection. Thus, we perform this structural generalization right after a solver returns a solution. Based on the information in the solution, the structural generalization traverses the circuit representation of the 1-step query and collects relevant EUF literals causing a violation until it reaches independent variables (primary inputs or state variables) or constants. In the case of the transition relation, $\hat{T}$, we traverse only the cone-of-influence of each next-state variable in $\neg\hat{P}^{+}$ and collect such EUF literals. For each next-state variable, we also find an equivalent term in its cone-of-influence. The resulting cube is a conjunction of the collected EUF literals and the equalities of the next-state variables and their equivalent terms. In our example, the structural generalization gives rise to the following cube:

$$\begin{aligned}\hat{C}_1(\hat{x}_1, \hat{x}_2, \hat{x}_1^+, \hat{x}_2^+) = &\mathrm{LT}(\hat{x}_1^+, \hat{x}_2^+) \wedge \neg\mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge \left(\hat{x}_1^+ = \mathrm{ADD}(\hat{x}_1, \mathrm{K1})\right) \wedge \\ &\left(\hat{x}_2^+ = \hat{x}_2\right) \wedge (\hat{x}_1 \neq \mathrm{KMAX}) \wedge (\hat{x}_1 \neq \hat{x}_2)\end{aligned} \tag{3.28}$$

From this cube, we can derive a state cube by applying the implicit projection procedure described in Section 3.1. The procedure consists of two steps. First, we compute a complete cube relative to the following set of state terms[4] and predicates[5] :

$$S = \{\mathrm{K0}, \mathrm{K1}, \mathrm{KMAX}, \hat{x}_1, \hat{x}_2, \mathrm{ADD}(\hat{x}_1, \mathrm{K1}), \mathrm{ADD}(\hat{x}_2, \mathrm{K1}), \neg\mathrm{LT}(\hat{x}_1, \hat{x}_2)\}. \tag{3.29}$$

From the uninterpreted predicate, LT, we obtain the following functional consistency

[4]In our definition, state terms include constant terms.

[5]If single-bit state variables are involved in the cube, we treat them as zero-arity predicates and apply this projection procedure.

constraint represented in two equivalent ways.

$$(\hat{x}_1^+ = \hat{x}_1) \wedge (\hat{x}_2^+ = \hat{x}_2) \rightarrow (\mathrm{LT}(\hat{x}_1, \hat{x}_2) = \mathrm{LT}(\hat{x}_1^+, \hat{x}_2^+))$$
$$\neg(\hat{x}_1^+ = \hat{x}_1) \vee \neg(\hat{x}_2^+ = \hat{x}_2) \vee (\mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge \mathrm{LT}(\hat{x}_1^+, \hat{x}_2^+)) \vee (\neg\mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge \neg\mathrm{LT}(\hat{x}_1^+, \hat{x}_2^+)) \tag{3.30}$$

This functional consistency constraint and $(\mathrm{LT}(\hat{x}_1^+, \hat{x}_2^+) \wedge \neg\mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge (\hat{x}_2^+ = \hat{x}_2))$ in $\hat{C}_1$ imply $\neg(\hat{x}_1^+ = \hat{x}_1)$. This disequality and $(\hat{x}_1^+ = \mathrm{ADD}(\hat{x}_1, \mathrm{K1}))$ in $\hat{C}_1$ imply $\neg(\hat{x}_1 = \mathrm{ADD}(\hat{x}_1, \mathrm{K1}))$. This literal involves only the terms in S, so this literal is included in the complete cube. The resulting complete cube relative to S is:

$$\hat{C}_1^{*S}(\hat{x}_1, \hat{x}_2, \hat{x}_1^+, \hat{x}_2^+) = \mathrm{LT}(\hat{x}_1^+, \hat{x}_2^+) \wedge \neg\mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge \left(\hat{x}_1^+ = \mathrm{ADD}(\hat{x}_1, \mathrm{K1})\right) \wedge \left(\hat{x}_2^+ = \hat{x}_2\right) \wedge (\hat{x}_1 \neq \mathrm{KMAX}) \wedge (\hat{x}_1 \neq \hat{x}_2) \wedge (\hat{x}_1 \neq \mathrm{ADD}(\hat{x}_1, \mathrm{K1})) \tag{3.31}$$

Second, we drop terms and literals involving primary-input variables and dependent variables (next-state variables and local variables) from this complete cube, which yields:

$$\hat{C}_2(\hat{x}_1, \hat{x}_2) = \neg\mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge (\hat{x}_1 \neq \mathrm{KMAX}) \wedge (\hat{x}_1 \neq \hat{x}_2) \wedge (\hat{x}_1 \neq \mathrm{ADD}(\hat{x}_1, \mathrm{K1})) \tag{3.32}$$

This state cube represents a set of abstract states that can reach $\neg\hat{P}$ in one transition, and it is much simpler than the simplified state minterm, $\hat{C}_0$.

If the following formula were UNSAT, we could apply minimal unsatisfiable subset (MUS) extraction to further enlarge $\hat{C}_2$ [67].

$$\psi(\hat{\mathbf{w}}, \hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \equiv C_2(\hat{x}_1, \hat{x}_2) \wedge \hat{T}(\hat{\mathbf{w}}, \hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \hat{P}(\hat{\mathbf{w}}^+, \hat{\mathbf{x}}^+) \tag{3.33}$$

However, ψ is satisfiable, so we terminate our generalization procedure. It is interesting to note that $\hat{C}_2$ can reach both $\hat{P}$ and $\neg\hat{P}$ at the same time in one transition. This means that the transitions from $\hat{C}_2$ are non-deterministic.

In this section, we applied our two-step generalization procedure, structural generalization followed by implicit projection, to derive a state cube from a 1-step query.

3.3 Feasibility Check and Datapath Refinement for Sequential Circuits

Applying the datapath abstraction and refinement framework for the verification of sequential circuits requires an extension of the combinational feasibility check and datapath refinement procedure in Section 2.5. Specifically, verifying a sequential circuit requires that we handle a multi-cycle counterexample, which is captured by a counterexample trace (CEXT). Performing an IC3-style reachability computation on the abstract term-level encoding of the circuit returns an abstract counterexample trace (ACEXT) if the given safety property is violated at the abstract level. This ACEXT must now be checked for bit-level feasibility by concretizing it to a corresponding CEXT. If the ACEXT yields a corresponding *continuous* CEXT, this bit-level trace is returned as a true bug witness to the violation of the property. Otherwise, datapath refinement (1) derives datapath lemmas to refute the spurious ACEXT by eliminating infeasible abstract states or transitions, and (2) if necessary, introduces a new state term or state predicate to increase the granularity of the abstract state space. The basic procedure for datapath refinement is the same as the one described in Section 2.5: we (1) derive an abstract-level solution cube from a structural analysis of the circuit, (2) concretize the solution cube, (3) extract MUSes if the concretized formula is unsatisfiable, and (4) derive the negation of each MUS's abstract-level counterpart. However, a major difference exists in the manner of deriving a solution cube, because our application is a sequential circuit, not a combinational circuit. That is, we need to derive a solution cube of current variables from a formula containing both current- and next-state variables. To achieve this goal, we

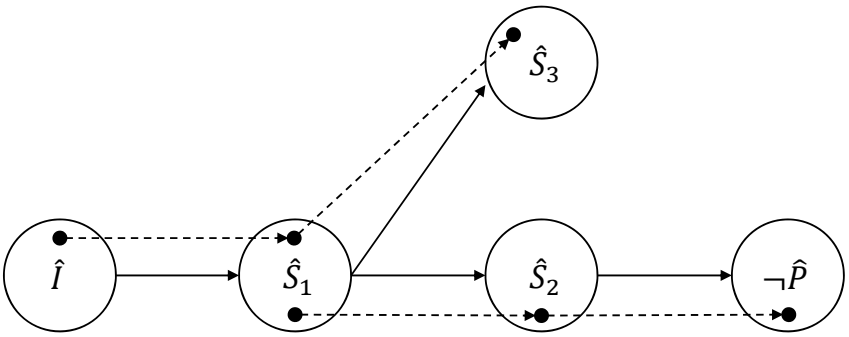


Figure 3.4: Nondeterministic Abstract Counterexample Trace

perform the structural generalization explained in Section 3.2 followed by the substitution of next-state variables with their equivalent terms and the elimination of redundant equalities. For example, from $\hat{C}_1$ in (3.28), we can obtain the following solution cube after replacing $\hat{x}_1^+$ and $\hat{x}_2^+$ with $\mathrm{ADD}(\hat{x}_1, \mathrm{K1})$ and $\hat{x}_2$ respectively and eliminating redundant equalities:

$$\hat{C}_3(\hat{x}_1, \hat{x}_2) = \mathrm{LT}(\mathrm{ADD}(\hat{x}_1, \mathrm{K1}), \hat{x}_2) \wedge \neg\mathrm{LT}(\hat{x}_1, \hat{x}_2) \wedge (\hat{x}_1 \neq \mathrm{KMAX}) \wedge (\hat{x}_1 \neq \hat{x}_2) \quad (3.34)$$

This solution cube includes only current variables and represent the conditions required for the transitions to the next states.

As explained in Section 1.2, state transitions at the abstract level can be nondeterministic. This gives rise to two problems. First, the success of an independent feasibility check on each segment of an ACEXT does not guarantee the existence of a continuous path from I to $\neg P$ at the bit level. Second, adding datapath lemmas is sometimes insufficient to refute a spurious ACEXT. We will review these two issues one by one.

Fig. 3.4 depicts a single ACEXT, $\hat{I} \rightsquigarrow \neg\hat{P}$. $\hat{S}_1$, $\hat{S}_2$, and $\hat{S}_3$ are abstract state minterms, i.e., each of them represents a single abstract state. The bit-level counterparts are represented without the circumflex accents as before. The dots in the circle are concrete states that map to the abstract state, and the solid and dashed arrows

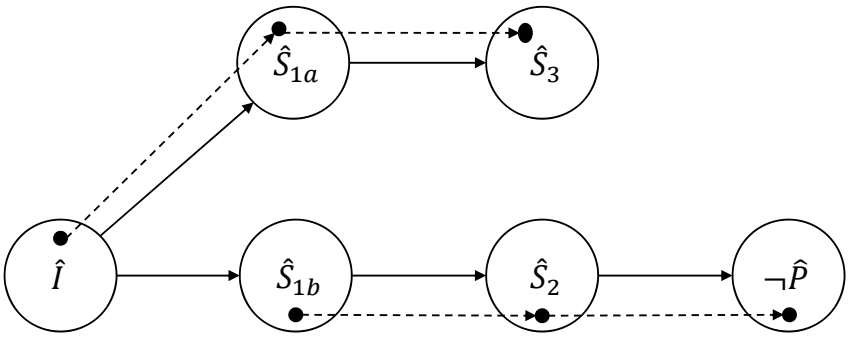


Figure 3.5: ACEXT After the Split of $\hat{A}_1$

represent the transitions between abstract states and concrete states, respectively. In the diagram, all abstract transitions have their bit-level counterparts (i.e., they are feasible abstract transitions). Therefore, the independent feasibility check on each segment in the ACEXT will succeed. As can be seen in the diagram, however, there is no continuous bit-level transition from I to $\neg P$. This is the first problem caused by the non-deterministic behavior, and this happens because the concrete state that goes to S_2 is different from the concrete state that comes from I. If the next state of $\hat{S}_1$ were deterministic (i.e., $\hat{S}_1$ goes only to $\hat{S}_2$), this problem would not happen, because all the concrete states in S_1 would go to the states in S_2.

We will now explain how this problem affects the feasibility check of an ACEXT. In practice, an abstract-level reachability computation returns a sequence of state cubes (i.e., not state minterms) that represents a set of ACEXTs. For example, it returns $< \hat{C}_{S1}, \hat{C}_{S2} >$ for the 3-step ACEXT in Fig. 3.4. $\hat{C}_{S1}$ and $\hat{C}_{S2}$ cover $\hat{S}_1$ and $\hat{S}_2$ respectively, and they satisfy the following formulas[6]:

$$\hat{C}_{S2}(\hat{\mathbf{x}}) \wedge \hat{T}(\hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \neg\hat{P}(\hat{\mathbf{x}}^+) \tag{3.35}$$

$$\hat{C}_{S1}(\hat{\mathbf{x}}) \wedge \hat{T}(\hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \hat{C}_{S2}(\hat{\mathbf{x}}^+) \tag{3.36}$$

$$\hat{I}(\hat{\mathbf{x}}) \wedge \hat{T}(\hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \hat{C}_{S1}(\hat{\mathbf{x}}^+) \tag{3.37}$$

[6]In this section, we assume no input in the system for simplicity.

The above formulas are all satisfiable at the bit level, but this does not guarantee the existence of a continuous CEXT as illustrated in Fig. 3.4. The granularity of state cubes is not fine enough to distinguish two sets of states heading to different sets of states, because a state cube is the projection of a solution cube on the abstract state space. A solution cube, on the other hand, can distinguish those sets of states due to the terms or predicates introduced during the substitution of its derivation process. Thus, we accumulatively derive solution cubes, $\hat{M}_2(\hat{\mathbf{x}})$, $\hat{M}_1(\hat{\mathbf{x}})$, and $\hat{M}_0(\hat{\mathbf{x}})$ from the following formulas respectively.

$$\hat{C}_{S2}(\hat{\mathbf{x}}) \wedge \hat{T}(\hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \neg \hat{P}(\hat{\mathbf{x}}^+) \tag{3.38}$$

$$\hat{C}_{S1}(\hat{\mathbf{x}}) \wedge \hat{T}(\hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \hat{M}_2(\hat{\mathbf{x}}^+) \tag{3.39}$$

$$\hat{I}(\hat{\mathbf{x}}) \wedge \hat{T}(\hat{\mathbf{x}}, \hat{\mathbf{x}}^+) \wedge \hat{M}_1(\hat{\mathbf{x}}^+) \tag{3.40}$$

We then perform feasibility checks on the solution cubes[7]. For example, we perform the feasibility check of the solution cube, $\hat{C}_3$, in (3.34) by checking the satisfiability of its corresponding bit-vector formula:

$$((x_1 + \text{`W'd1}) < x_2) \wedge \neg(x_1 < x_2) \wedge \neg(x_1 = \text{`W'b11...1}) \wedge \neg(x_1 = x_2), \tag{3.41}$$

where W is the bit width of x_1 and x_2. We obtain this formula by simply replacing uninterpreted terms and predicates in $\hat{C}_3$ with their bit-vector counterparts. The satisfiability of this formula is checked by an SMT solver. In this case, the SMT solver returns that no assignment of x_1 and x_2 satisfies the formula, from which we conclude that $\hat{C}_3$ is infeasible.

Once a set of ACEXTs is identified as infeasible traces, we need to derive a datapath lemma to refute them, which can be done by extracting MUSes from the bit-level

[7]In fact, the feasibility check on M_0 is enough, because it checks the existence of a continuous path from $\hat{I}$ to $\neg\hat{P}$. However, it is better to check M_2, M_1, and M_0 incrementally, because most spurious traces are found from the earlier checks.

counterpart of the solution cube. The datapath lemma is the negation of each MUS's abstract-level counterpart[8]. This datapath lemma eliminates either infeasible abstract states or transitions. In some cases, there can be no infeasible one to eliminate. This is the second problem caused by a non-deterministic state transition. In the ACEXT illustrated in Fig. 3.4, there are no infeasible ones to eliminate even though the trace is spurious. This happens because the granularity of the abstract state space is too coarse to eliminate the spurious ACEXT. Thus, we need to increase the granularity by introducing a new state term or state predicate so that we can distinguish the two concrete states in $\hat{S}_1$. This is done by investigating state terms or state predicates in solution cubes that are not present in their corresponding state cubes. Once $\hat{S}_1$ is split into two abstract states, the non-deterministic behavior is gone and we can refute the spurious ACEXT by eliminating the infeasible transition from $\hat{I}$ to $\hat{S}_{1b}$, as illustrated in Fig. 3.5.

3.4 The Averroes Algorithm

Averroes takes an input a hardware design and a property written in Verilog-HDL. They are parsed with a commercial Verilog-HDL parser, Verific [68], and then passed to a datapath abstraction and refinement framework, **DP-CEGAR**, which integrates an IC3/PDR-style reachability computation, **Reach-CEGAR**. In this section, we provide the detailed procedures of **DP-CEGAR** and **Reach-CEGAR**. We then conclude this section by proving the convergence of these procedures.

3.4.1 DP-CEGAR

Fig. 3.6 illustrates the high-level pseudo-code of **DP-CEGAR**. The initial datapath abstraction is performed by **DP-Abstract** which returns EUF formulas of

[8]We apply additional two processes to derive more powerful datapath lemmas, which will be explained in Chapter V.

```
1.  trace DP-CEGAR(T, I, P){
2.      (T̂, Î, P̂) = DP-Abstract(T, I, P);
3.      Δ = 1; // Initialize datapath lemmas
4.      while(true){
5.          ACEXT = Reach-CEGAR(T̂, Î, P̂, Δ);
6.          if empty(ACEXT)
7.          then return empty trace; //P holds
8.          CEXT = DP-Concretize(ACEXT);
9.          if Feasible(CEXT)
10.             then return CEXT; //P fails
11.             else Δ = Δ ∧ DP-Refine;
12.     }
13. }
```

Figure 3.6: High-Level Pseudo Code for CEGAR-Based Datapath Abstraction

the bit-level transition, initial, and property formulas (line 2) by, basically, replacing wide datapath signals with uninterpreted terms, and datapath operators and predicates with, respectively, uninterpreted functions and predicates. Single-bit control signals are not abstracted [61][9]. The abstract formulas are overapproximations of the bit-level versions and, thus, represent a sound abstraction. The procedure then initializes Δ (line 3) which serves as a database of derived datapath lemmas. The reachability computation is carried out by calling **Reach-CEGAR** (line 5) that operates on the abstract formulas. Note, in particular, that this version of **Reach-CEGAR** takes as a fourth argument a formula representing the learned datapath lemmas which it augments to all the queries it performs. If **Reach-CEGAR** returns an empty trace, **DP-CEGAR** terminates with the conclusion that the property holds (line 7). However, if **Reach-CEGAR** returns a non-empty abstract counterexample trace (ACEXT), a concrete bit-level version counterexample trace (CEXT) is constructed by **DP-Concretize** (line 8) and checked for feasibility (line 9). If found to

[9]For better classification of datapath and control, we supplement this classification method by applying two more sophisticated techniques that will be discussed in Chapter IV.

be feasible, CEXT is returned as a witness for the violation of the property (line 10). Otherwise, the ACEXT returned by **Reach-CEGAR** are concretized and checked for feasibility one transition at a time. Each infeasible state or transition in a counterexample triggers the generation of one or more datapath lemmas using a simplified version of the MUS extractor in [69]. If the abstract state space is not fine enough to refute the spurious ACEXT, a new state term or -predicate is introduced. Feasibility checking is done using the bit-vector theory in the Yices (version 1.0.35) SMT solver [12].

3.4.2 Reach-CEGAR

Fig. 3.7 highlights the major steps of the approximate reachability computation in **Reach-CEGAR**. The formulas processed by **Reach-CEGAR** are all in EUF and all reasoning is done using the Yices SMT solver. The procedure utilizes a queue Q of proof obligations each of which is a pair $(\hat{c}(\hat{\mathbf{x}}), k)$ where $\hat{c}(\hat{\mathbf{x}})$ is a state cube and k is a frame number. The following numbered list corresponds to the numbered boxes in Fig. 3.7:

1. At the start of major iteration k, frame k is overapproximated to $\hat{P}$ ($\hat{F}_k = \hat{P}$). The iteration then repeatedly checks for counterexamples to induction (CTIs) using the function 1-step which calls the SMT solver with the query:
   $\hat{F}_k \wedge \hat{T} \wedge \Delta \wedge \neg \hat{P}^+$
2. A satisfying solution $\hat{m}(\hat{\mathbf{x}}) \in \hat{F}_k(\hat{\mathbf{x}})$ to this query indicates a CTI that must now be checked for reachability from $\hat{I}(\hat{\mathbf{x}})$. Before proceeding with that check, however, the state minterm derived from the solution is "expanded" to a state cube. The detailed procedure can be found in Section 3.2. The enlarged cube $\hat{c}$ is now added to the Q as a proof obligation in frame k, meaning "can $\hat{c}$ be eliminated from F_k by showing that it is unreachable from I along paths whose length is at least k?"

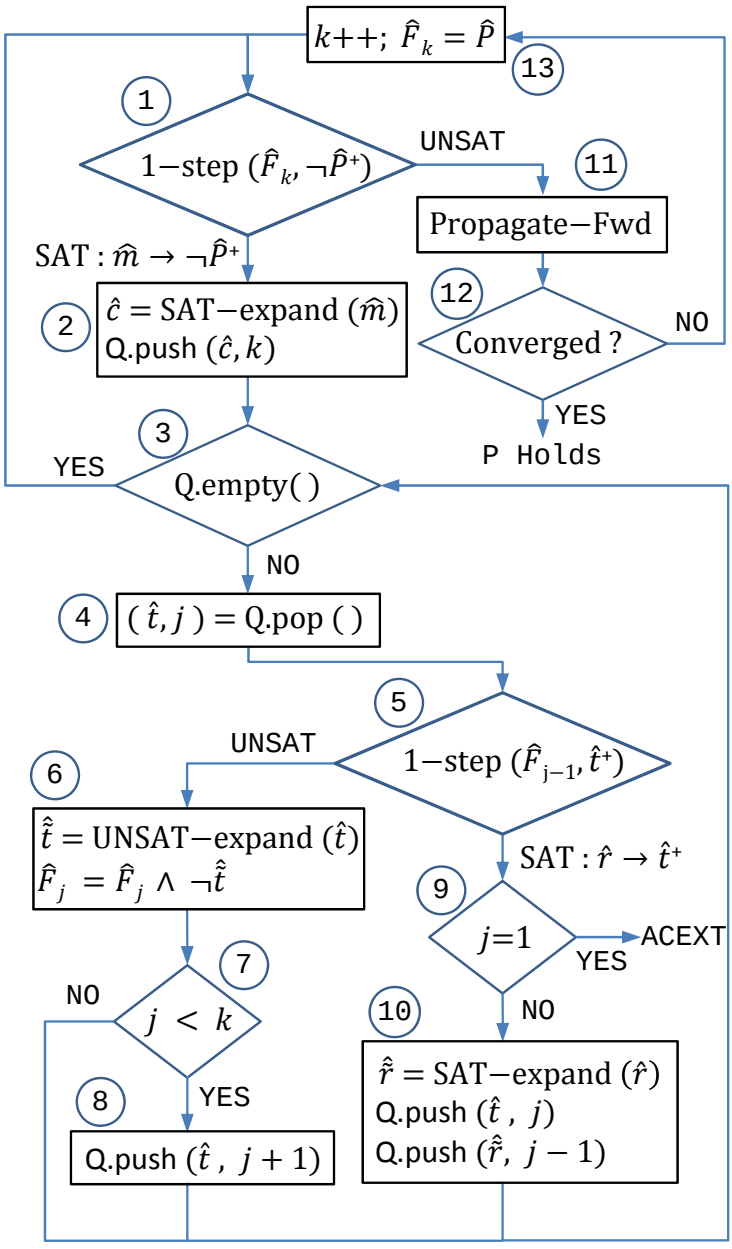


Figure 3.7: Implementation of **Reach-CEGAR** in the Averroes Verifier

3. An empty queue signifies that the current CTI has been successfully eliminated and the algorithm proceeds to check for the existence of another CTI from the current frame.
4. The reachability computation starts here by retrieving a proof obligation $(\hat{t}, j)$ from the queue.
5. The 1-step function checks the formula $\hat{F}_{j-1} \wedge \hat{T} \wedge \Delta \wedge \hat{t}^{+} \wedge \hat{P}^{+}$ to determine if $\hat{t}$ can be reached in one step from frame $j-1$.
6. If $\hat{t}$ is not reachable from frame $j-1$, it is enlarged to $\hat{\hat{t}}$ by extracting one or more MUSes from the UNSAT formula in step 5. The negation of $\hat{\hat{t}}$ is now added as a refinement clause to frame j (which means that all frames $1 \leq i \leq j$ are tightened as a result of the unreachability of $\hat{t}$ in frame j).
7. The processing of cube $\hat{t}$ terminates if we reach the last frontier k.
8. Otherwise, $\hat{t}$ is added as a proof obligation in frame $j+1$. This step is optional but, as pointed out in [3], it helps to improve performance and to find ACEXTs that are longer than $k+1$.
9. If the current proof obligation is $(\hat{t}, 1)$ and $\hat{t}$ is found to be reachable from frame 0, then we have found an ACEXT and the procedure terminates.
10. If $\hat{t}$ in frame j is found to be reachable (in one step) from frame $j-1$, the satisfying solution $\hat{r}$ to the query in step 5 is enlarged similarly to how $\hat{m}$ was enlarged in step 2. Processing continues by re-inserting $(\hat{t}, j)$ into the queue and adding $(\hat{\hat{r}}, j-1)$ as a new proof obligation.
11. When all CTIs from the current frontier k have been eliminated, refinement clauses from earlier iterations are checked to see if they can be moved forward to tighten later frontiers. A refinement clause $\omega \in \hat{F}_j$, $1 \leq j \leq k$ that causes the query $\hat{F}_j \wedge \hat{T} \wedge \Delta \wedge \neg\omega^{+} \wedge \hat{P}^{+}$ to be UNSAT indicates that cube $\neg\omega$ in frame $j+1$ is unreachable in one step from frame j and can thus be eliminated from frame $j+1$. This is accomplished by propagating clause ω forward: $\hat{F}_{j+1} = \hat{F}_{j+1} \wedge \omega$.

12. The procedure terminates proving that $\hat{P}$ holds if two successive frames become equal, i.e., if $\hat{F}_j = \hat{F}_{j+1}$ for some $1 \leq j \leq k$. This check is equivalent to finding the clause set associated with frame j has become empty.
13. Otherwise, a new frame is created and initialized to P and the procedure continues to check for CTIs corresponding to longer ACEXTs.

3.4.3 Proof of Convergence

Averroes is guaranteed to terminate. We will prove the termination of **Reach-CEGAR** and **DP-CEGAR** in turn. **Reach-CEGAR** in Averroes performs the reachability computation in a finite abstract state space that is fixed during the computation, so its termination proof is similar to the proof of its bit-level counterpart. At the k-th iteration, at least one abstract state is removed from one of k frontiers whenever a CTI is found. Thus, the k-th iteration must terminate after a finite number of CTI checks. At the end of the iteration, **Reach-CEGAR** is in one of the following three cases:

1. An abstract-level counterexample trace is found.
2. Two successive frontiers become equal.
3. No additional CTI is found.

In the first two cases, **Reach-CEGAR** terminates. In the last case, **Reach-CEGAR** increases k and repeats the main loop. In this case, each frontier must contain at least one more abstract state than the other frontiers in the earlier frames (otherwise, two successive frontiers would have become equal during the forward propagation in step 11 of **Reach-CEGAR**). Thus, k cannot be greater than the number of abstract states in the abstract state space, and the main loop will terminate in a finite number of iterations.

Next, we will prove the convergence of **DP-CEGAR**. The while loop in **DP-CEGAR** repeats only when **Reach-CEGAR** returns a spurious counterexample

trace. In this case, **DP-CEGAR** calls **DP-Refine**, which conducts one of the following tasks:

1. To increase the granularity of the abstract state space.
2. To eliminate infeasible abstract states.
3. To eliminate infeasible abstract transitions.

The first task is performed when we need to split a feasible abstract state that corresponds to more than one concrete state, so the number of repetitions of the task is bounded by the number of concrete states. Thus, the maximum number of abstract states and transitions in the finest abstract state space is finite. This leads to the finite number of the repetition of the last two tasks. Each elimination task deletes at least one abstract state or transition, so the number of repetitions is bounded by the maximum number of abstract states and transitions, which is finite. Therefore, the while loop in **DP-CEGAR** iterates for a finite number of times, and Averroes terminates. □

CHAPTER IV

Structural Abstraction

Our abstraction framework is called *structural abstraction* because it preserves the structure of the concrete system and makes it easy to map abstract components to their concrete counterparts and vice versa. It consists of two parts: advanced datapath abstraction and memory abstraction. In Section 4.1, we investigate smart ways to abstract irrelevant details in a hardware design to improve the scalability of the verification task. A conventional datapath abstraction approach classifies each circuit component into datapath and control according to the width of the component without any further analysis. This often leads to (a) the misclassification of datapath and control and (b) unnecessary terms or literals in the resulting abstract formula. We demonstrate these problems by analyzing datapath lemmas derived during our reachability computation with the conventional abstraction strategy. To solve those problems, we introduce advanced datapath abstraction which (a) decomposes unnecessarily concatenated signals and (b) applies a property-directed concretization to a small part of the hardware design that is misclassified as a datapath component. In Section 4.2, we describe a way to handle a large memory or array of vectors in an efficient way. One easy way to handle a memory would be to treat it as a set of registers. For example, we can convert a memory with 1024 entries of 32 bits each into 1024 individual 32-bit registers. Obviously, this conversion does not scale to a

large memory. Therefore, we instead abstract a large memory by replacing it with a *represented slot* incorporating additional circuits. We then add more slots if one slot is insufficient for our reachability computation.

4.1 Advanced Datapath Abstraction

In this section, we provide the results of our analysis of the datapath lemmas that we have collected. We then present our advanced datapath abstraction approach.

4.1.1 Patterns of Datapath Lemmas

If we abstract some control components that are directly related to a given control-centric property, we end up with many iterations of datapath refinement. Datapath lemmas derived during datapath refinements introduce constraints among misclassified control components to compensate for any information missing as a result of abstraction. Therefore, it is often useful to analyze datapath lemmas to diagnose problems in the abstraction strategy. In this section, we analyze the patterns of the datapath lemmas that were derived during our reachability computation with a conventional datapath abstraction strategy from a set of large-datapath industrial benchmarks.

The datapath lemmas can be classified into three categories based on the types of terms in the datapath lemmas. The datapath lemmas in the first category, *constant*, include only constant terms. These datapath lemmas represent an equality or disequality relationship among constant terms. The datapath lemmas in the second category include the uninterpreted functions (UFs) and uninterpreted predicates (UPs) of bit-field extraction and concatenation operators. The second category can be divided into *bit-select* and *part-select* according to the output bit width of the extraction operators involved (*bit-select* if the bit width is one, and *part-select* otherwise). The other datapath lemmas are classified as *others*. These datapath lemmas

Table 4.1: Proportions of Four Types of Datapath Lemmas

Type	Example (Corresponding Verilog expression)	# Lemmas	Proportion (%)
Constant	3'd6 != 3'd0	21	0.65
Bit-select	{a, b, 1'b1}[0]	2554	78.97
Part-select	!(m = n) || ({a, b, m, c}[5:3] == {d, e, n, f}[5:3])	577	17.84
Others	8d0 + 8d1 + 8d1 = 8d2	82	2.54
Total		3234	100

Table 4.2: Composition of Datapath Operators in a Set of Large Industrial Benchmarks

	Bit-Select	Part-Select	Concatenation	Others	Total
# Operators	35276	56739	89663	51833	233511
Proportions	15.11	24.3	38.4	22.19	100

constrain the UFs and UPs of other types of operators such as additions and multiplications. Table 4.1 lists these four different types of datapath lemmas and their proportions in a set of datapath lemmas collected from 211 large-datapath industrial benchmarks. The table includes an example for each type. Datapath lemmas are EUF formulas, but we represent them in the table as Verilog expressions to make them easier to understand.

As can be seen in the table, *bit-select* accounts for about 79% of the set of datapath lemmas. This is a very high percentage in contrast with the proportion of bit-select operators, which is about 15% as can be seen in Table 4.2. This kind of datapath lemma is introduced when control signals are concatenated together to simplify the RTL description of a hardware design. For example, an instruction register is usually a concatenation of register addresses and some control bits. Thus, individual control bits are represented by extracted signals of the instruction register, which are abstracted as uninterpreted predicates in a conventional datapath abstraction approach. That is, control signals are abstracted due to misclassification. In this case, many datapath lemmas are required to constrain the uninterpreted predicates especially when their corresponding control signals are involved in a given control-centric

property. The same thing happens to control logic circuits. They are sometimes described as datapath operations on concatenated signals, so they are abstracted in a conventional approach. This also causes many iterations of datapath refinements followed by many datapath lemmas.

Part-select makes up the second largest portion. This kind of datapath lemma is also introduced by concatenated signals. In the earlier example, the instruction register contains register addresses, and part-select operators are used to access the registers. The datapath lemmas in this category have two interesting features. First, most of them are implications. Like the example in the table, the datapath lemmas in this category assert that the value of a register remains unchanged even after the register goes through concatenation and extraction operations. Thus, it is natural for them to be expressed as implications. Second, concatenation operators are involved in all of the datapath lemmas. Extraction (bit-select and part-select) operators are closely related with concatenation operators, because extractions are usually applied to concatenated signals. Thus, most of the datapath lemmas in *part-select* also involve concatenation operators. As a result, concatenation operators are involved in more than 90% of the entire set of datapath lemmas.

In conclusion, an unnecessarily concatenated signal results in the misclassification of datapath and control on the individual signals concatenated in the signal as well as the control logic circuits connected to them. They also introduce many unnecessary extraction and concatenation operators, which are abstracted as uninterpreted functions or predicates in a conventional datapath abstraction approach. These problems lead to an inefficient abstract formula, which yields many datapath lemmas in the categories of *bit-select* and *part-select*. This is why more than 95% of the datapath lemmas are classified in those categories. To solve this problem, we developed two techniques: circuit components decomposition so as to reduce the number of datapath lemmas involving extraction and concatenation operators; and property-directed

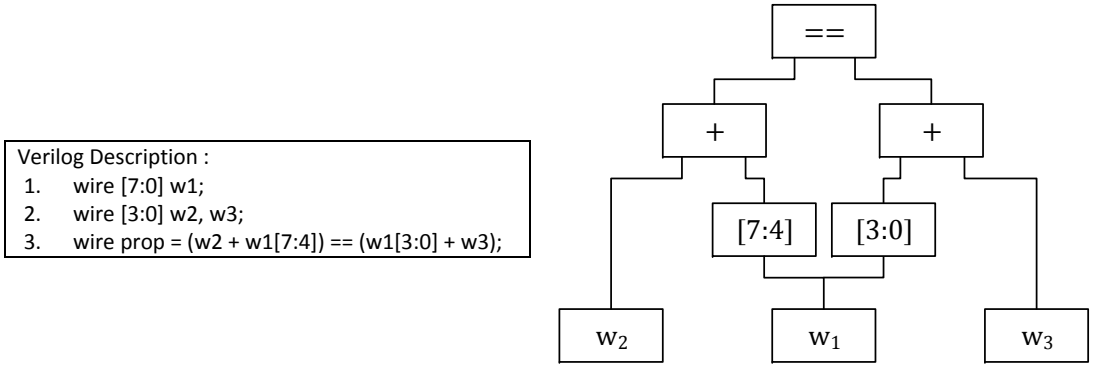


Figure 4.1: Motivating Example of the Optimization Technique

concretization in order to correct the misclassification of circuit components that play an important role in verifying a property.

4.1.2 Circuit Components Decomposition

An important contribution of Vapor [61] is that it pointed out the benefit of eliminating unnecessary concatenation and extraction operators in the design. In an abstract formula, they are abstracted as uninterpreted functions or predicates like other datapath operators, so they can cause spurious counter-examples followed by many iterations of datapath refinement. As demonstrated in Section 4.1.1, more than 95% of datapath lemmas derived from industrial benchmarks involve the abstract-level counterparts of one of these two operators. To reduce the number of such operators, Vapor decomposes signals in a formula into multiple pieces if the accesses to them are mutually disjoint. For example, the eight-bit signal, w_1 in Fig. 4.1, can be decomposed into two 4-bit signals, w_{1a} and w_{1b} which are the same as $w_1[7:4]$ and $w_1[3:0]$ respectively. Then, we can eliminate unnecessary extraction operators involved. When the resulting partition includes single-bit signals, they are treated as terms (i.e., abstracted) even though they are single-bit components.

The approach of Vapor is intended for a combinational (or unrolled) circuit, which is different from our target system, a sequential circuit, and Vapor does not have a

systematic procedure for the optimization. Therefore, we developed a systematic way of eliminating unnecessary concatenation and extraction operators that can be applied to a sequential circuit. We apply this technique during preprocessing, i.e., before applying datapath abstraction.

Fig. 4.2 shows the pseudo-code of our decomposition algorithm. We represent T and P as directed graphs consisting of nodes and edges. Each circuit component in T and P corresponds to a node in the graphs. For our decomposition algorithm, we introduce a *decomposition index list*, $N.indices$, for each node, which is initialized to $\{width, 0\}$. The list indicates how to decompose a node linked to the list. For example, a node M with a decomposition index list of $\{width, b, a, 0\}$, is decomposed into three pieces: $M[width - 1 : b]$, $M[b - 1 : a]$, and $M[a - 1 : 0]$. To construct decomposition index lists, we (1) traverse the graph of P from the node of a *prop* signal to sources (i.e., the nodes of primary inputs and state variables) and (2) update the decomposition index lists of every node we visit (lines 2 to 3). We then traverse the graph of T from the nodes of next-state variables to sources (lines 6 to 7).

$ConstructIndexes()$ (lines 15 to 34) fills in the index lists. This is done recursively. Suppose that $ConstructIndexes()$ is called from a direct successor, we insert every element in the list of the successor to the list of the current node (line 17). Once all the successors are traversed (line 18), we create a temporary decomposition index list, B, according to the type of the node and pass it to direct predecessors. If it is a concatenation operator (lines 20 to 24), we adjust the index list for each child node. If it is an extraction operator of $[msb\_index : lsb\_index]$ (lines 25 to 29), we increase every index in the list by lsb_index and put $msb\_index + 1$ and lsb_index into the list. We then call $ConstructIndexes()$ with the updated list for every direct predecessor. If the decomposition index lists of current-state variables have updated in $ConstructIndexes()$, We need to synchronize the lists of the current- and next-state variables of the same register after (lines 4 and 8) so as to decompose the two nodes

```
1.  (G_T,G_P) Decompose(G_T,G_P) {
2.      P = Node in G_P representing a property;
3.      ConstructIndices(P, ∅);
4.      Synchronize the indices of state variables;
5.      for ( i = 0; i < param_depth; i++ ) {
6.          for ( each node representing a next-state variable N in G_T )
7.              ConstructIndices(N, ∅);
8.          Synchronize the indices of state variables;
9.      }
10.     for ( each node representing a next-state variable N in G_T )
11.         G_T = G_T ( N ↦ DecomposeNode(N) );
12.     G_P = G_P (P ↦ DecomposeNode(P) );
13.     return (G_T,G_P);
14. }
15. ConstructIndices(N, S) {
16.     W = 0;
17.     N.indices = N.indices ∪ S;
18.     if(All the successors of N have been visited)
19.         for ( each direct predecessor N_P of N ) {
20.             if(N.type == Concatenation) {
21.                 for ( each index i in S )
22.                     if ( W + N_P.width <= i )      break;
23.                     else      B = B ∪ (i-W);
24.                 W = W + N_P.width;
25.             } else (N.type == Extraction ) {
26.                 (M, L) = MSB and LSB indices of the extraction;
27.                 for ( each index i in N.indices )
28.                     B = B ∪ (i+L);
29.                 B = B ∪ {M+1, L};
30.             } else      B = N.indices;
31.             if (N_P.width == 1)      ConstructIndices(N_P, ∅);
32.             else                     ConstructIndices(N_P, B);
33.         }
34. }
35. Nodes DecomposeNode(N){
36.     if ( N.type == Signal or Constant ) {
37.         return decomposed elements of N based on N.indices;
38.     } else {      // operators
39.         for ( each direct predecessor N_P of N )
40.             D = D ∪ DecomposeNode(N_P);
41.         if (N.type == Extraction ) {
42.             return extracted set of nodes from D;
43.         } else if (N.type == one of concatenation, conditional,
                    equality, reduction, or bit-wise operators ) {
44.             return Node incorporating decomposed nodes D
                        , which is functionally equivalent to N;
45.         } else {  // one of arithmetic, relational, and shift operators
46.             for ( each entry d in D )
47.                 C = C ∪ Node of a concatenation of d;
48.             N_t = Node of N.type with the direct predecessors of C;
49.             return decomposed elements of N_t based on N.indices;
50.         }
51.     }
52. }
```

Figure 4.2: Node Decomposition Algorithm

in the same way, because they are connected to the same register. This synchronizing process may require another updating process (lines 5 to 9), because the changes in the decomposition index lists of the nodes of next-state variables can affect the index lists of their predecessors. We can repeat the two processes of updating and synchronizing index lists until there is no further change anymore, but since this often leads to excessive decomposition, we repeat the procedure just once (i.e. $param\_depth$ is currently set to be one). Depending on the target transition relation and property, the best choice of the parameter may vary, but our default value of one led to the best results in our set of benchmarks.

After constructing decomposition index lists, we decompose each node based on its index list (lines 10 to 12). If a node's type is a signal or a constant (lines 36 to 37), we decompose it into multiple pieces based on its decomposition index list. For example, we decompose an 8-bit signal, x, with the decomposition index list, $\{8, 4, 1, 0\}$, into three pieces:

x_a $(= x[7:4])$, x_b $(= x[3:1])$, and x_c $(= x[0:0])$.

In the case of an operator node, we decompose its child nodes (lines 39 to 40) and then process the decomposed nodes according to its operation type recursively (lines 41 to 51). For an extraction operator (lines 41 to 42), we apply the extraction operation to the decomposed nodes and return a concatenation of the extracted nodes. For concatenation, conditional, equality, reduction, or bit-wise operators operators (lines 43 to 44), we decompose the operator node into multiple nodes and put a proper operator on top of the decomposed nodes in order to make the resulting compound node functionally equivalent to the original operator node. For example, the equality of two wide signals is converted into a conjunction of the equalities of each decomposed element pair. During the decomposition procedure, we eliminate the following unnecessary datapath operators.

- Extraction of entire bits

e.g. replace $A\,[\,(\text{bit width of } A) - 1 \,:\, 0\,]$ with A

- Concatenation of a single component

  e.g. replace $\{A\}$ with A

For other operators such as arithmetic, relational, and shift operators (lines 46 to 49), we do not decompose the operator node at this stage, because the circuit we would have to introduce to derive a functionally equivalent node from the decomposed operator nodes is very large and complicated[1]. Thus, we keep the operator node intact, and we put concatenation and extraction operators before and after the operator to connect it with decomposed child nodes (lines 46, 47 and 49). Suppose that there is an addition operator and our decomposition technique decomposes its child and parent nodes. Instead of decomposing the addition operator node, we insert (1) a concatenation operator to combine decomposed child nodes to connect them with the wide addition operator node (lines 46 to 47) and (2) extraction operators to the operator node (line 49) to connect it with multiple parent nodes that have been decomposed.

Our decomposition process improves on the approach of Vapor in two aspects without sacrificing the advantages of Vapor. First, we treat the single-bit FFs decomposed from multi-bit registers as control components. Vapor abstracts the single-bit FFs, but this abstraction is often not helpful in reducing the complexity of abstract formulas. Second, our decomposition process is more aggressive in that we decompose circuit components even when the accesses to them are not mutually disjoint. Vapor does not decompose the components in that case. Our aggressive approach offers more opportunities to eliminate unnecessary concatenation operators. In Fig. 4.3, we can reduce the number of datapath operators from six to three, which is not possible with Vapor because the accesses to the topmost concatenation operator are not mutually disjoint.

[1]It is better to concretize the operator instead. We decide whether to concretize this kind of operator during our property-directed concretization.

```
Verilog Description :
 1.  wire [15:0] w1, w2, w3;
 2.  wire [7:0] w4, w5;
 3.  wire w6;
 4.  wire [7:0] w7, w8, w9, w10;
 5.  wire [15:0] temp1;
 6.  wire [31:0] temp2;
 7.  assign temp1 = (w6 ? {w7, w8} : {w9, w10});
 8.  assign temp2 = {w4, temp1, w5};
 9.  assign w1 = temp2[31:16];
10.  assign w2 = temp2[23:8];
11.  assign w3 = temp2[15:0];
```

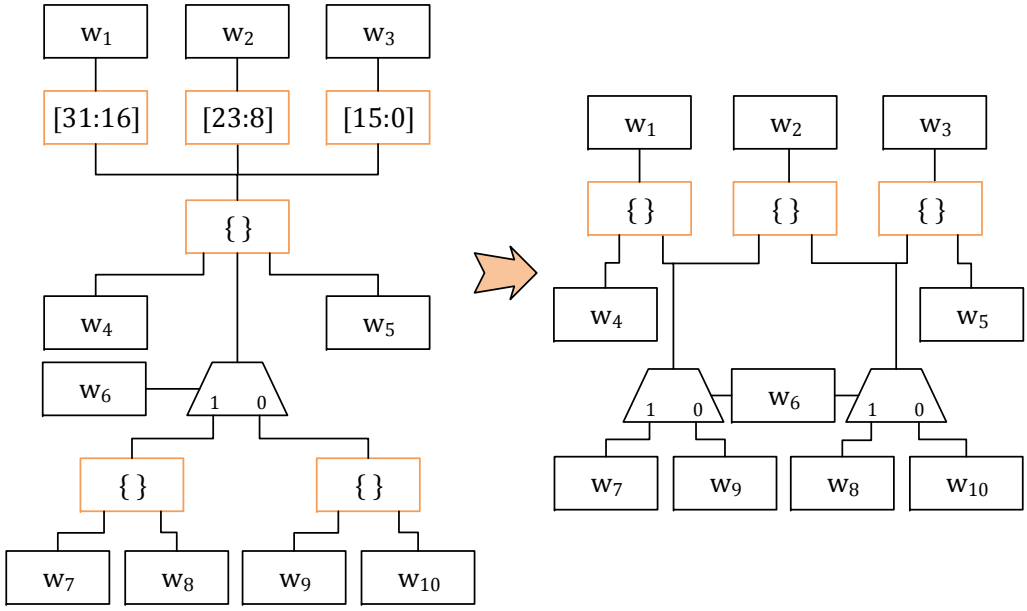


Figure 4.3: Benefit of the New Aggressive Decomposition Algorithm

```
Verilog Description :
1.    input wire clk;
2.    reg [7:0] x;
3.    wire y1, y2;
4.    wire [3:0] y3;
5.    always @(posedge clk)
6.              x <= y1 ? (y2 ? 8'd0 : x) : {4'd1, (y3 + x[7:4])};
7.    wire prop = (x[3:0] == 4'd0);
```

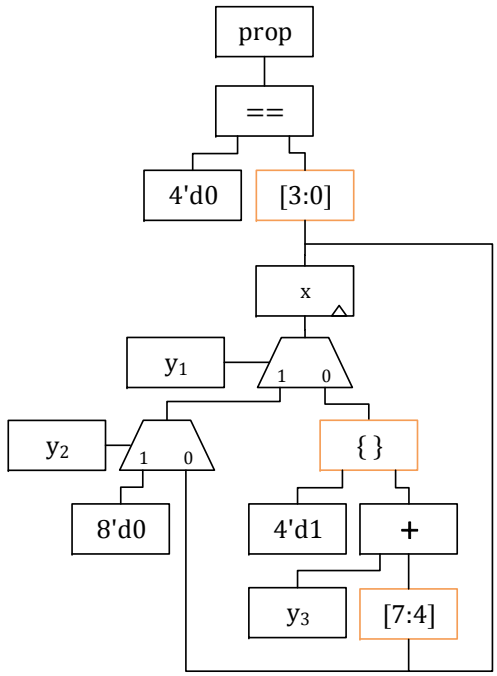


Figure 4.4: Simple Transition System for the Demonstration of the Circuit Components Decomposition Procedure

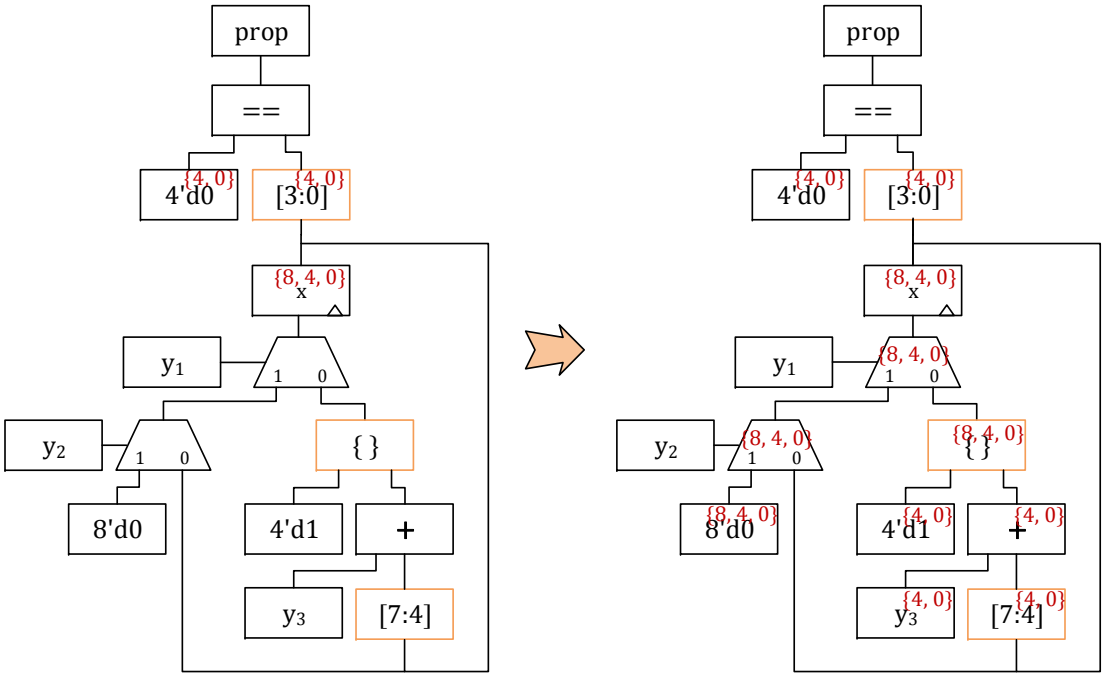


Figure 4.5: Constructed Decomposition Index Lists on the Simple Transition System

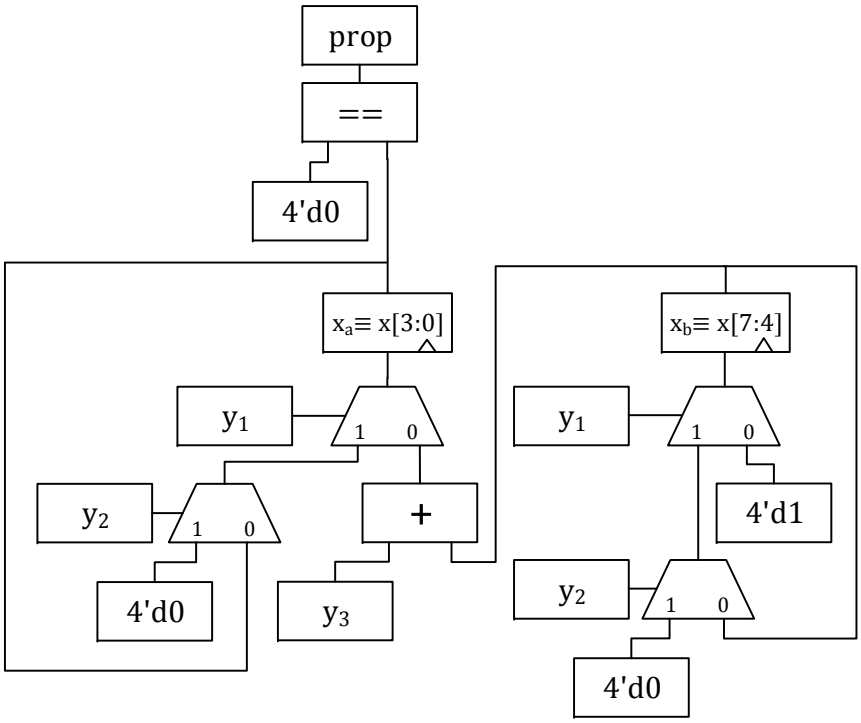


Figure 4.6: Resulting Circuit after Circuit Components Decomposition

Fig. 4.4, Fig. 4.5, and Fig. 4.6 illustrate how our algorithm works. A simple transition system in Fig. 4.4 contains one register, x, and the property compares the lower 4 bits of the register with a 4-bit constant zero. In our decomposition procedure, we traverse the graph of the *prop* signal first and fill the decomposition index lists of each node. Then, we move to the node of the register, x, and the same procedure for filling decomposition index lists is applied to the graph of the next-state circuit of the register. Fig. 4.5 shows this two-step procedure. The left diagram displays the decomposition index lists after the first traversal. The decomposition index lists are red in the diagram. After the second traversal, we obtain the decomposition index lists in the right diagram. Based on the decomposition index lists, we decompose circuit components, which results in the circuit in Fig. 4.6. Note that our decomposition procedure automatically eliminates three unnecessary operators, one concatenation and two extraction operators, from the original circuit.

Table 4.3: The Number of Extraction and Concatenation Operators Before and After Applying the Decomposition Algorithm

Abstraction Method	Bit-Select	Part-Select	Concatenation	Total
Default	35276 (21612)	56739 (43320)	89663 (7912)	181678 (72844)
Decomposition	43552 (11344)	31454 (4432)	84286 (6839)	159292 (22615)

Table 4.3 lists the number of extraction and concatenation operators before and after applying the decomposition technique. The number in parenthesis represents the number of extraction and concatenation operators that are connected with unnecessarily concatenated signals or multiplexers attached to those signals. These operators cause many datapath refinement iterations, and our decomposition technique eliminates them by splitting concatenated signals. As can be seen in the table, our decomposition technique eliminates about 70% of those kinds of operators, which leads to the 12% reduction in the entire set of extraction and concatenation operators. The gap between the two different percentages comes from the extraction and concatenation operators we introduce to preserve non-splittable datapath operators such as arithmetic, relational, and shift operators.

We don't decide whether to split those kinds of operators at this stage, because the careless split of a big datapath operator can offset the benefit of datapath abstraction. Instead, we put concatenation and extraction operators before and after the operator to connect it with split signals as explained earlier. Unlike the extraction and concatenation operators related with unnecessarily concatenated signals, these newly introduced operators usually do not introduce additional datapath refinement iterations, because they stick to the existing datapath operator most of the time. Thus, whenever the existing datapath operator needs to be refined, they all are refined together. Otherwise, none of them are refined. Some of those groups of operators may need to be concretized to reduce the number of datapath refinement iterations, and this is done during our property-directed concretization.

In sum, our decomposition technique traverses the graph representation of T and

P, and it minimizes the number of extraction and concatenation operators by decomposing unnecessarily concatenated signals. This technique allows us to derive an efficient abstract formula, which leads to a much smaller number of datapath refinement iterations. However, this optimization technique has one limitation; it considers only extraction and concatenation operators. That is, it does not correct the misclassification of other types of operators. For example, a counter is often represented by a wide register, and its operation is represented by an add operator. If this counter affects only some control components, it is usually better treated as a control component and concretized. This cannot be done by our decomposition technique. In the following section, we provide a systematic way to detect and concretize this kind of circuit component.

4.1.3 Property-Directed Concretization

One easy solution to the misclassification problem would be to ask users to provide some hints for the classification. For example, a user who designed a microprocessor can easily recognize that the instruction register in his or her design is a control register even though it is a wide register. This approach, however, has two problems. First, it is often too much to ask. Except some easy cases, it is not obvious how to characterize components in a design, even for the designers, and it usually requires a deep analysis. This obviously is a cumbersome task, so circuit designers are not willing to do it. Second, some hints can cause a counter-productive effect even when they are correct. This is because the best abstraction strategy depends on the property to be verified. For example, it is better to abstract a complicated control logic circuit that has nothing to do with the property even when a correct hint guides us to concretize it.

To resolve this issue, we developed a property-directed concretization algorithm that automatically detects a part of a circuit that is better to be concretized. Fig. 4.7

```
1.   (GT,GP) Concretize(GT,GP) {
2.       P = Node in GP representing a property;
3.       C = a set of nodes in the graphs to be concretized;
4.       if ( Width of the widest node in P's COI < param_width )
5.           C = C ∪ P;
6.       SS = Set of the nodes of state variables in P's COI;
7.       for ( i = 1; i < param_depth; i++ ) {
8.           for ( each node N in SS )
9.               S = S ∪ Set of the nodes of state variables in N's COI;
10.          SS = SS ∪ S;
11.      }
12.      for ( each node N in SS )
13.          if( Width of the widest node in N's COI < param_width )
14.              C = C ∪ N;
15.      for ( each node N in C )
16.          (GT,GP) = Bit-blast the entire COI of N;
17.      return (GT,GP);
18.  }
```

Figure 4.7: Property-Guided Concretization Algorithm

provides a pseudocode of the concretization algorithm. First of all, we check to see if it is better to concretize the cone-of-influence of a property node. This is done by calculating the widest bit width of the datapath operators in the cone-of-influence of the property node (line 4). During the analysis, we consider only arithmetic, relational, and shift operators, whose bit-level counterparts are huge. If the widest bit width of the datapath operators is smaller than a parameterizable threshold, *param_width*, currently set to be 32, we conclude that the circuit nodes we traversed are better to be concretized. We then collect a set of state nodes to apply the analysis of the widest bit-width. To maximize the benefit of datapath abstraction, we need to collect a minimal number of circuit components that are expected to play an important role during our reachability computation. We observed that the state variables in the cone-of-influence of a property are more involved in the reachability lemmas derived during the reachability computation than the other state variables. This is reasonable, because they directly affect the property to verify. With the same reason, misclassifications on the state variables are more critical than those of the

others. Thus, we consider only the state nodes in the cone-of-influence of a property node when we look for state nodes to be concretized (line 6). We can collect more state nodes if we apply a multi step cone-of-influence analysis (i.e., if we collect a set of state variables that affects a property in multi clock cycles) (lines 7 to 11). Another parameterizable threshold, *param_depth*, which is currently set to be 1, indicates the number of steps (or depth) for this analysis. Once we collect a set of state nodes to examine, we check to see if any of them needs to be concretized by using the same method applied to a property node (lines 12 to 14).

The collected nodes to be concretized can represent multi-bit registers, single-bit FFs, or a property. A node of a single-bit FF is collected when the next-state circuit of the FF includes a small datapath component that seems to be better to concretize. For example, if the next state of a single-bit FF depends on the sum of two three-bit signals, it sometimes is better to concretize the adder to prevent a spurious abstract counterexample trace that can be caused by the abstraction of the adder. Once the nodes to be concretized are collected, we concretize related circuit components accordingly (lines 15 to 16). For a multi-bit register, we decompose the register into single-bit FFs and concretize the next-state circuit of the register. For a single-bit FF, we just concretize its the next-state circuit. In the case of a *prop* signal, we concretize the entire cone-of-influence of the *prop* signal. This approach is based on our observation that the concretization of a single circuit component introduces many concatenation and extraction operators surrounding the component. Therefore, it is usually better to concretize the circuit component as well as the circuit surrounding the component at the same time. This can lead to an excessive concretization which slows down an SMT solver significantly, so we need to be careful when we decide a circuit component to concretize. This is the reason why we examined a small number of state nodes during our property-directed concretization procedure. After the concretization is done, we update a property node and a set of secondary output

nodes with the concretized ones.

In sum, our property-directed concretization technique analyzes a *prop* signal and the state variables appearing in the cone-of-influence of the *prop* signal and then detects the best part of a circuit to concretize. This approach uses a select and concentration strategy; it focuses on only a small number of important control components and corrects misclassification on them. Thus, we can achieve the best result when this technique is paired with the decomposition technique in Section 4.1.2, because the optimization technique covers the other part of a design ignored in this concretization technique.

4.2 Memory Abstraction

Many industrial designs contain large memories. Because of the large number of state bits involved in a memory, handling a memory affects the verification scalability significantly. In this section, we discuss an efficient memory abstraction technique for making a verification system more scalable. We limit our discussion to a simple random access memory (RAM), which can be represented by a two-dimensional array in Verilog-HDL, because this is the most common type of memory in industrial designs.

Suppose that we converted a memory into a set of registers incorporating a multiplexer and a demultiplexer. This is one easy way to handle a memory. For example, a 1024×32 bit memory is converted into (i) 1024 registers of 32-bit width, (ii) a 1024-to-1 multiplexer for a memory-read operation, and (iii) a 1-to-1024 demultiplexer for a memory-write operation. The multiplexer and demultiplexer allow us to access one of the 1024 registers corresponding to a 10-bit address. When we abstract these converted circuit components, we can simplify each 32-bit register as one state term. However, we cannot reduce the number of registers; we end up with 1024 state terms. In addition, the multiplexer and demultiplexer lead to a very big and complicated EUF formula. Therefore, this approach will undermine the scalability of our

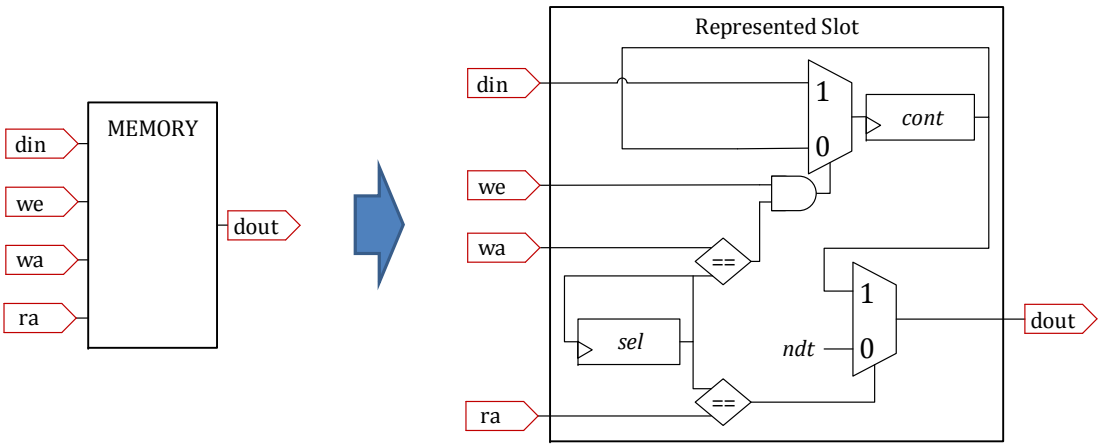


Figure 4.8: Represented Slot of Bjesse's Memory Abstraction

verification system.

Bjesse proposed a more scalable way to handle a memory, in which a memory is replaced with a set of represented slots [65]. The overall procedure follows a counter-example guided abstraction refinement framework:

1. An initial abstraction replaces a memory output with a fresh input variable.
2. If a given property holds with the initial abstraction, we finish the proof by concluding that the property holds.
3. If the property is proved to be violated, yielding a counterexample trace, we check to see whether the trace is valid. If it is valid, we return the trace. Otherwise, we apply a simulation-based analysis and collect the set of memory read operations that caused the invalid trace. A memory-read operation with an address contained in a signal v in T or P at d cycles before the violation is represented by an *abstraction pair*, (v, d).
4. A *represented slot* is constructed for each abstraction pair, (v, d). Fig. 4.8 provides the circuit diagram of the represented slot. *sel* and *cont* registers represent the address and the content of the memory slot respectively. The represented memory location (*sel*) is set arbitrarily at the beginning, and it does not change during the execution. Whenever we access the memory location specified in *sel*,

we access the *cont* register. When we access a memory location that is not represented by any slots, we return an unconstrained value represented by an input variable, *ndt*. The memory in T is replaced by this represented slot. We then update the property as $(prev^d(sel == v) \rightarrow P)$ to ensure that this slot represents the memory location at d cycles before the property check. $prev^d$ in the updated property is a temporal formula that is true at the kth cycle if (i) $k \geq d$ and (ii) $sel == v$ at the (k-d)th cycle. This property can be easily transformed to a circuit consisting of a flip-flop chain.

5. We check to see whether the updated property holds in the updated transition relation. If the property holds, we check for the existence of a counterexample trace whose length is less than d_{max} in the original design (without memory abstraction) by using standard bounded model checking. d_{max} is the longest time delay among the abstraction pairs that have been derived. If no such trace is found, we conclude that the property holds. Otherwise, we return the trace found during the bounded model checking.
6. If a counterexample trace is returned showing that the updated property is violated, we check the validity of the counterexample trace with simulation. If it is valid, we finish the proof by returning the trace. Otherwise, we need to add more represented slots with the same procedure described in steps 3 and 4. We then repeat steps 5 and 6 until we prove that the property holds or we find a valid counterexample trace.

This approach performs another counter-example guided abstraction refinement loop on top of Averroes, so it does not require any change in Averroes. In addition, this memory abstraction can be viewed as another instance of structural abstraction like datapath abstraction, because the structure of the original hardware design is preserved during abstraction. It simply applies a netlist-to-netlist transformation to a memory and keeps everything else in place.

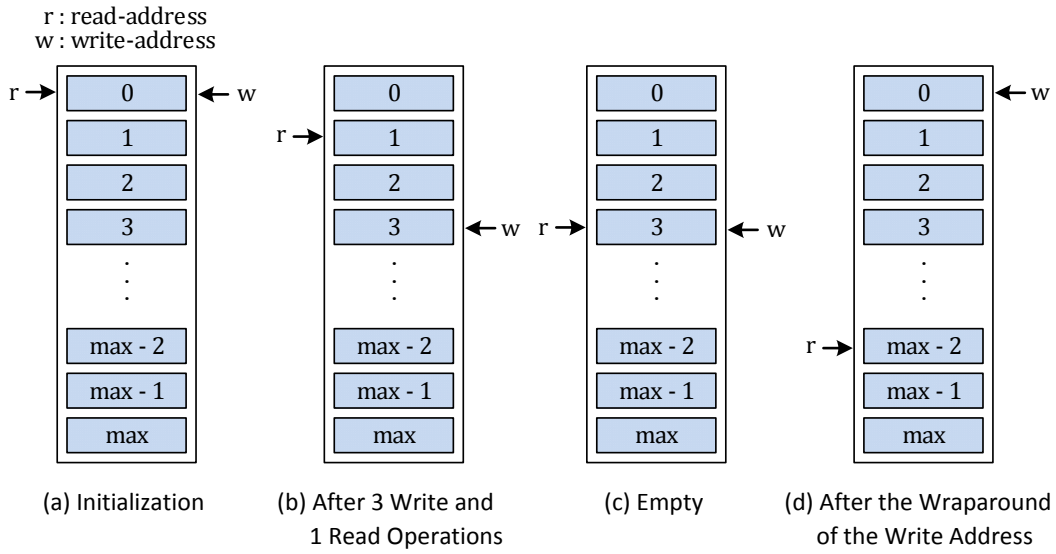


Figure 4.9: Four Different States of a FIFO

We applied this memory abstraction technique to our FIFO benchmark[2], which checks a "read-after-write" property. When each memory in the four FIFOs of the benchmark contained 16 entries, the abstraction technique replaced the 16 entries with a represented slot consisting of 2 registers. Because of the 1/8 reduction of the number of registers, we anticipated a speed-up of 8 times. However, we only doubled the speed. The effect of memory abstraction turned out to be quite limited, because the major bottleneck of the property checking was the enumeration of all the possible values of the read and write pointers in the FIFOs.

This enumeration is caused by an address wraparound, which may or may not occur. If an address cannot wrap around, the behavior of a FIFO becomes very simple. As illustrated in Fig. 4.9 (a), (b), and (c), both read and write pointers are initialized to zero at the beginning (a), and they are incremented whenever there is a read or write operation respectively (b). If all the written entries are read, both pointers are pointing the same memory address (c). In this case, an "empty" signal is enabled to prevent an additional read operation. Otherwise, the address of the read

[2]The detailed description of the benchmark will be provided in Chapter VI.

pointer (read address) is always smaller than that of the write pointer (write address). All the safe (or good) states can be categorized into these two cases. The bad states are categorized into the other case where the read address is greater than the write address. When we apply datapath abstraction to the FIFO, these three cases are simply represented by the equality and less-than relationship of the read and write addresses. Thus, induction-based proofs can quickly verify the "read-after-write" property.

When the address can wrap around, however, the problem becomes very difficult to verify. Now, the read address can be greater than the write address in a safe state, as shown in Fig. 4.9 (d). This happens when only the write address wraps around. The examination of whether the read address is pointing to a written memory entry requires the enumeration of the read and write addresses, because we need to trace every possible track of the two addresses. In this case, the exact values of the pointers play an important role, so abstracting these pointers will lead to many iterations of datapath refinement. Thus, we cannot abstract the pointers[3].

A simple but very powerful solution to this problem is to introduce a "wraparound" signal, which is enabled when the write address wraps around, to the FIFO and to check the property only when the signal is disabled. This simple helper signal simplifies the verification problem at the abstract level dramatically, because we do not need to consider the case shown in Fig. 4.9 (d). We can ensure that the "wraparound" signal can be automatically generated by analyzing the pattern of the read and write pointers of a memory. Different types of memories and properties may require different types of signals to be able to abstract read and write pointers. We call this technique *memory address abstraction*, because the introduction of a helper signal allows us to abstract memory address signals. Memory address abstraction combined with memory abstraction allows a constant or linear time verification of a control-centric

[3]Averroes automatically detects this potential problem through its structural analysis, and it does not apply datapath abstraction to the pointers.

property involving a memory regardless of the number of entries in the memory.

CHAPTER V

Datapath Refinement

In our counterexample-guided abstraction and refinement (CEGAR)-based datapath abstraction, feasibility check and datapath refinement methodologies affect the overall performance significantly. A feasibility check is done at the bit level, so we need to keep the bit-level formula passed to a solver as small as possible. Otherwise, the overhead of the feasibility check will outweigh the benefit of datapath abstraction. Datapath refinement also plays an important role. If we can derive much more powerful datapath lemmas during datapath refinement, our abstraction and refinement loop will finish in a much smaller number of iterations.

Reveal introduced an automatic procedure for the feasibility check and datapath refinement based on localization, generalization, and minimal unsatisfiable subset (MUS) extraction, but this procedure can be applied only to a combinational circuit. Thus, we have extended it for application to a sequential circuit as presented in Section 3.3. In addition, the procedure does not scale to large industrial designs, because it often generates similar datapath lemmas over and over. To solve this problem, we introduce localized datapath lemmas and constant propagation-based generalization techniques. The two techniques are capable of deriving powerful datapath lemmas, so we do not need to generate as many similar datapath lemmas. In this chapter, we explain these two techniques.

5.1 Localized Datapath Lemmas

A datapath lemma is a constraint on uninterpreted terms and literals that correspond to abstracted circuit components, and it tightens the initial abstraction by refuting spurious ACEXTs. As explained earlier, it is important to derive a powerful datapath lemma that refutes a large number of spurious ACEXTs, because this leads to a small number of refinement iterations. In Section 3.3, we explained the basic procedure for deriving a datapath lemma from a solution cube. We now introduce one additional step at the end to derive a *localized* datapath lemma. A localized datapath lemma locally constrains an abstract function block by making the connection between the function block and the peripheral blocks attached to it indirect. This is done by replacing some sub-expressions in a datapath lemma with local variables, $\mathbf{w}$, in a transition relation, $T(\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{x}^+)$. For example, a localized datapath lemma constrains the function block, M, in Fig. 5.1 by using the abstract-level counterparts of the local variables, $\hat{w}_1$ to $\hat{w}_k$. A localized datapath lemma can effectively constrain the abstract function block that needs to be refined, but it cannot constrain the other abstract function blocks with the same type that are connected to different local variables. A globalized datapath lemma, on the other hand, constrains the abstract function block with independent variables (primary inputs or state variables) and constants, so it can simultaneously constrain the other function blocks with the same type. However, a globalized datapath lemma cannot constrain the target function block effectively, because it constrains all the function blocks in the cone-of-influence of the target function block at the same time.

We use local variables to significantly reduce the size of a formula passed to a solver, because common expressions do not need to be repeatedly redefined. During the reachability computation, on the other hand, we do not use local variables to represent a state cube, because the number of literals in the state cube must be reduced. We need to keep the number of literals as small as possible for the efficient

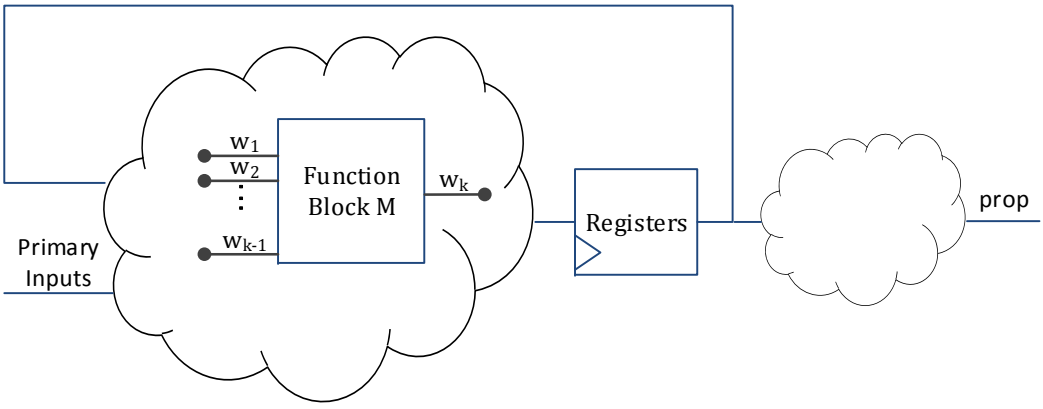


Figure 5.1: Internal Nodes of a Sequential Circuit

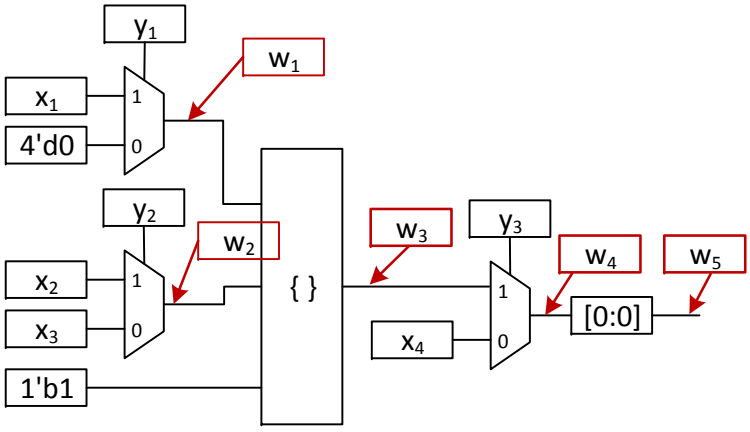


Figure 5.2: A Part of a Sequential Circuit for the Demonstration of a Refinement Process

computation of a minimal unsatisfiable subset during our cube enlargement process. By the same reasoning, we do not use local variables during the derivation of datapath lemmas, which is basically a process of computing minimal unsatisfiable subsets from a solution cube. Thus, we need to apply an additional process to derive localized datapath lemmas.

We will explain this process with a simple example, provided in Fig. 5.2. The circuit in the diagram is a part of a sequential circuit, and x_i, y_i, and w_i represent a state element, primary input, and internal wire respectively. Suppose that we found an ACEXT that assigns $y_1y_2y_3w_5$ to 1110 at a certain cycle. This is obviously a

spurious ACEXT, because w_5 must be true when $\hat{w}_3$ equals $\hat{w}_4$. A corresponding solution cube of the ACEXT is:

$w_1 \wedge w_2 \wedge w_3 \wedge \neg Ex\_0\_0\_9(Concat\_4\_4\_1\_9(\hat{x}_1, \hat{x}_2, true))$[1].

Note that this is a part of the entire solution cube, and it includes only the literals derived from the circuit part shown in Fig. 5.2. Its bit-level counterpart is:

$w_1 \wedge w_2 \wedge w_3 \wedge \neg\{x_1, x_2, true\}[0:0]$.

This is obviously unsatisfiable, and we can find the following bit-level MUS from the unsatisfiable formula.

$\neg\{x_1, x_2, true\}[0:0]$

The satisfiability of this formula remains the same even after we replace x_1 and x_2 with any other variables. Thus, we replace them with the local variables w_1 and w_2, which correspond to the internal nodes connected to the concatenation. We then obtain the following formula.

$\neg\{w_1, w_2, true\}[0:0]$

The resulting datapath lemma is the negation of the abstract-level counterpart of the above formula:

$Ex\_0\_0\_9(Concat\_4\_4\_1\_9(\hat{w}_1, \hat{w}_2, true))$.

This datapath lemma injects the simple constraint that the least significant bit of the concatenation whose last operand is one-bit true is the one-bit true. This datapath lemma is highly localized in the sense that it involves only the circuit components it is constraining. It is interesting to note that without the substitution of local variables, we would end up with the following weak datapath lemma.

$Ex\_0\_0\_9(Concat\_4\_4\_1\_9(\hat{x}_1, \hat{x}_2, true))$.

This lemma refutes only spurious ACEXTs that assign w_1w_2 to 11, so we need the following three more datapath lemmas to refute the other spurious ACEXTs that assign w_1w_2 to 00, 01, or 10.

[1]In this formula, $Ex\_0\_0\_9$ and $Concat\_4\_4\_1\_9$ are the uninterpreted predicate of extraction and the uninterpreted function of concatenation respectively as defined in Section 1.1.6.

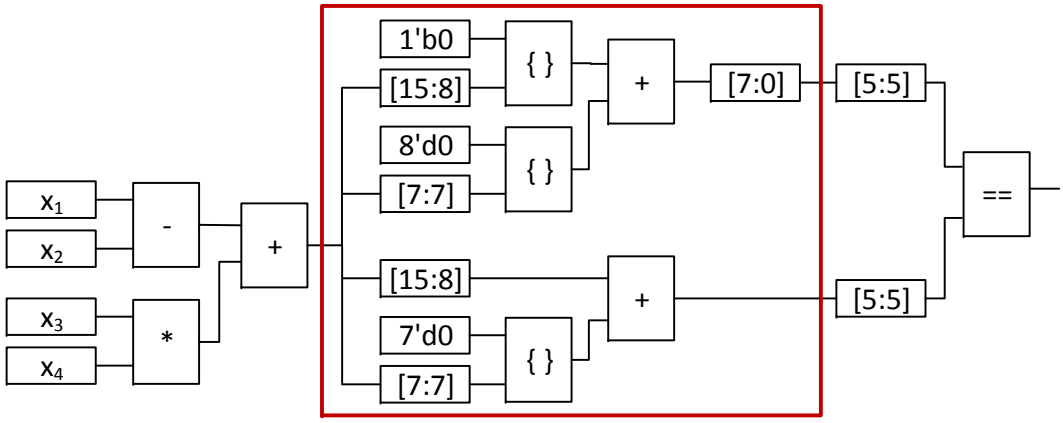


Figure 5.3: Graph Representation of a Weak Datapath Lemma

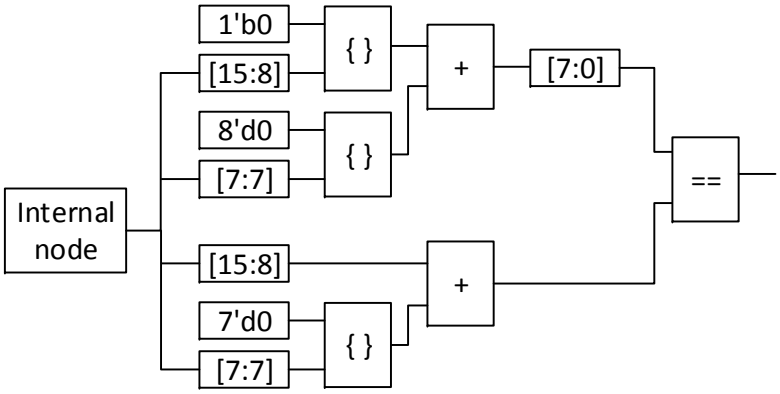


Figure 5.4: Graph Representation of a Localized Datapath Lemma

$Ex\_0\_0\_9(Concat\_4\_4\_1\_9(K0, \hat{x}_3, true))$

$Ex\_0\_0\_9(Concat\_4\_4\_1\_9(K0, \hat{x}_2, true))$

$Ex\_0\_0\_9(Concat\_4\_4\_1\_9(\hat{x}_1, \hat{x}_3, true))$

That is, we need four datapath lemmas (with four datapath refinement iterations) to inject the same constraint without the use of local variables. The number of datapath lemmas required to inject the constraint increases exponentially (i) if more multiplexers are involved in the cone-of-influence of the concatenation operator and/or (ii) if the concatenation operator has more operands. Therefore, it is essential to derive a localized datapath lemma in a scalable refinement methodology.

We have considered a situation where local variables can be used in a datapath

```
1.  δ LocalizedLemmaDerivation(δ) {
2.      G_δ = DAG representation of δ;
3.      for ( each converging node or source N in G_δ) {
4.          F = node of free variable;
5.          G_δ = G_δ ( N ↦ F);
6.          δ_t = EUF formula represented by G_δ;
7.          if ( δ_t is always true at the bit-level ) {
8.              (N_1, N_2) = matching nodes in G_δ;
9.              if ( N_1 is always equivalent to N_2 at the bit-level ) {
10.                 G_δ = DAG representation of (N_1 == N_2);
11.             }else if ( N_1 is always equivalent to N_2 at the bit-level ) {
12.                 G_δ = DAG representation of (N_1 != N_2);
13.             }
14.             N_r = node in G_T or G_P that N originated from;
15.             G_δ = G_δ (F ↦ N_r);
16.             δ = EUF formula represented by G_δ;
17.         }
18.     }
19.     return δ;
20. }
```

Figure 5.5: Localized Lemma Derivation Algorithm

lemma, but the same idea can be applied to a more complicated case where an uninterpreted function or predicate can be replaced by local variables. Fig. 5.3 shows the graph representation of a datapath lemma's bit-level counterpart (the actual datapath lemma is represented as an EUF formula). The datapath lemma in the diagram means that even after we append one bit to the most significant bit of an adder's two inputs and remove the most significant bit of its output, the result does not change. The circuit components in the red box are what we want to constrain, and the circuit components outside the red box make the datapath lemma too specific or weak to constrain them. By using local variables, we can derive a much more powerful datapath lemma as represented in Fig. 5.4. Again, this diagram is the graph representation of the datapath lemma's bit-level counterpart, and the actual datapath lemma is represented as an EUF formula.

The pseudocode of the localized lemma derivation is given in Fig. 5.5. LocalizedLemmaDerivation() in the pseudocode takes a datapath lemma as an input and

returns a localized datapath lemma. From the input datapath lemma, we try replacing each converging node or source (i.e., a node that has no predecessors) with a corresponding internal node (line 3). A converging node is a good candidate for substitution, because it corresponds to a common expression shared by multiple terms. The literals in a datapath lemma are usually an equality and disequality among terms, and the substitution of a common expression in both of the sides often does not affect the evaluation of the literals. In a for loop, we replace the converging node with a node of an arbitrary free variable (lines 4 to 5) and check to see whether an EUF formula represented by the replaced graph is always true at the bit level (lines 6 to 7). If it is always true, we replace the free variable with an appropriate local variable in T or P (lines 14 to 15). The substitution for a free variable in a formula does not change the satisfiability of the formula, so the datapath lemma after the substitution is always true (i.e., correct) at the bit-level as well. We can further generalize a datapath lemma by eliminating irrelevant parts of the datapath lemma through structural analysis (lines 8 to 13). This time, we traverse G_δ starting from a sink (i.e., a node that has no successors) and find a pair of nodes at the same distance from the sink that represent different types of operators. Once we find two such nodes, we check to see whether they are always the same at the bit-level (line 9). If they are always the same at the bit-level, we replace G_δ with a graph representing $N_1 = N_2$ (line 10). If they are always different at the bit-level (line 11), we replace G_δ with a graph representing $N_1 \neq N_2$ (line 12). This generalization technique can replace a large number of datapath lemmas with few powerful datapath lemmas, so we can verify a property with a smaller number of datapath lemmas, which leads to more efficient runtime and memory usage.

5.2 Constant Propagation Based Generalization

During the analysis on the datapath lemmas derived from our industrial benchmarks, we found that some of them follow the pattern below.

$$\neg[(w_a = K_a) \wedge L(w_a, w_b, K_b)] \tag{5.1}$$

In the formula, w_a and w_b are uninterpreted terms; K_a and K_b are constant terms; and $L(w_a, w_b, K_b)$ is an EUF literal involving w_a, w_b, K_b. In this pattern, a signal is assigned to a constant, so we can generalize the lemma by applying constant propagation as in the following formula:

$$\neg L(K_a, w_b, K_b). \tag{5.2}$$

For example, a datapath lemma following the pattern,

$\neg((x = K0) \wedge (x = ADD(x, K1)))$

where $K0$ and $K1$ are constant terms of zero and one respectively, can be generalized to $(K0 \neq ADD(K0, K1))$ after constant propagation. Contrary to the original datapath lemma, which is valid only if x equals to $K0$, the generalized one can be applied to any signal equal to $K0$. In other words, the generalized one constrains the uninterpreted function, ADD, in a more direct or effective way.

The generalized datapath lemma, (5.2), is *correct*. That is, its negation is unsatisfiable at the bit level (i.e., it is always true at the bit level) if the negation of (5.1) is unsatisfiable at the bit level:

Proof.

$(w_a = K_a)$ implies $(L(w_a, w_b, K_b) = L(K_a, w_b, K_b))$. ($\because$ Substitution of Equational logic)
Thus, the negation of (5.1), $((w_a = K_a) \wedge L(w_a, w_b, K_b))$, is equivalent to
$((w_a = K_a) \wedge L(K_a, w_b, K_b))$.

```
1.   MUS ConstantPropagation(MUS) {
2.       do {
3.           M = set of (variable, constant) pairs;
4.           for ( each literal L in MUS ) {
5.               if(L → (v = c)){
6.                   M = M ∪ (v, c);
7.                   MUS = MUS - L;
8.               }
9.           }
10.          for ( each pair (v, c) in M ) {
11.              MUS = MUS (v ↦ c);
12.          }
13.      } while ( MUS has been updated );
14.      return MUS;
15.  }
```

Figure 5.6: Constant Propagation Algorithm

The former expression is UNSAT at the bit level, so the latter expression is also UNSAT at the bit level.

In the latter expression, $(w_a = K_a)$ is SAT, and w_a is not involved in $L(K_a, w_b, K_b)$. Therefore, $L(K_a, w_b, K_b)$,the negation of (5.2), is UNSAT at the bit level. □

The constant propagation-based generalization technique can be easily applied to more complicated patterns by applying it multiple times as in the following example.

$$\neg[(w_a = K_a) \wedge (w_b = K_b) \wedge (w_c = w_a) \wedge L(w_a, w_b, w_c, K_c)]$$
$$\rightarrow \neg[(w_b = K_b) \wedge (w_c = K_a) \wedge L(K_a, w_b, w_c, K_c)]$$
$$\rightarrow \neg[(w_c = K_a) \wedge L(K_a, K_b, w_c, K_c)]$$
$$\rightarrow \neg[L(K_a, K_b, K_a, K_c)]$$

Fig. 5.6 provides a pseudocode of a function, $ConstantPropagation()$, which conducts this iterative procedure. As discussed in Sections 2.5 and 3.3, a datapath lemma is a negated conjunction of an MUS, and $ConstantPropagation()$ applies a simple constant propagation algorithm to the MUS. For each EUF literal in the MUS (line 4), we collect variable and constant pairs if the literal implies their equality relationship

(lines 5 to 8). We then replace each collected variable with its corresponding constant in the MUS (lines 10 to 12). We repeat this collection and substitution process until no further changes in the MUS are possible (line 13). With this constant propagation-based generalization, we can reduce the number of datapath lemmas required for a proof at the abstract level.

CHAPTER VI

Empirical Evaluation

In this chapter, we demonstrate the scalability of Averroes on three different sets of proprietary industrial benchmarks. We then discuss the effects of our four advanced abstraction and refinement optimizations. We would like to note that we use AVR instead of Averroes in our tables and figures to save space.

6.1 Statistics of Benchmarks and Experimental Setup

Anecdotally, abstracting a design's datapath is commonly believed to yield scalable verification of its control logic. However, unlike verification at the bit level, which enjoys a large corpus of benchmarks and published results, there is little documentation in the open literature of the effectiveness of datapath abstraction on a diverse set of word-level benchmarks. The dearth of publicly-available RTL benchmarks that preserve the word-level semantics of a design was one of the main challenges we faced when evaluating the effectiveness of Averroes. Realizing that reporting on hand-crafted synthetic benchmarks would not be convincing, we opted instead to evaluate performance on a set of 350 industrial Verilog benchmarks that we obtained under

Table 6.1: Statistics of the Large Industrial Benchmarks

Benchmark	Regs	FFs	State Bits	%Regs	%Reg Bits	AIG Size
mult_hold_1	6	2	258	75	99	24452
mult_hold_2	6	2	514	75	100	98052
mult_hold_3	6	2	1026	75	100	392708
mult_hold_4	6	10	266	38	96	24638
mult_viol_1	7	2	268	78	99	25008
mult_viol_2	7	2	524	78	100	99119
mult_viol_3	7	2	1036	78	100	394797
mult_viol_4	7	10	279	41	96	25193
mult_viol_5	7	10	279	41	96	25193
mult_viol_6	7	10	279	41	96	25190
mult_viol_7	7	10	279	41	96	25190
fifo_hold_2	28	10	474	74	98	6848
fifo_hold_3	44	10	866	81	99	17968
fifo_hold_4	76	10	1642	88	99	53904
M0+_hold	56	26	1306	68	98	41630

non-disclosure agreements [1]. Of these, 124 were medium-sized "generic" benchmarks that were used for initial calibration. Their code sizes ranged between 298 and 805 lines; in terms of flip-flops, the smallest had 514 and the largest had 931. Another 211 "big-datapath" benchmarks came from RTL designs consisting of big complicated datapath components. These benchmarks are the instances of equivalence checks between the two designs before and after applying high-level optimization. Their code sizes ranged between 135 and 16658 lines, and they included 6 to 3788 flip-flops. The remaining 15 benchmarks included 11 large multipliers, 3 FIFO designs, and the ARM Cortex-M0+ core [70]. The code sizes for these ranged from 116 to 10,226 lines.

Table 6.1 lists additional statistics of the 15 benchmarks including the number of multi-bit registers (Regs), the number of single-bit flip-flops (FFs), the total number of state bits (FFs + the number of bits in the registers), the percentage of registers and register bits in the benchmark, and the number of AND nodes in the AIG repre-

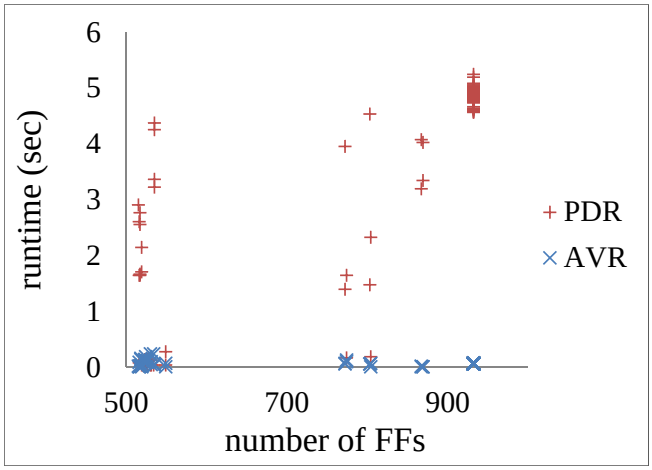
[1] Unfortunately, we are not allowed to disclose the word-level benchmarks. In fact, the company that provided the benchmarks did not even want to be identified. We understand this situation, because the company representatives want to protect the IP of their, or their customers', RTL designs. However, to spur further research in this space, it is important to find a way to make such RTL designs publicly available without compromising their owners' IP rights.

sentation [71] of its synthesized bit-level netlist. The multiplier benchmarks involved checking the sequential equivalence before and after clock gating optimizations; in four of these the property holds, and in the remaining seven it fails. The FIFO benchmarks check a "read-after-write" property for different FIFO depths. Finally, the M0+ experiment involved checking self-equivalence under partial initialization (i.e., when only a subset of the state bits are initialized on reset); this is sometimes referred to as self-equivalence with don't-cares or SEQX. In all cases, the verification involved an *unbounded check* to determine if the given safety property holds, on all, or is violated, by some, reachable states.

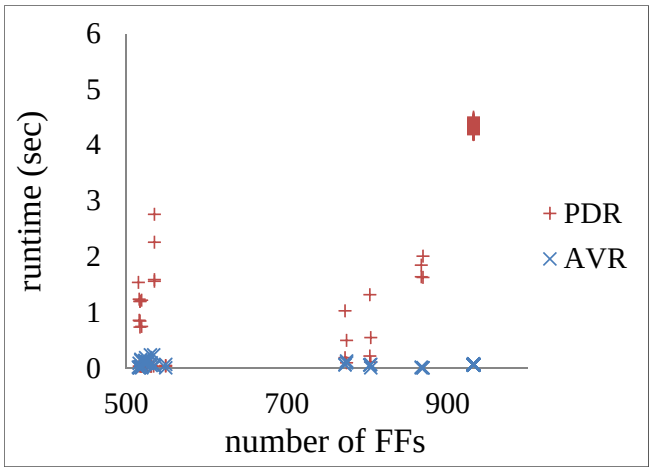
We compared the performance of Averroes to that of PDR *with and without pre-processing*, because PDR is one of the most reliable and efficient bit-level verification tools. In its default mode, PDR simplifies the input design before it starts the approximate reachability loop: PDR invokes the ABC dprove command [72]. Such pre-processing can greatly reduce the size of the input circuit, which helps with the subsequent reachability computation. All experiments were run on a 3.2GHz Xeon desktop computer with a 16 GB memory. A time-out of 10,000 seconds was used for each verification run.

6.2 Emperical Results for Generic Industrial Benchmarks

Each of the 124 generic benchmarks was provided with a single specified safety property and were meant to calibrate the performance of Averroes against that of PDR. Fig. 6.1 compares the runtime of Averroes against that of PDR as a function of the number of flip-flops in these benchmarks. In all cases, Averroes is faster than PDR, and, unlike PDR, its performance is largely independent of the number of flip-flops. This validates the hoped-for benefit of datapath abstraction. Oddly, the performance of PDR with pre-processing was worse than without! This seems to be due to the fact that there was not much structural reduction due to pre-processing



a. PDR with pre-processing vs. Averroes



b. PDR without pre-processing vs. Averroes

Figure 6.1: Verification Results of the Generic Industrial Benchmarks

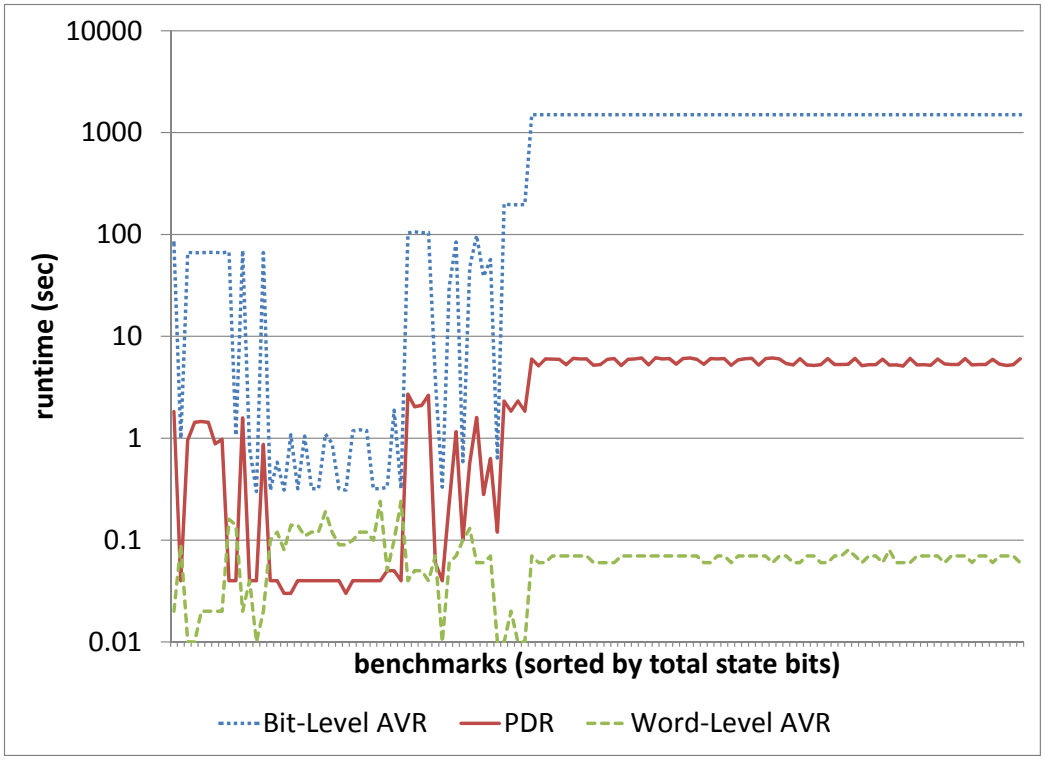


Figure 6.2: Runtimes of Averroes With and Without Datapath Abstraction

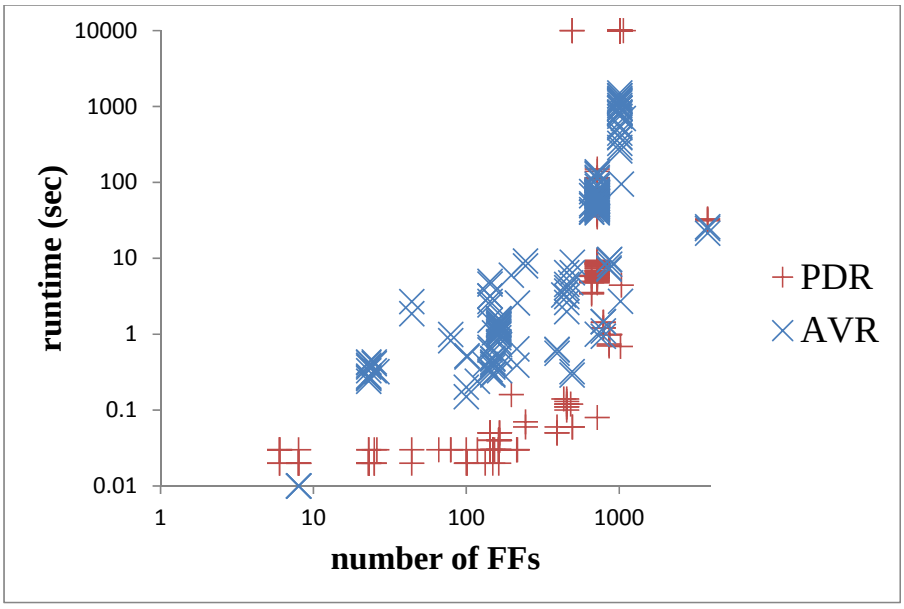
causing pre-processing overhead to outweigh its benefit.

We also checked the effectiveness of datapath abstraction by running Averroes both with and without datapath abstraction and comparing the results. In this experiment, we used a time-out of 1,500 seconds to save time. In Fig. 6.2, the effectiveness of datapath abstraction can be seen by comparing the green dashed line and blue dotted line, which correspond to runs of Averroes with and without datapath abstraction respectively. Note that bit-level Averroes, shown in the blue dotted line, times out on all large benchmarks and is about 3 orders of magnitude slower than the word-level abstract version on the smaller ones. Note also that bit-level Averroes is 1 to 2 orders of magnitude slower than PDR, shown in the red solid line. This is primarily because our current prototype implementation of the IC3/PDR framework lacks many of its optimizations and utilizes a much slower solver. Despite this handicap, word-level Averroes is close to 2 orders of magnitude faster than PDR as a result

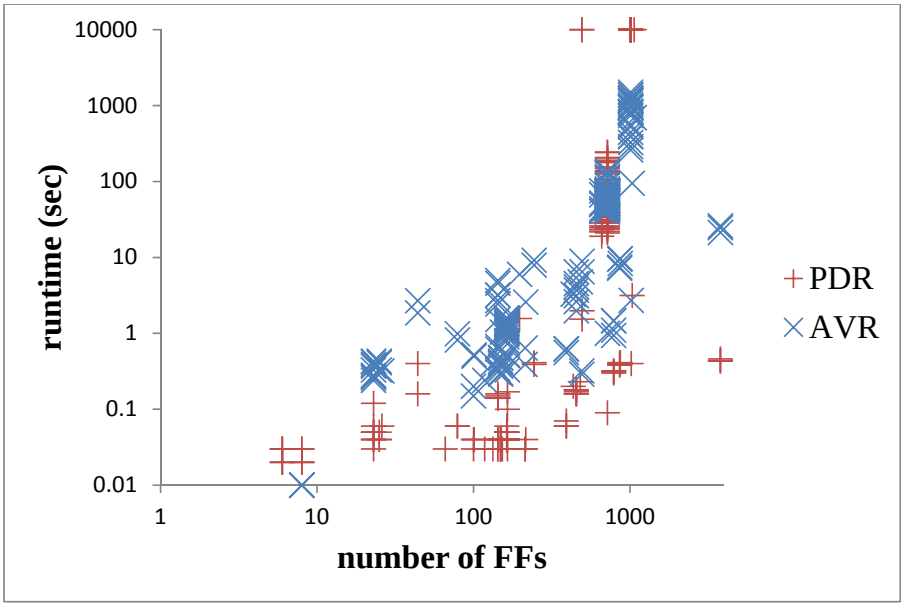
of datapath abstraction. Assuming that the performance of the reachability engine in Averroes can be optimized so that it is comparable to that of PDR, we can expect the runtime performance of word-level Averroes to improve by one or two orders of magnitude. The seemingly erratic runtime curves for the smaller benchmarks are due to the fact that in 15 of these benchmarks, P happens to be an inductive invariant, that is, it is closed under T. Thus, no reachability computation is needed to establish P and all three solvers finish very quickly in these cases.

6.3 Emperical Results for Big-Datapath Industrial Benchmarks

The 211 "big-datapath" benchmarks contain complex datapath components and their properties are designed to compare a design with an optimized one. Fig. 6.3 compares the runtime of Averroes with the runtimes of PDR with and without preprocessing. In the graphs, the x- and y-axes are in a logarithmic scale. Because of the complexity of the datapath components involved, the runtimes of PDR and Averroes are slower than those of the generic benchmarks. PDR is faster than Averroes when the number of flip-flops is small. However, PDR slows down very quickly as the number increases, and it times out in 30 large benchmarks, as can be seen in the topmost red crosses. The performance of Averroes, on the other hand, is less affected by the number of flip-flops, and it finishes all the proofs. This runtime patterns of PDR and Averroes are clearly shown in Fig. 6.4. Averroes becomes faster in the benchmarks whose PDR runtime is longer than about 100 seconds. The slower runtimes of Averroes in easier problems are caused by its lack of implementation-level optimizations, as discussed in Section 6.2. The generic benchmarks in Section 6.2 are much smaller than these benchmarks; The average code size of the generic benchmarks (about 574 lines of code) is about 11 times smaller than that of these big-datapath

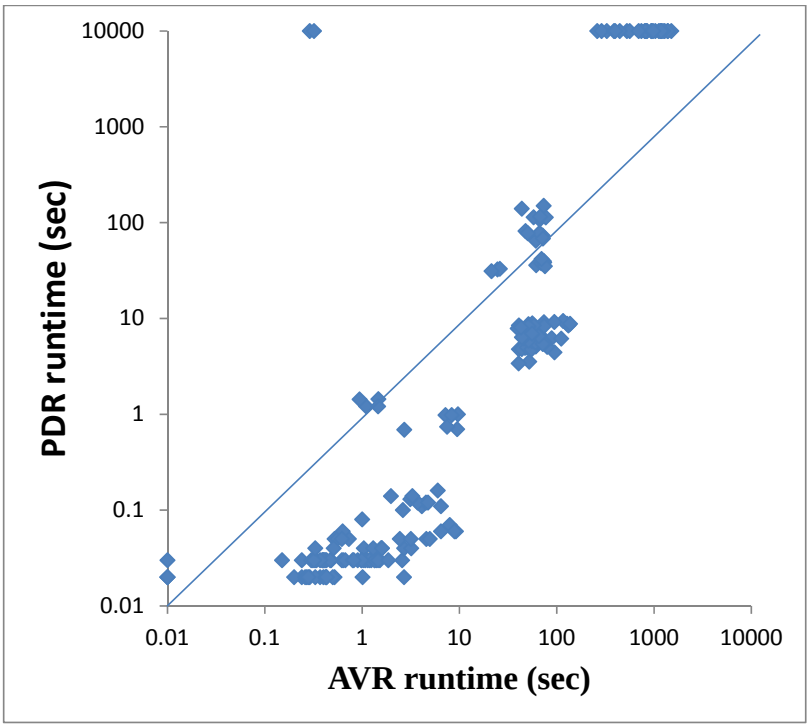


a. PDR with pre-processing vs. Averroes

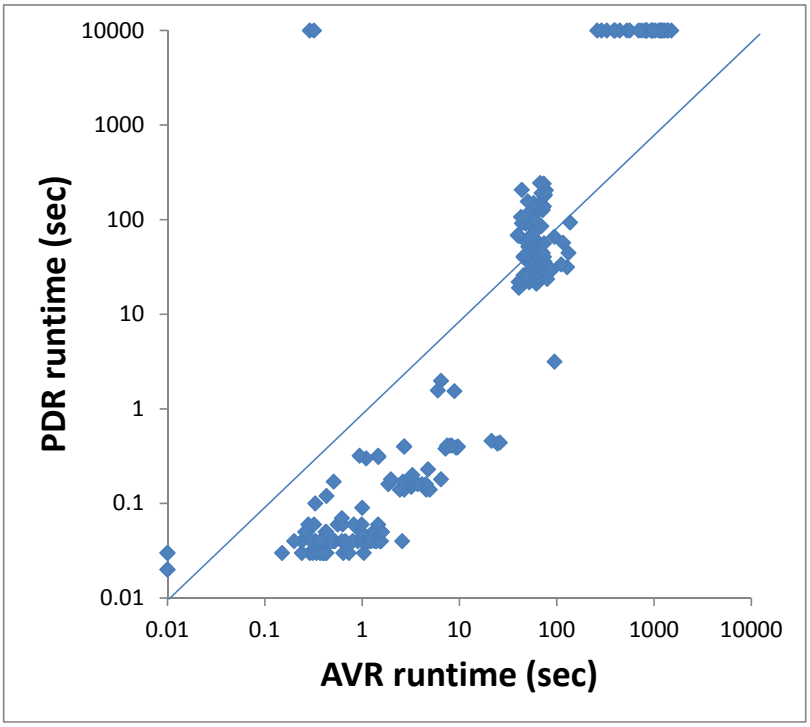


b. PDR without pre-processing vs. Averroes

Figure 6.3: Verification Results of the Big-Datapath Industrial Benchmarks



a. PDR with pre-processing vs. Averroes



b. PDR without pre-processing vs. Averroes

Figure 6.4: Verification Results of the Big-Datapath Industrial Benchmarks 2

Table 6.2: Verification Results of the Large Industrial Benchmarks.

Benchmark	Runtime, sec		Frames		CTI Checks		Refinement Clauses		Solver Calls	
	PDR	AVR	PDR	AVR	PDR	AVR	PDR	AVR	PDR	AVR
mult_hold_1	T.O.	0.02	3	1	3	4	131720	3	6331714	22
mult_hold_2	T.O.	0.02	3	1	3	4	13008	3	1032718	22
mult_hold_3	T.O.	0.02	3	1	3	4	5986	3	546718	22
mult_hold_4	T.O.	0.04	2	2	2	8	1	8	10	56
mult_viol_1	116.15	0.05	2	1	2	5	3	3	21	15
mult_viol_2	256.32	0.18	2	1	2	5	2	3	16	15
mult_viol_3	1483.92	0.75	2	1	2	5	2	3	16	15
mult_viol_4	T.O.	0.54	2	7	2	30	262365	29	8177493	335
mult_viol_5	T.O.	11.88	2	22	2	47	252987	69	7961852	3040
mult_viol_6	T.O.	299.23	2	115	2	120	247035	275	8102702	55251
mult_viol_7	T.O.	1884.52	2	451	2	536	239809	754	7826919	425892
fifo_hold_2	14.87	1.35	12	8	12	115	4030	115	94230	1574
fifo_hold_3	201.58	12.88	20	16	20	355	17772	317	612147	9711
fifo_hold_4	746.94	264.85	31	28	36984	1804	24609	1403	1611008	103590
M0+_hold	T.O.	917.76	8	17	5315	1154	3783	911	75898	45755

a. PDR was run *with* pre-processing.

mult_hold_1	T.O.	0.02	2	1	134	4	217	3	1307	22
mult_hold_2	T.O.	0.02	2	1	257	4	512	3	3147	22
mult_hold_3	T.O.	0.02	2	1	521	4	612	3	2915	22
mult_hold_4	T.O.	0.04	2	2	189	8	250	8	1611	56
mult_viol_1	0.62	0.05	2	1	256	5	383	3	1439	15
mult_viol_2	10.86	0.18	2	1	386	5	532	3	2129	15
mult_viol_3	219.93	0.75	2	1	537	5	798	3	3396	15
mult_viol_4	T.O.	0.54	2	7	181	30	284	29	1821	335
mult_viol_5	T.O.	11.88	2	22	191	47	252	69	1596	3040
mult_viol_6	T.O.	299.23	2	115	177	120	273	275	1751	55251
mult_viol_7	T.O.	1884.52	2	451	179	536	260	754	1660	425892
fifo_hold_2	21.12	1.35	16	8	7191	115	5544	115	192592	1574
fifo_hold_3	1252.89	12.88	29	16	28402	355	39937	317	2525164	9711
fifo_hold_4	T.O.	264.85	32	28	98122	1804	153114	1403	7618089	103590
M0+_hold	T.O.	917.76	8	17	5532	1154	4363	911	77808	45755

b. PDR was run *without* pre-processing.

benchmarks (about 6147 lines of code). Thus, this performance handicap did not appear in the runtime graph of word-level Averroes, shown in Fig. 6.1.

6.4 Emperical Results for Large Industrial Benchmarks

Table 6.2 shows the results of our experiments on the 15 large benchmarks; time-outs are indicated as T.O. As with the generic benchmarks, Averroes was faster than PDR across this entire set of 15 benchmarks. PDR had particular difficulty with the

multiplier benchmarks. PDR timed out on eight out of the eleven cases. A possible explanation for this behavior is that the combinational logic in the multiplier benchmarks, which involves wide (32- to 256-bit) datapath signals, led to bit-level formulas that were too large and complicated for PDR to handle effectively. An examination of the runtime per solver call for mult_hold_2 and fifo_hold_2 confirms this. These two benchmarks have similar sizes in terms of state bits, but mult_hold_2 leads to an AIG whose size is more than 14 times larger than that of fifo_hold_2, as shown in Table 6.1. PDR made 3,147 solver calls in 10,000 seconds for the multiplier benchmark, averaging about 3.18 seconds per call. The corresponding data for the FIFO benchmark were 192,592 calls in 21.12 seconds, an average of 110 micro seconds per call which is more than four orders of magnitude faster. Additionally, the peculiarly low number of solver calls for mult_hold_4 in Table 6.2-a seemed too suspicious; on closer examination we found out that the first 10 calls were very quick, but the solver timed out on the 11th. This again suggests a difficult formula that thwarted the solver.

In contrast to PDR's performance, Averroes was able to solve all 11 cases, most in fractions of a second. Other performance metrics, such as the number of net refinement clauses and number of solver calls, are significantly less than those for PDR suggesting that datapath abstraction was effective in reducing the "size" of the reachability search space and that the abstract refinement clauses were much stronger than their bit-level counterparts in pruning the space. The cases requiring longer runtimes, about 30 minutes for mult_viol_7, were due to extremely long counterexample traces that require the traversal of many frames which, in turn, translate into many solver calls. For instance, the counterexample trace for mult_viol_7 consisted of 1002 transitions which required the traversal of 451 frames and making 425,892 solver calls.

The three FIFO benchmarks involved checking a "read-after-write" property for the FIFO entries. The FIFO depths (number of entries) ranged from 4 (for fifo_hold_2)

to 16 (for fifo_hold_4) and each benchmark had two FIFOs whose width is 32 bits and two FIFOs whose width is 16 bits. Again, Averroes outperforms PDR on these benchmarks, on average being about 20 times faster. This is another indication of the effectiveness of datapath abstraction. To dramatize this, we carried out a parametric experiment by increasing the width of the FIFO entries. As expected, the runtime of Averroes did not change, whereas the runtime of PDR exhibited exponential behavior. However, all three verifiers exhibited exponential behavior as FIFO depths were increased! Upon reflection, this too should have been expected since FIFOs are basically "small" memories and datapath abstraction alone is insufficient to handle them. We present in Table 6.3 data showing the performance of Averroes when it is augmented with the structural memory abstraction described in [65]. This type of abstraction can be layered on top of any model checking verifier and can certainly be added to PDR. But as the column labeled AVR_MA in this table shows, memory abstraction scales the performance of Averroes only to a FIFO depth of 32. Further scaling requires integrating memory abstraction with datapath abstraction of the memory addresses, as explained in Section 4.2. This is shown in column AVR_MAA. Clearly the combination of memory abstraction and memory *address* abstraction yields a verification flow that is largely independent of memory size. The linear increase in the runtime of Averroes is due to the bit-level feasibility checks on wider memory addresses as memory size increases.

The last benchmark in Table 6.2 is the SEQX instance of the Cortex-M0+. The verification goal here was to show that the M0+ core is self-equivalent when 41 of its state bits are left uninitialized on reset (i.e., their initial value is X or don't care). Specifically, SEQX holds when none of these don't-care values propagate to observable outputs. Effectively, the verifier is establishing the state equivalence of 2^{41} possible initial states. We should note that SEQX becomes quite trivial if the number of uninitialized state bits is small. In fact, bit-level verifiers can quickly

Table 6.3: Runtimes (in Seconds) of FIFO on Various Depths

depth	State Bits	PDR	AVR	AVR_MA	AVR_MAA
2^2	474	14.87	1.35	1.8	8.28
2^3	866	201.58	12.88	10.92	20.57
2^4	1642	746.94	264.85	120.51	21.93
2^5	3186	T.O.	T.O.	2538.49	23.31
2^6	6266	T.O.	T.O.	T.O.	19.17
2^7	12418	T.O.	T.O.	T.O.	24.02
2^8	24714	T.O.	T.O.	T.O.	20.58
2^9	49298	T.O.	T.O.	T.O.	21.15
2^{10}	98458	T.O.	T.O.	T.O.	27.9
2^{11}	196770	T.O.	T.O.	T.O.	29.02
2^{12}	393386	T.O.	T.O.	T.O.	23.57
2^{13}	786610	T.O.	T.O.	T.O.	33.79
2^{14}	1573050	T.O.	T.O.	T.O.	46.85
2^{15}	3145922	T.O.	T.O.	T.O.	57.04
2^{16}	6291658	T.O.	T.O.	T.O.	79.19

solve such problems using structural hashing techniques. However, as the number of uninitialized state bits increases, structural hashing ceases to be effective (not very many equivalent signals to merge) and bit-level verifiers fail. This is clearly shown in Table 6.2: PDR was not able to prove self-equivalence; Averroes required about 15 minutes to show that SEQX holds for M0+.

6.5 Effects of Advanced Features in Averroes

In Chapters IV and V, we discussed two advanced abstraction optimizations, circuit components decomposition and property-directed concretization; and two advanced refinement optimizations, localized datapath lemmas and constant propagation. To demonstrate the effectiveness of these optimizations, we need a reasonably large number of benchmarks that require many datapath refinement iterations for the proofs. Among our industrial benchmark suites, the big-datapath benchmarks meet this requirement. Thus, we focus on these benchmarks in this section. To check the effectiveness of the advanced abstraction optimizations, circuit components decomposition and property-directed concretization, we ran Averroes without each of the

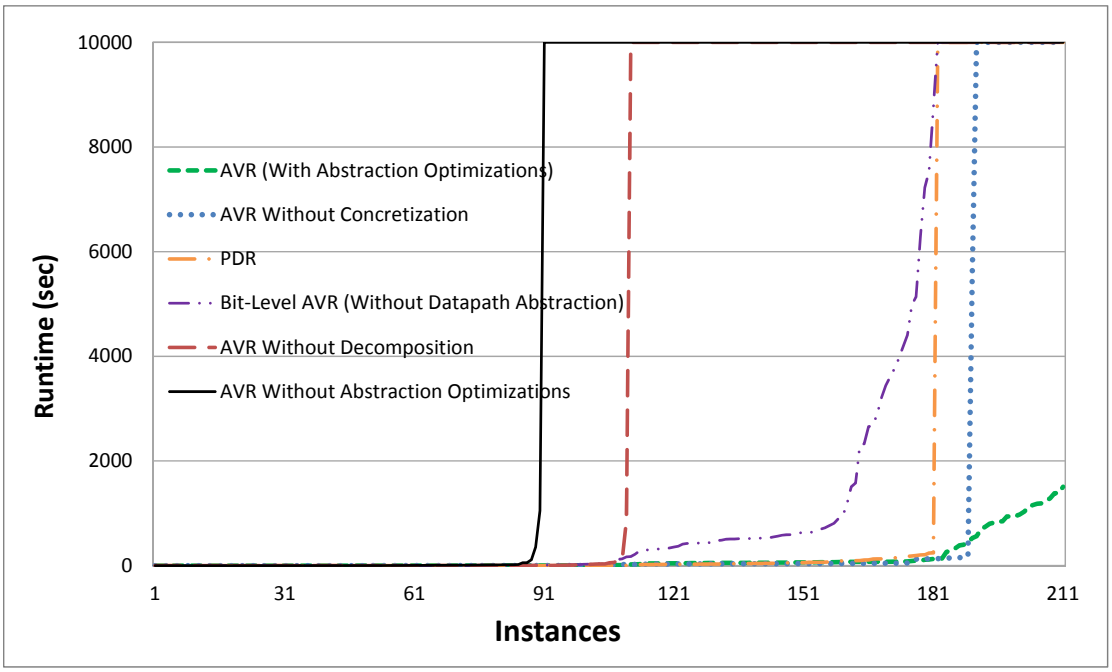


Figure 6.5: Comparison of PDR and Averroes With Various Settings

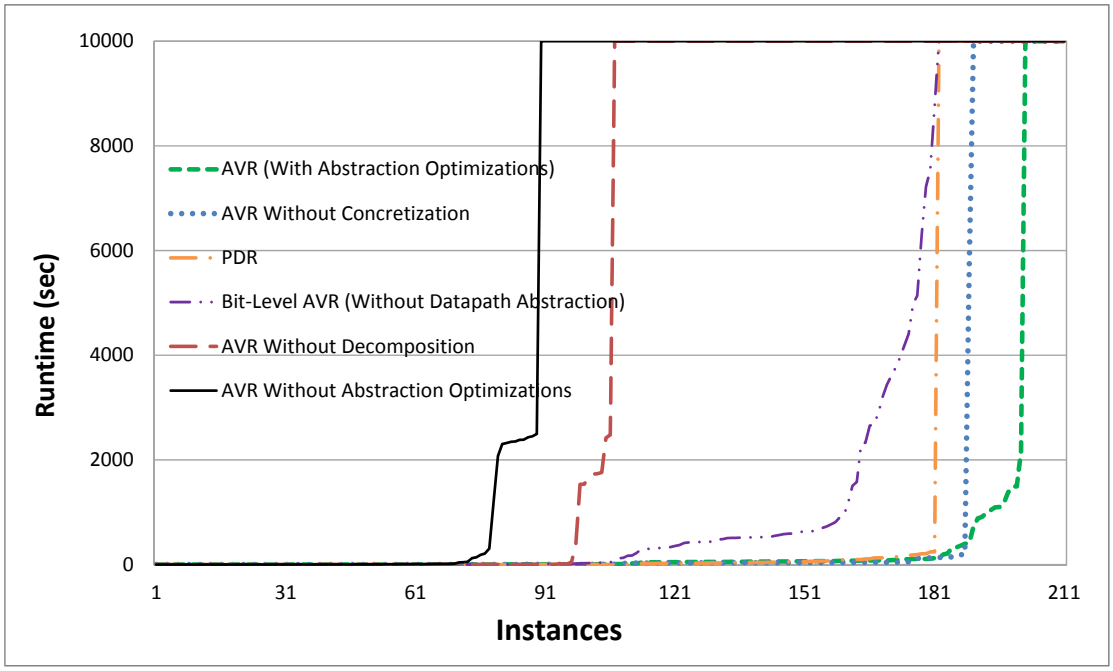


Figure 6.6: Comparison of PDR and Averroes With Various Settings (Advanced Refinement Optimizations are Disabled)

optimizations and Averroes with neither optimizations. Fig. 6.5 shows the runtime distribution with a timeout of 10,000 seconds. As can be seen in the green dashed line and the black solid line, Averroes (with all the optimizations) finished all the 211 proofs, but it finished only about 40% of them without the abstraction optimizations. Between the two optimizations, the effect of the circuit components decomposition optimization is more dramatic; Averroes finished the proofs in only about half of the instances as shown by the red long-dashed line. This dramatic difference shows how important the classification as data or control is. The misclassification can cause many iterations of datapath refinements, leading to the slower runtimes.

The effect of the property-directed concretization optimization is less dramatic, but it leads to a non-trivial difference; Averroes timed out in about 10% of the entire instances without the concretization optimization. We can achieve the best results by combining the two optimizations. Circuit components decomposition separates entangled signals, so property-directed concretization can detect a smaller part of a circuit to be concretized. This yields a smaller formula, which leads to the faster runtime of a solver and thus Averroes. Therefore, Averroes with the two optimizations proved all the instances, which could not be achieved with only one of them.

It is also interesting to compare Averroes with bit-level approaches. The orange dash-dot line and purple dash-dot-dot line represent the results of PDR and bit-level Averroes (i.e., Averroes without datapath abstraction) respectively. They both proved 181 instances (about 86% of the entire set of instances) before timing out, but bit-level Averroes is much slower than PDR. It took only about 300 seconds for PDR to finish the proofs of the 181 instances, but it took about 8,800 seconds for bit-level Averroes. This again shows that the performance of Averroes can be further improved by implementation level optimizations. Even without those optimizations, our (word-level) Averroes finished all the proofs in about 1,500 seconds, including the 30 instances that bit-level approaches could not solve in 10,000 seconds.

Table 6.4: Number of Learned Datapath Lemmas

Index	No-Optimizations	Localized-Lemma	Constant-Propagation	Both-Optimizations
1	7560	4	7374	4
2	36	36	9	9
3	189	16	14	14
4	14	16	14	14
5	2217	18	2216	18
6	2208	20	2206	20
7	2236	20	2243	20
8	2217	21	2232	21
9	2224	22	2220	22
10	3671	22	2221	22
11	2227	22	2226	22
12	2238	24	2243	24
13	402	24	576	26
14	2221	36	2229	36
15	100	69	94	69
16	102	74	98	74
17	102	82	98	82
18	166	218	223	95
19	194	131	188	131
20	214	211	164	132
21	173	147	175	137
22	182	141	182	141
23	192	141	190	141
24	190	142	190	142
25	184	142	182	142
26	180	143	186	143
27	192	146	182	146
28	196	148	188	148
29	301	168	308	153
30	278	193	172	154
31	309	157	358	159
32	316	160	330	165
33	541	277	541	277
34	220	290	335	303
Average	999.76	102.38	953.15	94.29

Fig. 6.5 shows the performance results with the datapath refinement optimizations enabled. Without the refinement optimizations, Averroes timed out in 10 instances (about 5% of the instances) as shown by the green dashed line of Fig. 6.6. It is obvious that they helped improve the performance of Averroes, but their effects are not clearly observable in this graph. Thus, to highlight the effects of the datapath refinement optimizations, we disabled advanced abstraction optimizations and compared the number of datapath lemmas learned during the proof instead of runtime

Table 6.5: Runtimes (seconds) of PDR and AVR With Various Settings

	without refinement optimizations	with refinement optimizations
AVR Without Abstraction Optimizations	5905.21	5743.68
AVR Without Decomposition	5049.2	4793.63
AVR Without Concretization	1089.87	1042.68
AVR	575.59	137.5
Bit-Level AVR (Without Datapath Abstraction)	1942.16	
PDR	1446.35	

as shown in Table 6.4. After the disabling of the abstraction optimizations, Averroes solved 90 out of 211 instances in 10,000 seconds. Among the 90 benchmarks, only 34 benchmarks required datapath refinements during the proofs, so we made the comparison of the number of datapath lemmas in these 34 benchmarks. Table 6.4 lists the 34 benchmarks that yielded datapath lemmas. As the table makes clear, the localized datapath lemmas optimization yields about a 10 times smaller number of datapath lemmas on average. In the best case, the first instance, it reduced the number of datapath lemmas from 7,560 to only 4. This dramatic effect comes from a localized datapath lemma that focuses on a small subset of circuit components. The localized lemma directly and effectively constrains the datapath components involved, which leads to this huge reduction. The effect of constant propagation is not as impressive; it leads to about a 1.05 times (up to 13.5 times) smaller number of datapath lemmas on average. Constant propagation turns a localized lemma into a globalized datapath lemma, which can simultaneously constrain scattered datapath components, and this result shows the limitation of a globalized datapath lemma. Even though the effect of the constant propagation is limited on average, it sometimes plays an important role when the localized datapath lemmas optimization is not that effective. This effect can be seen in the 2nd, 18th, and 20th instances. Using only the localized datapath lemmas optimization, we had no improvement, but with the addition of the constant propagation, the improvement was marked: the number of datapath lemmas was reduced by about two thirds with the combined optimizations.

The runtime comparison gives us a better intuition about the performance impact

of our optimizations. As can be seen in Table 6.5, the average runtime of Averroes with all the optimizations (137.5 seconds) is about 43 times less than that of Averroes without any optimizations (5,905.21 seconds). Averroes (with all the optimizations) is also faster than PDR by about 11 times. In the table, we counted the runtime of an unsolved instance as the time-out value, 10,000 seconds. Thus, the actual runtime gap can be much longer than this, because Averroes (with all the optimizations) proved all the instances before timing out when the other ones timed out in some instances[2]. The comparison between bit-level Averroes and PDR can be misleading because of the timed-out instances; their average runtimes are reduced to 606.6 and 28.62 seconds respectively when we exclude the 30 out of 211 instances that timed out. That is, bit-level Averroes is about 21 times slower than PDR.

In sum, our experimental results clearly demonstrate that the scalability of Averroes is significantly improved by the advanced abstraction and refinement optimizations we have developed. In the 211 large-datapath benchmarks, Averroes with the optimizations proved about more than two times more instances with a speed-up of more than 43 times. Among the optimizations, the effects of the circuit components decomposition and the localized datapath lemmas were much more notable than the other optimizations. However, the other optimizations also made measurable contributions when these optimizations were less effective.

[2]When we conducted the same experiment with a time-out of 15,000, Averroes (with all the optimizations) was about 65 times faster than Averroes without any optimizations and about 16 times faster than PDR

CHAPTER VII

Conclusions and Future Work

In this chapter, we discuss conclusions and future directions for our research.

7.1 Conclusions

The computational complexity of verification problems is often caused by irrelevant details associated with them; we can achieve a scalable way to attack those problems by abstracting away the irrelevant details. We combine two orthogonal abstraction techniques, structural abstraction and approximate reachability, which yields a scalable system for the verification of control-centric properties in hardware designs containing wide datapaths and complex control logic. Other abstraction approaches have been shown to work well in different domains, but structural abstraction seems to provide the most scalability for the particular control logic bugs targeted in our approach.

Our verification system, Averroes, introduced in Chapter III, applies structural abstraction, which classifies circuit components into datapath and control logic and abstracts the datapath as uninterpreted functions or predicates, producing abstract formulas in the EUF logic. Averroes then conducts an IC3/PDR style approximate reachability computation on the abstract formulas. Averroes defines the abstract state

space with a set of state terms and predicates, which is fixed during the reachability computation. If the granularity of this abstract state space is not fine enough to prove the property, the granularity is increased by introducing more state terms or predicates during refinement. There are two key advantages in our approach. First, the reachability computation is performed at the abstract level in EUF logic. Thus, every query to a solver is made with EUF formulas, which tend to be simpler and easier to solve than the corresponding bit-level equivalents. Second, the reachability computation is conducted in the small fixed-size abstract state space. This makes the computation more efficient and guarantees the termination of the computation.

The performance of Averroes, however, is highly dependent on the initial abstraction and refinement strategies. Inappropriate choices of these can slow the computation at the abstract level and lead to a huge number of abstraction refinement iterations. For better initial abstraction, Averroes introduced the circuit components decomposition and the property-directed concretization optimizations described in Chapter IV. The optimizations detect misclassification of datapath and control logic by a structural analysis, and then correct the misclassification by decomposing circuit components in the word-level netlist and then concretizing a small part of the netlist. For better refinement, Averroes introduced the localized datapath lemmas and the constant propagation-based generalization optimizations detailed in Chapter V. The localized lemmas optimization derives datapath lemmas that constrain datapath components directly, and thus effectively, by means of their local variables. Those datapath lemmas, called localized datapath lemmas, allow us to refute a large number of spurious counterexamples with a small number of simple lemmas. The constant propagation-based generalization optimization applies constant propagation to datapath lemmas to eliminate redundant terms, yielding more compact and powerful datapath lemmas.

Our experimental evaluation, presented in Chapter VI, demonstrated that Aver-

roes outperforms a bit-level approach significantly on three different sets of industrial benchmarks. This result suggests that scalability is quite achievable by augmenting bit-level reasoning with RTL word-level abstractions. In the subsequent experiment, we examined the effect of each of the four advanced abstraction and refinement optimizations in Averroes. The effects of the circuit components decomposition and the localized datapath lemmas optimizations were much more dramatic than those of the other optimizations, showing that the decomposition of unnecessarily concatenated circuit components and the derivation of localized datapath lemmas are indispensable parts of our abstraction and refinement framework.

7.2 Future Work

While this thesis presented a complete and fully automatic procedure for verifying the safety properties of a hardware design, it could be further extended in a number of ways, as follows.

1. **Supporting liveness properties**

   Averroes currently supports only safety properties, but we can extend it for liveness properties. One easy solution would be to adopt the liveness-to-safety conversion algorithm proposed in [73, 74]. We may have to make an extra effort to make it suitable for abstract-level verification. This extension will allow us to support various features in SystemVerilog assertion (SVA) [75], which is widely used in industry.

2. **Investigating a more efficient way to derive a localized datapath lemma**

   We demonstrated that a localized datapath lemma can effectively constrain datapath components. Currently, we use a two-step procedure to derive the localized lemma. We first derive a datapath lemma from a solution cube repre-

sented by global variables such as primary inputs and state variables. We then convert it into a localized lemma by using a structural analysis followed by satisfiability checks. Instead of this two-step approach, we can derive a localized lemma directly from a solution cube represented by local variables. When we represent a solution cube with local variables, we end up with a solution cube consisting of many equalities between each local variable and its corresponding term. Thus, one main challenge of this new approach would be to efficiently compute minimal unsatisfiable subsets from the solution cube including many term-level literals.

3. **Constructing a database of datapath lemmas**

   Some datapath lemmas can be shared across many verification problems originated from similar designs. Thus, we can avoid re-deriving the same datapath lemma by constructing a database of the lemmas. There are two challenges in this approach. First, if we introduce many datapath lemmas at the beginning of each verification run, a formula passed to an SMT solver will become large. This can slow down an SMT solver significantly. Thus, we need to select a small number of relevant datapath lemmas from the database at each verification run. Second, we may need to modify some of the datapath lemmas in the database for different variable names in different designs.

4. **Extending the memory abstraction technique**

   We adopted Bjesse's memory abstraction technique and made an improvement for our abstract-level reasoning. We have tested our improved technique, memory address abstraction, only with FIFOs. Thus, we need to test our technique with different types of memories such as stack and content-addressable memory (CAM). Furthermore, some opportunities for improving Bjesse's memory abstraction technique remain. For example, we might be able to replace his simulation-based refinement process in the abstraction technique with another

one utilizing an SMT solver.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV'00. Springer Berlin / Heidelberg (2000) 154–169

[2] Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI'11, Springer-Verlag (2011) 70–87

[3] Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD'11, IEEE (2011) 125–134

[4] Lee, S., Sakallah, K.A.: Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In: CAV'14, Springer (2014) 849–865

[5] Rota, G.C.: The number of partitions of a set. American Mathematical Monthly (1964) 498–504

[6] Silva, J.P.M., Sakallah, K.A.: GRASP-new search algorithm for satisfiability. In: ICCAD'96, IEEE Computer Society (1996) 220–227

[7] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC'01, ACM (2001) 530–535

[8] Een, N., Sörensson, N.: Minisat: A SAT solver with conflict-clause minimization. Sat **5** (2005)

[9] Smullyan, R.M.: First-order logic. Courier Corporation (1995)

[10] Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. Handbook of satisfiability **185** (2009) 825–885

[11] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS'08. Springer (2008) 337–340

[12] Dutertre, B., De Moura, L.: The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf (2006)

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

[14] IEEE Standard VHDL Language Reference Manual: IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002). (2009)

[15] Alpern, B., Schneider, F.B.: Defining liveness. Information processing letters **21**(4) (1985) 181–185

[16] Mneimneh, M., Sakallah, K.: SAT-based sequential depth computation. In: ASP-DAC'03, ACM (2003) 87–92

[17] Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD'07, IEEE (2007) 173–180

[18] Manna, Z.: Temporal verification of reactive systems: safety. Volume 2. Springer Science & Business Media (1995)

[19] Agnew, D., Claesen, L., Camposano, R.: Automated high-level verification against clocked al-gorithmic specifications. In: IFIP WG10.2'93. Volume 32., Elsevier Science Ltd (1993) 147

[20] Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: CAV'94, Springer (1994) 68–80

[21] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. Springer (1982)

[22] Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: ISP'82, Springer (1982) 337–351

[23] Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking. Springer (2008) 1–26

[24] Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Tools for Practical Software Verification. Springer (2011) 1–30

[25] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. Springer (1981)

[26] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In: POPL'83, ACM (1983) 117–126

[27] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. TOPLAS'86 **8**(2) (1986) 244–263

[28] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS'90, IEEE (1990) 428–439

[29] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)

[30] Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: A new symbolic model checker. STTT **2**(4) (2000) 410–425

[31] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS'99, Springer-Verlag (1999) 193–207

[32] Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1) (2001) 7–34

[33] Silva, J.P.M., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. Computers, IEEE Transactions on **48**(5) (1999) 506–521

[34] McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV'03, Springer (2003) 1–13

[35] Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD'00, Springer (2000) 127–144

[36] Bjesse, P., Claessen, K.: SAT-based verification without state space traversal. In: FMCAD'00, Springer (2000) 409–426

[37] Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press (1994)

[38] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (September 2003) 752–794

[39] Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and SAT-based reachability in hardware model checking. In: FMCAD'12. (2012) 173–181

[40] Baumgartner, J., Ivrii, A., Matsliah, A., Mony, H.: IC3-guided abstraction. In: FMCAD'12, IEEE (2012) 182–185

[41] Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT'12. Springer (2012) 157–171

[42] Cimatti, A., Griggio, A.: Software model checking via ic3. In: CAV'12, Springer (2012) 277–293

[43] Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: TACAS'14. Springer (2014) 46–61

[44] Welp, T., Kuehlmann, A.: Qf bv model checking with property directed reachability. In: DATE'13, EDA Consortium (2013) 791–796

[45] Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV97, Springer (1997) 72–83

[46] Colón, M.A., Uribe, T.E.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV98, Springer (1998) 293–304

[47] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Model Checking Software. Springer (2003) 235–239

[48] Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In: Integrated formal methods, Springer (2004) 1–20

[49] Jain, H., Kroening, D., Sharygina, N., Clarke, E.: Word level predicate abstraction and refinement for verifying RTL verilog. In: DAC'05. (june 2005) 445–450

[50] Wang, D., Jiang, P.H., Kukula, J., Zhu, Y., Ma, T., Damiano, R.: Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In: DAC'01, ACM (2001) 35–40

[51] Chauhan, P., Clarke, E., Kukula, J., Sapra, S., Veith, H., Wang, D.: Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In: FMCAD'02, Springer (2002) 33–51

[52] Wang, C., Li, B., Jin, H., Hachtel, G.D., Somenzi, F.: Improving ariadne's bundle by following multiple threads in abstraction refinement. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **25**(11) (2006) 2297–2316

[53] Hojati, R., Brayton, R.K.: Automatic datapath abstraction in hardware systems. In: CAV'95, Springer (1995) 98–113

[54] Paruthi, V., Mansouri, N., Vemuri, R.: Automatic data path abstraction for verification of large scale designs. In: ICCD'98, IEEE (1998) 192–194

[55] Johannsen, P.: Speeding up hardware verification by automated data path scaling. PhD thesis, Christian-Albrechts Universität Kiel (2002)

[56] Bjesse, P.: Word level bitwidth reduction for unbounded hardware model checking. Formal Methods in System Design **35**(1) (2009) 56–72

[57] Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV'02, Springer (2002) 78–92

[58] Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Reveal: A Formal Verification Tool for Verilog Designs. In: LPAR'08. Volume LNCS 5330., Doha, Qatar, Springer (November 2008) 343–352

[59] Brady, B.A., Bryant, R.E., Seshia, S.A.: Learning conditional abstractions. In: FMCAD'11, IEEE (2011) 116–124

[60] Brady, B.A., Seshia, S.A., Sinha, R., Lahiri, S.K., Bryant, R.E.: A user's guide to UCLID version 3.1. (2015)

[61] Andraus, Z.S., Sakallah, K.A.: Automatic abstraction and verification of verilog models. In: DAC'04. (2004) 218–223

[62] Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Refinement Strategies for Verification Methods Based on Datapath Abstraction. In: ASP-DAC'06, Yokohama, Japan (January 2006) 19–24

[63] Brady, B.A., Bryant, R.E., Seshia, S.A., O'Leary, J.W.: ATLAS: automatic term-level abstraction of RTL designs. In: MEMOCODE'10, IEEE (2010) 31–40

[64] Mitchell, T.M., et al.: Machine learning (1997)

[65] Bjesse, P.: Word-level sequential memory abstraction for model checking. In: FMCAD '08. (Nov 2008) 1–9

[66] Brayton, R.K., Hachtel, G.D., McMullen, C., Sangiovanni-Vincentelli, A.: Logic minimization algorithms for VLSI synthesis. Volume 2. Springer Science & Business Media (1984)

[67] Chockler, H., Ivrii, A., Matsliah, A., Moran, S., Nevo, Z.: Incremental formal verification of hardware. In: FMCAD'11. (2011) 135–143

[68] Verific Design Automation. http://www.verific.com

[69] Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning **40**(1) (January 2008) 1–33

[70] ARM: Cortex-m0+ processor. http//www.arm.com/products/processors/cortex-m/cortex-m0plus.php (2012)

[71] Biere, A.: The aiger and-inverter graph (aig) format. Available at fmv.jku.at/aiger (2007)

[72] Mishchenko, A., Case, M., Brayton, R., Jang, S.: Scalable and scalably-verifiable sequential synthesis. In: ICCAD'08, IEEE (2008) 234–241

[73] Long, J., Ray, S., Sterin, B., Mishchenko, A., Brayton, R.: Enhancing abc for ltl stabilization verification of systemverilog/vhdl models. Ganesh Gopalakrishnan University of Utah USA (2011) 38

[74] Schuppan, V., Biere, A.: Efficient reduction of finite state model checking to reachability analysis. International Journal on Software Tools for Technology Transfer **5**(2-3) (2004) 185–204

[75] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language: IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009). (2013)