On Invariants to Characterize the State Space for Sequential Logic Synthesis and Formal Verification

Mike Case

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-46
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-46.html

April 2, 2009

Acknowledgement

I would like to thank my adviser, Bob Brayton, for his support and guidance throughout my studies. From our regular meeting to spontaneous coffee shop rendezvous when I wanted to discuss the latest results, Bob was always available, encouraging, and willing to steer me in the right direction.

Alan Mishchenko who served as my unofficial co-advisor. Alan was always willing to hear my latest crazy ideas, and he initially planted into my head many of the seeds that grew into the methods presented in this thesis.

Lastly, my colleagues at IBM. They provided important feedback on my work, and they employed me during my 18 months of study. This gave me

a grounding that 1) motivated the need to explore scalable algorithms, and 2) the ability to evaluate my techniques in a realistic setting.

On Invariants to Characterize the State Space for Sequential Logic Synthesis and Formal Verification

by

Michael Lee Case

B.S. (Oregon State University) 2004

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering—Electrical Engineering and
Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Robert Brayton, Chair
Professor Andreas Kuehlmann
Professor Lee Schruben

Spring 2009

The dissertation of Michael Lee Case is approved:

Professor Robert Brayton, Chair	Date
Professor Andreas Kuehlmann	Date
Professor Lee Schruben	Date

University of California, Berkeley

Spring 2009

On Invariants to Characterize the State Space for
Sequential Logic Synthesis and Formal Verification

by

Michael Lee Case

Abstract

On Invariants to Characterize the State Space for
Sequential Logic Synthesis and Formal Verification
by
Michael Lee Case
Doctor of Philosophy in Engineering—Electrical Engineering and Computer Science
University of California, Berkeley
Professor Robert Brayton, Chair

Because of the large size of industrial designs, modern sequential logic synthesis and formal verification techniques cannot afford to accurately characterize the state space of a design. This limits the ability to both optimize designs and to formally prove the the designs behave as required.

Invariants are properties that hold in all reachable states. They can be generated in an automated manner, and the set of generated invariants provides a characterization of the design's state space. This characterization can be utilized sequential logic synthesis and formal verification.

In total, this thesis provides 1) a framework to efficiently generate invariants, 2) extensions to sequential logic synthesis to make it more capable of reducing the size of complex designs, and 3) extensions to formal verification to increase its scalability on complex industrial designs.

Professor Robert Brayton, Chair Date

Acknowledgements

I would like to thank my adviser, Bob Brayton, for his support and guidance throughout my studies. From our regular meeting to the spontaneous coffee shop rendezvous when I wanted to discuss the latest results, Bob was always available, encouraging, and willing to steer me in the right direction.

This work would not have been possible without Alan Mishchenko who served as my unofficial co-advisor. Alan was always willing to hear my latest crazy ideas, and he initially planted into my head many of the seeds that grew into the methods presented in this thesis.

Lastly, my colleagues at IBM were very influential in the later stages of my research. They provided important feedback on my work, and they employed me during my 18 months of study. This gave me a grounding that 1) motivated the need to explore scalable algorithms that actually work on industrial designs, and 2) the ability to evaluate my techniques in a realistic setting.

I would like to thank my loving wife Luciena for having the courage to marry a student and for her support and patience through my time at Berkeley.

Contents

List of Examples

Chapter 1

Introduction

1.1 Motivation

This thesis is concerned with building efficient and correct computer chips. Implementing a chip to be smaller, faster, and use less power increases the chip's value to the consumer. Guaranteeing that the chip is designed correctly avoids the need for costly debugging and recalls after the chip has manufactured and shipped.

1.2 Chip Design and CAD Tools

Computer chip designs are normally specified in a **Hardware Description Language** (**HDL**), as illustrated in Figure 1.1. This is a text-based specification that can be used to derive a circuit implementation which can then be manufactured into a **die**, the functional part of the computer chip. The process by which the text based specification is translated into a circuit implementation is referred to as **Computer Aided Design** (**CAD**).

Chip designs can be roughly classified into two types: **analog** or **digital**. Analog designs are typically specified at the transistor level and thus the designer has much

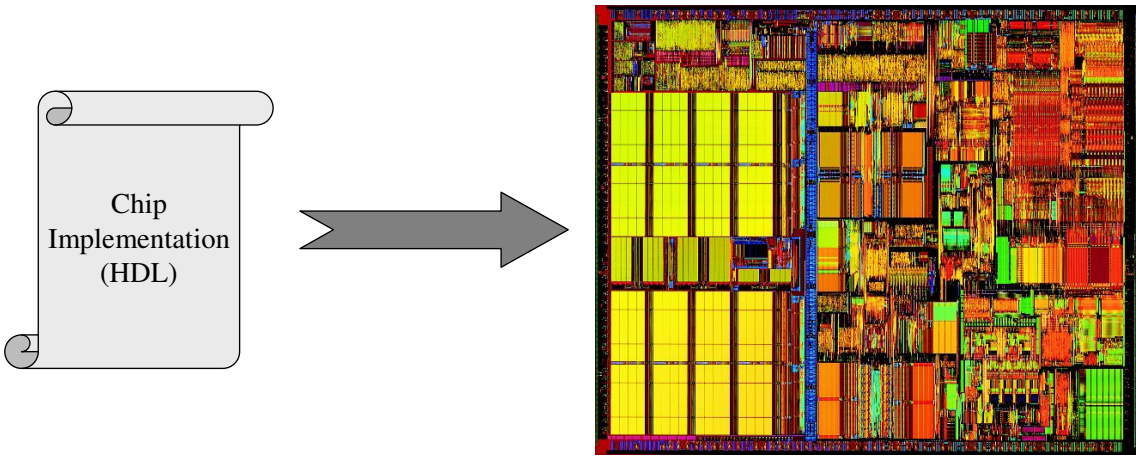


Figure 1.1: Chip designs normally start with a Hardware Description Language (HDL). CAD tools aid in preparing the HDL design for manufacturing. The pictured die is an Intel Pentium 3 CPU [Intel Corporation, 2001].

more control over the final performance of the chip, but this comes at the cost of a complicated design. In contrast, digital designs are specified at the logic level. This allows the designer to specify the intended logical function of the chip yet ignore the details of how the electronics will implement this logic. This abstraction enables much greater design productivity, and for that reason most designs are specified as **digital logic** designs. This thesis is concerned with such designs.

A chip designer typically employs several different **CAD Tools** in his or her work. Each tool has a different purpose. In this thesis we are concerned with two such families of tools: CAD tools that take an existing design and optimize the logic to produce an equivalent yet better design, known as **synthesis** tools, and tools that check that the chip will function as intended, known as **verification** tools. This thesis introduces techniques that can improve the quality of synthesis and verification tools.

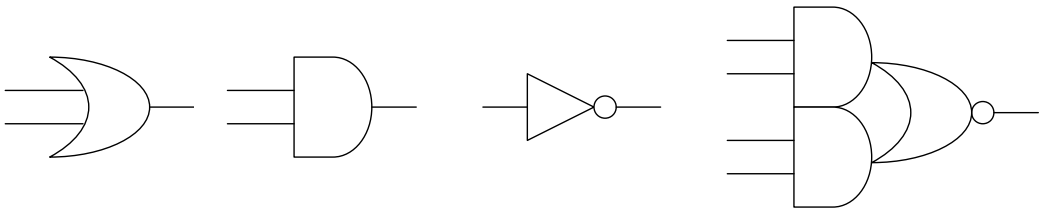


Figure 1.2: Examples of digital logic gates.

1.2.1 Digital Logic Components

Digital logic designs can be specified in high-level English-like languages such as VHDL [Wikipedia, 2009], Verilog [IEEE, 2008], or System C [OSCI, 2008]. While convenient for the human user, English-like languages are burdensome to use within CAD software. Instead, a representation that directly discusses the digital logic implementation, known as an **Register-Transfer Level** (**RTL**) design, is used. In high performance chips such as **Central Processing Units** (**CPUs**), a design is often specified directly in terms of RTL because this tends to result in higher-performance chips. Such RTL designs are the focus of this work.

RTL designs are specified in terms of logic gates, and there are several different types of gates that appear in such designs. While dozens of gate types may be found in an RTL design, an example of four such gate types is illustrated in Figure 1.2. An **or gate** takes two inputs and outputs a 1 if either of these inputs is 1. An **and gate** similarly takes two inputs and outputs a 1 if both inputs are 1. A **not gate** takes a single input and outputs a 1 if the input is 0 and is sometimes referred to as an **inverter**. Additionally, there may be more complex gate types present. For example, consider the **aoi22 gate**. This gate takes 4 inputs and outputs a 1 if certain pairs of inputs are not both 1 at the same time.

A family of gate types is known as a **complete logic family** if any compli-

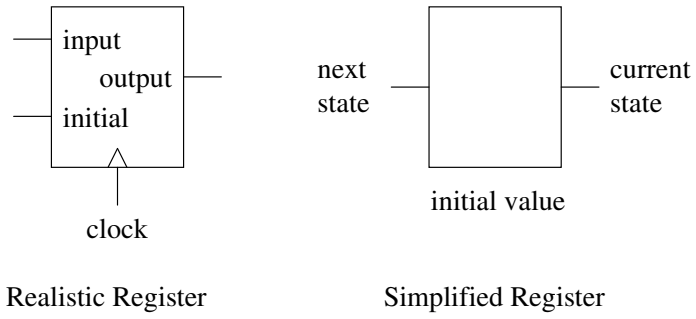


Figure 1.3: Examples of registers that may appear in sequential logic.

cated gate type can be constructed from a combination of gates from the family. The simplest complete logic family has only two members: *and* and *not*. Designs that take gates from only this simple family are known as **And-Inverter Graphs** (**AIGs**) [Kuehlmann and Krohm, 1997]. AIGs have come to be the preferred style for representation of a design inside a CAD tool due to the simplicity of this small logic family.

1.2.2 Registers and Sequential Logic

Logic gates can be arranged in a network such that a single gate in the network takes as input either 1) the output of some other gate in the network, or 2) an input to the network as a whole. Such a structure is often referred to as a **logic network** or **netlist**. Such a logic network computes 1 or more signals as a function over a set of inputs. In order that this function be deterministic, we impose the constraint that the logic network be free of cycles. That is, a logic gate should not take its output as one of its inputs, either directly or through a series of intermediate gates.

In addition to logic gates, a logic network commonly includes **registers**. A register is a state holding element, capable of storing a logic value that was computed in the past. Two examples of registers, a realistic one, and a simplified one that will be used in this thesis, are shown in Figure 1.3. The design has one or more signals referred to

as **clocks** that control the function of the registers. These clocks control the behavior of the design over time, and we shall refer to **time** j as the j'th time step, or the j'th event that has occurred on the clock signals.

An example of a realistic register can be see in Figure 1.3. The register takes as input an initial value that controls the register's output at time 0. After time 0, the output is controlled by the input signal, the clock signal, and the register's internal **state**. When an event on the clock signal, a transition from 0 to 1 for example, occurs the input signal will be read and the result stored in the internal state. This stored value will be output from the register. Thus the clock signal controls the point(s) in time that the register samples, or **latches**, the input.

A design can have multiple clock signals and multiple different types of registers. Through a technique known as **phase abstraction** [Bjesse and Kukula, 2005], the clocks and registers can be normalized such that a CAD tool only sees one clock signal and one type of register. This dramatically simplifies the implementation of the CAD tool, and we will use this simplified view in this thesis. The simplified latch takes an input that we shall refer to as the **next state function** because it computes the value that the register will have at the next point in time. The output shall be referred to as the **current state** because it is simply a reflection of the register's state at the current time. All registers have an implicit **initial state** which is driven by a designated signal in the design, allowing for a constant initial state, a nondeterministic initial state, or an initial state that is a complex function of initial conditions. All simplified registers take as input the single clock, and because there is no variation across registers, we omit this clock signal from all future discussion of registers. The clock is implicitly part of the design but is customarily not drawn. The terms **latch** and register are used interchangeably in this thesis to refer to this simple register model.

A logic network that is free of registers is known a **combinational logic network**.

Because this type of logic network is free of internal state, it is simpler to understand and to process in CAD tools. Such logic networks have been studied extensively in the past, and there now exist high quality CAD tools to optimize and/or verify these networks.

More complex systems can be constructed from logic networks that also include registers. These logic networks incorporate internal state and are referred to as **sequential logic networks**. Traditionally, sequential logic networks have been handled by removing their registers and processing the remaining combinational logic network in isolation. However, tools that take the registers into consideration can generate significantly more-optimal designs and verify more interesting design properties. Unfortunately, the analysis of sequential logic networks is complex, but the techniques presented in this thesis help to mitigate that complexity.

1.2.3 Finite State Machines

Sequential logic networks can be used to implement **Finite State Machines** (**FSMs**). FSMs form the building blocks of more complex digital logic systems by providing a methodology by which a design has a meaningful internal state that evolves in a chosen manner over time.

Definition: *A Finite State Machine (FSM) is a 6-tuple* $(\Sigma, \Gamma, S, s_0, \delta, \omega)$ *where* Σ, Γ *are respectively the sets of inputs and outputs of the FSM.* S *is the set of state variables in the FSM with* s_0 *being the FSM's initial state. At any time* t*, the state at time* $t+1$ *is given by* $\delta(\Sigma, S)$ *and the valuations of the outputs* Γ *are given by* $\omega(\Sigma, S)$*.*

Consider the following example of a small yet realistic FSM.

Example 1. A String-Matching FSM

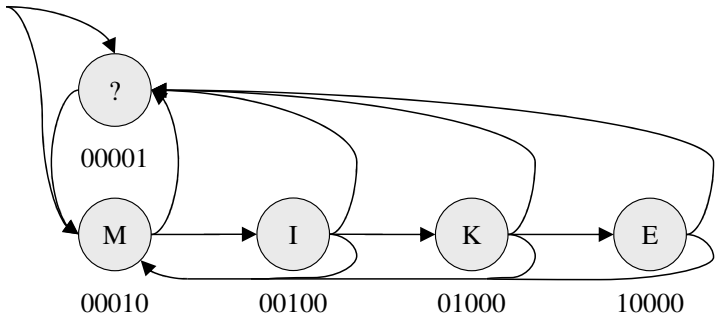


Figure 1.4: State transition graph of a simple FSM.

Suppose we have an FSM that takes as input Σ a sequence of keystrokes and outputs Γ a signal that is 1 whenever the user has typed the string "MIKE". We can define the following states S for this machine:

? *Means that the past input is indecipherable.*

M *Means that the user has just typed "M".*

I *Means that the user has just typed "MI".*

K *Means that the user has just typed "MIK".*

E *Means that the user has just typed "MIKE".*

The FSM assumes exactly one of these states at any given time, and we can graphically describe the evolution of the FSM's state over time with a ***State Transition Graph*** *(**STG**) as illustrated in Figure 1.4.*

*Initially, the FSM starts in states s_0 which is either the **?** or **M** state, depending on the current input. This is illustrated with a set of arrows from an undefined state in the upper-left of the figure. At any state of the FSM, the state at the next point in time shall be either 1) the next state in the **M**-**I**-**K**-**E** sequence if the appropriate input is given, or 2) the state **?** if the current input is not the next key in the sequence, or 3) the state **M** if the current input is wrong but*

*could possibly serve as the starting point for a new **M**-**I**-**K**-**E** sequence. This defines the state transition function δ.*

*An FSM's internal state allows it to produce more meaningful outputs than is possible in a state-free machine. In this case, we want to output a 1 whenever we have seen "MIKE". In other words, the output function ω is simply to output a 1 whenever we are in state **E**.*

FSMs can be easily implemented in digital logic. First, label each state in the FSM with a fixed-length unique binary code, as shown in Figure 1.4. Each bit in this code will be stored in a separate register, and so the number of bits in the code define the number of registers in the design. There are many ways to code the states, and some coding styles result in few registers while others trade registers for a higher-performance implementation. This is referred to as the **state encoding problem** and is beyond the scope of this work. The codes in Figure 1.4 were chosen arbitrarily. The valuation of all the registers is referred to as the implementation's **state vector** or simply **state**.

After the states are encoded, the initial values for each register can be derived by simply noting which states (and corresponding codes) are initial states in the state transition graph. The inputs to the registers, called the next state functions, define the value the registers should store at the next point in time. These can be directly translated from the FSM's encoded δ function. The set of all next state functions shall be referred to as the **transition relation**.

All that remains is to define the outputs of the sequential logic network. This can be derived easily from the FSM's ω function.

It should be noted that the logic implementation of an FSM is not unique. There always exist multiple equivalent implementations, resulting in a spectrum of area, power, and delay trade-offs. This thesis will discuss synthesis and verification, prob-

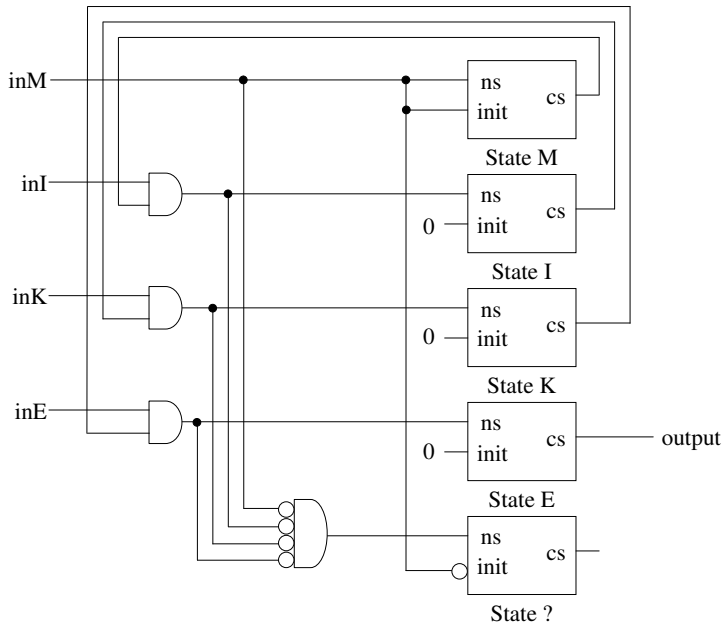


Figure 1.5: Example sequential logic network corresponding to Figure 1.4. Bubbled inputs are a shorthand representation of an inverter feeding the logic gate input.

lems that stem from the non-uniqueness of FSMs. Synthesis is the problem of moving from one implementation to an equivalent but better implementation. Verification involves checking that an implementation is correct or that a pair of implementations are equivalent.

Example 2. A Sequential Logic Implementation of Our String-Matching FSM

As an example of how to implement an FSM as a sequential logic network, here we implement the FSM of Figure 1.4. The implementation that will be developed is illustrated in Figure 1.5.

Using the state encodings given, there are 5 registers in the sequential logic, and each register corresponds to a single state of the FSM. We label the registers accordingly to get registers "State M" through "State ?".

The FSM takes keystrokes as input, and to simplify this suppose we simply consider 4 signals. Input "inM" shall be 1 iff the user typed "M". Inputs "inI", "inK", "inE" are defined similarly.

*From Figure 1.4, we can see that the next state is **M** iff inM=1. The next state is **I** if the current state is **M** and inI=1. Similarly, the next state is **K** if the current state is **I** and inI=1 and the next state is **E** if the current state is **K** and inE=1. Otherwise, the next state is **?**. This defines the transition relation of the sequential logic network.*

*The initial states are similarly given by translation from Figure 1.4. The FSM starts in state **M** if inM=1 and in state **?** if inM=0. It is illegal to start in any other states, and the initial state functions for the corresponding registers are all constant 0.*

*The single output is 1 iff the FSM is in state **E**. This implementation is trivial, just a wire.*

*Note that this implementation is very inefficient. The state **?** is clearly not needed, and the FSM could be implemented with fewer registers. Furthermore, how do we know that this implementation is correct? These questions will be explored below.*

1.2.4 The State Space and Design Complexity

The set of states that are possible to represent with an FSM's registers are known as the **state space**. Note that in general the cardinality of this set of states can be much larger than the number of states present in the original state transition diagram. If an FSM is implemented with N registers then the state space has 2^N states. Many synthesis and verification algorithms perform computations over the state space, and the large size of this space makes those computations difficult.

A pair of states (a, b) in the state space are connected by a **path** if starting from state a the FSM can assume state b after a fixed number of time steps if provided with suitable input stimulus. A **reachable state** r is any state for which there exists a path from an an initial state i to r. An **unreachable state** is any state for which no such path exists. Note that in a manufactured chip an unreachable state will never be observed during operation.

If the reachable states can be enumerated then this information can be leveraged in two interesting ways:

- Optimizations can be performed such that the behavior of the design is not preserved on the unreachable states. Because the unreachable state will never be observed in the lifetime of the chip, any change in behavior will go undetected. This freedom to change the behavior on a set of states provides flexibility to optimize the design in exotic ways, and such optimizations are known as **sequential synthesis** transformations.

- Verification that the design always behaves in a particular manner can be done by simply checking that this behavior holds on each of the reachable states. Because only these reachable states will be observed when the chip is operating, by checking the behavior on the reachable states the behavior of the chip has been checked for all time. This process is known as **formal verification**.

Clearly, there are advantages in partitioning the state space into reachable and unreachable states. A design with N registers has 2^N states. The FSM in Example 2 has only 5 registers for a total of 32 states, and partitioning these 32 states into reachable and unreachable states is fairly straightforward. However, realistic designs are far more complex.

Example 3. The Complexity of Product Machines

While simple FSMs such as seen in Example 2 do exist in practice, complex chip designs can not be constructed manually from a single FSM. Humans have been able to construct complex systems through modularity. By successively partitioning a large design, a designer can limit his focus to a very small window. In this small window an FSM can be used to implement the intended design, and the total design will be composed of the conjunction of several small FSMs from each of these windows. These FSMs often interact with each other in complicated ways.

A ***product machine*** *is a view of a design that includes two smaller designs. It is an important piece of the hierarchy where two sub-designs come together to form a design that is more complex than either of its constituents. While the size of the total design increases, the complexity of its state space increases as well.*

Suppose a product machine is composed of designs α and β. α has R_α registers and S_α reachable states ($S_\alpha \leq 2^{R_\alpha}$). Similarly β has R_β registers and S_β reachable states. Individually, the two designs have 2^{R_α} and 2^{R_β} states in their state space, respectively. However, the product machine has $R_\alpha + R_\beta$ registers and $2^{R_\alpha+R_\beta} = 2^{R_\alpha} \cdot 2^{R_\beta}$ states. The sizes of the state spaces have been multiplied, hence the name product machine.

If the two sub-designs α and β act independently then the product machine will have $S_\alpha \cdot S_\beta$ reachable states. However, often the designs will interact in nontrivial ways. For example, α may take as input one of the outputs of β. In practice, this limits the reachable state space, and the effective reachable state space will be much smaller than $S_\alpha \cdot S_\beta$.

Because the set of reachable states is not predictable, the partitioning of state space into reachable and unreachable states must be done anew on the product

machine. However, the product machine has $R_\alpha + R_\beta$ registers, and this increase in the number of registers to process severely complicates this analysis.

Example 4. The Complexity of Datapath and Control Logic

Realistic chip designs are not built of FSMs alone. They often contain more complex blocks. For example, an ***adder*** *is a block that inputs two binary-encoded numbers and outputs their sum. This binary encoding could be large. For example, in today's CPUs it is common to see 64-bit encodings, resulting in $2 \cdot 64 = 128$ inputs and 64 outputs.*

Logic blocks such as adders manipulate data as it flows through a chip. Multiple such blocks can be arranged in a network to compute complex functions over the chip's data. For example, by chaining adders and multipliers, polynomials can be computed. Such networks of data-processing blocks are known as a ***datapath****.*

Throughput *is a measure of how much computation a chip can complete in a unit of time. In order to increase throughput it is common to* ***pipeline*** *the datapath. This process involves cutting signals in the datapath and inserting registers. The combinational logic between two sets of introduced registers is known as a* ***pipeline stage****. The registers mean that the datapath will require multiple clock cycles to compute a result. In general, a computation will require as many clock cycles as there are pipeline stages, and this cycle requirement is known as the* ***latency****. However, because the registers partition the design, the data in stage j is independent from all stages $i < j$. This allows new computations to start before the previous computation completes, resulting in overlapping waves of computation. These waves result in significantly higher throughput.*

Registers that carry data as it flows through the design are known as ***datapath registers****. For example, the registers that are used for pipelining are datapath*

registers. They do not store the state of an FSM but instead store intermediate results for a larger data computation. One resultant characteristic is that a set of datapath registers typically have a very high percentage of reachable states.

Arithmetic blocks such as adders and multipliers require a large number of logic gates and correspondingly take a large area on the manufactured die. Additionally, the pipelining can introduce undesirable latency into the design. In order to improve this area and timing, computer architects employ a variety of tricks. The nature of these techniques is beyond the scope of this thesis, but it is sufficient to know that these techniques introduce several inputs to the datapath logic that control its function. These ***control inputs*** *must be configured in a particular way to ensure that the design operates as intended.*

FSMs are typically employed to operate the control inputs. An FSM block known as ***control logic*** *is built, and its outputs are tied to the control inputs of the design.*

The resulting design is a composition of datapath and control logic. The datapath has a high percentage of reachable states, and the control logic, because of the nature of product machines, typically has a low percentage of reachable states. Characterizing the reachable state space of the composite machine can be complicated because, similar to a product machine, it is not possible to predict the composite reachable state space from the individual spaces of the two components. The composite machine typically contains a large number of registers, further complicating this analysis.

1.3 Logic Synthesis

1.3.1 The Synthesis Problem

Logic synthesis is the process of optimizing a logic netlist. A **logic synthesis tool** is a CAD tool that inputs a logic netlist, optimizes it in some way, and outputs a netlist with equivalent functionality yet better implementation.

The goal of a logic synthesis tool is to make a logic netlist "better," but the definition of "better" can vary greatly depending on the overall objective. The designer's objective is captured mathematically as a **cost function** inside the synthesis tool, and the synthesis process can be viewed as an optimization problem that is trying to minimize a cost function. Some example cost functions are as follows:

- Often the designer wants to minimize the amount of area that a logic network will take on the die. This reduces the manufacturing cost as more dies can be fit on a single **wafer**. Area can be optimized by implementing the logic network with fewer gates and/or gates that can be arranged more efficiently on the die.

- Power is often another important consideration, and the logic can be optimized to minimize power. **Static power** is related to the area of the logic network and can be optimized with an area-focused cost function. **Dynamic power** can be minimized by re-implementing the logic such that the probability of observing the logic network's signals transitioning is minimized.

- Speed is another common designer objective. Improving the **delay** necessary to propagate logic values through a logic network is important because this enables the network to be run with a higher-frequency clock, improving the speed of the chip. Delay can be optimized by reducing the number of gates that lie on paths from the logic networks input to the outputs.

1.3.2 Technology Independent Synthesis

There are many CAD tools that a designer must use when implementing a chip, and logic synthesis is just one of these tools. Other important tools include **technology mapping**, **placement**, and **routing**. These form a **toolchain** such that the output of logic synthesis is used as the input to technology mapping, technology mapping feeds placement, and placement feeds routing.

Technology independent synthesis is the process of performing logic synthesis in isolation without considering the effects of the logic transforms on the later parts of the toolchain. Technology independent synthesis only considers the logic of the netlist and does not consider factors that affect the later manufacturability of the design. This was the focus of much of the early research in logic synthesis due to its simplicity and limited scope.

Modern CAD tools have shifted focus away from technology independent synthesis and instead invest efforts in **technology dependent synthesis**. This is also known as **physical synthesis**. In a physical synthesis tool, one or more components of the toolchain will be implemented together and performed at the same time. For example, logic synthesis, technology mapping, and placement might be all implemented in the same tool and performed simultaneously. This allows the logic synthesis to see the effect of its transformations on the placed design, allowing it to choose logic transformations that not only satisfy the logic synthesis' cost function but also the cost function in mapping and placement. Such a broader view of the optimization problem results in much more efficient implementations.

Physical synthesis leverages technology independent synthesis algorithms and improves them by integrating other CAD tools. In recent years there has been much research in physical synthesis while technology independent synthesis has been largely ignored. This thesis will discuss technology independent synthesis and will make two

key contributions to the field:

- Sequential logic synthesis techniques will be developed, and this type of synthesis is significantly more powerful than conventional methods. These techniques are new and are commonly developed in technology independent synthesis first due to its simplicity. Later they can be applied to physical synthesis.
- Synthesis will be used to simplify verification problems, and in this domain the goal is to verify properties of a design instead of manufacturing a chip. Physical synthesis makes no sense in this domain and therefore the focus should be on technology independent synthesis.

1.3.3 Combinational and Sequential Synthesis

There are several algorithms that have been developed for technology independent synthesis, and these algorithms can be partitioned roughly into two groups: combinational synthesis and sequential synthesis algorithms. Most practical designs incorporate registers, and hence most designs include sequential logic networks. Combinational and sequential algorithms differ in how they handle the design's registers.

Combinational synthesis algorithms do not change the number or function of the registers. All synthesis algorithms, both combinational and sequential, are constrained to not change the function of the logic network's outputs. Combinational synthesis takes this a step further by not changing the registers either. This simplifies the analysis because the current state functions can be treated as pseudo-inputs and the next state functions can be treated as pseudo-outputs. After performing this abstraction, combinational synthesis techniques focus on the remaining combinational logic network.

Sequential synthesis algorithms are capable of changing the number and function of the registers. They do not abstract the registers and therefore can optimize

the combinational logic and the registers at the same time. This is more complex because it requires the algorithms to understand the state space, but this also enables sequential synthesis to do much more powerful design optimizations that are impossible with combinational synthesis.

1.3.4 Important Synthesis Examples

Let us explore a few synthesis algorithms that are important to in this thesis. Note that logic synthesis has enjoyed more than 40 years of active research, and many algorithms have been developed to optimize logic networks. Here we focus on the few algorithms most relevant to this work. Consequently, many of these synthesis algorithms in this section are sequential. All algorithms will be explained through example, and their effects on the logic of Figure 1.5 will be examined.

Example 5. Synthesis With Cone of Influence Reduction

```
 1: function findCoi(gate, coiSet)
 2:   if (gate ∈ coiSet) then
 3:     return
 4:   else
 5:     add gate to coiSet
 6:
 7:     // Recurse
 8:     if (gate is an AND) then
 9:       for all inputs in of gate do
10:         findCoi(in, coiSet)
11:       end for
12:     else if (gate is a register) then
13:       findCoi(gate.nextState, coiSet)
14:       findCoi(gate.init, coiSet)
15:     end if
16:   end if
17: end function
```

Algorithm 1: Recursive Procedure to Find a COI

*The **Cone of Influence** (**COI**) of a logic gate is the set of gates and registers in its input or in the input of one of its inputs. Typically the COI of a gate is found in a manner similar to what is shown in Algorithm 1. The procedure starts at the gate and recursively traverses the inputs, adding gates to the set* coiSet*. The procedure terminates when no new gates can be added to this set. Note that sometimes a COI computation stops at the registers, but this implementation traverses through registers to process both the register's next state function and initial value.*

A COI of a gate g is a superset of all the logic necessary to produce the values seen at the output of g. If a gate does not belong to the COI of g then that gate can be removed from the logic network without influencing the values observed at g.[1]

*The COI gives rise to a very simple type of logic synthesis optimization: **COI Reduction**. The outputs of a logic network are more important than any other internal signals. In a COI reduction, the COI of each output is computed, and these COI's are unioned. This gives a set of gates such that if a gate k is not in the set then k can be removed without affecting any of the outputs. Therefore the size of the logic network can be reduced without disturbing any of the outputs.*

Figure 1.6 shows the effect of COI reduction on the logic network previously examined in Figure 1.5. In Figure 1.5, the register labeled "State ?" and the subset of the logic driving its next state function are not in the COI of the output. Therefore this logic can be removed, yielding a smaller logic network with equivalent output functionality.

COI reduction is a sequential transformation because it can change the number

[1]Note that in general there may be gates in the COI of g that can also be removed without changing the values at g. This happens because of logical redundancies that frequently occur in logic networks.

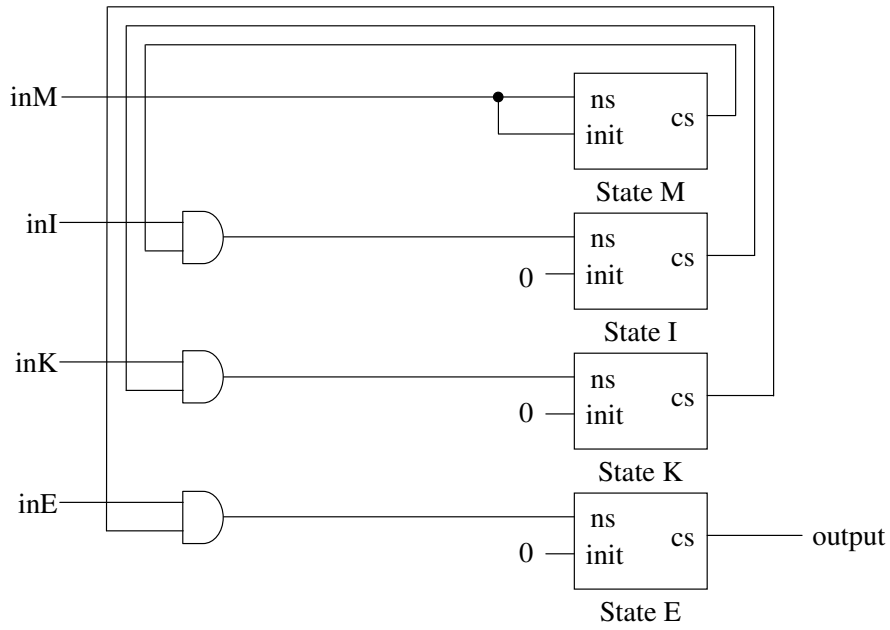


Figure 1.6: Cone of Influence reduction applied to Figure 1.5.

of registers. It is the simplest such sequential transformation, but because of its simplicity it is often used as a subroutine in other synthesis algorithms. It will be used several times throughout this thesis.

Example 6. Synthesis By Merging Signals

Merging of signals is another simple type of synthesis transformation. The ***fanouts*** *of a gate g are the gates in the logic network that take g's output as one of their inputs.* ***Merging*** *is an operation on two gates x and y in a logic network such that the fanouts of x are modified to input the logical value from y instead of x. After the inputs of the fanouts are thus modified, gate x no longer has fanouts and can be removed from the logic network through a subsequent COI reduction*[2]*.*

[2]In general, after x is removed some of its fanins will no longer have outputs and can be removed as well.

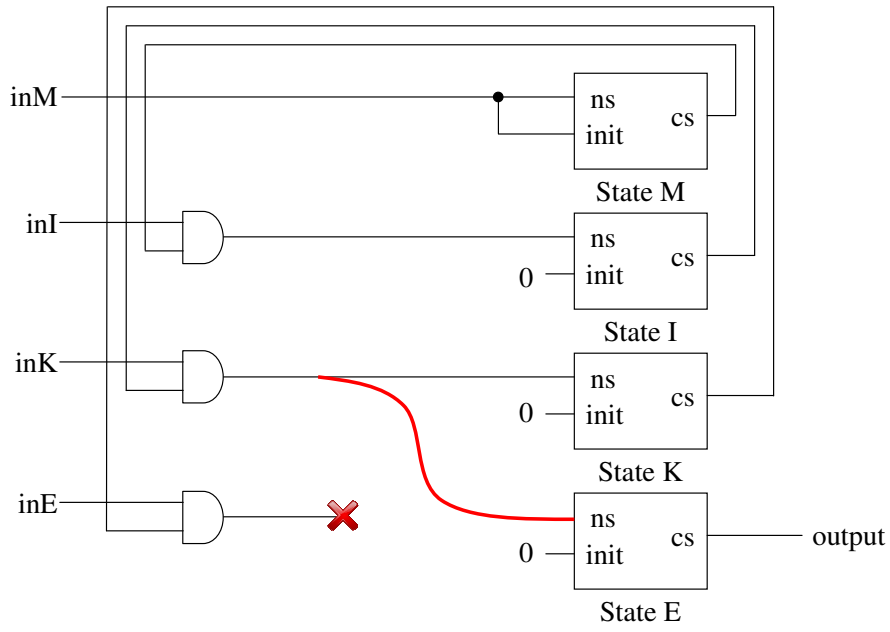


Figure 1.7: Merging applied to Figure 1.6.

Figure 1.7 shows an example of merging within the logic network of Figure 1.6. One AND gate fans out to the register denoted "State K," and the another AND gate fans out to the register denoted "State E." If the gates are merged then a single AND gate will be used to drive values into both registers. The other AND gate is now unnecessary because it has no outputs, and a COI reduction can be used to remove this gate.[3]

In general, merging is a symmetric operation in that in the merging of x and y we could use x to replace y or y to replace x. Such a merge is known as an ***undirected merge****. There are advantages to each of the two merging directions, and the direction that is used is usually selected depending on the logic network structure. Logic networks must be acyclic, with the exception of cycles that involve registers. Cycles that do not involve registers are known as* ***combi-***

[3]Note that this operation changes the behavior of the output and is therefore not a legal logic synthesis transformation. It is shown here for illustrative purposes only.

national cycles *and can be easily avoided by replacing the gate that is furthest from the registers with that which is closest to the registers. More formally, we can define the* ***level*** *of a gate g to be the number of gates on the shortest path from any register to g. If merging always replaces the higher-level gate h with the lower-level gate l then there can be no cycle involving l because the level constraint implies that h cannot fan out to l. Therefore after merging it is impossible for l to fan out to itself, and this is sufficient to guarantee acyclicity of the modified logic network.*

In some applications in this thesis it is advantageous to consider ***directed merges****. In this type of merge, the directionality is explicitly specified. That is, it is legal to replace gate x with gate y but not vis-a-versa. Directed merges are usually seen in applications where only one merge direction is guaranteed to preserve the functionality of the logic network.*

There are many synthesis algorithms that involve merging gates. One of the simplest applications is known as ***SAT/BDD Sweeping****. In this algorithm pairs of gates are proved to be equivalent and then they are merged and COI reduction is called. Because the gates are equivalent, the merge direction is not important and undirected merges are used. The merging is guaranteed to preserve the output behavior of the logic network, as is required with all logic synthesis algorithms. Later in this thesis methods will be introduced where strict equivalence between gates is not proved yet merging still preserves the output behavior. Therefore equivalence is a sufficient but not necessary condition to enable merging to preserve output behavior.*

Example 7. Synthesis With State Re-encoding

The algorithms discussed above require little or no knowledge of the state space of the design. For this reason they are highly ***scalable****, or fast running and able*

to handle large design sizes, but they lack the power to dramatically impact the logic network. As an example of a more powerful (and expensive) optimization, we consider state re-encoding here.

State re-encoding *is the process of changing the binary state encodings on the STG and then re-expressing the logic network in terms of these changed encodings. The logic network is highly sensitive to these encodings, and small changes in the codes used can have a large impact on the number of gates and the delay of the network. Additionally, the length of the codes used can be varied, resulting in a sequential logic network with a different number of registers.*

Logic synthesis tools work on logic networks and not on state transition graphs, making re-encoding of the individual states difficult. While an STG can be extracted from a logic network, often the resultant STG will be too large to manipulate efficiently. Practical re-encoding is therefore limited to only re-encoding the reachable states, but this requires one to exactly know which states are reachable. In large sequential logic networks, computing the exact set of reachable states is computationally infeasible. For all of these reasons, state re-encoding is computationally difficult and therefore rarely used. It is introduced here because later algorithms in this thesis will be compared with state-encoding.

As an example of state encoding, consider changing the state encodings of the sequential logic network shown in Figure 1.6. From Figure 1.4 we know that there are 5 reachable states: "M", "I", "K", "E", and "?". Previously a length-5 encoding was used, resulting in a logic network with 5 registers. For N reachable states, the length of the code words is bounded below by $\lceil \log_2(N) \rceil$. Therefore the states in this simple STG can be encoded using only 3 bits. Suppose the 3 bits of the encoding are labeled A, B, C and the encoding is specified like so:

- *"M" is denoted as* $A = 0, B = 0, C = 0$.

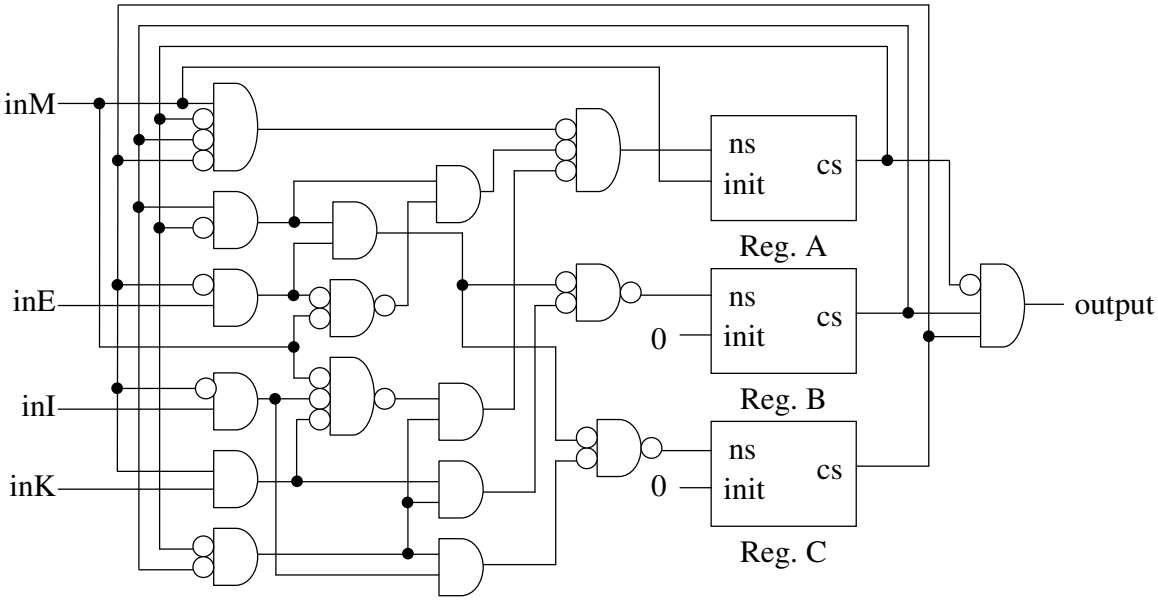


Figure 1.8: State Re-encoding applied to Figure 1.6.

- *"I" is denoted as* $A = 0, B = 0, C = 1$.
- *"K" is denoted as* $A = 0, B = 1, C = 0$.
- *"E" is denoted as* $A = 0, B = 1, C = 1$.
- *"?" is denoted as* $A = 1, B = 0, C = 0$.

With these encodings, a sequential logic network can be derived that has 3 registers: A*,* B*, and* C*. This logic network is shown in Figure 1.8. Clearly, this logic network has many more gates than the original network, but it has fewer registers. Depending on the objective function of the synthesis tool, such a transformation might be either accepted or rejected.*[4].

Example 8. Synthesis With Resubstitution

[4]This re-encoding example was done by hand, but due to its complexity such manual analysis can be error prone. The equivalence between Figures 1.6 and 1.8 was formally verified using IBM's SixthSense tool [Baumgartner, 2006]

Often in synthesis reducing the number of registers is critically important. State re-encoding has the potential to dramatically reduce the number of registers but is computationally difficult. Reduction in the number of registers can be accomplished by several methods, state re-encoding being just one. Any method that reduces the number of registers can be viewed as a weak form of state re-encoding because the encodings of all the reachable states will be shortened by at least one bit.

Resubstitution is a algorithm that can be used to re-express a logic function in terms of other logic functions. More formally, if a logic network has gates A, B, and C then resubstitution can be used to find a function F such that $A = F(B, C)$. $F(B, C)$ can then be constructed in the logic network and merged with A. A COI reduction is then used to completely remove A from the logic network.

In this thesis we will focus on resubstitution applied to the registers. In this case, A, B, and C will be registers and the removal of A is equivalent to the removal of a register from the sequential logic network. Removing registers in this way takes advantage of redundancies in the state encodings such that a bit of the code can be removed without changing the functionality of the STG.

The state encoding used in the STG of Figure 1.4 is called ***one-hot*** *because exactly one of the registers will store a 1 at any point in time with the remaining registers storing 0. A one-hot encoding usually results in a small number of logic gates in the logic network but a large number of registers.*

Any one-hot encoding is redundant in that exactly one of the registers can be expressed as a function of the others. Take for example the FSM shown in Figure 1.5. Register E can be expressed as $E = \overline{M \cdot I \cdot K \cdot ?}$. Introducing this new logic function into the logic network and doing a merge followed by a COI

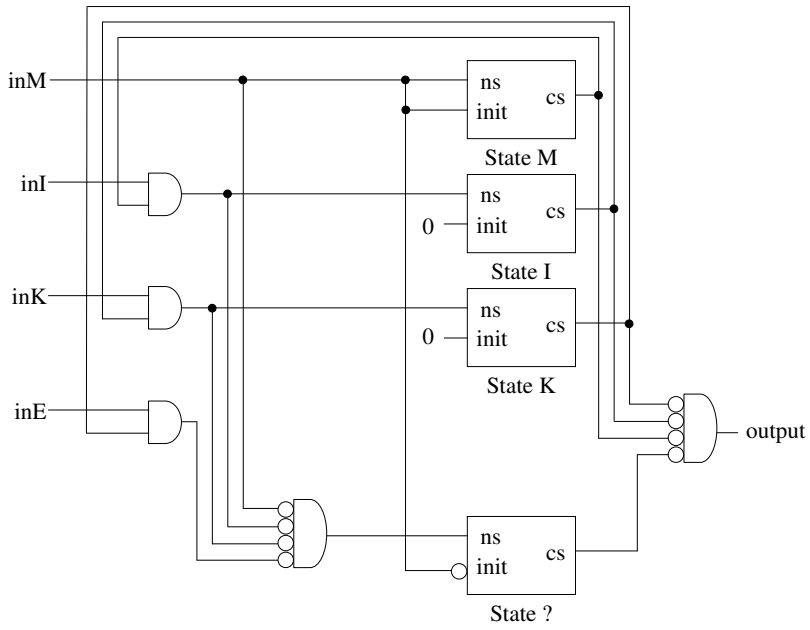


Figure 1.9: Resubstitution applied to Figure 1.5.

reduction gives the logic network shown in Figure 1.9. Some of the register-reduction power present in the state re-encoding algorithm is captured here, and resubstitution can be implemented in an efficient and scalable manner.

1.3.5 Synthesis Challenges

Physical synthesis has been a great advance in the traditional CAD toolchain, but there remains much research to be done in traditional technology-independent synthesis. Specifically, sequential synthesis provides several ways to optimize a logic network by taking advantage of the state space. There are three main challenges in sequential synthesis:

1. Sequential synthesis is inherently difficult because it usually requires knowledge about the state space. The set of reachable states is very difficult to obtain, and optimizations that utilize this information can be computationally expensive.

This thesis will address ways to approximate the set of reachable states and then use this information in an efficient manner to optimize a logic network.

2. All software tools are likely to contain bugs, and sequential logic synthesis tools are no different. In order to prevent a logic synthesis bug from causing an error in a manufactured chip, all synthesis results must be checked. A correct logic synthesis tool will not change the behavior of the outputs of the logic network, and so often one wishes to prove that the outputs of the original and optimized logic networks are pairwise equivalent, guaranteeing that sequential synthesis did not alter the design functionality. This verification problem can be quite challenging. This thesis will discuss advances in the field of formal verification, and these advances can be used to verify the results of sequential synthesis. In this way, formal verification enables sequential synthesis because without verification, the results of exotic sequential synthesis algorithms cannot be trusted or used in industrial settings.

3. Sequential synthesis changes the number and/or function of the registers in the sequential logic network. The registers today serve as points at which a designer can fully understand the operation of his or her design, and by changing the registers a synthesis tool may optimize a design yet produce something that the original designer is unable to understand. This makes debugging and **Engineering Change Orders** (**ECOs**) difficult. This is an important obstacle that stands in the way of mass adoption of sequential synthesis. While this obstacle can be overcome, industry leaders often wish to see tangible benefits of sequential synthesis before re-educating their designers. This thesis will advance the field of logic synthesis, laying the foundation for the later effort to change the way chips are understood and designed.

1.4 Formal Verification

1.4.1 The Verification Problem

Chip designers not only need efficient ways to construct complex devices, but they also need ways to verify that these devices operate as intended. The cost of a bug in a computer chip can be quite large, and therefore the need to design the chips correctly and fix any problems early in the design flow is great.

Modern chip designs are very complex, and to overcome this complexity chips are designed hierarchically. Errors can be introduced at any level of the hierarchy, but the errors are more costly to fix at higher levels of the hierarchy. This is because higher level fixes have more side-effects and require the coordination of many designers across the project. Typically, if a bug costs X to fix at one level of the hierarchy, it will cost $10 \cdot X$ to fix at the next higher hierarchy level [Brand, 2007].

Example 9. The Cost of a Bug

Modern chips are designed hierarchically, and a logic designer typically works on the lowest-level of the design, called a logic block. Suppose the designer introduces a bug in his or her logic block, and this bug takes the designer 3 days to debug and fix. At a salary of $100,000 per year, the bug costs the company about $1000 in just the designer's salary for these 3 days.

Now consider what happens in terms of the hierarchy of the chip:

Logic block: *This is the most basic design unit. Typically only one designer works on a block, and that designer is wholly responsible for fixing bugs in the block. In our example, a bug here may take 3 days to fix at a cost of $1000.*

Unit: *A unit is a collection of logic blocks. If a bug in a single block is not fixed, it may manifest itself as a subtle bug in the interaction of the different blocks*

that make up the unit. This might take 10 designers 3 days to fix for a total cost of $10,000.

Core: *In a CPU, a core is the smallest block that is capable of executing all of the architecture's instructions. Cores are composed of several units, and many designers are responsible for the logic that ultimately goes into the core. Because of the increased complexity of the design and the increased number of people involved, a $10,000 bug in a unit may become a $100,000 bug in the core.*

Chip: *Modern chips are composed of several cores that interact independently. Suppose our $100,000 core-level bug was not fixed but instead manifests itself as a complex inter-core bug at the chip level. The increased complexity will mean that the bug costs $1,000,000 to find and fix at the chip level.*

System: *A system is composed of many chips. Indeed, the system is enormously complex, and all of the designers in a company could be working on one system. The example $1,000,000 chip bug could take $10,000,000 to fix at this level.*

If a bug is not caught while the system is in development then the costs can be even greater. In 1994, Intel released their first Pentium processor. The CPU had a bug in a block that was responsible for floating-point division, the infamous FDIV bug [Nicely, 2008]. The bug was not detected until after the CPU had shipped and was in the hands of consumers. Intel had to initiate a recall in order to fix the problem, and in total this bug cost Intel $475,000,000.

Clearly, bugs in a chip design can be very costly. There are efforts underway to allow the designer to specify designs in a way that is less bug-prone [OSCI, 2008], but even with these new design styles bugs will persist because of the simple fact that humans sometimes make mistakes.

Verification is the process of detecting bugs or checking that a design is free of bugs. Because a design is not manufactured until late in the design process, it is necessary to verify software models of the design. This is known as **pre-silicon validation**. The goal of verification is to check every level of the design hierarchy and to detect bugs early in the design cycle and in the lowest possible level of the hierarchy in order to minimize costs.

1.4.2 Simulation Vs. Formal Techniques

Typically pre-silicon validation is accomplished with **simulation**. A **logic simulator** is a CAD tool that can be used to read in a model of the logic design and simulate its logical behavior over time. A designer can provide inputs to the design, and the simulator will compute the output values that will result. By providing a temporal sequence of inputs, a simulator can be made to simulate a sequential logic design over a time interval.

Simulation has long been the workhorse of verification, but it has a major problem that is reducing its usefulness in modern chip designs. Simulators simulate one input vector at a time, and as chips have increased in complexity the number of possible input vectors has increased exponentially. Simulators are simply not able to simulate every possible input vector, and this results in poor design **coverage**, or the percentage of the design's behavior that has been checked.

Example 10. Verifying a 64-bit Multiplier With Simulation

A common unit in a modern CPU is a ***multiplier****. This unit inputs two binary-encoded numbers and outputs their binary-encoded product. Some modern CPUs use 64-bit encodings for numbers, and therefore the multiplier has* $2 \cdot 64 = 128$ *binary inputs and 64 binary outputs. The number of possible inputs is therefore* 2^{128}.

> *Suppose a designer wishes to verify a 64-bit multiplier with simulation. To ensure 100% coverage it is necessary to simulate each possible input vector. Suppose the simulator is able to simulate 1,000,000 input vectors per second, extremely fast by today's standards. To simulate all* 2^{128} *input vectors would therefore take* $1.08 \cdot 10^{25}$ *years. In comparison, the universe is only* $1.4 \cdot 10^{10}$ *years old [Wikipedia, 2008].*

Simulation is handicapped by its ability to process only a single input vector at a time, and formal techniques are the answer to this dilemma. A **formal technique** is any algorithm that is able to process all inputs at the same time. Often this is accomplished by treating the inputs **symbolically** and analyzing the circuit behavior in terms of these symbols. **Formal verification** is the application of formal techniques to verification.

Formal verification is the future of verification. As chips become increasingly complex, the number of input vectors will continue to grow. Today it is impossible to simulate every possible input vector, and in years to come the problem will get even worse. Formal verification arose in the 1980's as a theoretic curiosity, but recently formal verification has become a major focus of research in CAD tools.

This thesis will discuss ways to improve formal verification, and because this thesis is focused on formal verification rather than simulation, the word "verification" should be understood to mean formal verification in the text to follow.

1.4.3 Sequential Equivalence Checking

One compelling use of verification is to check that synthesis did not introduce unintended changes into the design. Synthesis tools by definition will optimize a design while preserving the behavior at the outputs of the logic network, but as with all software there is no guarantee that a synthesis tool is free of bugs. While a design

might operate as intended, the possibility that a synthesis tool could introduce a bug into the design means that synthesis results must always be verified.

Equivalence checking is the process of driving two similar designs with identical input vectors and then checking that the output vectors are identical. Because logic synthesis tools input a logic network and output an optimized and supposedly equivalent network, to check the correctness of the synthesis tool it is sufficient to check the equivalence of the input and output networks.

Sequential Equivalence Checking (**SEC**) is the extension of equivalence checking to check the correctness of sequential synthesis. The two logic networks whose equivalence is being checked are fed identical temporal sequences of inputs vectors, and the SEC tool checks that the output vectors are always equivalent. Because every state observed in either of the two designs will always be reachable, SEC allows for sequential synthesis to change the behavior of the design on the unreachable states. Therefore, while sequential synthesis is more powerful than combinational synthesis because it allows more flexibility with respect to the state space, it requires SEC in order to check the equivalence of the input and output logic networks. SEC often involves complex and computationally expensive algorithms, but this thesis will contribute methods to make SEC more scalable.

Example 11. Sequential Equivalence Checking The Logic Networks of Section 1.3.4

Section 1.3.4 discussed numerous optimizations of the basic sequential logic network in Figure 1.6. SEC can be used to guarantee that this synthesis was correct.

As an example, consider the result of resubstitution, shown in Figure 1.9. SEC can be performed on the ***product machine*** *formed from these two simpler logic network. Both logic networks are fed identical outputs, and the resulting composition is shown in Figure 1.10.*

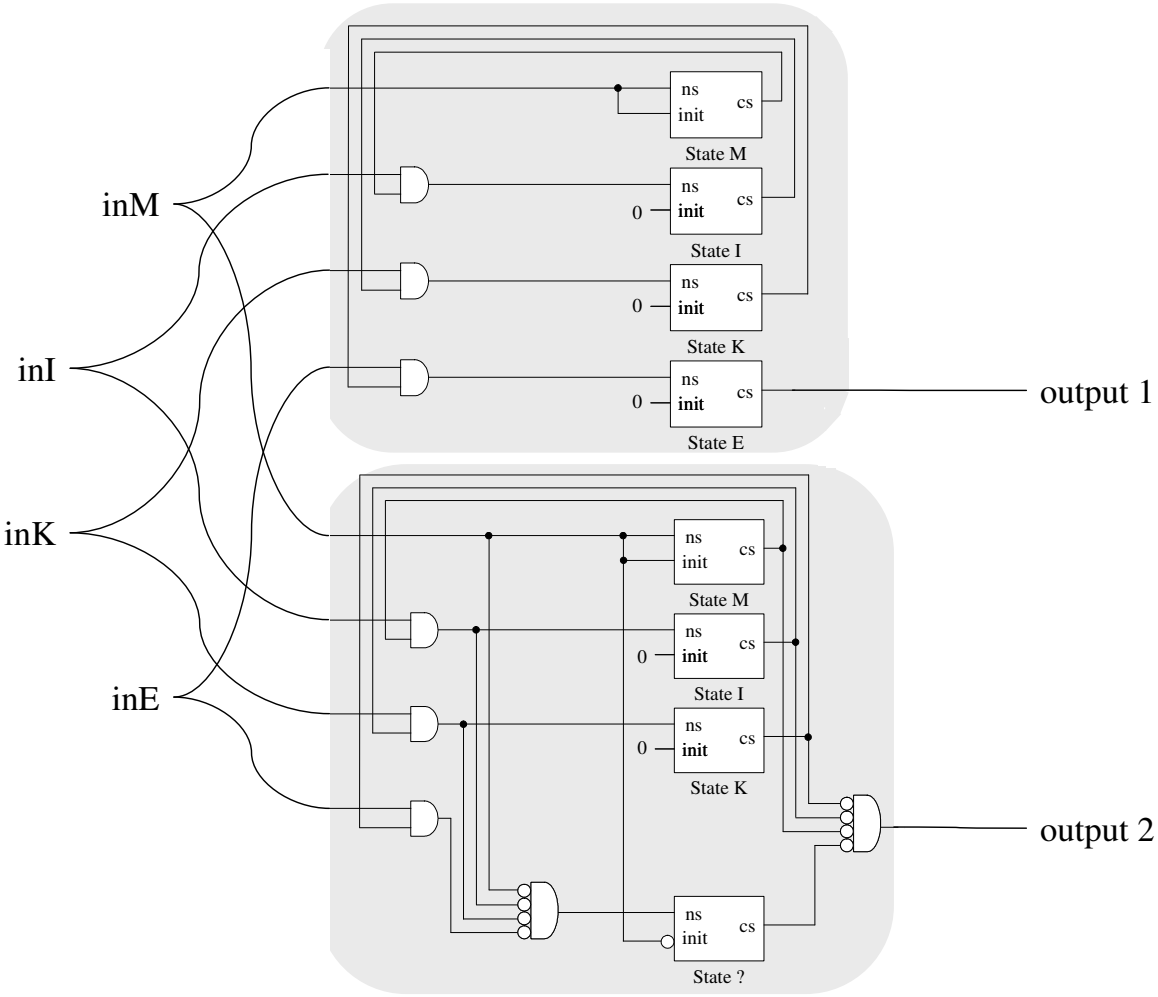


Figure 1.10: Checking the sequential equivalence of Figures 1.6 and 1.9.

The product machine has two separate outputs, one for each component logic network. If synthesis operated correctly then these outputs should always be equivalent. SEC is the process of driving this product machine with every possible sequence of input vectors and checking that the outputs are always equivalent.

1.4.4 Property Checking

Property checking is a more general type of verification. The designer can specify one or more **properties** of the design that should be verified. For example, in programming it is common to use **assertions** to check that your code is operating correctly. Formal verification would treat each assertion as a property and would either 1) prove that for all input sequences it is impossible to violate the assertion or 2) produce a **counterexample trace**, a temporal sequence of inputs that demonstrates how the assertion may be violated.

Many verification tools allow properties to be embedded directly into the designer's HDL, and therefore the designer can specify the properties to be checked as he or she is describing the logic itself. This allows for very intricate properties to be specified, often capturing design assumptions that would otherwise only exist in the designer's head.

Property checking is more general than SEC in that SEC can be expressed as a property checking problem. In SEC, a product machine is constructed and the outputs of the two component designs are proved to be pairwise equivalent. This pairwise equivalence check can be thought of as a property that must be proved on the product machine. In general, for two designs each with n outputs, SEC can be performed by building a product machine and checking n properties on that machine.

Because SEC can be viewed as a special case of property checking, this thesis will focus on the broader field of property checking. Therefore the term verification in the

text to follow should be understood to mean formal verification of properties.

Example 12. SEC Specified as Property Checking

Consider once more the numerous sequential logic networks presented in Section 1.3.4. The most basic such logic network was presented in Figure 1.5, and the most complex logic network, the result of state re-encoding, was presented in Figure 1.8. The state re-encoding example was complex, and while constructing the example the author used a property checking tool to verify the SEC and thus the correctness of the example.

The IBM verification environment was used to check the correctness of the re-encoding. Algorithm 2 shows the VHDL implementation of the product machine used to check the equivalence of Figures 1.5 and 1.8. The two machines each have a single output, output *and* output2 *respectively. A single assertion labeled* fail0 *is defined such that* fail0 *will be asserted when these two outputs are different. The IBM formal verification tool* SixthSense *was then used to prove that* fail0 *is impossible to assert.*

This example also has ***constraints****, or conditions that the formal tool must hold to be true. Constraints are often used to model the environment of a circuit, and constraints used in this sense would guarantee that a design is not being stimulated with an illegal input that would never appear in its typical environment. In the case of the two designs examined here, the input signals* inM*,* inI*,* inK*, and* inE *represent the conditions that the user has pressed the "M", "I", "K", or "E" key, respectively. It is assumed that the user presses at most one key at any point in time, and the constraints labeled* c1 *through* c5 *in Algorithm 1.8 are used to model these assumptions.* SixthSense *is thus disallowed from providing any input stimulus that violates the constraints. This is important because the two designs are actually not equivalent without this environmental*

```
 1: architecture intro_synth of intro_synth is
 2:         signal inM, inI, inK, inE : std_logic;
 3:
 4:          -- implementation 1: 1 hot
 5:         signal cs : std_logic_vector(0 to 3) := "0000";
 6:         signal ns : std_logic_vector(0 to 3);
 7:         signal m, i, k, e : std_logic;
 8:
 9:          -- implementation 2: logarithmic encoding
10:         signal cs2 : std_logic_vector(0 to 2) := "100";
11:         signal ns2 : std_logic_vector(0 to 2);
12:         signal a, b, c, output2 : std_logic;
13: begin
14:          -- decode everything
15:         inM  <= input(0);
16:         inI  <= input(1);
17:         inK  <= input(2);
18:         inE  <= input(3);
19:          --!! [constraint; c0]  <= not inM or not inI;
20:          --!! [constraint; c1]  <= not inM or not inK;
21:          --!! [constraint; c2]  <= not inM or not inE;
22:          --!! [constraint; c3]  <= not inI or not inK;
23:          --!! [constraint; c4]  <= not inI or not inE;
24:          --!! [constraint; c5]  <= not inK or not inE;
25:
26:         ...
27:
28:          -- output logic
29:         output  <= e;
30:         output2  <= not A and B and C;
31:
32:          --!! [ fail; fail0; "machines are different" ]  <= (output xor output2);
33: end intro_synth;
```

Algorithm 2: Using property checking to check the equivalence of Figures 1.5 and 1.8

constraint. The fact that the in *signals are mutually exclusive was used in the state re-encoding, and failing to represent these constraints in verification would result in a* ***spurious counterexample****, or counterexample that arises simply because of an inadequate verification setup.*

To illustrate the importance of SEC, the author checked the correctness of his state re-encoding before drawing the complex circuit of Figure 1.8. In the original state re-encoding derivation, the next state function for register "A" was specified incorrectly, and SixthSense *found a counterexample that demonstrated that the designs were legitimately inequivalent. Therefore without SEC, Figure 1.8 would be incorrect.*

In the field of property checking, the encountered properties can be classified into two broad types: safety properties and liveness properties. **Safety properties** are properties that express bad design behavior that should be avoided. A counterexample to a safety property is a finite-length sequence of input vectors that can be used to drive the design into a state that violates the property. **Liveness properties** are properties that express good behavior that should always be observed. A counterexample to a liveness property is an infinite length sequence of input vectors that demonstrate that it is possible for the design to always avoid the good behavior specified by the liveness property[5].

Safety properties have been the focus of much past research because they are conceptually simpler and therefore preferred by logic designers. Therefore a verification tool that is good at checking safety properties is adequate for nearly all logic designers. Recently it has been demonstrated that liveness properties can be transformed into safety properties [Schuppan and Biere, 2006], and therefore the space of liveness properties is contained in the space of safety properties. In this work we study the

[5]Because of the finiteness of the state space, all such counterexamples are composed of a tail of states followed by a loop of states.

broader, and conceptually simpler, space of safety properties. In the text to follow, the term property should be understood to mean safety property.

1.4.5 Important Verification Examples

In this section three verification algorithms will be discussed. Similar to synthesis, formal verification has a rich history, and many verification algorithms have been proposed in past research. Many of those algorithms are outside the scope of this thesis, but three algorithms that will be important in future discussions will be explored.

Example 13. Explicit State Model Checking

In the formal verification of a safety property, the goal is to demonstrate that the property is satisfied on all reachable states of a finite state machine or to find a temporal sequence of input vectors that can drive the machine from an initial state to a state where the property fails. The most conceptually simple way to do the verification is to simply enumerate all reachable states. If each such state is checked and the property is found to always be true then it is verified. If a reachable state is examined where the property fails to hold then by careful bookkeeping it is possible to produce a sequence of states and corresponding input sequences that form a path from an initial state to the failing state. This is known as ***explicit state model checking*** *or* ***reachability analysis****.*

Explicit state model checking proceeds methodically to explore the reachable state space, and to do this it maintains a set of states S that it has examined in the past. This process is illustrated in Figure 1.11 where the state space is gradually explored in a sequence of steps, each step progressing further from the initial states I and looking for states where the property fails B. The steps of the algorithm can be summarized like so:

1. *The algorithm examines the initial states, denoted I in Figure 1.11 and*

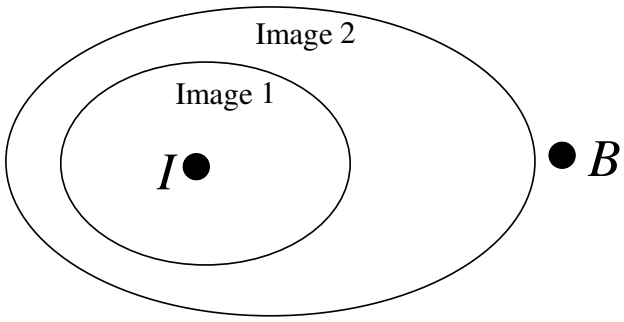


Figure 1.11: Reachability analysis on a state space.

checks that the property holds on these states. The set of examined states is now $S_1 = \{I\}$.

2. *Next, the set of states that are reachable in a single transition from* S *are examined. This set of states is known as the* ***image*** *of* S*, and the set of examined states is now* $S_2 = S_1 \bigcup image(S_1)$*. If the property fails on any of these states then a counterexample trace can be produced such that the temporal sequence of input vectors defines a path from* I *through* S_1 *and terminating in a bad state* B *in* S_2.

3. *The sequence of images progresses until either a fixed point is reached, indicating that all reachable states have been explored and the property is verified, or an image operator exposes a bad state* B *where the property fails to hold.*

Reachability analysis is very effective for small designs but suffers from the ***state explosion problem****. A design with* N *registers can have as many as* 2^N *states, and designs with 10k - 100k registers are not uncommon. Representing a set that has* $2^{50,000}$ *distinct elements is well beyond today's current computing capacity. Although there have been great advances in the ability to compactly represent sets of states [Bryant, 1992], performing reachability analysis on a design with*

more than 50 - 75 registers is usually difficult.

Reachability analysis is commonly used today for small designs, and even for big designs it is useful to understand reachability analysis. It is conceptually simple, and it is the model that will be used in this thesis to understand more complex verification algorithms.

Example 14. Induction Based Model Checking

In most designs the reachable state space is too large to explore exhaustively, but thankfully many properties can be proved with ***temporal induction****, eliminating the need for exhaustive exploration. Consider that a FSM progresses through a series of states. Consider an FSM with a property p and the following two temporal conditions:*

Base Case: *The property p holds in all of the FSM's initial states.*

Inductive Step: *For all states s where p holds, p also holds in every state reachable in 1 transition from s.*

If these two conditions hold then we can conclude that p holds for all reachable states of s.

The two conditions that comprise the inductive proof can be easily formulated using a ***Satisfiability Solver (SAT)****, and very efficient implementations can be developed. Note that if the base case fails, then a length-1 counterexample can be returned to the user. If the inductive step fails then, unlike reachability analysis, it is not possible to explore in detail the sequence of states and corresponding input vectors that led from an initial state to a state where p fails*[6]*. Therefore induction usually forms the basis of efficient algorithms that can return proofs*

[6]Indeed, no counterexample may exist because the property might be true yet not inductive.

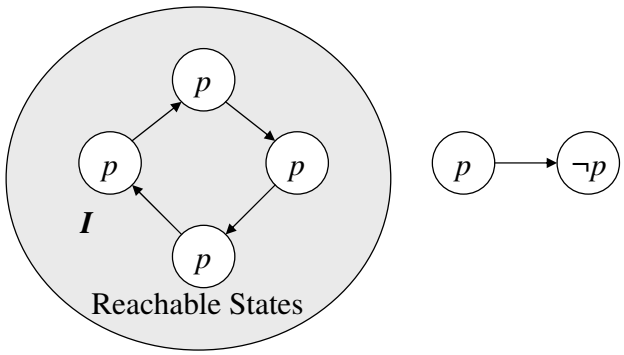


Figure 1.12: Induction is an incomplete method that cannot prove p.

for properties, but other algorithms like ***Bounded Model Checking*** *(**BMC**) are used to search for counterexamples.*

Although induction can be efficiently implemented, it is unfortunately unable to complete many verification proofs. This is because induction is an incomplete method that is unable to prove some types of true properties. Consider the simple state transition graph in Figure 1.11. This FSM has 6 states but only 4 are reachable from the initial state I. Clearly, property p holds on all 4 reachable states and thus is true and verifiable with reachability analysis. Induction would be unable to prove this because in the unreachable state space there is a transition from a state satisfying p to a state satisfying $\neg p$. Such a transition violates the inductive step and means that the property is not provable by induction. The unreachable state satisfying $\neg p$ is known as an ***induction leak*** *because it is the reason the inductive proof was unable to proceed. The property p is said to be* ***non-inductive****.*

There exist techniques to strengthen induction, notably unique state induction and k-step induction [Bjesse and Claessen, 2000]. While both of these techniques increase the number of properties provable with induction, the resulting method is still incomplete in that there are true properties that remain unprovable. In other

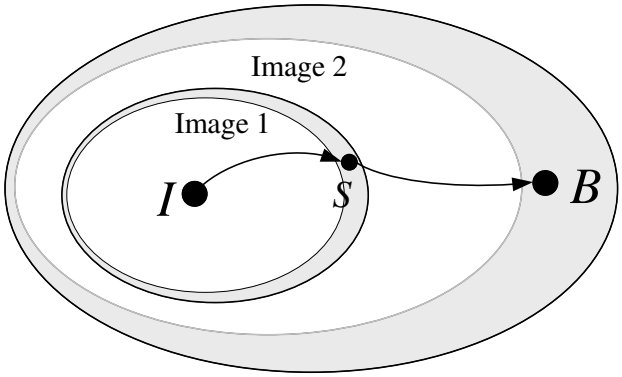


Figure 1.13: Interpolation is similar to approximate reachability analysis.

words, despite the efforts of past researchers there remain inductive leaks. This thesis will discuss a way to strengthen induction by focusing on the elimination of some of these inductive leaks.

Example 15. Interpolation Based Model Checking

Interpolation *is a model checking algorithm that is used in modern verification flows along with induction. It avoids the pitfalls of reachability analysis by not computing exact images. Instead, a SAT solver and a method known as* ***Craig Interpolation*** *are used to compute an over-approximation to an image, and model checking is performed using these approximations.*

Figure 1.13 depicts the behavior of interpolation on a state space. Similar to Figure 1.11, the white ovals represent the exact images. Interpolation involves taking an approximation to this image, which can be depicted as growing each image slightly to include the gray fringe regions. This over-approximation dramatically simplifies the computation and may allow the computation to come to a fixed point sooner because the state space will be covered more quickly. It can also lead to spurious counterexamples.

Figure 1.14 depicts a flowchart of the interpolation algorithm and can be used

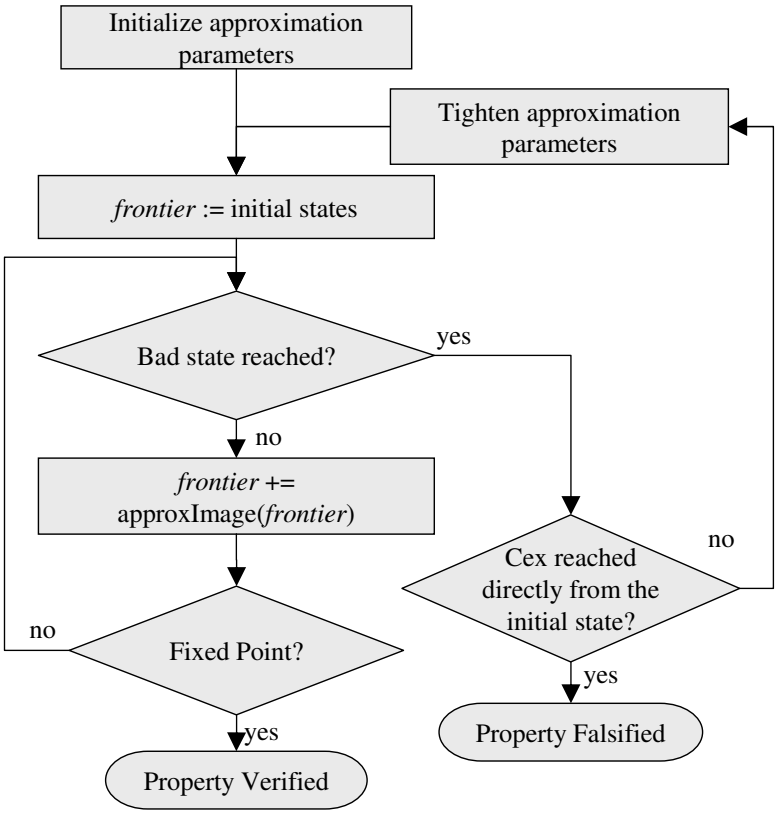


Figure 1.14: High-level flowchart depicting model checking via interpolation.

to understand its behavior in more detail. The set of already explored states is known as the ***frontier****. The algorithm proceeds as follows:*

1. *Parameters necessary for the image approximation are setup, including an integer value k. The frontier is initialized to the initial state(s).*
2. *Bounded model checking is used to see if there exist bad states reachable in k or fewer steps from the frontier set. If this is true and the frontier set was equal to the initial state(s) then a path from the initial states to the bad states has been found and this represents a valid counterexample. If the frontier set was not the initial state(s) then because interpolation involves approximate images there is no guarantee that the states in the frontier set are reachable, and the path from the frontier set to a bad state may represent a spurious counterexample. In this case, return to 1 and use parameters that give a tighter image approximation (eg: increase k). With these tighter images the spurious counterexample will hopefully not be encountered in the next attempt.*
3. *If BMC does not find a bad state, enlarge the frontier set by adding to the image of the frontier set. This is analogous to the reachability algorithm but uses an approximate image operation.*
4. *If the previous step did not change the size of the frontier set, then a fixed point has been reached and all reachable states have been explored. The property is therefore verified.*
5. *If a fixed point is not reached, return to step 2.*

Because of the existence of possibly spurious counterexamples, interpolation frequently needs to tighten its approximation parameters and restart the frontier computation. This is a costly operation because it discards all the effort that

went into the previous frontier computation. In this thesis we propose a method to handle spurious counterexamples by demonstrating that certain states along the counterexample path are unreachable. By incorporating this unreachability information into the underlying SAT solver that is used in interpolation, the algorithm can safely disregard this counterexample and continue its frontier development. By avoiding the costly restart, the runtime of interpolation can be dramatically improved.

1.4.6 Challenges in Formal Verification

Formal verification of safety properties is a very difficult problem. The number of reachable states in a sequential design with N registers may be as many as 2^N and developing a tool capable of operating on 2^N states for large N is a difficult engineering problem.

Several ways to perform approximate computations on 2^N states have been developed, but every approximate method to date is incomplete in some sense. Some methods can only produce counterexamples but not formal proofs, some fail to prove certain types of true properties, and some simply fail to converge and return a meaningful answer within feasible time limits.

Formal verification today is limited to very small pieces of a chip design. Because of the computational difficulties associated with many registers, small separate pieces of a design are considered in isolation. These small **windows** of the design have a limited number of registers and are manageable with formal verification. However, this windowing approach requires that the interactions between the logic in the window and the logic outside of the window be modeled, a very labor intensive job.

The goal of formal verification is to handle larger design sizes and eventually a full chip design. The methods presented in this thesis will help to make formal verification

more scalable and are a meaningful contribution toward this final goal.

1.5 Synergy Between Synthesis and Verification

Synthesis and verification are closely related disciplines. Synthesis techniques have come to help the verification community, and likewise verification techniques are now utilized in the synthesis community. This thesis will present methods that are applicable to both domains, and so it is worthwhile to understand how these two fields are related.

1.5.1 Synthesis For Verification

Formal verification of safety properties on sequential logic networks is very difficult when the size of those networks is large. In industrial verification flows, synthesis techniques are applied prior to verification in order to simplify the logic networks as much as possible before the expensive verification algorithm is invoked.

Verification algorithms typically do not scale as well as synthesis algorithms and so it is possible that a network is too large for processing with a verification algorithm yet can be processed with a synthesis algorithm. If the synthesis algorithm reduces the size of the network such that it can be processed with a formal verification algorithm, then this synthesis has enabled verification. Without the requisite synthesis, verification would have been unable to prove properties on the large logic network.

Often, many properties are of a simple nature and can be proved with synthesis alone. A property is a signal that evaluates to 1 whenever an error state is encountered. A true property is therefore 0 on all reachable states, and a sufficiently powerful synthesis algorithm (on a sufficiently weak property) can detect this condition and simplify the network by replacing the property with the constant 0. Synthesis can

therefore filter out many of the easy-to-prove properties in the design and simplify the later verification pass by allowing it to focus on only the most difficult design properties.

1.5.2 Verification For Synthesis

Formal verification algorithms present a way to verify properties across every reachable state. In recent synthesis advances, formal verification techniques have been adopted to leverage this state space exploration for proving that candidate synthesis transformations are valid.

Suppose for example a sequential synthesis algorithm wishes to modify the logic network yet does not have a guarantee that the output functionality will be preserved. The algorithm can formulate a property that expresses the correctness of the proposed transformation, and a formal verification algorithm can be invoked to verify the property. If the property is verified then the proposed synthesis transformation was correct, and the logic network can be modified accordingly. If a counterexample is found, then the synthesis algorithm must not modify the logic network in the intended manner because doing so would guarantee to change the design functionality.

Formal verification techniques can therefore be used inside of logic synthesis algorithms to enable them to perform intricate sequential transformations with the confidence that the design functionality is preserved. However, even with this guarantee of correctness, bugs in the synthesis tool itself could result in a synthesized logic network that is not equivalent to its pre-synthesis counterpart. For this reason, an equivalence check must be preformed between the pre-synthesis and post-synthesis logic networks.

In academia many complex sequential synthesis algorithms have been proposed, but many of these algorithms are not usable in industry because they introduce

complicated design changes that are difficult to verify in the later equivalence check. As verification algorithms improve and are able to handle larger designs and more complex properties, equivalence checks of more complicated synthesis algorithms can be completed. As a result, industry can make use of the advances in logic synthesis because they have a guarantee that the synthesis software is operating correctly. In this way, advances in verification are enabling the use of more complex logic synthesis.

1.6 Contributions of This Thesis

This thesis is centered on one topic: invariants. **Invariants** are design properties that hold in every reachable state. Each design property that has verified is an invariant, but in addition to this many invariants can be mined from the design in an automated way. The conjunction to these invariants forms an over-approximation to the set of reachable states, and this approximation can be applied to both sequential synthesis and formal verification.

Chapter 2 will discuss invariants. Efficient methods have been developed to mine the candidate invariants from the design and then to prove that they are valid invariants. The invariants can be used either to simplify the circuit or to approximate reachability.

Chapter 3 will discuss the application of the approximate reachability to sequential synthesis. Both resubstitution, examined in Example 8, and sequential ODC-based merging, similar to Example 6 but with a complicated criterion for merging signals, will be examined. These two methods are advanced sequential synthesis techniques that have been developed. Each technique can be strengthened by considering the states that fall outside the reachability approximation to be **don't cares** where the synthesis is free to change the design behavior. Synthesis results both with and without invariants will be examined in order to judge the effectiveness of invariants

in a synthesis setting.

Chapter 4 will discuss the application of approximate reachability to formal verification. Invariants can be used as constraints on the state space that is explored in both induction, examined in Example 14, and interpolation, examined in Example 15. Both of these techniques can explore unreachable states and therefore are susceptible to spurious counterexamples. By constraining the explored states to fall inside the reachability approximation, the runtimes of the algorithms are improved because less states must be explored, and also the algorithms are made more robust against spurious counterexamples.

Additionally in Chapter 4 an advanced technique will be discussed: deriving invariants on demand such that the invariants found are few in number and will immediately help a currently running verification task. This represents one promising future research direction in the area of invariants. Many other minor research directions will be discussed throughout.

Chapter 2

Invariants

2.1 Motivation and Basic Algorithm

2.1.1 Basic Definitions

An **invariant** is any design property that holds in every reachable state. In this way, the set of states that satisfies the invariant is a superset, or over-approximation to, the reachable states.

User-specified properties that have been verified are one example of invariants, but invariants can come from many other sources. Of particular interest are invariants that are automatically discovered. An **invariant family** is selected and properties that belong to this family can be mined automatically from the design. This concept can be developed into a push-button solution to find design invariants.

A **candidate invariant** is a property that has been automatically mined from the design but not yet proved. Upon being proved, the candidate invariant is referred to as simply an invariant.

Many different methods can be used to prove that candidate invariants hold on all reachable states. In most designs the most scalable method is induction (Example

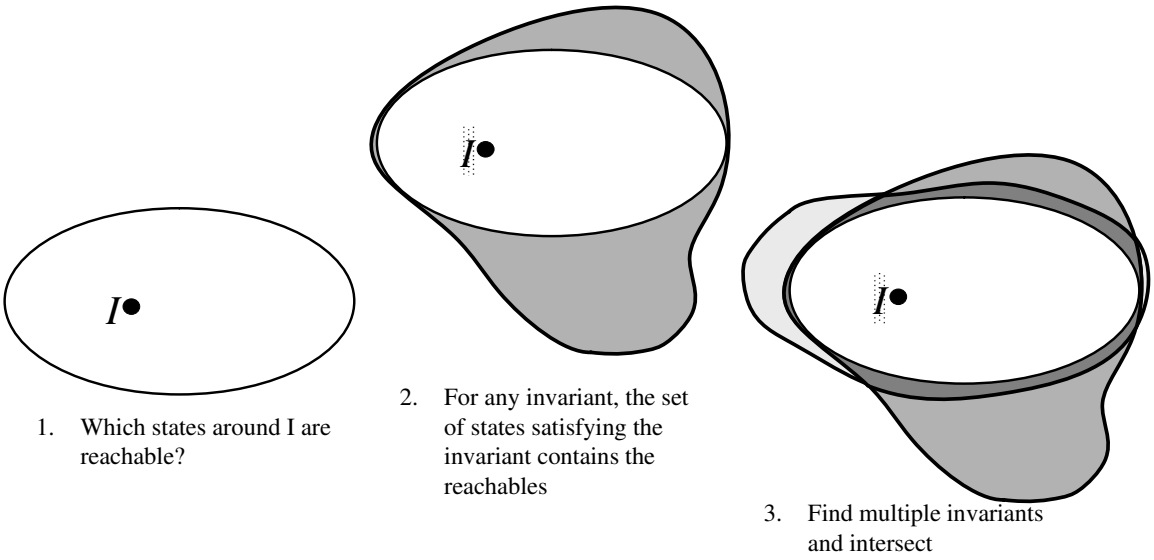


Figure 2.1: Approximating reachabilty with invariants.

14), and an invariant that has been proved by induction is referred to as an **inductive invariant**.

2.1.2 Uses For Invariants

Invariants will be used for two purposes in this thesis: to approximate reachability and to simplify a netlist.

Reachability analysis (Example 13) can be used to discover which states are reachable. It is precise in that a state is reachable if and only if it is explored by reachability analysis, but the algorithm is very non-scalable as a result of this precision. In practice, reachability analysis can only be applied on very small designs, and larger designs require the computation to be approximated in some way. Many ways to approximate reachability have been proposed (Section 2.8), but here we will focus on the use of invariants.

Figure 2.1 illustrates how invariants can be used to approximate reachability.

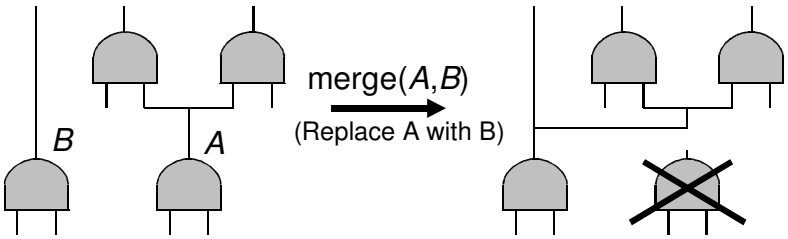


Figure 2.2: Invariants can be used to simplify a netlist.

In a state space, the set of reachable states can be thought of as a region of states surrounding the initial state(s) I, depicted in Figure 2.1.1. Any invariant by definition holds on all reachable states, and therefore the set of states satisfying the invariant contains the reachable states, as depicted in Figure 2.1.2. If an additional invariant is proved, the state containment relationship holds for it as well. Furthermore, the set of states that satisfy both invariants also contains the reachable states. As shown in Figure 2.1.3, this intersection is a smaller set than either of its two constituents and therefore it is a better approximation to the set of reachable states.

Another use of invariants in this thesis is even more fundamental. Some invariants enable simple synthesis transformations. For example, suppose the invariant family is node equivalences. That is, all found invariants will be properties expressing the equivalence of a pair of nodes in the design. If such a candidate invariant is discovered and successfully proved then a merge (see Section 1.3.4, Example 6) can be performed to simplify the design. For example, consider A and B to be nodes in the logic network and suppose the invariant $A = B$ has been proved. Then A and B can be merged and the logic can be simplified, as illustrated in Figure 2.2.

All invariants can be used for synthesis even if merges cannot directly be performed. The negation of an invariant gives a set of **sequential don't cares**. This set of states is an under-approximation of the unreachable state space, and it is guaranteed that the machine will not enter this state space during normal operation.

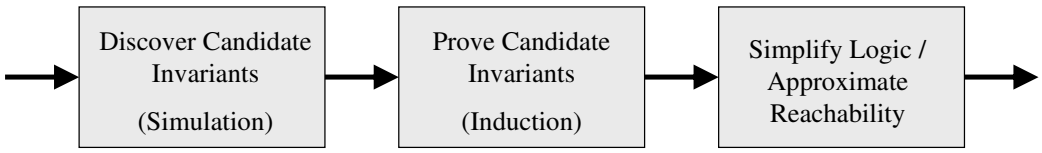


Figure 2.3: Basic invariant discovery algorithm.

Therefore sequential synthesis is free to change the behavior of the design on these states.

2.1.3 Basic Invariant Discovery Algorithm

Given a design and an invariant family, the basic algorithm for discovering invariants is quite simple, as shown in Figure 2.3.

1. Candidate invariants are discovered. The simplest method involves using **random simulation** (Section 2.3.1) to inject random values at the inputs of the logic network and then compute the resultant values at every node in the network. The simulation can be performed over several cycles in order to get a view of the reachable behavior of the design. Candidate invariants that hold within this view of the behavior are selected for the next step in the algorithm.

2. Candidate invariants are proved. Typically induction, a scalable yet incomplete method, is used for this proof. As a result, many true invariants may not be provable given this incomplete proof technique, but because of the large number of candidate invariants, the failure to prove a small percentage is not of concern.

3. The proved invariants are used. As described in Section 2.1.2, this means forming a reachability approximation and/or simplifying the design through synthesis.

The above algorithm is quite simple and works well for small designs. However, there are several inefficiencies that prevent its scaling to larger designs.

- Selection of the invariant family is important factor that impacts the quality of the reachability approximation. Section 2.2 will discuss choices for the invariant family, and Section 2.7 will discuss ways to utilize these families to obtain a good reachability approximation.

- Random simulation usually cannot explore every design behavior and therefore may select a candidate invariant that does not hold in every reachable state. In practice, this means that the number of candidate invariants selected for a proof can be overwhelming. Section 2.3 will discuss methods to improve the quality of the simulation and to reduce the candidate invariants to those that are most likely to give a good reachability approximation.

- In proving the candidate invariants, the number of candidate invariants may overwhelm the proof technique used. Selection of an efficient proof technique and incorporating invariant-specific optimizations is therefore very important and will be discussed in Section 2.5.

- Throughout the invariant discovery process, the number of candidate invariants can be very large and processing these invariants can be challenging. Efficient ways to store a large number of candidate invariants will be discussed in Section 2.6.

2.2 Types of Invariants

Many invariant families exist, and here we discuss five families that are important in this work. The families range from the computationally cheapest yet least expressive

to the computationally most expensive and most expressive.

2.2.1 Constants

Constants are the simplest invariant family. Given a set of nodes in the logic network, certain nodes may take a constant value in all reachable states. Consider candidate invariants expressing that a set of nodes are constant.

Typically the number of constant nodes is few. This means that the number of constant invariants that can be proved is low and the resulting invariant conjunction in Figure 2.1 is weak.

One notable exception to this weakness is a constant register. If it can be proved that a register assumes a constant value on all reachable states then any states where this register assumes the incorrect value are guaranteed to be unreachable. This can reduce the number of states in the reachability approximation by up to 50%.

Constant invariants have properties that make them very attractive in practice:

- Constants can be proved very efficiently using the methods that will be described in Section 2.5.
- Any proved constant invariant can be used to simplify the circuit. The constant node can be merged with its constant value, and the resulting **constant propagation** and COI reduction (Example 5) can dramatically simplify the circuit.

2.2.2 Equivalences

Equivalences is a property family that expresses the equivalence between two nodes in the logic network. That is, for all reachable states, two nodes will always assume the same logical value.

While being more numerous than constants, equivalences also tend to be few in number. This means that the proof obligation is low and invariant discovery using equivalences is scalable.

Because equivalence invariants are few in number, the resultant reachability approximation is usually weak. Again, the key exception is in the case of equivalences over pairs of registers. Suppose $A = B$ is proved and consider the value combinations ab that registers A and B can assume. $ab = 01$ and $ab = 10$ are guaranteed to be unreachable, reducing the size of the reachability approximation by up to 50%.

Similar to constants, equivalences have properties that make them a desirable property family:

- Equivalences can be proved efficiently using the methods that will be described in Section 2.5.
- Any proved equivalence can be used to simplify the circuit. If $A = B$ is proved then a merge can be performed to eliminate one of the nodes and drive its fanouts with the remaining node.

Equivalences are usually found modulo inversion, as first described in [van Eijk, 2000]. An **antivalence** is a relationship between a pair of nodes A and B such that $A = B$ or $A = \neg B$, for all reachable states. In this work we broaden the equivalence invariant family to include antivalences. This makes the family more effective in approximating the reachable states and simplifying the logic network without increasing the computational complexity of finding and proving invariants.

The equivalence invariants family can be further expanded by considering pairs of nodes that do not come from the same time frame. Let **next** refer to the value of a node in the next time frame. An example of such an extended equivalence is $A = next(B)$, expressing that node A assumes logical values equivalent to the value

B will assume at the next point in time, for all reachable estates. This is an interesting idea proposed by others, but in this work we do not make use of this concept due to the increased complexity of finding such candidates with simulation. Additionally, extending equivalences in this way complicates logic network simplification because a simple merge cannot be done. One or more registers may need to be introduced to make use of this inter-frame relationship, making the net benefit of such a transformation unclear.

2.2.3 Implications

Implications are a more expensive yet significantly more expressive property family. An implication is a relationship between two nodes A and B expressing the condition that $A \implies B$ holds for all reachable states. That is, if A is assigned value a and B is assigned b then $ab = 10$ will never appear in a reachable state.

Similar to how equivalences were expanded into antivalences, we consider implications modulo inversion. That is, for nodes A and B, four candidate implications may be gathered from the circuit: $A \implies B$, $A \implies \overline{B}$, $\overline{A} \implies B$, and $\overline{A} \implies \overline{B}$.

The implications invariant family contains both the constants and equivalences families. Consider the case that a node A always takes a constant value in every reachable state. If $A = 0$ then the implication $A \implies X$ is true for every node X. If $A = 1$ then $\overline{A} \implies X$ is true for every node X. Therefore the constants family is contained in the implications family. Similarly, consider an equivalence $A = B$. If this holds in all reachable states if and only if the following two implications hold: $A \implies B$ and $B \implies A$. The equivalences family is contained in the implications family as well.

Invariants complicate the overall invariant discovery algorithm (Figure 2.3) because the number of candidate invariants can be very large. If a logic network has

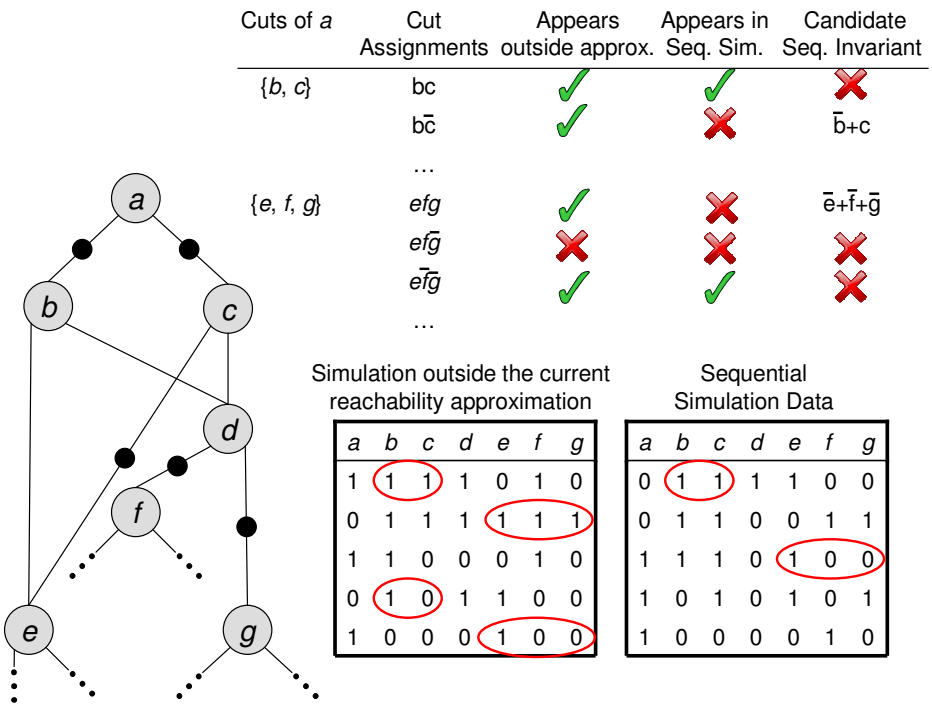


Figure 2.4: Extraction of invariants from a k-cut.

n nodes then there might be $O(n^2)$ candidate implication invariants. In practice, because an invariant is relatively difficult to falsify, simulation has trouble refuting many candidate implications and many of these candidates survive to the proof stage of the algorithm. This means that storage of the candidates as well as proof strategies must be carefully considered.

Implications are numerous and for this reason tend to give a tight approximation to reachability. However, they have little use beyond that. While there has been previous research in using implications for synthesis [Kunz *et al.*, 1997], there is no simple merging that can be done to simplify the circuit once a candidate implication invariant is proved. In this work, implications are not used for synthesis.

2.2.4 k-Cuts

k-Cuts are less numerous and less expensive to prove than invariants-implications. k-Cuts are clauses of length k that are derived from a localized region of the circuit. Because the number of localized regions is small, the number of cuts is also small.

A **cut** of a node X is a set of nodes such that any path from an input to X must pass through exactly one node in the cut. For example, in Figure 2.4 node a has numerous cuts: $\{b, c\}$, $\{e, f, g\}$, ...

Candidate invariants are formed as clauses over the nodes in a cut. A candidate invariant should 1) be able to refine the current reachability approximation, and 2) not be easily falsifiable with random sequential simulation. Simulation can be used to mine candidate invariant clauses from cuts that satisfy these two properties.

Example 16. Mining candidate k-cut invariants

Figure 2.4 shows an example of k-cut invariants. Two simulations of the circuit are done: one on states that lie outside the current reachability approximation (if available), and another that only simulates reachable states, possibly on a random sequential walk (Section 2.3.2). Then candidate invariants can be derived as follows:

1. *Iterate over all nodes in the AIG. In the simple AIG pictured, suppose the current node is a.*
2. *The cuts of a can be enumerated.*
3. *For each cut, consider all cubes formed from the literals in the cut. It is important to consider all polarity assignments of each literal, meaning that for n literals 2^n cubes will be considered. With n small (4-5) this exponential is not troubling.*
4. *If a cube appears outside the current reachability approximation then the*

inverse of the cube, a clause over the cut nodes, is false at at least one point outside the reachability approximation. If this clause is proved then the invariant clause is conjoined with the reachability approximation, and the conjoined approximation will be stronger than the original approximation. Therefore, we limit the enumerated cubes to only those that appear outside the reachability approximation.

5. *If the cube appears in the sequential simulation, then there is a reachable state for which the cube's inverse, a clause over the cut nodes, evaluates to 0. Therefore this clause cannot possibly hold on all reachable states. We limit the enumerated cubes to those that do not appear in the sequential simulation.*

6. *All cubes that survive the above filtering steps are inverted and the resulting clauses are taken as the candidate invariants.*

On average, there tends to be few candidate k-cut clauses for each node a, and for this reason the number of candidate invariants tends to be linear in the number of nodes in the network. This makes the proof of these candidates manageable.

Furthermore, each candidate invariant involves nodes that are close together in the circuit. Each candidate is a local property in this sense, and this locality lends itself to fast SAT runtimes. Because the solver does not need to relate nodes from opposite sides of the circuit (as may be the case with candidate implication invariants), the solver tends to run very quickly with k-cut candidates.

2.2.5 Random Clauses

The final invariant family is random clauses. A fixed integer n is chosen and n or less nodes from the design are selected. Polarities are assigned to these nodes and they

are conjoined to form a clause. For example, consider the design nodes A, B, and C. $(A + \overline{B} + C)$ and $(\overline{A} + \overline{B})$ are both candidate invariant clauses.

Random clauses are useful for expressing intricate node relationships that are not expressed by any other property family. In this way, random clauses can fill in for the shortcomings of the other families. Any invariant family can be made to form tighter reachability approximations if candidate random clauses are also considered.

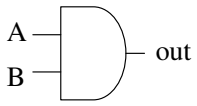
Random clauses can be very numerous, and care must be taken to not exhaust computational resources. A network with m nodes will have $\binom{m}{n} \cdot 2^n$ random clauses of length n. Clearly, even for small n and m exhaustive exploration of all random clauses is infeasible. Therefore it is necessary to bound the search for random clauses. In this work we find it useful to limit our algorithms to only explore a predetermined number of random clauses.

2.3 Improving the Quality of Simulation

Simulation is a valuable tool in the processing of candidate invariants, and here we discuss several ways in which it is used efficiently.

2.3.1 Random Simulation and Bit Parallel Simulation

Simulation is the process of determining the logical value of a node in a logic network as a function of the values of other nodes in the network. When simulation is called to provide a value for node A it will return either 0, 1, or X indicating that the value of A could not be uniquely determined from the values assigned to the other nodes. **Random simulation** refers to the process of injecting randomly selected Boolean values at the inputs of a logic network and using simulation to derive the value of every other node in the network. Because each node is a deterministic function of



Vectors to Simulate

A	B	Other nodes
0	0	…
1	0	…
1	1	…
1	1	…

Machine Word View

A = 0111…
& B = 0011…
out = 0011…

(all vectors simulated in 1 machine instruction)

Result

out
0
0
1
1

Figure 2.5: Bit parallel simulation is used to simulate many vectors in parallel.

the inputs, all nodes should be assigned Boolean values by the simulation procedure if all inputs are assigned Boolean values.

An **input vector** is a logical assignment to each input in a logic network. Random simulation can be used to explore the behavior of a design under a single input vector, but to characterize the behavior of a design one vector is often not sufficient. What is needed is a view of the design's behavior under many input vectors.

The development of efficient simulation routines requires one to understand the underlying architecture of the machine on which the CAD tool is run. A **machine word** is a collection of bits that the machine is able to process as one unit, typically either 32 or 64 in today's CPUs. Simulation involves many iterations of computing the logical value of a gate as a function of the values of its inputs, and this computation involves word level operations. In an AIG, the operation is reduced to a single bitwise AND.

When a CPU performs an AND operation, it processes two input machine words. Suppose the words have 32 bits and on the bit level are represented as: $a_0, a_1, \ldots, a_{31}$ and $b_0, b_1, \ldots, b_{31}$. The output of the AND operation will be the vector $(a_0 \cdot b_0), (a_1 \cdot b_1), \ldots, (a_{31} \cdot b_{31})$. Observe that for any bit position $j \in [0, 31]$, the output bit in position j is influenced by only the two input bits in position j. Therefore all of the

input bit positions act independently. If a machine has n bits in its machine word, then n distinct simulation vectors can be assigned a bit position in the machine word being simulated. As the AND is computed, the output values for all n simulation vectors will be computed in parallel. This concept is referred to as **bit-parallel simulation** and is illustrated in Figure 2.5.

Bit-parallel simulation dramatically increases the speed of random simulation. Often one wants to perform random simulation over many different input vectors. If there are m input vectors to be considered and n bits in a machine word then the number of simulation passes that need to be performed is therefore m/n.

2.3.2 Random Walks

Many applications of random simulation require that all the state vectors explored be reachable states. The simplest way to guarantee this is to perform simulation over a random walk from the initial state(s).

A **random walk** is a simulation that proceeds from a given state through a series of adjacent states in the state transition graph. A random walk from the initial state(s) is a random walk that starts from the initial state(s). Because each state seen along a random walk from the initial state has a concrete path leading to it from the initial state(s), the state is reachable. Therefore a random walk from the initial state(s) only explores reachable states.

Algorithm 3 illustrates the concept of a random walk from the initial states. Two parameters to the method are *width* and *length*, specifying how many vectors should be present in the bit-parallel simulation and how many simulation steps should be executed, respectively. Initially, the *state*, or valuation of the registers, is simply initialized to the initial state values. Each register has a vector that is *width* bits long, corresponding to the desired bit-parallel width. Then in a loop the following

```
1: function randomWalk(width, length)
2:     state := width * initial_state_values
3:     for step = 1 to length do
4:         Consume state in the calling application
5:         input := width * random_bit()
6:         state := bit_parallel_simulate(state, input)
7:     end for
8: end function
```

Algorithm 3: A random walk through the state space.

things occur:

1. The *state* is consumed by the calling application. *state* represents *width* different valuations of the designs registers, and each valuation is guaranteed to be a reachable state.

2. A set of random vectors are chosen to represent each input that is not a register.

3. Bit parallel simulation is used to propagate the *input* and *state* vectors through the circuit, resulting in a new *state*.

A random walk is an algorithm that is built on top of bit parallel simulation, and there exist techniques that make its implementation very efficient. Therefore it is a useful tool in quickly deriving a set of reachable states.

2.3.3 Input Vector Randomization

Bit-parallel simulation enables multiple input vectors to be simulated simultaneously. Often an application has only one input vector that must be simulated, and normally the parallel-simulation capability would be wasted in these applications. Trace randomization helps to use this remaining simulation capacity in an intelligent way.

Trace randomization refers to the process of mixing an input vector or sequence of input vectors with a stream of random values to derive a random trace that is

near the original trace. With trace randomization one can expand a trace into a series of related traces, and this can dramatically increase the resolving power of the simulation.

The **Hamming distance** between two binary vectors is defined as the count of the number of bits positions for which the vectors differ. Often the input vectors that come from a particular application are special in that a purely random vector fails to sensitize the circuit in the same way that the chosen vector does. For example, a vector can be carefully selected such that for two nodes A and B in the logic network, $A \neq B$ under the chosen input vector. Vectors sensitizing this condition may be rare, and a purely random vector may not be able to sensitize the inequivalence. However, by using trace randomization, random vectors can be derived such that their Hamming distance to the original vector is small. These pseudo-random vectors can often sensitize the same conditions that the original vector can.

There are two key methods that have been explored to randomize traces: distance-1 simulation and random mixing.

2.3.3.1 Distance-1 Simulation

Distance-1 simulation is the process of randomizing an input vector by generating a series of other vectors of Hamming distance 1 to the original vector. If the logic network has n inputs then there are n distance-1 vectors for any given input vector.

The distance 1 vectors can be found using a simple XOR scheme. Consider for example an input vector *abcd*. The following 4 patterns can be used to compute the complete set of distance-1 vectors:

1000	$abcd \oplus 1000 = \overline{a}bcd$
0100	$abcd \oplus 0100 = a\overline{b}cd$
0010	$abcd \oplus 0010 = ab\overline{c}d$
0001	$abcd \oplus 0001 = abc\overline{d}$

Each of the 4 patterns acts as a mask that indicates which input bit should be flipped. Because each mask has a single 1 bit, the result of the XOR operation will be a vector with Hamming distance 1 to the original vector.

2.3.3.2 Random Mixing

If the goal is to derive pseudo-random vectors that sensitize interesting conditions then distance-1 simulation is a good way to expand a good vector into a series of good pseudo-random vectors. However, the number of pseudo-random vectors is bounded by the number of circuit inputs and often the traces can be randomized such that they are not distance 1 yet are still able to sensitize interesting circuit conditions.

Random mixing refers to the process of XORing the input vector with a purely random vector. Suppose we wish to derive pseudo-random vectors around the input vector *abcd*. In a uniformly random vector $r_0r_1r_2r_3$, the probability of a single bit $r_j = 1$ is 0.5. Therefore the expected Hamming distance between $abcd \oplus r_0r_1r_2r_3$ and *abcd* is approximately $4 \cdot 0.5 = 2$. For input vectors of length n, the Hamming distance will be approximately $n/2$.

By choosing m random vectors and performing m XOR operations, a set of m pseudo-random vectors can be generated such that each is approximately $n/2$ distance from the original input vector.

If a tighter distribution is desired then two purely random vectors can be chosen. In the product of two random vectors $r_0r_1r_2r_3 \cdot r_4r_5r_6r_7$ a single bit will assume a value 1 with probability 0.25. Generalizing this concept to random mixing, for an input

vector with n inputs and for k randomly chosen input patterns $R_1, \cdot R_k$, the vector $abcd \oplus \prod_{j=1}^{k} R_j$ will be distance $n/2^k$ from the original input vector, on average.

Random mixing provides a way to quickly generate a large number of input vectors in the vicinity of a reference vector, and in practice is often able to generate a series of interesting vectors from a seeded interesting vector.

2.4 Mining Candidate Invariants

Recall that candidate invariants are invariants that are likely to hold in all reachable states yet have not been proved. A set of candidate invariants can be derived through simulation techniques, and several iterations of a basic simulation algorithm can be used to derive candidate invariants that are of high quality.

2.4.1 Methods to Mine Candidates

```
1: function mineCandidateInvariants(logic, invariantFamily)
2:     candidates := ()
3:     for all invariants I in invariantFamily do
4:         candidates += I
5:     end for
6:     return candidates
7: end function
```

Algorithm 4: Algorithm to mine candidate invariants, version 1.

First consider the most basic candidate invariant mining algorithm, shown in Algorithm 4. A logic network and invariant family are known, and the task of the algorithm is to return a list of candidate invariants that come from the invariant family and likely hold in every reachable state in the logic network. The most simple algorithm is one that considers every candidate invariant from the logic family. For some invariant families such as equivalences (Section 2.2.2) this is efficient because a

single equivalence class containing every node in the logic network can be represented compactly. However, there are other logic families such as implications (Section 2.2.3) for which this is very inefficient. For a logic network with n nodes, n^2 candidate invariants would be returned by this simple algorithm. The algorithm can be improved by considering a simulation of the logic network.

```
 1: function mineCandidateInvariants(logic, invariantFamily)
 2:    candidates := ()
 3:    stateSet := randomWalk(logic)
 4:    for all invariants I in invariantFamily do
 5:       if (I holds ∀ states in stateSet) then
 6:          candidates += I
 7:       end if
 8:    end for
 9:    return candidates
10: end function
```

Algorithm 5: Algorithm to mine candidate invariants, version 2.

Consider the improved mining algorithm shown in Algorithm 5. A random walk (Section 2.3.2) is used to derive a set of known-reachable states, labeled *stateSet*. Then each candidate invariant from the invariant family is checked against this set of states, and only candidates that hold on all of these states are returned. Because the candidate invariants will be proved to hold on all reachable states, discarding candidates that do not hold on *stateSet* is equivalent to discarding those candidates that are falsifiable with random simulation. In a sense these are the easy-to-falsify candidates that do not require more expensive algorithmic machinery to falsify, but the vast majority of candidates are of this type.

It is important to carefully tune the algorithm such that *stateSet* is of high quality. If any unreachable states are contained in *stateSet* then candidates may be discarded that are only falsifiable on these unreachable states. This results in the discarding of valid, provable invariants and hence hurts the overall quality of the reachability

approximation that the invariants provide. Another danger is a *stateSet* that does not contain enough reachable states. In this case, few candidate invariants will be falsified and the set of candidates returned by Algorithm 5 will be too large to process efficiently. This can be controlled by tuning the length and width of the random walk. This results in a trade-off between the time spent doing the random walk and the time required to handle an overly-large set of candidate invariants.

```
1: function mineCandidateInvariants(logic, invariantFamily, cexStates)
2:    candidates := ()
3:    for all invariants I in invariantFamily do
4:       if (I holds ∀ states in cexStates) then
5:          candidates += I
6:       end if
7:    end for
8:    return candidates
9: end function
```

Algorithm 6: Algorithm to mine candidate invariants, version 3.

If candidate invariant discovery is being performed after other sequential analysis algorithms have already been run on the logic network, then a more efficient simulation method can be used. Suppose a method such as **Bounded Model Checking** (**BMC**) has been run and has developed a series of counterexamples. Because of the nature of BMC, each of these counterexamples is guaranteed to be reachable and usually sensitizes a rare circuit condition that is difficult for a random walk to explore. The invariant mining algorithm can be improved by using these counterexample states in the place of the states that were explored by the random walk, and this improved algorithm is shown in Algorithm 6.

An example use of counterexample states is explored in Table 2.1. In this application, candidate merges are derived by simulation such that the merges do not affect any of the circuit outputs in any reachable states (Section 3.4). Such candidate invariants are difficult to falsify, and the states explored on a random walk will

	Number of Candidate Merges		
Design	Random Walk	Reach. Cexs.	Ratio
ibm4	1,274,272	16,217	1.3%
ibm5	6,577	8,460	128.6%
ibm6	4,556	4,721	103.6%
ibm7	1,766,864	101,409	5.7%
ibm9	20,429	25,648	125.5%

Table 2.1: Comparing random walks and counterexamples

falsify the candidates with a relatively low probability. Often this results in a large number of candidates and slows the proof of these candidate invariants. By using counterexamples that were previously derived from BMC, the number of candidate merges can be dramatically reduced. In two of the five benchmarks examined, the use of counterexamples dramatically reduced the number of found candidate invariants and made the proof step much more scalable. Note also that if the counterexample states are few in number then they may be able to falsify fewer candidate invariants than states from a random walk. This is why three of the cases in the table saw an increase in the number of candidates when using counterexample states. An industrial application would be wise to use a combination of random walk states and counterexample states.

2.4.2 Candidate Prioritization

In some applications the number of candidate invariants is too large to handle efficiently, even after careful tuning of the random walk and use of reachable counterexamples. In this case, only a subset of the candidate invariants can be passed to the proof algorithm, and this subset must be carefully chosen such that the discarded candidates do not hurt the overall algorithm much. If the overarching goal is to form a tight reachability approximation then a technique can be used to prioritize the candidate invariants and keep only those that contribute most to the reachability

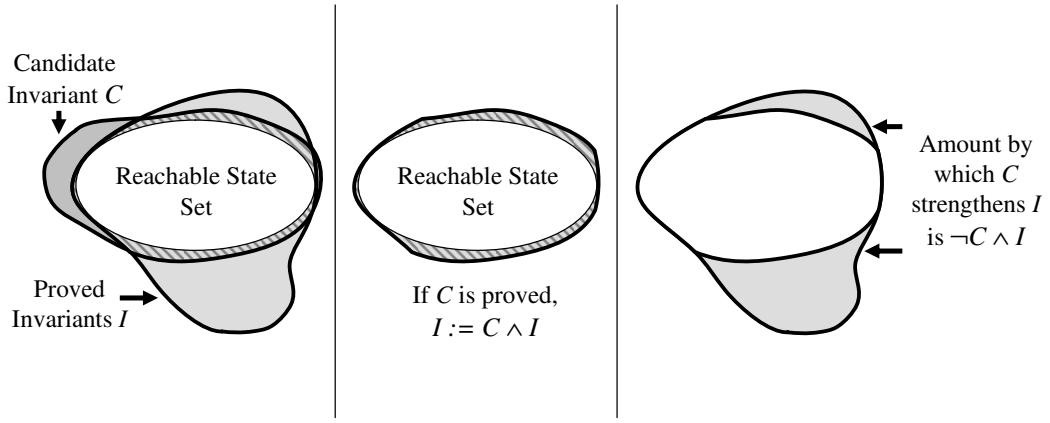


Figure 2.6: Using simulation to determine which candidate invariants are most valuable.

approximation.

The intuition for the invariant prioritization scheme is shown in Figure 2.6. Suppose multiple iterations of invariant discovery are being used and a set of invariants I has already been proved. These invariants provide an over-approximation to the set of reachable states (Section 2.1.2). Any new candidate invariant C will, if successfully proved, also over-approximate the reachable states. If C is proved then the reachability overapproximation will be tightened to $I \cdot C$. The amount by which the reachability overapproximation shrinks is therefore equal to the number of input vectors satisfying $I \cdot \overline{C}$. By prioritizing candidates C that have a large $I \cdot \overline{C}$ area, we can bias the computation to find invariants that are best able to tightening the existing reachability approximation.

This algorithm is formalized in the pseudocode shown in Algorithm 7. A set of candidate invariants and an existing reachability approximation are given and the task is to reduce the candidates to a set no larger than N that best refines the reachability approximation. Because the area of $I \cdot \overline{C}$ is expensive to compute exactly, it is approximated through simulation. A set of randomly chosen input vectors *randomInputVectors* is selected, and the number of these vectors that satisfy

```
1: function prioritizeCandidates(logic, candidates, reachabilityApprox, N)
2:    randomInputVectors := enumerateRandomVectors(logic)
3:    for all candidates C in candidates do
4:       candContribution := reachabilityApprox · ¬C
5:       C.score := 0
6:       for all vectors V in randomInputVectors do
7:          C.score += ( candContribution(V) == 1 )
8:       end for
9:    end for
10:   sortedCandidates := sortByScore(candidates)
11:   return best N candidates from sortedCandidates
12: end function
```

Algorithm 7: Candidate prioritization for reachability approximation.

$I \cdot \overline{C}$ is computed. This count is used as an approximation to the area of this function, and the N candidates C with the best approximate areas are returned to the user. These candidates have the greatest ability to refine the reachability approximation, if proved.

A practical application of this method is shown in Figure 2.7. Several rounds of invariant discovery are run, and the invariant families are varied between iterations. The time needed to discover the candidates (`getCandidates()`), the time needed to prove the candidates, the count of the number of candidates, and the count of the number of proved invariants are shown for each iteration. In each iteration the candidate invariants are filtered such that the N candidates that are best able to refine the previous reachability approximation are selected for a proof attempt. Typically, the number of candidate implications (Section 2.2.3) tends to be quadratic in the number of nodes in the logic network while the number of candidate cut invariants (Section 2.2.4) tends to be linear in the number of nodes. However, comparing iterations 3 and 4 in Figure 2.7, we see that the number of candidate implications was less than the number of candidate cut invariants. The reason for this is that a tight reachability approximation was formed in iterations 1-3, and the candidates from iteration

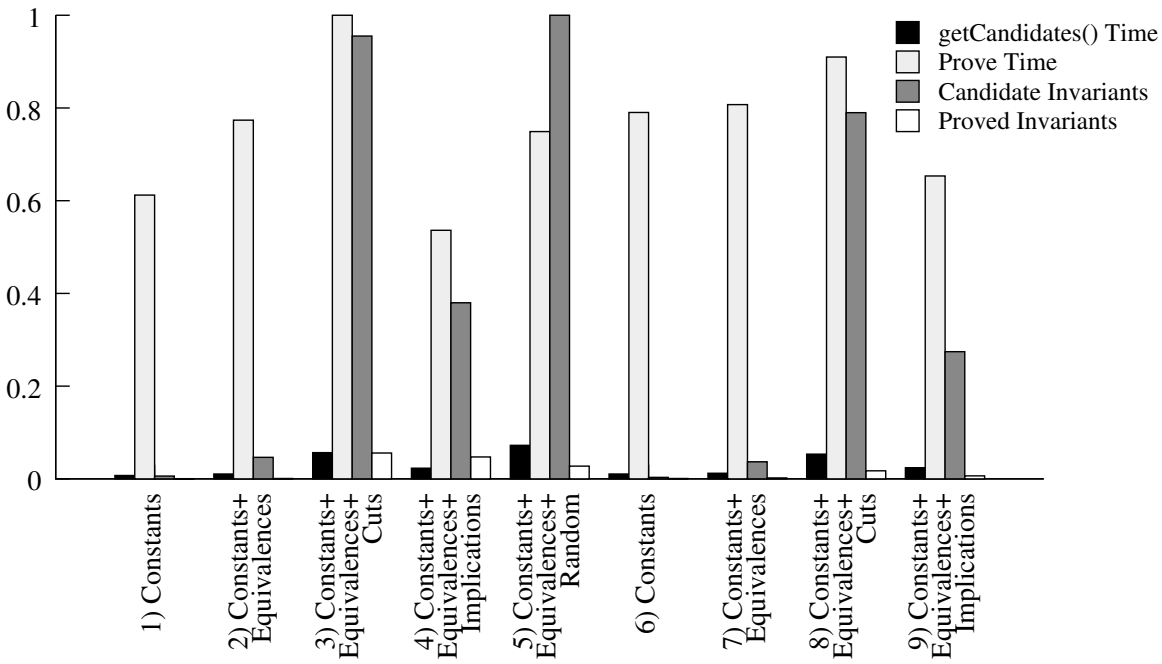


Figure 2.7: Example illustrating several rounds of invariant discovery.

4 that are able to refine this approximation are few in number. Therefore, while in general the number of candidate implications is often too large to efficiently handle, simulation-based prioritization effectively reduces the number of candidates to a small set that is both effective in forming a tight reachability approximation and efficient to process.

2.5 Efficient Proving of Candidate Invariants

After mining candidate invariants, the candidates must be proved to hold on all reachable states. Without this proof, it is unsafe to utilize the invariants for circuit optimization or reachability approximation. However, this proof step can be quite expensive, and this section will discuss methods by which the proof can be made scalable.

2.5.1 Proof Techniques

There exist several methods to prove that a property holds on all reachable states. Any of these methods can be utilized in order to prove that the candidate invariants are valid. Consider the following three popular techniques:

Reachability analysis (Section 13) can be implemented with BDDs to exhaustively explore every reachable state. However, this method is not scalable and often cannot be used to process industrial designs. Our goal is to make invariants scalable even on large industrial designs.

Interpolation (Section 15) is a method of proving difficult properties and scales to large industrial designs. However, interpolation decomposes a set of candidate invariant proofs and proves each candidate separately. The set of candidates is large and such an approach would be prohibitively expensive. Additionally, interpolation tends to be an inefficient way to find counterexamples, and the risk that a candidate invariant will fail to hold on all reachable states makes interpolation a gamble.

Induction (Section 14) is a very scalable way of simultaneously proving a large number of candidate invariants, and it scales well to large industrial designs. However, induction is not complete in that it can fail to prove invariants that do indeed hold on all reachable states. But this is an acceptable trade-off for the scalability of induction. In this work we utilize induction. All found invariants proved in this way are referred to as **inductive invariants**.

The method used to prove invariants in [Case *et al.*, 2006b; Case *et al.*, 2008b; Case *et al.*, 2008c] is k-induction without unique state constraints [Bjesse and Claessen, 2000], illustrated in Algorithm 8. A natural number k is input and the candidate invariants are checked in two phases:

```
1: function proveInvariants(candidates, k)
2:    // Base Case
3:    while (checkBaseCase(candidates, k) == counterexample) do
4:       candidates := refineCandidates_base(candidates, counterexample)
5:    end while
6:
7:    // Inductive Step
8:    while (checkInductiveStep(candidates, k) == counterexample) do
9:       candidates := refineCandidates_ind(candidates, counterexample)
10:   end while
11:
12:   return candidates
13: end function
```

Algorithm 8: Invariant proof with induction.

Base Case: Here we verify that the candidates C hold in all states reachable in k or less transitions from the initial state(s) I. Formally:

$$\forall\, S_0, S_1, \ldots, S_k \;.\; (S_0 \in I) \wedge (S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_k),$$
$$(S_0 \models C) \wedge \cdots \wedge (S_k \models C)$$

This can be efficiently implemented using a SAT solver. The solver can be used to find a counterexample refuting one or more candidate invariants or to prove that no such refutation exists.

Inductive Step: Here we verify that for any sequence of states of length k for which all candidate invariants hold in all k states, then the candidate invariants also hold on all states reachable in the $k+1$'st step. Formally:

$$\forall\, S_0, S_1, \ldots, S_k, S_{k+1} \;.\; (S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_k \rightarrow S_{k+1}),$$
$$\big((S_0 \models C) \wedge (S_1 \models C) \wedge \cdots \wedge (S_k \models C)\big) \implies (S_{k+1} \models C)$$

Again, this can be efficiently implemented using a SAT solver, and a series of

counterexamples can be derived that refute particular candidate invariants.

As explored in Section 14, induction is not a complete technique. The inductive step can produce counterexamples that refute true properties. There have been methods proposed that are effective in limiting these spurious counterexamples, and here we consider how they relate to this current work:

- [van Eijk, 2000] argues that induction is made stronger if multiple candidate invariants are proved simultaneously. The reason is that the left hand side of the implication in the inductive hypothesis $\big((S_0 \models C) \wedge (S_1 \models C) \wedge \cdots \wedge (S_k \models C)\big)$, referred to as the **inductive constraint** or **inductive hypothesis**, becomes stronger as C becomes stronger. If elements are added to C then it becomes harder to satisfy every candidate in C, and therefore fewer sequences $S_0, S_1, \ldots, S_k$ are able to satisfy this constraint. This makes the inductive step more difficult to refute and hence candidate invariants are more likely to be proved. For this reason, induction is a good choice for proving candidate invariants because often the candidate invariants will be numerous and induction will naturally be strengthened.

- [Bjesse and Claessen, 2000] demonstrates that induction produces fewer spurious counterexamples if the sequence of states $S_0, S_1, \ldots, S_k, S_{k+1}$ is constrained to be a simple path. That is, no two states explored by the inductive step should be identical. This can be guaranteed using **unique state constraints** [Kroening and Strichman, 2003]. While unique state constraints are effective in strengthening induction, they complicate the SAT problem and slow the overall proof method. The focus in this work is on speed and scalability, and because candidate invariants are numerous we choose to spuriously drop some invariants and keep our method scalable by ignoring uniqueness constraints. Exploration

of efficient methods to incorporate uniqueness constraints will be considered in future work.

2.5.2 Refining Candidate Invariants

Both the base case and inductive step in Algorithm 8 involve the discovery of counterexamples and subsequent refinement of the candidate invariants. Because candidates are iteratively discarded until a stable set is derived, this is an example of a **greatest fixed-point method**.

Efficiently refining the candidate invariants with a given counterexample is key to the scalability of Algorithm 8. Simulation-based routines can be used to propagate the counterexample values through the logic network and derive the set of candidates that are falsified. Simulation is important for two main reasons:

1. In some implementation styles, it may be difficult to determine precisely which candidate invariant is falsified by the counterexample. For example, suppose all candidate invariants are conjoined to form a single property that is to be proved on all reachable states. In this case, the counterexample refutes the conjoined property, but additional analysis is needed to determine the set of candidates that are falsified. Simulation provides such analysis.

2. Often the space of counterexamples is dense in that several valid counterexamples can be found such that the Hamming distance between any two counterexamples is small. By randomizing the counterexample (Section 2.3.3) and using bit-parallel simulation (Section 2.3.1), many similar counterexamples can be explored in parallel. This enables the algorithm to automatically discover related counterexamples and refute additional candidate invariants without invoking the SAT solver. This greatly reduces the total runtime of the inductive proof.

```
 1: function refineCandidates_base(candidates, counterexampleVector)
 2:    goodCandidates := ()
 3:    sim := randomizedSimulation(counterexampleVector)
 4:    for all candidates C in candidates do
 5:       if (C holds on sim) then
 6:          goodCandidates += C
 7:       end if
 8:    end for
 9:    return goodCandidates
10: end function
```

Algorithm 9: Candidate refinement for a single counterexample vector.

Algorithm 8 calls two refinement subroutines, `refineCandidates_base()` to refine the candidate invariants using a base case counterexample, and `refineCandidates_ind()` to refine using an inductive step counterexample.

First consider the simpler of the two, the procedure to handle a base case counterexample, shown in Algorithm 9. A set of candidate invariants and a base case counterexample trace are given, and the task is to produce a subset of the candidate invariants that all hold on the given counterexample. The logic network is simulated with the procedure `randomizedSimulation()` that performs a bit-parallel, randomized simulation of the counterexample trace. The simulation is sequential and includes at least as many time steps as there are steps in the counterexample trace. The randomization is limited to the circuit inputs, and preventing the random data from intermingling with the registers' state guarantees that all states explored by this simulation are reachable. Next, each candidate is iteratively tested against these states. The candidates that hold on this state set are recorded and returned from the procedure.

The more complicated of the two simulation-based refinement procedures is the method to refine the candidates using a counterexample from the inductive step. This method is outlined in Algorithm 10. Again, a set of candidates and a counterexample

```
 1: function refineCandidates_ind(candidates, counterexampleTrace)
 2:    goodCandidates := ()
 3:    sim := randomizedSimulation(counterexampleTrace)
 4:    mask[ ] := 1111...
 5:    for (i := 0; i le width(sim; i++) do
 6:       if (sim[i] ⊭ the inductive hypothesis for candidates) then
 7:          mask[i] := 0
 8:       end if
 9:    end for
10:    for all candidates C in candidates do
11:       if  (C holds in (sim || ¬ mask)  then
12:          goodCandidates += C
13:       end if
14:    end for
15:    return goodCandidates
16: end function
```

Algorithm 10: Candidate refinement for a sequence of inductive counterexample vectors.

trace are given. The trace is simulated using bit parallel simulation and input randomization, just as the base case method does. This input randomization does not guarantee that the states satisfy the inductive hypothesis because randomizing the inputs at time j may affect the state at time $j+1$.

The inductive hypothesis can be accounted for by masking out the explored paths of states that do not satisfy the constraint. Suppose we have a bit-parallel simulation that defines n different states that range over $k+1$ different time steps. That is, the simulation data can be sliced into n different state traces, each of temporal length $k+1$. A length-n *mask* is formed such $mask[i] = 1$ if trace i satisfies the inductive hypothesis on the first k of its states. For all traces i that have $mask[i] = 1$, the $k+1$'st state is examined. A candidate invariant that fails on this state has a counterexample trace that satisfies the inductive hypothesis. Therefore the invariant is not provable by induction and can be discarded.

The masking procedure described above is useful to narrow the simulation data to

a subset that obeys the inductive hypothesis. In practice it is useful to constrain the randomized simulation such that at least one of the n simulated traces is equivalent to the original counterexample trace that was passed to Algorithm 10. This trace will refute at least one candidate invariant, and thus constraining the simulation guarantees that Algorithm 10 will make forward progress and return a proper subset of the candidate invariants.

```
1: function refineCandidatesLoop_ind2(candidates, counterexampleTrace)
2:    repeat
3:       startingCandidates := candidates
4:       candidates := refineCandidates_ind(candidates, counterexampleTrace)
5:    until (startingCandidates == candidates)
6:    return candidates
7: end function
```

Algorithm 11: Outer loop around Algorithm 10.

As Algorithm 10 discards candidate invariants the inductive hypothesis weakens. This may mean that additional randomized counterexample traces that were previously masked out now satisfy the weaker inductive hypothesis. This indicates that the *mask* can be weakened and the number of ones in the mask can increase. However, in practice the mask is expensive to compute and incremental re-computation is not practical.

It is important to consider the effects of the weaker inductive hypothesis in order to remove as many candidate invariants as possible before resorting to another SAT call. SAT is expensive, and a method that relies heavily on simulation to refute candidate invariants is usually more scalable. Therefore consider an enhanced procedure that iteratively weakens the inductive hypothesis, shown in Algorithm 11. In this, the procedure `refineCandidates_ind()` (Algorithm 10) is repeatedly called until the set of candidate invariants reaches a fixed point. This allows the mask to be recomputed, allowing more counterexample traces to be compared against the set of candidate

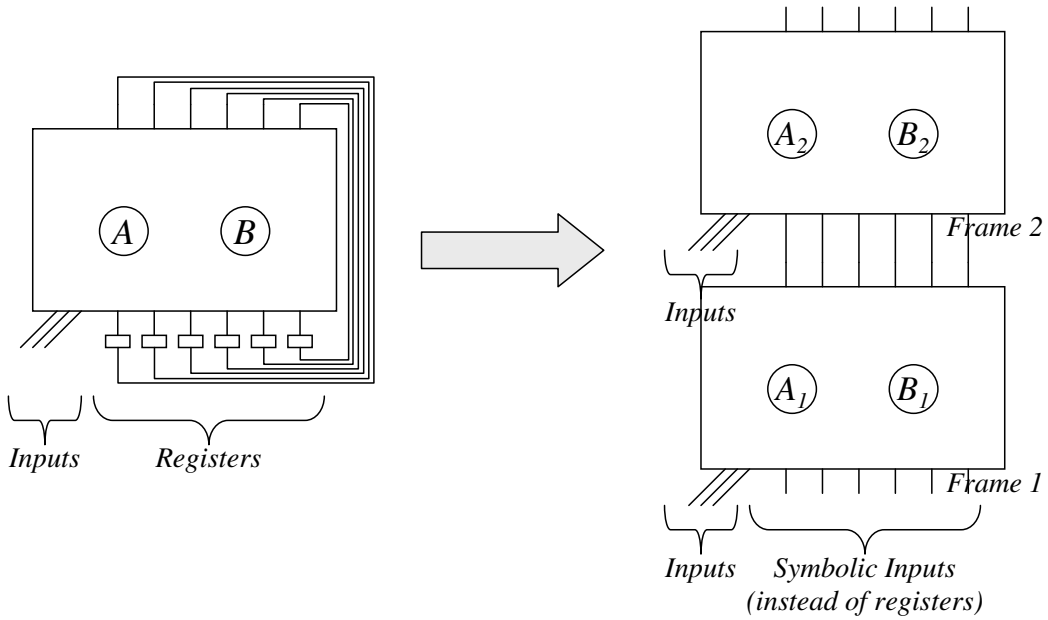


Figure 2.8: Unrolling the transition relation to check the inductive step.

invariants and hence further refining the candidates. Algorithm 11 is an effective way to take advantage of an ever-weakening inductive hypothesis without paying a large runtime price to recompute the mask.

2.5.3 Speculative Reduction

Induction can be implemented efficiently using a SAT solver, and here we will discuss an important technique that makes this implementation so efficient: speculative reduction.

Suppose there is a single candidate invariant $A = B$ that is to be proved with $k = 1$ induction. The inductive step of the proof can be accomplished by unrolling the circuit as shown in Figure 2.8. The original sequential logic network is composed of three pieces: inputs, registers, and a transition relation. We construct an **unrolled logic network** by instantiating multiple copies of the transition relation, each with its own

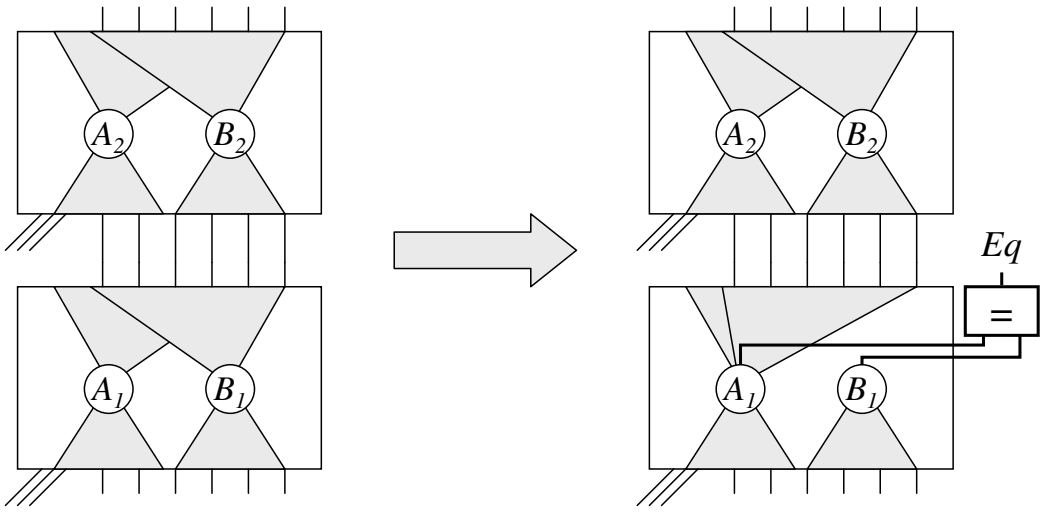


Figure 2.9: Using speculative reduction to simplify the inductive step check.

dedicated set of inputs. The registers in the first copy are driven by symbolic inputs, and the registers in every other copy are driven by the next state functions from the previous copy. In this way, the unrolled logic network represents a time-unrolled model of the logic. The lower copy, referred to as frame 1, will input a symbolic state S_1 and output a next state S_2. The upper copy, frame 2, takes this state S_2 as input. In this way the two copies represent two adjacent temporal snapshots of the logic network.

In order to check the inductive step of the $A = B$ proof it is sufficient to check that for all inputs, $(A_1 = B_1) \implies (A_2 = B_2)$. The unrolled logic network can be given to a SAT solver, and this check can be formulated as a SAT problem.

The SAT problem can be simplified using speculative reduction [Mony *et al.*, 2005], a method of simplifying the circuit using an invariant before the invariant is proved. Consider the circuit transformation shown in Figure 2.9. Each instance of A and B has a **fanin cone** which includes all the logic whose output values can propagate through the node and a **fanout cone** comprised of the logic that the nodes value can propagate through. These cones are highlighted in the figure. We transform the

unrolled logic network by driving B_1's fanout cone with A_1 and adding logic to check the equality of A_1 and B_1, letting $Eq = (A_1 \equiv B_1)$[1]. The inductive step check then reduces to $Eq \implies (A_2 = B_2)$.

Theorem 2.5.1. *The speculatively reduced expression holds if and only if the original inductive step expression holds.*

Proof. First suppose $(A_1 = B_1) \implies (A_2 = B_2)$. Then either $(A_1 = B_1) \wedge (A_2 = B_2)$ or $A_1 \neq B_1$. In the first case Eq holds and so the speculative reduction transformations do not logically change A_2 or B_2, hence $(A_2 = B_2)_{orig} = (A_2 = B_2)_{spec.reduced}$. Therefore we have $Eq \implies (A_2 = B_2)$. If $A_1 \neq B_1$ then we have $\neg Eq$ and $Eq \implies (A_2 = B_2)$ holds vacuously.

Next suppose $(A_1 = B_1) \not\Longrightarrow (A_2 = B_2)$. Then $A_1 = B_1$ and $A_2 \neq B_2$. Hence we have Eq and $A_2 \neq B_2$ and so $Eq \not\Longrightarrow (A_2 = B_2)$. □

Speculative reduction dramatically simplifies the SAT problem because the unrolled logic network can be significantly compacted. In moving fanout cone logic from B to A, similar functions present in both fanout cones will become redundant. These redundancies can be eliminated using a technique such as **structural hashing** [Kuehlmann and Krohm, 1997], and the unrolled logic network can be significantly compacted. In practice this leads to large runtime improvements.

Now examine the impact of speculative reduction on the proving of invariants from each of the invariant families discussed in Section 2.2:

Constants : Speculative reduction of constant invariants can greatly simplify the inductive invariant check. A check that the original node is constant must be preserved, and in all other contexts the equivalent constant can be used in place of the node. This means that **constant propagation** can greatly simplify the circuit.

[1] This logic can be implemented with a simple **xor gate**.

Equivalences : Similar to constants, these can also simplify the circuit. These were examined in Figure 2.9. Logic testing the equivalence must be constructed, and then the fanout cones can be manipulated such that only one of the nodes in the equivalence drives both fanout cones. This generalizes to larger equivalence classes, and exactly one node from each class can be chosen as the representative and used to drive all the fanout cones. This can greatly simplify the logic network.

All others : There is no known method to utilize the other invariant types for speculative reduction. Because of the benefit speculative reduction has for constants and equivalences, this is an interesting area for future research.

2.5.4 Partitioning the Proof

Despite techniques such as induction and speculative reduction, proving that candidate invariants hold on all reachable states can still be a difficult task. This is especially true if the number of candidate invariants is numerous. In this case, the inductive formulation gets translated into a large SAT problem with many constraints, and such a problem can sometimes be difficult to solve.

If other proof optimization methods fail, one crude yet effective method remains: partitioning. **Partitioning** refers to the process of decomposing the set of candidate invariants into disjoint subsets, or partitions [Case *et al.*, 2006b]. Each partition can be proved in isolation, thereby decomposing a difficult proof obligation into a series of simpler problems.

An example of partitioning is shown in Figure 2.10. The candidate invariants are partitioned and each partition is proved separately. For each partition, the number of candidate invariants initially is quite large but quickly decays to a fixed point where the candidate invariants are finally proved to hold on every reachable state. The

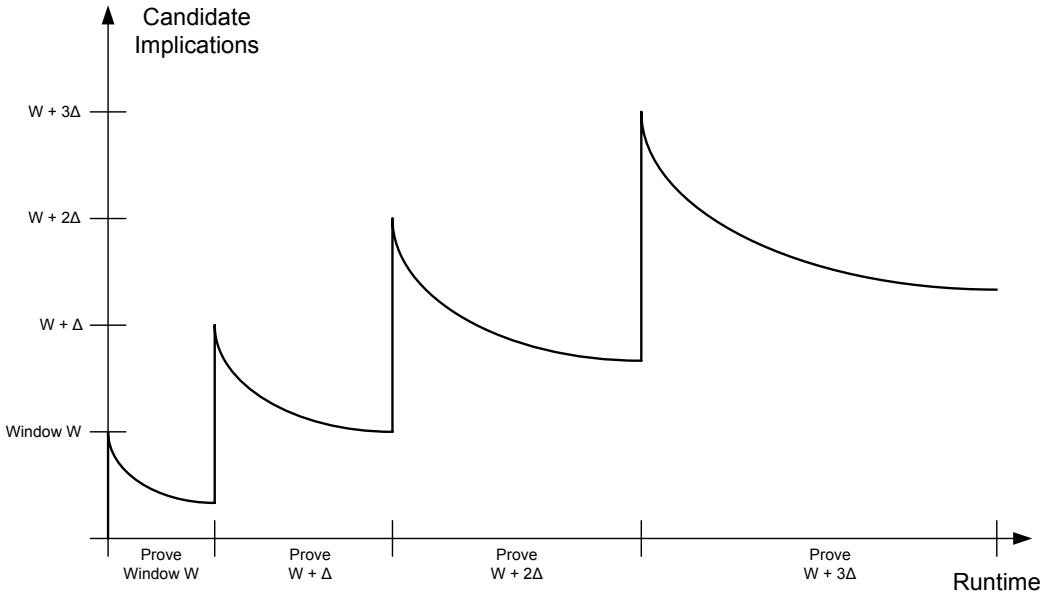


Figure 2.10: Use of partitioning to simplify the proof obligation.

total number of invariants, both candidate and proved, grows as new partitions are processed. Note that the number of proved invariants grows monotonically.

Partitioning can greatly improve the scalability of induction but is not without its drawbacks. Induction is a strong technique when multiple candidate invariants are proved in conjunction, and this is because many candidate invariants can combine to yield a strong inductive hypothesis, as discussed in Section 2.5.1. If the set of candidate invariants is decomposed, each partition will be smaller than the original set of candidate invariants and therefore have a weaker inductive hypothesis. This means that in total, a partitioned proof might be able to prove fewer invariants than a monolithic proof, due to the incompleteness of induction.

One way to partially counter this incompleteness is to use the proved invariants from one partition in the proof of all future partitions. The proved invariants in one partition give a reachability approximation, and the future induction proofs can be strengthened by constraining the counterexamples to only come from this approx-

imate space. This reduces the number of spurious counterexamples and therefore strengthens induction. Using proved invariants to strengthen future partition proofs is a good way to recover some of the incompleteness inherent in partitioning. The strengthened method is not as strong as a monolithic proof, but the speed improvement that can be realized by partitioning makes this small trade-off appealing in practice.

2.6 Efficient Storage of Candidate Invariants

In order to make an invariant discovery algorithm scalable, it is necessary to be able to handle a potentially large number of candidate invariants. It is important to invest in high-quality simulation (Section 2.3) and scalable proof techniques (Section 2.5). However, equally important is to efficiently store the candidate invariants.

2.6.1 Storage of Constants and Equivalences

Constant and equivalence candidate invariants can be easily stored using equivalence classes. An *equivalence class* is simply a set of equivalent objects. All candidate equivalences can be analyzed and sets of equivalent nodes can be gathered. These sets can be stored as equivalence classes.

This concept also generalizes to constant candidate invariants by considering a constant node to be either equal to the constant 0 or constant 1. Therefore constant invariants can be stored in an equivalence class along with the appropriate constant node.

Example 17. Equivalence Classes

Suppose the following candidate invariants are discovered through simulation:

$$\begin{array}{cccc} a = b & d = e & e = g & i \\ b = c & d = f & h & j \end{array}$$

Because of the transitivity of equivalence, the nodes a, b, and c are all equivalent. Therefore $\{a, b, c\}$ is an equivalence class. Similarly, $\{d, e, f, g\}$ is also an equivalence class.

The candidate constant invariants h, i, and j are among the invariants listed above. These can be recorded as a single equivalence class $\{1, h, i, j\}$ since each of these constant candidates is presumed to be equal to constant 1.

Equivalence classes provide an efficient way to store constant and equivalence candidate invariants in memory. Each class can be represented in memory as a linked list, and the set of equivalence classes is simply a linked list of linked lists.

Equivalence classes can also be efficiently proved. Note that an equivalence class with n nodes represents $\binom{n}{2} = \frac{n(n-1)}{2}$ or $O(n)$ different equivalences. For large n this can be prohibitively expensive. Alternatively, one can choose a **representative** node from each class and simply prove that each node is equivalent to the representative. This gives $n-1$ or $O(n)$ equivalences to prove rather than the more expensive $O(n^2)$.

2.6.2 Storage of k-Cuts and Random Clauses

The best known method for storage of k-cut and random clause candidate invariants in simply with a linked list. The storage size is linear in the number of candidate invariants. While it may seem that more research is needed in this area, in reality the storage size for these invariant types is not a problem because k-cut invariants are usually few in number and the number random clause candidates is usually closely controlled and limited to a small number.

2.6.3 Storage of Implications

When deriving candidate implication invariants, it is tempting to enumerate candidate implication invariants over the entire logic network. This gives a large number of candidate invariants, $O(n^2)$ for a network with n nodes. This motivates methods to store the candidate implication invariants in a compact manner, and this has been the subject of our past research [Case *et al.*, 2006a; Case and Brayton, 2007].

2.6.3.1 Underlying Graph Theory

The set of candidate implication invariants can be represented as a directed graph, known as the **implication graph**. The nodes involved in the candidate implication invariants become the nodes of the graph, and for each candidate implication $a \rightarrow b$ there is one directed edge in the implication graph from node a to node b. Because of the transitivity of implications ($a \rightarrow b$ and $b \rightarrow c$ imply $a \rightarrow c$), the implication graph contains an implication $x \rightarrow y$ if y is reachable from x along the directed edges in the graph. The problem therefore is to minimize the graph and maintain the reachability relationships between pairs of nodes.

In any directed graph, if one is only interested in preserving reachability information, then there is flexibility in choosing the set of edges to represent. For example, in the simple graph $a \rightarrow b \rightarrow c$, a can reach c because there is a path from a to c. In this graph the presence of the edge $a \rightarrow c$ is optional. Since it does not add any new reachability information to the graph, depending on the application an implementer may choose to exclude it.

If the graph contains all optional edges, it is known as a **transitive closure**. The transitive closure has the maximal edge set, and this can dramatically increase the resources required to store it. However, checking for the existence of a path between two vertices reduces to checking for the existence of a single edge.

Alternately, we may wish to minimize the number of edges in the graph by excluding all optional edges. This gives rise to the transitive reduction. The **transitive reduction** $r(G)$ of a directed graph G is a directed graph satisfying the following properties:

1. The vertex sets of G and $r(G)$ are identical.
2. There is a directed path from vertex u to vertex v in G if and only if there also one in $r(G)$.
3. There is no graph with fewer edges than $r(G)$ satisfying the above conditions.

Since the set of edges is minimized, storage requirements are also minimized. However, the induced graph sparsity can increase the time required to check for the existence of a path between two vertices. In both memory and runtime, the transitive reduction has characteristics opposite the transitive closure. Unfortunately, while the transitive closure has been extensively studied, the transitive reduction has received less attention in the literature.

We are motivated to study the transitive reduction because of its applications in minimizing the storage requirements for candidate implication invariants. We utilize a conservative approximation to a transitive reduction to maintain internal data-structures, benefiting from the reduced memory requirements. Also, because each edge represents one implication to be proved, reducing the number of edges helps to improve the performance of the proof step without compromising the underlying information content.

New algorithms have been developed to efficiently maintain a conservative approximation of the transitive reduction. It is maintained in an online manner through any number of edge addition and removal operations. Here we give the algorithms, their complexities, proofs of correctness, and some experimental results to empirically verify the theoretic results.

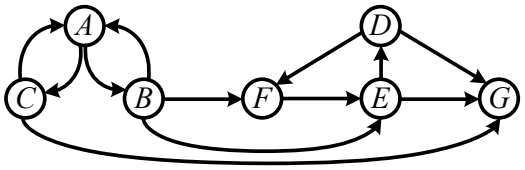


Figure 2.11: An example graph G.

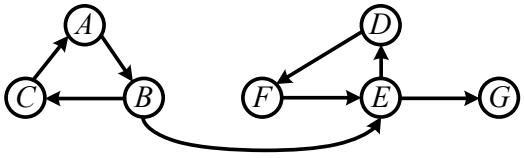


Figure 2.12: Corresponding $r(G)$.

2.6.3.2 Approximating the Transitive Reduction

Due to problems that arise in the online maintenance of a true transitive reduction, we choose to approximate the transitive reduction.

Consider the directed graph G and a transitive reduction $r(G)$ of it shown in Figures 2.11 and 2.12. The reduction can be built by first greedily removing redundant edges. This eliminates the need for $C \rightarrow G$, $B \rightarrow F$, and $D \rightarrow G$. For all of these edges, there exists an alternate path joining the pair of vertices, and this path makes the directed edge redundant. The next step in creating the transitive reduction is to identify all **Strongly Connected Components** (**SCCs**) and replace each with a simple cycle. This simple cycle maintains the connectedness of the component with the minimum number of edges. In the figure the SCC $\{A, B, C\}$ has been thus processed.

The transitive reduction is simple to build, but difficult to maintain. Suppose that we now wish to remove the edge $A \rightarrow B$ in the reduction shown in Figure 2.12. In the resulting graph, A and B should be placed in separate components, but it is unclear to which component C should be assigned. There is not sufficient information in $r(G)$ to refine this component.

The solution is to relax the constraint that components be joined by simple cycles. This leads us to explore an approximation called the **Minimum Equivalent Graph** (**MEG**). Given a directed graph G, the MEG $meg(G)$ is a transitive reduction

satisfying the following:[2]

1. $edges(meg(G)) \subseteq edges(G)$.

2. $\forall$ edges $u \rightarrow v$ in $meg(G)$, $\nexists$ an alternate simple path $u \rightarrow v$ in $meg(G)$.

2.6.3.3 Maintaining the MEG Under Edge Addition

Consider the addition of an edge to an MEG. This edge can introduce any number of new paths which may make other edges redundant. Any edge addition algorithm must identify and remove these now-redundant edges.

```
1: // a→b := Edge to be added
2: if ∄ path from a to b then
3:    Add edge a→b
4:    Color ancestors of a red
5:    Color descendants of b blue
6:    for all edges from a red r
                 to a blue b do
7:       if ∃ simple path r→b
              through a→b then
8:          Remove r→b
9:       end if
10:   end for
11: end if
```

Algorithm 12: Edge addition algorithm.

Algorithm 12 takes redundant edges into account to maintain the MEG under edge addition. An example application of this algorithm is shown in Figure 2.13. The left-most graph is an MEG to which we will add edge $C \rightarrow D$. To identify the edges that are made redundant, we color the ancestors of C and the descendants of D. The edges $A \rightarrow D$ and $C \rightarrow E$ are between colored vertices, and both of these edges have alternate simple path through $C \rightarrow D$. The two edges are redundant, and they

[2] If G is acyclic then the $meg(G) = r(G)$.

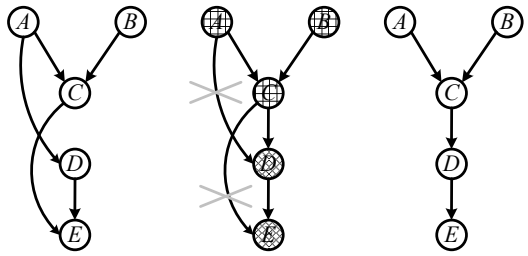


Figure 2.13: Example edge addition.

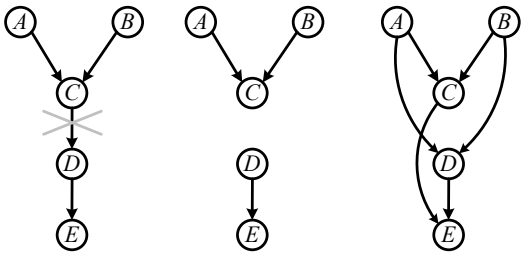


Figure 2.14: Example edge removal.

are removed to make the output MEG. Because we are maintaining an MEG, adding one edge caused the total number of edges to decrease by 1 while the reachability information content increased.

Theorem 2.6.1 (Correctness of the addition algorithm)**.** *If the input to Algorithm 12 is an MEG then the output is also an MEG.*

Proof. Let $a \rightarrow b$ be the edge added by Algorithm 12, and let the input and output graphs be given by I and O respectively. We assume I is an MEG and now proceed to show that O is an MEG.

Consider $a \rightarrow b \in O$. If I had a path from a to b then $O = I$ (by line 2) and so is an MEG. Therefore assume I had no such path. Algorithm 12 adds only one edge $(a \rightarrow b)$, and so is incapable of forming an alternate simple path from a to b.

Now consider all $u \rightarrow v \in O$ such that $u \neq a$ or $v \neq b$. Because Algorithm 12 adds only $a \rightarrow b$, we must have $u \rightarrow v \in I$. Because I is an MEG, there is no alternate simple path from u to v. Suppose algorithm 12 introduced such an alternate path. Then $a \rightarrow b$ would have to be on this path since all new paths go through this edge. In this case, $u \rightarrow v$ is removed in line 8 and so could not possibly be in O. By contradiction, $u \rightarrow v$ has no alternate simple path in O.

Because no edge in O has an alternate simple path, O is an MEG. □

To analyze the time complexity of Algorithm 12, suppose the input MEG has v

vertices and e edges. On line 2, path existence can be implemented as a depth-first search which has complexity $O(v + e)$. Adding the edge on line 3 can be done in constant time, if implemented properly. Coloring sets of vertices on lines 4-5 can be done with two more depth-first searches. In lines 6-10, a path existence check and possibly an edge removal must be done for each edge. This dominates all other steps with complexity $O(e \cdot ((v + e) + 1)) = O(ev + e^2)$. Thus the complexity of Algorithm 12 is $O(ev + e^2)$.

2.6.3.4 Maintaining the MEG Under Edge Removal

```
1: // a→b := Edge to be removed
2: Remove edge a→b
3: for all parents p of a do
4:    Add p→b with algo. 12
5: end for
6: for all children c of b do
7:    Add c→c with algo. 12
8: end for
```

Algorithm 13: Edge removal algorithm.

Removal of an edge from an MEG is conceptually harder and has higher complexity than edge addition, but the algorithm is simpler. In an MEG a single edge may lie on multiple paths in the graph. These paths represent reachability information, and the removal of an edge must not violate this reachability. Therefore, edge removal involves both the identification of these disturbed paths as well as the addition of edges to preserve the paths in the absence of the now removed edge.

Algorithm 13 effectively removes an edge without disturbing any other paths in the MEG. An example application of this algorithm is shown in Figure 2.14. The left-most graph is an MEG from which we wish to remove the edge $C \rightarrow D$. Note that in this graph, A can reach $\{D, E\}$, but removal of $C \rightarrow D$ disturbs this. Several

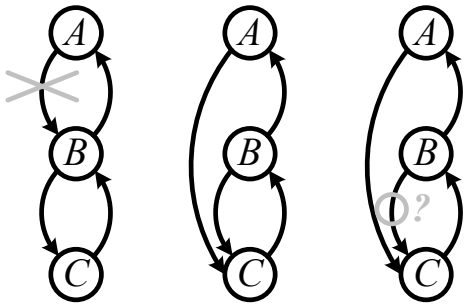


Figure 2.15: Edge removal needs Algorithm 12.

other reachability relationships are similarly disturbed. We can maintain the graph by adding the edges $A \rightarrow D$, $B \rightarrow D$, and $C \rightarrow E$, as described in lines 3-8 of the algorithm. Because we maintain an MEG, removal of an edge caused the total number of edges to increase by 2 while the total reachability information decreased.

The use of Algorithm 12 as a subroutine to Algorithm 13 may at first seem unnecessary, but it is vitally important in order to maintain the MEG. Consider Figure 2.15 as an example. Removal of edge $A \rightarrow B$ causes $A \rightarrow C$ to be added which makes $B \rightarrow C$ redundant. Unless we use Algorithm 12 for edge addition, we will not detect this redundant edge and the output will not be an MEG.

Theorem 2.6.2 (Correctness of the removal algorithm)**.** *If the input to Algorithm 13 is an MEG then the output is also an MEG.*

Proof. Algorithm 13 can be subdivided into two parts: removal of the specified edge and calls to Algorithm 12.

Since the input is an MEG, and because removal of an edge cannot introduce any paths in the graph, the graph after the edge removal is an MEG.

By Theorem 2.6.1 we know the graph after calls to is also a Algorithm 12 is also an MEG. □

Consider now the time complexity of Algorithm 13. Let the input MEG have v vertices and e edges. The edge removal on line 2 can be done in constant time. In

lines 3-8 we call Algorithm 12 possibly v times for a complexity of $O(v \cdot (ev + e^2)) = O(ev^2 + e^2v)$. The total complexity of our edge removal algorithm is $O(ev^2 + e^2v)$.

2.6.3.5 Graph Theoretic Results

The graph algorithms described here were implemented in a C++ library which was highly tuned for maximum performance. The algorithms were incorporated inside an implication invariant discovery engine.

In this first set of results, the MEG library is run by itself using a small driver application to generate a sequence of random edge additions and removals. This sequence is sufficiently long to provide reliable statistics, and the number of vertices in the graph is varied to expose the underlying sensitivity to this parameter. The performance is compared against a normal graph package which just updates the graph and does no reduction whatsoever.

Figure 2.16 shows the average number of edges present in the graph. It shows this for both the version maintaining $meg(G)$ and that maintaining G, and it shows this over a range of vertex set sizes. From this, we can see that the MEG has dramatically fewer edges, and the difference grows with the vertex set size. This was the original motivation for our study of the transitive reduction and MEG.

Figure 2.16 also shows the runtime performance of the addition algorithm by showing the normalized time for 1,000,000 edge additions[3]. Recall that the edge addition algorithm has complexity $O(ev + e^2)$, and we can see this linear dependence on v in the figure. For this algorithm, the difference between the two graph implementations is not very large.

The edge removal is studied in the last graph in Figure 2.16 where the normalized time for 1,000,000 edge removal operations is shown. Recall that we expect complexity $O(ev^2 + e^2v)$, and the graph shows that the performance is roughly quadratic in the

[3]All tests were run on a 1.6 GHz Pentium-M laptop.

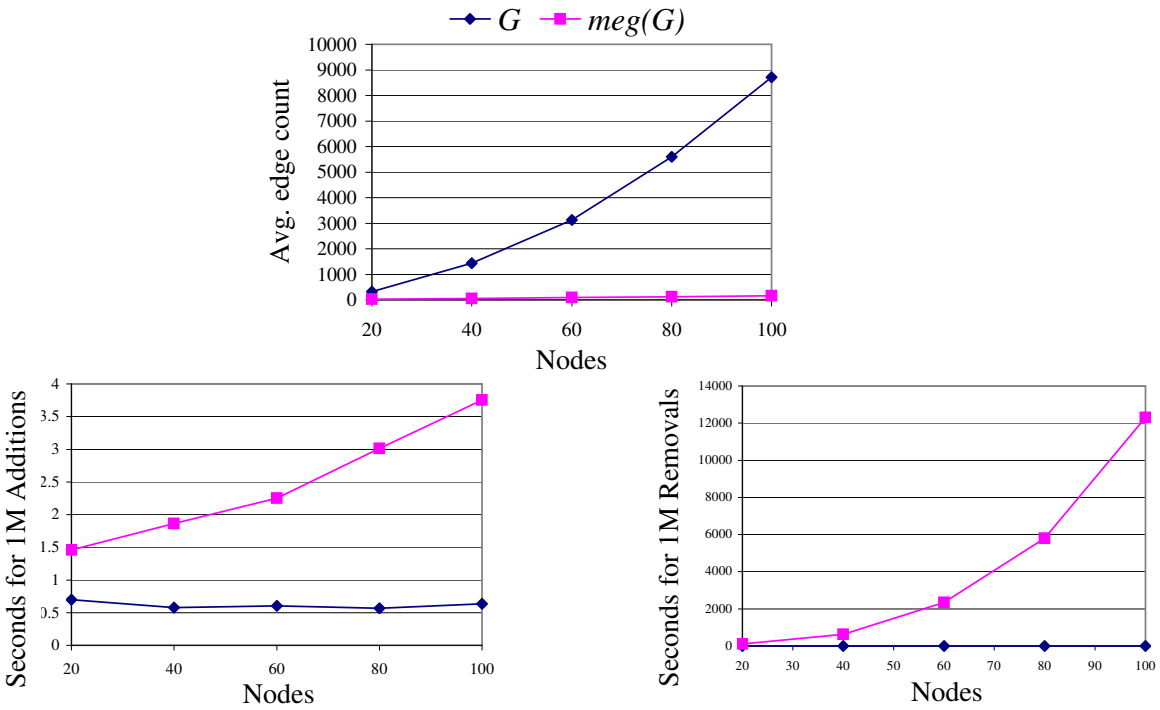


Figure 2.16: Empirical Analysis of the MEG Algorithms.

number of vertices. The performance gap between the two graph packages is great, and the cost of maintaining the MEG under edge removal is non-trivial. Finding a more efficient edge removal algorithm could be the focus of further research.

As expected, the tests show great savings in storage but increased runtime. We need to look at the performance of the implication discovery framework as a whole to see benefit from the MEG library. In the following experiments we examine the performance of the algorithms in this application with vertex and edge numbers as they would occur naturally. This gives a much more informative picture of the true performance.

2.6.3.6 Invariant Discovery Results

In our motivating invariant discovery application the two graph libraries, maintainers of $meg(G)$ and G respectively, were used as the implication graph manager. The code is structured such that the user may select which graph library to use.

Using this framework, we ran the tool on 15 circuit designs: 10 small academic designs and 5 obtained from industry. The tool was run using each graph library, and the total runtime and memory was recorded.

In the following graphs, the x-axis gives the measured quantities for the $meg(G)$ package, and the y-axis gives the quantities for the G package. Each point is a single design as measured under both implementations. If the point appears above the diagonal, the $meg(G)$ implementation was better than the G implementation, and the performance gap corresponds to the distance above (or below) the diagonal.

Figure 2.17 shows the runtimes of the implication invariant discovery tool. Note that the runtime of the tool with the $meg(G)$ package is roughly constant over these 15 designs, and in almost all cases the tool is made faster by using $meg(G)$. However, this figure is not a true indication of the performance gap because in running the experiments, it was necessary to terminate the flow of the tool early. Without this

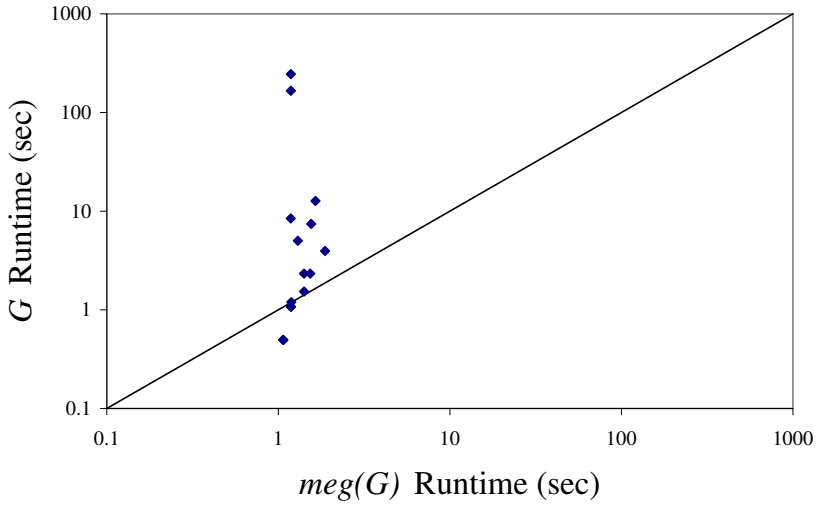


Figure 2.17: Invariant discovery with the MEG: runtime.

early termination, the tool with G failed to run in less than 10 minutes (on 13 designs) while the tool with the $meg(G)$ package took roughly the same amount of time as shown. Thus the performance gap between the two implementations is very large and grows as the cutoff time limit is increased.

It is interesting to understand why the tool runs faster when using $meg(G)$. Although using this reduction introduces polynomial-time overhead in the maintenance routines, for each edge in the graph, the tool must prove the implication the edge represents. This proof can be exponential in complexity, and the reduced edge set of the MEG allows a trade-off between exponential and polynomial complexity. This enables the dramatic performance improvements. In fact, without these improvements, use of implication invariants is impractical, and the MEG is what enables it to be run at all.

Figure 2.18 shows the peak memory allocated by the logic synthesis tool. This is recorded for both graph packages over the 15 circuit designs. Again we see that the memory used by the tool with the MEG package is roughly constant, and in nearly all cases this beats the memory used with the normal package. The performance gap in

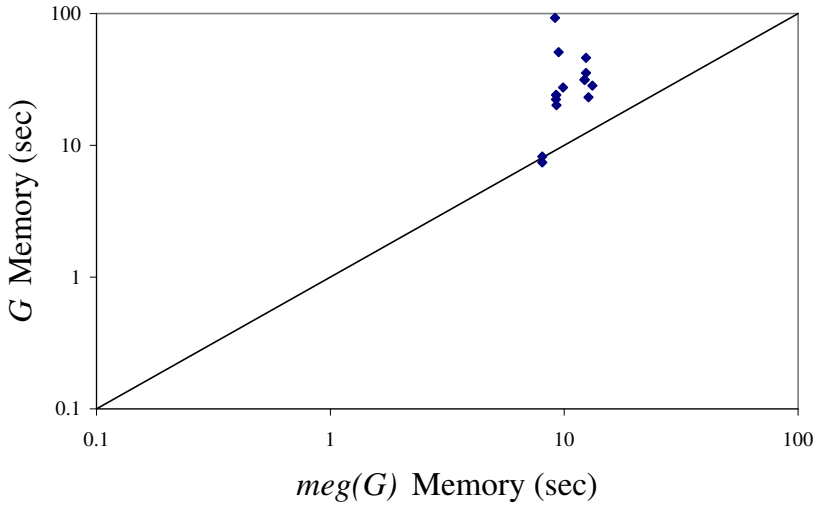


Figure 2.18: Invariant discovery with the MEG: memory.

memory is not always large because we focus on small designs where the implication graph typically does not dominate the memory consumption. With larger designs, we have observed that without the MEG package we cannot fit the application into our memory space, making the logic synthesis tool impractical.

Clearly the MEG and its associated maintenance algorithms are vital for efficient storage and processing of candidate implication invariants.

2.7 Orthogonality of Invariant Families

So far methods to efficiently process invariants from 5 different invariant families have been discussed: constants, equivalences, implications, k-cuts, and random clauses. Each family has its strengths and weaknesses, both in terms of computational characteristics as well as ability to approximate the reachable state space.

Often the invariant families are able to approximate the reachable state set in orthogonal ways. The approximation offered by implications, for example, is fundamentally different than the approximation offered by k-cuts. By finding invariants

from different families and conjoining the results, we can harness these orthogonal strengths to produce a higher quality reachability approximation than is possible with any single invariant family.

Furthermore, it is often useful to compute invariants in waves. Similar to the windowing proof technique of Section 2.5.4, we again decompose the invariants proof. Here we go a step further and decompose the invariant discovery as well. This has several advantages:

1. Proving invariants in waves provides an effective way to manage computational resources. In practice, a global timeout is always specified because there are almost always more candidate invariants than there is time available to discover and prove them. The computation can be terminated between waves, and the results are sound because all candidate invariants have been proved at these points.

2. Decomposing the invariant discovery into waves means that each wave can proceed with different parameters. For example, the invariant family can change between waves, allowing the invariants from one wave to strengthen the approximations found by all previous waves. (There is also the reverse strengthening where the previously proved invariants are used as constraints in the proof of the current wave.)

3. Having a number of previously completed waves means that at the start of each wave a crude reachability approximation is available. Using the techniques of Section 2.4.2, candidate invariants in the current wave can be prioritized such that the ones that are most useful in refining the reachability approximation are proved while those that are less useful are ignored. This is important for quickly converging on a tight reachability approximation.

```
 1: interestingStates := randomWalk()
 2: provedInvariants := ∅
 3: while (computational resources not exceeded) do
 4:    invariantFamilies, parameters := readConfigFile()
 5:    // Algorithm 6
 6:    candidates := mineCandidateInvariants(interestingStates,
 7:                                          invariantFamilies,
 8:                                          parameters)
 9:    // Algorithm 7
10:    candidates := prioritizeCandidates(candidates,
11:                                       provedInvariants,
12:                                       parameters)
13:    // Algorithm 8
14:    provedInvariants += proveInvariants(candidates, parameters)
15: end while
16: return provedInvariants
```

Algorithm 14: Total Invariant Discovery Algorithm, using Orthogonality.

The total invariant discovery algorithm, a summary of all methods presented so far, is shown in Algorithm 14. Initially a set of interesting states is derived from a random walk (Section 2.3.2), and the set of proved invariants is empty. Then proceed to prove waves of invariants until the computational resources are exceeded. Each wave proceeds as follows: the invariants are mined from the simulation information, they are prioritized relative to the previously proved invariants, and they are then proved.

An example illustrating the execution of Algorithm 14 was shown in Figure 2.7 in Section 2.4.2. Here the computation proceeds in waves such that each wave selects a different mix of invariant families. The progression of waves is tuned such that the proof complexity increases over time, meaning that the easier-to-prove invariants are tried first. This is important if little runtime is available.

Each wave uses candidate prioritization to select candidates that are most likely to refine the reachability approximation from the previous waves. This means that in

Wave 4, the number of candidate implication invariants is small. Usually the number of implication invariants is large, motivating the MEG study in Section 2.6.3. Like the MEG, candidate prioritization is another effective way of handling a large number of candidate implication invariants.

Wave computation allows for the variation of other parameters besides simply the invariant families. Note in Figure 2.7 that for all j, Waves j and $j + 5$ use the same invariant families. This is because Waves 1-5 use $j = 1$ induction as the proof technique while Waves 6-9 use $j = 2$ induction. This allows us to gradually increase our proof effort as time permits, and this is important for complex designs or highly runtime-constrained runs.

In practice the wave computation that utilizes both orthogonal strengthening as well as gradually increasing proof effort provides an effective way to quickly form a good reachability approximation. In an application where the reachability information was used to perform sequential synthesis, often the execution of Algorithm 14 with a time limit of 60 seconds was sufficient to approximate reachability to the degree necessary for the synthesis algorithm. If the time limit was increased and more invariants found then the quality of the synthesis did not improve, indicating that the reachability approximation quickly converges to a set of states and after a point in time (about 60 seconds) changes only very little. This was experimentally verified on hundreds of industrial designs with a wide range of sizes.

2.8 Alternative Methods For Approximating Reachability

Invariants provide a way to approximate the set of reachable states, but there are other ways to approximate reachability without the use of invariants. For the sake of

completeness, here we discuss two other reachability approximation methods: BDD supersetting and interpolation.

2.8.1 BDD Supersetting

Reachability analysis using BDDs (Example 13) works well on small circuits, but as the number of registers increases, the BDDs used in the computation become too large to handle efficiently.

One approach to deal with these large BDDs is to approximate them. There are numerous techniques [Somenzi, 2005] for introducing onset minterms into the function represented by the BDD such that the number of BDD nodes is reduced. Such methods are used to replace the BDD with a function that is a superset and has a more compact representation.

In this work, we do not discuss BDD supersetting because we attempt to produce synthesis and verification algorithms that are entirely free of BDDs. This avoids the computational pitfalls that come with BDDs. While supersetting addresses these pitfalls, it is not sufficient to prevent all computational difficulties. For this reason, in this work we focus on techniques rooted in SAT solving.

2.8.2 Reachability by Interpolation

Interpolation-based model checking presented in Example 15 can be used to approximate reachability. Suppose a property has been verified to hold on all reachable states by interpolation. Then the interpolation routine has produced an over-approximation to the reachable states such that the property holds on all states in this over-approximation.

It is tempting to use interpolation as a general-purpose engine for approximating reachability. However, by its nature the interpolation-based model checking algorithm

needs a true property to verify in order to produce this reachability approximation. It is not obvious how to manufacture meaningful properties that give good reachability approximations, and for this reason such a reachability approximating algorithm hasn't been researched in the past.

Example 18. Reachability by Interpolation: Practical Experience

Suppose we want to use interpolation-based model checking to approximate the set of reachable states in a sequential circuit. We manufacture a property and verify it with interpolation. After the property has been verified, the interpolation routine has produced an over-approximation to the set of reachable states such that all the property is satisfied on all the over-approximate reachable states.

Further suppose a very simple technique is used to manufacture properties to be checked. In any sequential circuit, the number of unreachable states is usually much larger than the number of reachable states. Therefore, a randomly chosen state has a high probability of being unreachable. Suppose we formulate a property that simply says that the FSM is never in a predetermined, random state. We pass this property to interpolation, and if the property is verified then a reachability approximation is obtained.

Unfortunately, while this method works, the reachability approximations produced are not valuable. Interpolation is aimed at producing a reachability approximation that is sufficient to prove the property, not one that is a tight approximation to the true set of reachable states. For this reason, often the reachability approximations resulting from our simple manufactured properties are equivalent to every state being reachable except for the single pre-selected random state. Clearly, this approximation is very crude and not useful in any of the contexts examined in Section 2.1.2.

If in the future a method is invented that can manufacture meaningful properties

over arbitrary sequential logic networks, then interpolation-based reachability may be an attractive option. The properties would need to be such that 1) the property is true, 2) the property fails to hold in many unreachable states, 3) the failing states are "close" to the reachable states, requiring the resultant interpolants to have sufficient information to distinguish the reachable states from the almost-reachable states, thereby giving a tight approximation to reachability.

Chapter 3

Applying Invariants to Sequential Synthesis

3.1 Motivation

Invariants are useful for over-approximating the set of reachable states, but by itself this accomplishment is not directly useful to the end user of an **Electronic Design Automation** (**EDA**) tool. The value of invariants comes when they are leveraged to improve an existing EDA process.

Here we consider logic synthesis. Invariants provide a set of states that are guaranteed to be unreachable (they under-approximate the unreachable state space), and on these unreachable states a synthesis algorithm is free to change the design's behavior. These induced **sequential don't cares** provide flexibility that can be exploited to optimize the design.

While there exist ways to directly use invariants for synthesis purposes by merging signals that invariants found to be equivalent (Sections 2.2.1 and 2.2.2), in this chapter we will consider the more indirect use of invariants as providers of sequential don't

cares. By viewing invariants in this way they will be able to benefit many different synthesis algorithms, even when the invariants themselves cannot directly justify the merging of signals.

3.2 Direct And Indirect Sequential Synthesis

There exist many sequential synthesis algorithms that do not rely on invariants, and here we seek to differentiate those types of algorithms from their invariant relatives.

We define a **direct sequential synthesis** method as any method that is able to reason directly about the state space in order to perform design optimizations. Many such algorithms have been reported in past research literature.

We define an **indirect sequential synthesis** method as a method that is able to change the behavior of the design on certain states, or even change the number of states, because the synthesis is leveraging sequential don't cares. Normally these sequential don't cares come from invariants, and any combinational algorithm that is able to use don't cares can be made sequential by using invariants in the form of sequential don't cares. Such synthesis techniques haven't been well studied because of the absence of invariant generation techniques that are able to quickly develop substantial sequential don't cares.

Direct and indirect sequential synthesis methods have distinct advantages and disadvantages. Table 3.1 compares these methods in more detail.

3.3 Case Study: SAT-Based Resubstitution

To further compare direct and indirect sequential synthesis, two case studies will be presented: SAT-based resubstitution and ODCs. Both are combinational synthesis methods that can be extended into either direct or indirect sequential synthesis al-

Direct Sequential Synthesis	Indirect Sequential Synthesis
Description: A direct sequential synthesis method incorporates state space analysis into the synthesis algorithm.	**Description:** An indirect sequential synthesis method does not intrinsically understand the state space but instead does sequential synthesis by leveraging the sequential don't cares from a set of invariants.
Examples: Performing sequential SAT-sweeping by embedding inductive reasoning into the synthesis algorithm [van Eijk, 2000].	**Examples:** 1) Prove a number of invariants, and 2) do combinational SAT-sweeping. Only look for counterexamples that disprove equality of node pairs on the set of states that satisfy the invariants.
Synthesis Power: Able to reason about complex state space properties necessary to perform synthesis optimizations. As a result, these methods can dramatically simplify the design.	**Synthesis Power:** Invariants express general facts about the shape of the state space but may not be exactly what is required by the synthesis. As a result, the quality of the synthesis is slightly worse.
Runtime: State space analysis is expensive, and direct sequential synthesis methods can be quite slow as a result.	**Runtime:** Because indirect sequential synthesis methods are inherently combinational, they often run quickly. However, the time for invariant computation must be included and this may be a significant contribution to the total runtime.
Other Advantages: N/A	**Other Advantages:** If many indirect sequential synthesis methods are invoked then the invariants need only be computed once and then used in all synthesis steps. The runtime cost of invariant computation can be amortized, and the total runtime for a sequence of indirect methods can be much less than the total runtime for a similar sequence of direct methods.

Table 3.1: Comparing direct and indirect sequential synthesis.

gorithms. Examining these two algorithms and their sequential extensions will give insight into issues related to using invariants in synthesis.

3.3.1 Introduction to SAT-Based Resubstitution

A **dependent register** is one which may be expressed as a function over other registers in the design. Once identified, the overall register count of the design may be reduced by replacing the dependent registers by the corresponding equivalent functions. Such a reduction in registers is beneficial in a logic synthesis context where each register has a finite cost in terms of circuit area and power.

Additionally, elimination of dependent registers is useful in a verification context because many verification algorithms are sensitive to the number of registers present in the design. For example, BDD-based reachability analysis generally requires exponential resources with respect to register count, hence it may dramatically benefit from the elimination of dependent registers [Jiang and Brayton, 2004]. The effectiveness of induction is also generally sensitive to the implementation of the logic, particularly in the presence of unreachable states [Wedler *et al.*, 2004]. Dependent register elimination inherently reduces the fraction of unreachable states of a design, thereby enhancing inductiveness.

Dependency is traditionally identified using BDD-based algorithms (e.g., [Jiang and Brayton, 2004]), which practically limits its application to smaller designs or requires approximate compositional analysis, resulting in suboptimalities. Recently, it has been demonstrated that *combinational dependency* of next-state functions may be identified using purely SAT-based analysis [Lee *et al.*, 2007], enabling the analysis to scale to much larger designs. However, the previous research lacks several elements that make this method effective in practice. Here we present several improvements to the proposed algorithm and extend it to sequential synthesis using both direct and

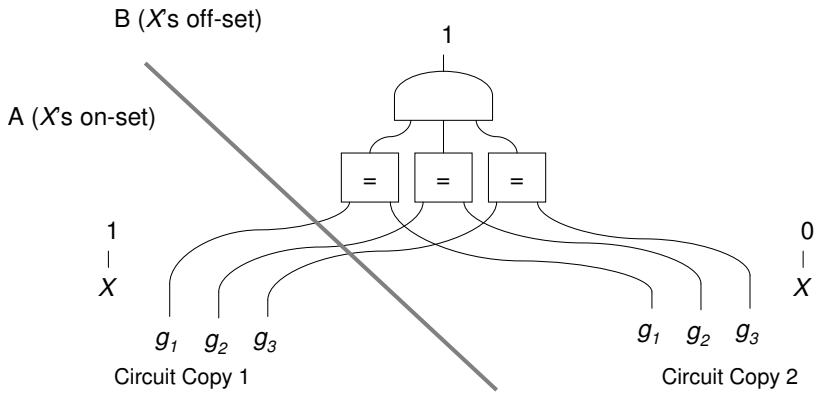


Figure 3.1: SAT-based dependency formulation

indirect methods.

Identification of dependent registers is a special type of resubstitution. In logic synthesis, **resubstitution** refers to a process that recasts a Boolean expression as a function over other pre-existing Boolean expressions [Brayton *et al.*, 1987]. For example, suppose there are Boolean signals X, g_1, g_2, ..., g_n. Resubstitution may be used to build a function $F(\cdot)$ such that $X = F(g_1, g_2, \ldots, g_n)$, or to prove that no such function exists. The functions g_1, g_2, ..., g_n are referred to as the **basis** and $F(\cdot)$ as the **dependency function**. Upon finding F, the old implementation of X can be removed and replaced with the new implementation of F. Often resubstitution yields a reduction in the size of the AIG, and for this reason has been the focus of much synthesis research.

Recently resubstitution has been posed as a SAT problem [Lee *et al.*, 2007] which improves on the scalability of previous approaches. Suppose that we wish to express X as a function over signals $g_1, \ldots, g_3$. This is possible if and only if for each valuation to signals $g_1, \ldots, g_3$ there is a single possible X valuation. We can test if such a resubstitution exists using the circuit shown in Figure 3.1. Two separate copies of X, g_1, g_2, and g_3 are instantiated. Each pair of g's is constrained to have the same value, and the pair of X's is constrained to have differing values. A resubstitution exists if

and only if this test circuit is unsatisfiable. Many SAT solvers can be configured to record a proof of unsatisfiability [Zhang and Malik, 2003], and interpolation on this proof provides the dependency function.

Given Boolean formulas $A(x, y)$ and $B(y, z)$, if $A(x, y) \cdot B(y, z) = 0$ then there exists an **interpolant** [Craig, 1957] I such that I refers to only the common variables y of A and B, and $A \implies I \implies \overline{B}$. [Pudlák, 1997] provides an algorithm to extract the interpolant I from the proof of unsatisfiability of $A \cdot B$. This technique was first introduced to the verification community in a SAT-based unbounded verification algorithm [McMillan, 2003] where the interpolant represents an overapproximate image computation.

In the resubstitution context, interpolation will be used to derive a dependency function. We may partition the resubstitution test circuit in Figure 3.1 into two halves A and B. A represents the set of g's where $X = 1$, the on-set of X. Similarly, B represents the off-set of X. Because $A \implies I \implies \overline{B}$, I is a function that lies in between the on and off-sets of X, and we can replace X with I. Furthermore, I only refers to the common variables between A and B, namely the g's, hence I provides the dependency function.

While this SAT-based formulation of [Lee *et al.*, 2007] enables substantially greater scalability than BDD-based techniques, this formulation is limited in four key ways:

1. The prior research can only re-express combinational logic and cannot directly eliminate registers. Minimizing the number of registers is important in both synthesis and verification contexts.

2. The prior research does not address incompatibilities present in the set of found dependencies. Often, dependencies must be discarded to avoid creating combinational cycles in the logic. We discuss an efficient way to compute a compatible set of dependencies in Section 3.3.2.2.

3. The prior research does not address the logic bloat that may result from interpolation. In general, logic generated by interpolation is highly redundant.

4. The prior research does not address ways to choose the basis set over which the resubstitution is to be attempted. Careful selection of the basis set is important for scalability.

5. The prior research cannot identify dependencies which hold in all reachable states but not in arbitrary unreachable states. In our experience, the reduction potential of this combinational analysis is often a subset of that possible using min-register retiming with integrated resynthesis [Baumgartner and Kuehlmann, 2001]. We use invariants as well as direct sequential synthesis to overcome this limitation.

3.3.2 Improvements to SAT-Based Resubstitution

In this section, we discuss our enhanced resubstitution procedure (function `eliminateRegisters` in Algorithm 15). Our resubstitution setup is illustrated in Figure 3.2, which is similar to Figure 3.1 but modified in several ways. This section will discuss how we target the formulation to find dependent registers as well as enhancements that make SAT-based resubstitution effective in practice. We iteratively call this procedure for every next-state function in the design in order to find the set of all dependent registers. This discussion is based on the publication [Case *et al.*, 2008c].

3.3.2.1 Register Elimination

If the resubstitution formulation illustrated in Figure 3.2 is unsatisfiable, then a dependency function will be obtained that may be used as a replacement for $next(S)$.

```
 1: function eliminateRegisters(design, invariants)
 2:    depends := ∅
 3:    for all (registers R in design) do
 4:       test := buildResubTest(next(R), other next-states)
 5:       if (satSolve(test) == unsat) then
 6:          proof := getResolutionProof()
 7:          curr := getDependencyFunc(R, proof)
 8:          notCurr := getDependencyFunc(¬R, proof)
 9:          depends += pickBest(curr, notCurr)
10:       end if
11:    end for
12:    depends := makeCompatible(design, depends)
13:    return simplifyDesign(design, depends)
14: end function
```

Algorithm 15: Improved SAT-based resubstitution.

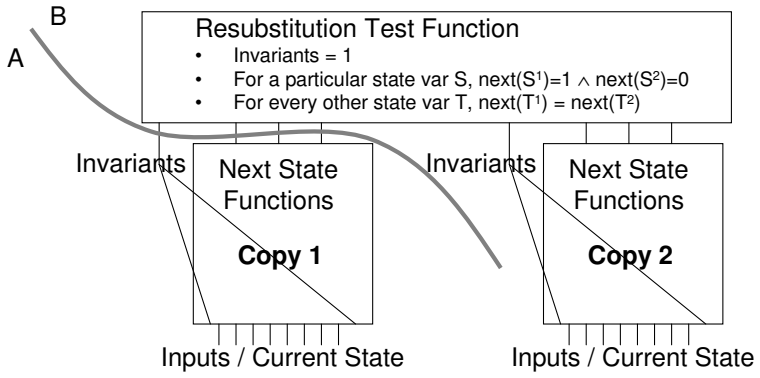


Figure 3.2: Our enhanced resubstitution framework.

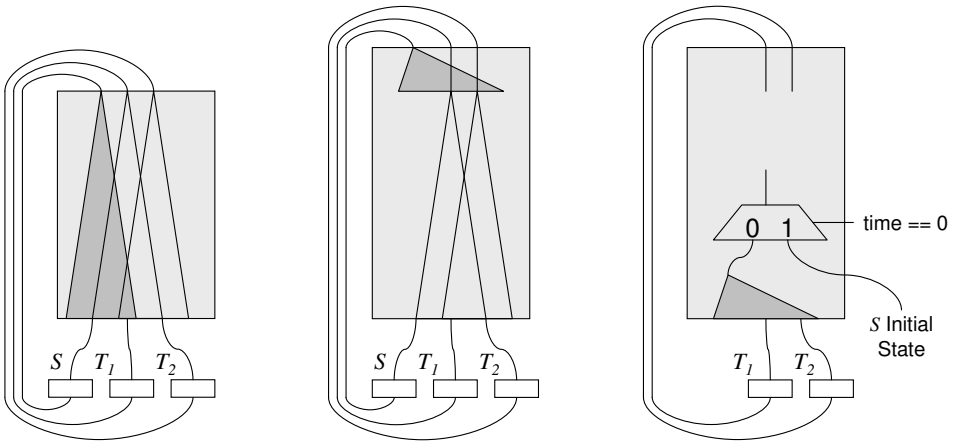


Figure 3.3: Register elimination process

Trading the existing implementation of $next(S)$ with the dependency function may yield a savings in the number of ANDs in the AIG. We specifically want to reduce the number of registers in a design because this is almost always beneficial to both synthesis and verification applications, and we are willing to tolerate a modest increase in ANDs to achieve this goal. Here we present a formulation by which a dependency function can be used to directly eliminate a register in the design.

Consider Figure 3.3a where the logic needed to implement $next(S)$ is highlighted. If a dependency function exists, it will express $next(S)$ as a function of the other two next-state signals $next(T_1)$ and $next(T_2)$. The implementation of $next(S)$ may be replaced with this dependency function, as illustrated in Figure 3.3b. We can further simplify the design by expressing this dependency function over the current states instead of the next-states, thereby eliminating register S.

Define an **orphan state** to be any state σ_1 for which there does not exist a state σ_2 such that σ_1 is reachable from σ_2 in one transition. Note that every reachable state, with the possible exception of the design's initial state(s), is not an orphan state.

Theorem 3.3.1. *For registers S, T_1, ..., T_n, if there exists an $F(\cdot)$ such that $next(S) = F(next(T_1), \ldots, next(T_n))$ then for every state that is not an orphan state $S = F(T_1, \ldots, T_n)$.*

Proof. Let σ_1 be a state of the design and a concrete valuation of the registers S, T_1, ..., T_n. If σ_1 is not an orphan state then there exists a state σ_2 such that σ_1 can be reached in one transition from σ_2. Let $X(\sigma_j)$ denote the valuation of register X in state σ_j and note that there exists inputs such that $next(X(\sigma_2)) = X(\sigma_1)$. From the hypothesis we have $next(S(\sigma_2)) = F(next(T_1(\sigma_2)), \ldots, next(T_n(\sigma_2)))$. Rewriting we see that $S(\sigma_1) = F(T_1(\sigma_1), \ldots, T_n(\sigma_1))$. □

Theorem 3.3.1 allows for the dependency function computed over next-state functions to be expressed over current states, provided the initial state(s) are accounted for. The result of this process is illustrated in Figure 3.3c. The dependency function between Figure 3.3b and 3.3c is identical; only the logic driving its inputs has changed. The register S has been completely eliminated at the cost of initial state correction logic. To correct the initial state, we introduce a multiplexor which at time 0 will drive the initial value of the register being eliminated, and thereafter will drive the dependency function for the next-state function of the eliminated register. To enable selection of these two values, a register may need to be introduced to the design which initializes to 1 and thereafter drives 0. This register is reused across all resubstitutions.

While Theorem 3.3.1 is not guaranteed to hold in the initial state, in some cases the initial value may be preserved by that dependency function. That is, the value produced by the dependency function at time 0 may be identical to the initial value of the register being removed, and the initial state correction logic can be omitted. On a benchmark suite for which 3,390 resubstitutions were performed, the initial state had to be corrected 67% of the time (Table 3.2). Our designs had complex

Design	Resubs.	Needed Correction
Set1 / IBM01	522	348
Set1 / IBM02	465	310
Set1 / IBM03	663	442
Set1 / IBM04	1200	800
Set1 / IBM05	51	34
Set1 / IBM06	276	184
Set1 / IBM07	213	142
	3390	2260 (67%)

Table 3.2: Necessity of initial state correction.

initialization functions due to retiming [Mony *et al.*, 2004] whose value the dependency function could replicate with relatively low probability. This illustrates the power of our technique to enhance register reduction capability particularly in the presence of complex initial states[1].

3.3.2.2 Compatible Dependencies

The `eliminateRegisters` function from Algorithm 15 will attempt to resubstitute each next-state function present in the design. Through this process, a large number of dependency functions may be identified that can replace existing registers as depicted in Figure 3.3. Unfortunately, the set of dependencies found in this manner are generally not **compatible**, and if multiple dependency simplifications are performed simultaneously then often a **combinational cycle** will be created in the AIG resulting in an illegal design. A compatible set of dependencies is one in which all dependencies can be applied simultaneously with no resultant combinational cycles. Therefore, once the dependencies have been identified, one must identify a subset of compatible dependencies contained therein, and this chosen subset may impact the

[1]This synthesis technique can dramatically simplify verification efforts because a reduction in the number of registers corresponds to a reduction in the state space that must be searched. See Section 1.5.1.

size of the resulting design.

```
 1: function makeCompatible(design, dependencies)
 2:    scored := ∅, compatible := ∅
 3:    for all (Dep. D in dependencies) do
 4:       red = D.redundant_AIG_node
 5:       repl = D.replacement_AIG_node
 6:       gain = aigSize(design) - aigSize(design - red + repl)
 7:       scored[D] = scoreFunction(gain.regs, gain.ANDs)
 8:    end for
 9:    sortDescending(scores)
10:    for all (Dep. D in scored) do
11:       red = D.redundant_AIG_node
12:       repl = D.replacement_AIG_node
13:       if (!isCyclic(repl, red, compatible)) then
14:             compatible += D
15:       end if
16:    end for
17:    return compatible
18: end function
```

Algorithm 16: Selecting a set of compatible dependencies

Example 19. Resubstitution for one-hot encoded registers

To illustrate the notion of incompatible resubstitution, consider a design with registers $R_1, \ldots R_n$ which have a one-hot encoding where in every reachable exactly one of these n registers will evaluate to 1. Given adequate invariants to characterize this one-hot condition, the following dependencies may be identified:

$$R_1 = \overline{R_2} \wedge \overline{R_3} \wedge \overline{R_4} \wedge \cdots, \quad R_2 = \overline{R_1} \wedge \overline{R_3} \wedge \overline{R_4} \wedge \cdots$$

It is not possible to express R_1 as a function of R_2 and simultaneously express R_2 as a function of R_1 without creating a combinational cycle.

Finding a compatible subset of dependencies is a computationally difficult task. Finding an optimal subset might entail enumerating and testing every possible subset, and this is feasible for only very small sets of dependencies. Instead, we utilize a heuristic to quickly find a near-optimal subset of compatible dependencies.

After the complete set of dependencies is found, we reduce this to a set of compatible dependencies as illustrated in Algorithm 16. Each found dependency consists of two signals: a **redundant** signal that will be eliminated and a **replacement** signal that will be introduced in its place. We first sort the dependencies in the order of their ability to simplify the circuit, computed as a function `scoreFunction`[2]. The list of sorted dependencies is then iterated over, and a subset of compatible dependencies is greedily found. For each dependency, we test if performing this optimization in the presence of the other compatible dependencies will introduce a combinational cycle using `isCyclic`. If so, the candidate merge is discarded. Otherwise the merge is added to the compatible set.

While this search is greedy, prioritizing the dependencies by score enables the algorithm to capture most of the optimization potential present in the original set of dependencies. This is illustrated on a set of industrial designs in Table 3.3. For each design, the found dependencies and compatible subset of these found dependencies are examined. Usually only a small percentage (24%) of the total dependencies must be discarded to form a compatible subset. If the total gain is summed over all possible dependencies, we see that the sum gain from the compatible dependencies is similar. This indicates that most of the AIG optimization potential present in the full set of dependencies was captured by the compatible subset.

3.3.2.3 Reducing Dependency Function Interpolants

The dependency functions are obtained from the interpolant of a proof of unsatisfiability. Using the method given in [Pudlák, 1997], the resultant logic will have size that is linear in the size of the resolution proof. The proof of unsatisfiability for com-

[2]Through trial and error, we have found that the function $20 \cdot gain.\text{regs} + gain.\text{ANDs}$ works well. This captures the fact that we're willing to tolerate a bloat of 20 AND gates for each register that is removed.

	Total Deps.		Compatible Deps.	
Design	Count	Sum Score	Count	Sum Score
IBM01	376	-1714	325	-91
IBM02	194	-659	154	-725
IBM03	428	-18278	374	-16950
IBM04	678	643	579	909
IBM05	35	312	22	258
IBM06	102	573	58	308
IBM07	142	139	102	-97

Table 3.3: Compatible dependencies on a set of IBM benchmarks. The number of functional dependencies between registers is shown along with the sum of the scores that is used to approximate the synthesis value of these dependencies.

plex SAT problems may be large, and this may result in an interpolant and resulting dependency function that are very large. Here we explore several ways to control the size of the obtained dependency functions.

The most basic way to control the size of the dependency functions is with combinational synthesis. Logic that comes from interpolants is usually highly redundant and amenable to combinational synthesis techniques [Cabodi *et al.*, 2006]. While combinational synthesis is effective in reducing the size of these interpolants, it is too slow to be used on every interpolant prior to reducing to a compatible subset in Section 3.3.2.2. Here we focus on ways to more directly optimize our dependency functions before combinational synthesis is applied.

One simple way to control the size of the interpolants before synthesis is applied is to use incremental SAT. Our implementation attempts to resubstitute each next-state function in the sequential AIG, and through this process many similar SAT problems are encountered. Using one incremental solver instance to solve all of these problems is advantageous for two reasons:[3]

[3]Incremental SAT is generally preferred, but such solvers can store a large number of learned clauses. If memory is a concern, the solver instance may need to be periodically refreshed.

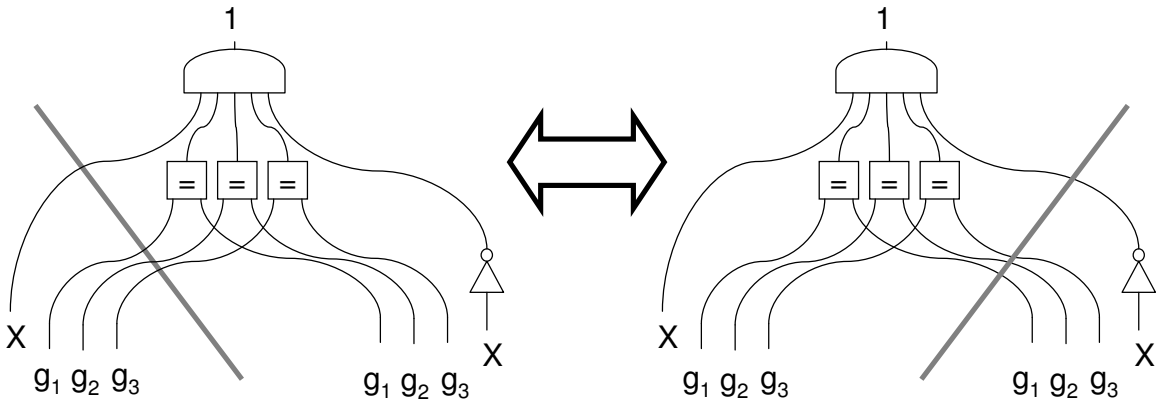


Figure 3.4: There is flexibility in how to partition the problem for interpolation.

1. One incremental solver typically learns fewer clauses than many non-incremental solvers. The size of an interpolant is related to the number of learned clauses, and using incremental SAT will result in a reduction in the total size of all interpolants.

2. In incremental SAT the learned clauses from one problem are preserved and may contribute toward the search for a satisfiable solution to a future problem. If the same learned clause participates in two proofs of unsatisfiability then the two interpolants will share common logic. This also reduces the total cost of the logic needed to implement all interpolants.

In addition to using incremental SAT, we propose a more intelligent approach to mitigate logic bloat. Consider the resubstitution framework shown in Figure 3.2 that is able to eliminate a register S by resubstituting $next(S)$. Copy 1 of the transition relation represents the on-set of $next(S)$, and Copy 2 represents the off-set. Using the partitioning that separates Copy 1 from the rest of the circuit as shown, we get an interpolant I such that $next(S) \implies I \implies \overline{next(S)}$, and the resulting dependency function is able to replace S using the concepts of Section 3.3.2.1. However, the

Table 3.4: Dependency function use on a set of IBM benchmarks

Design	Depend. Count	Avg. Score by Repl. Type S	$\overline{S}$
IBM01	1128	0.70	-0.87
IBM02	582	0.78	0.04
IBM03	1284	-6.56	-20.20
IBM04	2034	1.71	2.11
IBM05	104	3.10	6.06
IBM06	306	2.42	4.44
IBM07	426	1.44	2.21

problem is symmetric and we could alternatively partition to separate Copy 2 from the rest of the circuit. This alternative partitioning scheme is illustrated in Figure 3.4. In this case, $\overline{next(S)} \implies I \implies next(S)$, and the dependency function is be able to replace $\overline{S}$.

Thus, we have the flexibility to compute the interpolant in two different ways to either replace S or $\overline{S}$. These two replacements affect the size of the modified AIG in different ways, and to quantify this each possible replacement is scored in a manner identical to Section 3.3.2.2. By selecting the highest-scoring replacement, the dependency function can be used to its best advantage.

This is examined on a suite of industrial benchmarks in Table 3.4. For each design, the number of found dependencies is given. Each dependency is scored as if it were used to replace one of the two signals: S or $\overline{S}$, and the average score for each replacement type is given. A negative score indicates that the number of ANDs introduced to build the dependency function was greater than the cost of the logic being removed. Note that the signal we would prefer to replace is benchmark-dependent. In general, it is also dependency-specific, and our implementation individually scores each dependency in two ways in order to best utilize each dependency function.

3.3.2.4 Automated Discovery of the Basis Set

The previous work in SAT-based resubstitution assumed a basis set to be given. In attempting to resubstitute every register and express it in terms of the other registers, it is possible to implement a naive basis set that can be used as this given basis set. Every next state function other than the next state function for the register being resubstituted can be grouped into a basis set, and the resubstitution can be performed over this set.

While this approach works well for small circuits, it has difficulty scaling to designs with thousands of registers. The size of the SAT problem is very sensitive to the number of elements in the basis set, and so in a circuit with many registers the basis set is large which leads to very poor run time.

It is helpful to observe that when a register can be resubstituted, the dependency function typically only touches a few of the design's other registers. This means that the resubstitution can be performed using a very small basis set. Unfortunately, this small basis set will be different for each register that is resubstituted, but the basis set can be derived in an efficient manner such that rederivation for each resubstitution attempt is practical.

Consider the basis set discovery routine in Algorithm 17. This algorithm starts with an empty basis set and grows the basis set as needed until the resubstitution is possible. In this way the algorithm uses a **dynamic basis** that grows monotonically.

The core of the algorithm is a call to the function `resubstitute`() with the current basis set. This function sets up the SAT problem to test if a resubstitution is possible with the current basis set. If this SAT problem is satisfiable then resubstitution is not possible with the current basis set. A satisfying counterexample is returned that can be used to either grow the basis set or conclude that resubstitution is impossible. If the SAT problem is unsatisfiable then the resubstitution is possible with the current

```
 1: function resubWithDynamicBasis(design, X)
 2:   basisSet := ∅
 3:   while (1) do
 4:     if (resubstitute(X, basisSet) = SAT then
 5:       if (∃ new basis element B that can block the counterexample) then
 6:         basisSet += B
 7:       else
 8:         return "resub failed"
 9:       end if
10:     else
11:       // SAT problem was unsatisfiable. Resubstitution is possible.
12:       Interpolate and simplify the circuit.
13:       return "resub succeeded"
14:     end if
15:   end while
16: end function
```

Algorithm 17: Resubstitution with a dynamic basis set.

basis set, and we proceed to interpolate and simplify the circuit.

Example 20. Expanding a basis set.

The method by which the basis set is expanded in Algorithm 17 is critically important. Here we will explain this method by example.

Consider the resubstitution test function shown in Figure 3.5. We attempt to resubstitute to eliminate register S*, and in this iteration the basis set is empty. Two copies of the transition relation are instantiated, and we constrain the SAT problem such that the pairwise copies of* next(S) *differ. Additionally, if there were basis elements we would constrain their pairwise copies to be equivalent.*

Suppose our satisfiability solver finds a counterexample. This means that with the current basis set it is possible for the onset of S *to intersect the offset of* S*, and a counterexample is returned that demonstrates this overlap. The counterexample is a set of assignments to the circuit inputs, and this assignment is possibly incomplete. Let* X *denote an input that has no assigned value.* ***Ternary***

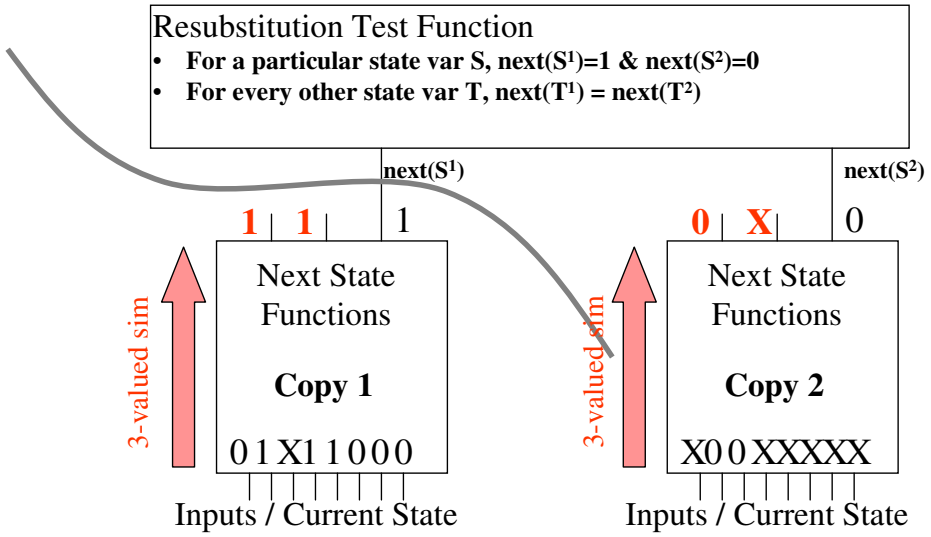


Figure 3.5: Automated expansion of the basis set.

simulation *(sometimes called* ***3-valued*** *simulation) can be used to propagate this input assignment throughout the circuit to get an assigned logic value at every circuit node.*

Suppose under the simulated assignments, a pair of next state functions takes opposing values, as happens in the leftmost next state function in Figure 3.5. If the corresponding register is added to the basis set then the constraint that the next state functions assume equal logic values will be sufficient to block this counterexample. Therefore this register is selected as the next register to add to the basis set.

In this way, registers can be added to the basis set until all counterexamples are blocked and the problem is unsatisfiable, denoting a valid resubstitution. If a counterexample is found such that no additional registers are able to block it then the resubstitution is ultimately not possible without expanding the basis set beyond just the next state functions.

The dynamic basis allows a large and complex SAT problem to be decomposed

into a series of smaller and simpler SAT problems. When combined with incremental SAT solving, dramatic speedups in the run time of the total routine can be obtained.

3.3.3 Extension to Indirect Sequential Synthesis

The SAT-based resubstitution method presented in [Lee *et al.*, 2007] is a purely combinational technique in that it cannot change the state space. The improvements presented above, specifically Section 3.3.2.1, make the method into a sequential synthesis algorithm because they change the number of registers in the design. However, the power to change the state space can be enhanced even more by using indirect sequential synthesis. Here we will consider the use of invariants as a source of sequential don't cares to enhance resubstitution.

Suppose that prior to resubstitution a number of invariants have been discovered and proved in the sequential logic network. These invariants provide an over-approximation to the set of reachable states, and hence an under-approximation to the unreachable states. This under-approximation gives a set of sequential don't cares on which the design's behavior can be changed without affecting the correctness of the design.

Suppose a functional dependency exists between the registers in a design and one register can be expressed as a function of the others. It is possible that the resubstitution test function shown in Figure 3.2 is satisfiable but that each satisfying counterexample requires the two circuit copies to be driven by unreachable states. The satisfiability demonstrates that the resubstitution is impossible, but because all counterexamples are unreachable the resubstitution can be performed without impacting the behavior of the design while in normal operation. We need a way to ignore these unreachable counterexamples, and the sequential don't cares provide this ability.

	Preprocessed Design			Algorithm 15			Algorithm 15 + Invariants			
Design	Inp.	ANDs	Regs	ANDs	Regs	Time	Invars.	ANDs	Regs	Time
Set1 / IBM01	36	2083	393	1997	223	15.33	190	1995	224	91.53
Set1 / IBM02	26	906	223	931	144	9.63	389	911	146	82.05
Set1 / IBM03	44	5977	625	9602	429	69.90	166	9520	439	197.52
Set1 / IBM04	34	3536	704	3600	415	52.87	58	3588	414	128.94
Set1 / IBM05	189	19989	743	13828	733	77.24	514	14611	728	370.94
Set1 / IBM06	40	4373	698	4330	669	32.44	17	4321	669	104.35
Set1 / IBM07	44	1124	241	1082	189	10.79	288	1030	190	92.89
Summary[2]				1.04	0.75			1.03	0.75	
Set2 / IBM08	8	502	101	499	101	6.22	12	499	101	73.17
Set2 / IBM09	16	3258	683	3238	678	48.81	10	3232	677	175.71
Set2 / IBM10	89	4463	823	4471	814	55.29	29	3164	690	141.08
Set2 / IBM11	26	2402	530	2433	523	37.10	522	2386	521	214.00
Set2 / IBM12	114	770	199	768	197	7.69	431	754	195	88.15
Set2 / IBM13	189	5111	193	4878	167	11.84	160	4882	166	100.60
Summary[2]				0.99	0.97			0.94	0.94	
Set3 / IBM14	38	2773	470	2773	470	10.76	123	2735	461	100.83
Set3 / IBM15	125	15796	668	11972	655	113.74	399	13113	655	256.80
Set3 / IBM16	68	2757	680	2743	675	36.55	16	2740	675	98.53
Set3 / IBM17	127	15867	654	11882	640	99.87	464	13017	640	521.74
Set3 / IBM18	125	15675	668	11977	658	127.87	306	13165	658	336.07
Set3 / IBM19	84	5958	776	6219	762	93.52	146	5560	689	154.88
Summary[2]				0.88	0.99			0.90	0.97	

[1] All experiments were run on a 1.83 GHz laptop running Linux 2.6.
[2] Ratios are relative to the preprocessed AIG size.

Table 3.5: Resubstitution performance on three sets of IBM benchmarks[1], with and without invariants

A set of invariants can be leveraged by considering the invariant logic shown in Figure 3.2. Each circuit copy has an independent logic cone that asserts that all invariants hold in that transition relation. The resubstitution test function is then modified to enforce that all invariants in the two copies hold. This will block unreachable counterexamples. It is important to note that invariants give an under-approximation to the unreachable states and hence cannot block all unreachable counterexamples. However, the counterexamples they are able to block are helpful in enhancing the power of sequential synthesis.

Table 3.5 shows the performance of our enhanced SAT-based resubstitution on

three sets of IBM benchmarks. Each design was heavily preprocessed using combinational and sequential synthesis techniques. These preprocessing steps are able to simplify the logic dramatically, and in our experience the original published method of [Lee *et al.*, 2007] is rarely able to improve on these designs further.

Because of the numerous enhancements discussed in Section 3.3.2, resubstitution is able to reduce the size of the preprocessed designs. Note that the first set of benchmarks is quite amenable to resubstitution while the second two sets have several designs that cannot be optimized with the weak sequential synthesis provided by our enhanced resubstitution algorithm.

Next, we reverted to the preprocessed designs and tried resubstitution again. This time invariants were found and leveraged in the resubstitution effort. There are many designs for which invariants were vital in allowing resubstitution to reduce the number of registers in the design. This shows that reducing the occurrence of spurious counterexamples with sequential don't cares is important.

3.3.4 Extension to Direct Sequential Synthesis

Section 3.3.3 above showed that resubstitution can benefit from sequential don't cares because this information enables resubstitutions to be considered that may be invalid on unreachable states. It is interesting to consider the type of state space information that is helpful to resubstitution. Here we explore direct sequential synthesis techniques applied to resubstitution by modifying the resubstitution algorithm to incorporate direct reasoning about the state space. This direct synthesis technique is formed by incorporating induction (Section 14) within the inner loop of the resubstitution routine.

For a basis set $g_1, \ldots, g_n$ and a target signal X, it is possible to express X in terms of the basis signals if and only if for any two logical assignments to all nodes in the

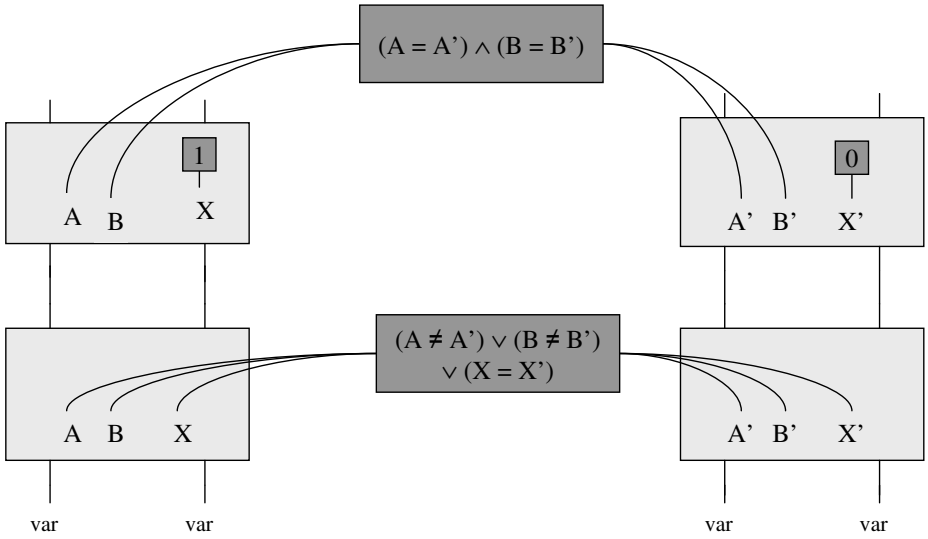


Figure 3.6: Inductive step of resubstitution by induction.

design (where the assignments to signal α are denoted α and α' respectively):

$$\big((g_1 = g_1') \wedge \cdots \wedge (g_n = g_n')\big) \implies (X = X') \tag{3.1}$$

Figure 3.1 checks Equation 3.1 by considering two independent circuit copies that can each take independent logical values. The formula is checked with additional logic that bridges the two circuit copies. This is sufficient to check that Equation 3.1 holds combinationally in every possible state, but more sophistication is needed to check that Equation 3.1 holds on the reachable states. We will achieve this by using induction to check that Equation 3.1 holds.

Figure 3.6 shows the inductive step check for the inductive check of Equation 3.1. Consider two basis elements A and B and let X be the resubstitution target. We unroll the circuit of Figure 3.1 in order to represent two adjacent time steps. The lower time step is driven by a symbolic state, and we constrain that Equation 3.1 holds in this time frame. The upper time step takes as input the next state produced by the lower time step, and we check to see if it is possible to violate Equation 3.1 in

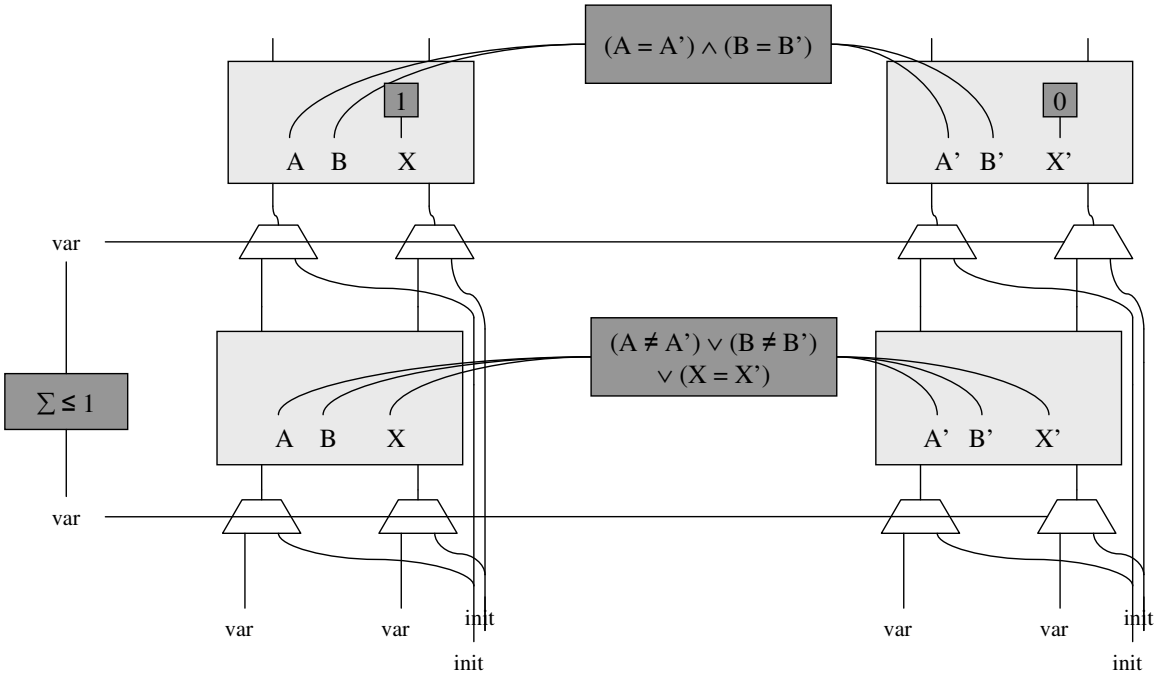


Figure 3.7: Simultaneous base case and inductive step of resubstitution by induction.

this time step. If it is not possible to violate Equation 3.1 then we have successfully checked the inductive step of the $k = 1$ inductive proof.

Unfortunately, doing a full inductive check is more complicated than just checking the inductive step. The base case must also be checked. Typically, the base case and inductive step are checked using two separate SAT problems. However, because resubstitution involves using the interpolant to derive the dependency function we would like a single proof of unsatisfiablity from which we can extract the interpolant. Therefore it is necessary to combine the base case and inductive step into one single SAT problem.

Figure 3.7 shows the combined base case and inductive step for our example basis $\{A, B\}$ and our target signal X. We introduce two nondeterministic variables, one for each time frame. If the nondeterministic variable evaluates to 1 then we drive

the time frame with the initial state instead of the state that would have been driven in Figure 3.6. This means that depending on the valuations of the nondeterministic variables this SAT problem checks the following three things:

1. If all nondeterministic variables are 0 then the circuit reduces to the inductive step check.

2. If the nondeterministic variable feeding the upper frame is 1 then the circuit is checking Equation 3.1 on all initial states.

3. If the variable for the upper frame is 0 and the variable for the lower frame is 1 then the circuit is checking Equation 3.1 on all states reachable in 1 step from an initial state.

Two minor observations here are that 1) this checks more than is required for the base case, and 2) there is symmetry in the nondeterministic variable assignments. The base case for a $k = 1$ proof requires that we check that Equation 3.1 holds in all initial states, but we actually check more than this. This may hurt runtime but does not affect the results since anything that holds on all reachable states will hold on the states that we check as well. The second observation is regarding the symmetry in the problem formulation. Note that if both nondeterministic variables are equal to 1 then what is checked is equivalent to the case that the upper variable is 1 and the lower is 0. We break this symmetry by adding the constraint that the sum of the nondeterministic variables is ≤ 1.

The SAT problem depicted in Figure 3.7 allows us to check Equation 3.1 inductively. When combined with a dynamic basis (Section 3.3.2.4) it is possible to detect the registers that can be expressed in terms of other registers. Each register that can be reexpressed can be removed from the design and gives a net reduction in the number of design registers. Table 3.6 examines the number of registers that can be

Implementation	Resubstitutions Found	Runtime
Section 3.3.2 (no invariants)	158	64 sec
With $k = 1$ induction	169	50 sec
With $k = 2$ induction	195	242 sec
With $k = 3$ induction	238	360 sec

Table 3.6: Direct sequential synthesis on Set1 / IBM06

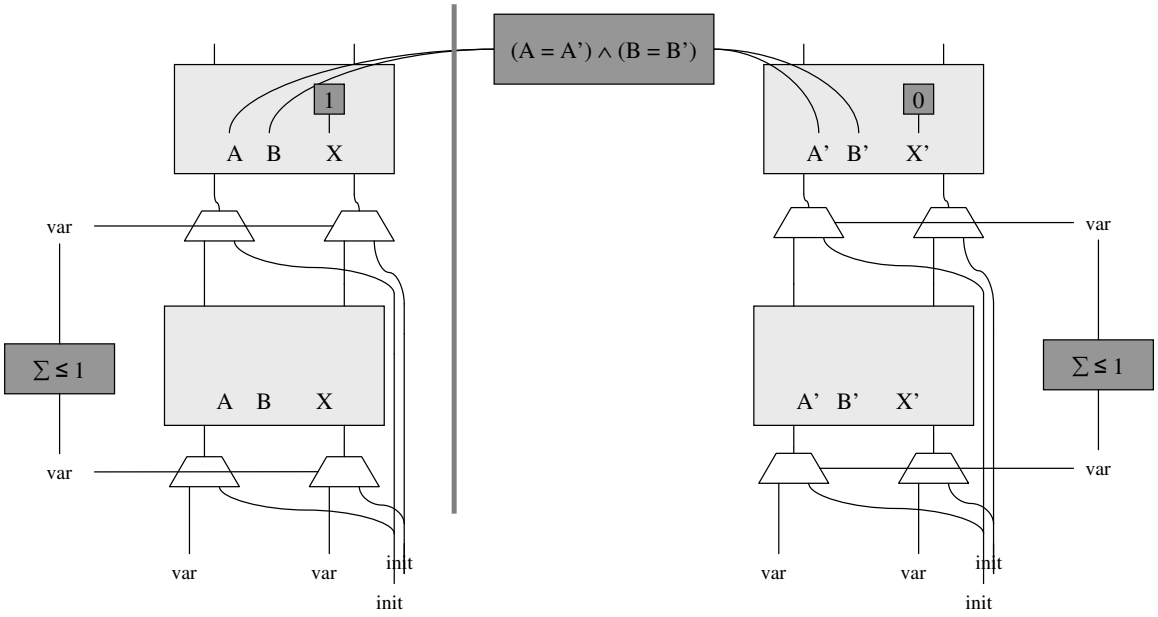


Figure 3.8: Simplified version of Figure 3.7 that is suitable for interpolation.

removed as a function of the k used in the k-step induction. It is interesting to note that on the examined benchmark, the number of resubstitutions found appears to be quadratic in k. Clearly, the runtime also increases as k increases and the SAT problem becomes more complicated, but these results indicate that the proof effort can dramatically affect the synthesis results.

Table 3.6 shows the registers for which a basis was found such that Equation 3.1 holds in all reachable states. These registers are redundant and can be expressed in terms of their basis sets. Unfortunately, to actually remove the redundant registers

interpolation is needed, and similar to Section 3.3.1, the problem must be partitioned into two parts before it can be interpolated. Each signal that is shared between the partitions will appear in the interpolant.

There is no clear way to partition the problem in Figure 3.7. In each possible partitioning there exists many signals that are shared between the two parts, and there is no partitioning such that the resulting interpolant can be interpreted as the dependency function. In order to get a clean partitioning, we simplify the problem as shown in Figure 3.8. The constraint on the lower frame of the inductive check was dropped, and the nondeterministic variables were replicated. This simplification weakens induction and means that many valid resubstitutions that previously could be proved will now be falsified by spurious counterexamples. Note that the resubstitutions found in this manner are still sequential in that the resubstitutions may not be valid on unreachable states. The simplification allows the problem to be cleanly partitioned as in Section 3.3.1 where one partition represents the onset of X and the other the offset of X. This means that the resulting interpolant can be interpreted as the dependency function, and resubstitution can proceed as before.

Table 3.7 examines the performance of three versions of the resubstitution algorithm:

1. The enhanced combinational resubstitution from Section 3.3.2.
2. A direct sequential synthesis resubstitution using $k = 1$ induction.
3. Another direct sequential synthesis resubstitution using $k = 2$ induction.

Each of these three algorithms was examined in the following way:

1. A preprocessed design was optimized using the selected resubstitution algorithm. The number of registers removed was recorded.
2. Invariants were discovered and used as a source of sequential don't cares.

Design	Section 3.3.2			k=1 induction			k=2 induction		
	Δ Regs (no invar.)	Δ Regs (invar.)	Sec.	Δ Regs (no invar.)	Δ Regs (invar.)	Sec.	Δ Regs (no invar.)	Δ Regs (invar.)	Sec.
0	138	8	193.85	123	18	188.79	131	16	248.5
1	82	3	183.98	150	3	300.5	150	21	418.24
2	184	2	342.6	193	2	383.85	189	56	855.37
3	263	22	240.71	247	30	217.41	261	27	235.63
4	10	9	359.98	11	7	266.74	20	9	787.53
5	28	0	289.51	34	2	352.38	108	22	670.76
6	48	3	184.38	49	3	162.06	49	4	178.62
7	0	0	168.16	0	0	140.83	0	0	152.59
8	5		218.04	5	1	273.03	265	21	227.28
9	6	6	229.49	6	9	189.54	12	10	368.93
10	8	2	301.17	7	3	306.9	8	3	833.01
11	2	1	179.68	3	2	148.92	10	2	189.9
12	2	0	288.9	2	0	215.29	2	0	620.15
13	0	6	218.39	0	6	342.07	0	6	479.38
14	33	1	193.47	32	2	157.71	32	3	191.31
15	15	0	396.18	14	1	356.03	18	2	1024.35
16	1	2	257.73	2	1	201.7	14	1	1219.71
17	11	0	334.35	11	1	323.04	17	1	966.37
18	16	1	376.1	16	1	294.92	19	2	686.36
19	15	5	404.21	17	6	233.93	22	14	437.34
313	25	3	210.66	26	2	173.25	29	2	277.26

Table 3.7: Direct sequential synthesis with resubstitution. The relative change in design size increased by both 1) the inductive formulation, and 2) the use of invariants.

3. The design was optimized again, but this time the sequential don't cares were used. This allows us to quantify the benefits of indirect sequential synthesis in terms of the number of registers removed.

Looking only at the direct sequential synthesis results in Table 3.7, it is clear that using an inductive proof technique is much more effective in reducing the number of registers than is the combinational proof technique. In our implementation, $k = 1$ is currently the default proof technique, and the user can specify a larger k if he or she is willing to spend the runtime for a better synthesis result.

Note in these experiments, sequential don't cares are used in conjunction with an inductive proof. The result is a hybrid of direct and indirect sequential synthesis,

and the resultant proof leverages more information about the state space than either direct or indirect synthesis techniques alone.

3.3.5 Comparing Direct and Indirect Synthesis

Table 3.7 gives us an insight into the merits of direct and indirect sequential synthesis as applied to resubstitution. Indirect sequential synthesis is performed by using invariants as a source of sequential don't cares, and direct sequential synthesis is done by performing an inductive proof inside of the resubstitution algorithm itself.

Direct and indirect techniques benefit resubstitution in orthogonal ways. Direct sequential synthesis can often remove more registers than indirect sequential synthesis, indicating that the state space properties exploited by the inductive proof can be stronger than those discovered by invariants. Conversely, adding invariants to an inductive proof can increase the number of registers removed. This indicates that at times the invariants discover important state relationships not captured by the inductive formulation. In this way the direct and indirect techniques are orthogonal. Each can strengthen the synthesis in unique ways, and the best results are obtained by combining both direct and indirect sequential synthesis techniques.

Although the techniques are orthogonal, the direct sequential synthesis (inductive proof) seems to be more beneficial. If the combinational method of Section 3.3.2 is strengthened using either direct or indirect methods then the direct sequential synthesis will yield more register reductions than the indirect techniques. This is because induction seems to be more effective in uncovering the state relationships that are important for proving that a resubstitution is valid (Equation 3.1). This indicates that if both direct and indirect techniques cannot be performed in the allotted runtime then the direct techniques are preferred because they will better benefit resubstitution.

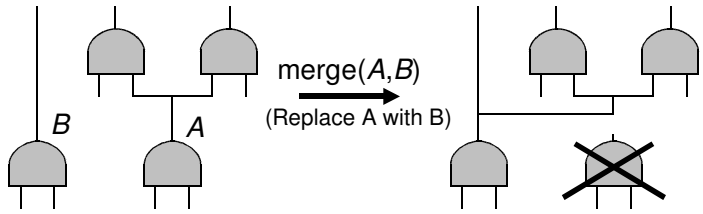


Figure 3.9: Merging signals A and B by rewiring and removing dangling nodes.

Although the direct sequential synthesis seems stronger, indirect methods have one distinct advantage: the results can be recycled. The invariants that are discovered (and the resultant sequential don't cares) can be used in many synthesis algorithms in addition to resubstitution. In this way, the runtime needed to derive these invariants can be amortized, and the invariants can effectively be very cheap to compute. In contrast, the runtime needed for an inductive proof is comparable for that needed to prove invariants, but the inductive proof runtime cannot be amortized. Thus a sequence of direct sequential synthesis algorithms is far slower than a single invariant discovery pass followed by a sequence of indirect sequential synthesis algorithms.

3.4 Case Study: Sequential ODCs

3.4.1 Merge-Based Algorithms

Consider **merging** signals as a basic synthesis operation. This operation, illustrated in Figure 3.9, involves rewiring such that the fanouts of a signal are driven by a different signal. This leads to circuit simplification as 1) one of the signals can be removed from the circuit and 2) the merge may cause downstream logic to appear redundant. This merge operation can be applied in multiple contexts to form the basis of many synthesis algorithms.

Synthesis algorithms based on merging signals have been extensively studied in

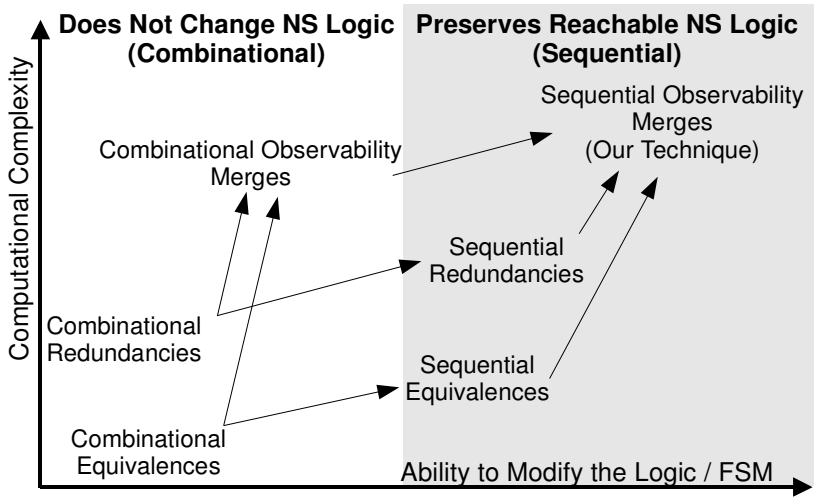


Figure 3.10: Taxonomy of merge-based optimizations. Arrows show that an algorithm generalizes another.

the past, as examined in Figure 3.10. Significant circuit optimizations can be realized using merge-based algorithms, but there is more potential for optimization with this simple yet powerful operation. All previous approaches are either limited in scope or do not take advantage of the sequential nature of the design.

SAT and BDD sweeping are two early algorithms based on merging signals [Kuehlmann and Krohm, 1997; Brand, 1993; Kunz, 1993; Kunz *et al.*, 1997; Goldberg *et al.*, 2001; Lu *et al.*, 2003]. These algorithms attempt to prove that pairs of signals are combinationally equivalent and then merge the pairs successfully proved. These algorithms are used extensively in modern synthesis and verification frameworks [Baumgartner, 2006; Synthesis and Group, 2008] because they scale well and provide significant reductions in the number of AIG nodes.

Generalizing on the previous algorithms, [van Eijk, 2000] finds sequentially equivalent signals by proving that pairs of signals are equivalent in every reachable state, a superset of the merges found by SAT and BDD sweeping. The success of this method depends on the design style, but in general the method is effective in reducing the

AIG node count.

Techniques based on **redundancies** represent another class of merge-based algorithms. These techniques prove that a signal, not necessarily constant, can be safely merged with a constant without producing a difference at the COs, improving the AIG node count by enabling constant propagation. Combinational redundancy methods are used also in industrial flows to enhance stuck-at-testability [Devadas *et al.*, 1994].

Recently there has been a renewed interest in **combinational observability** merges [Zhu *et al.*, 2006; Plaza *et al.*, 2007]. This class of algorithms finds ordered pairs of signals that may not be strictly equivalent yet will not produce a difference at the COs after merging. This takes advantage of logic reconvergence that masks out the signal differences as the values propagate toward the outputs. The merges found by this method are a superset of those found by SAT/BDD sweeping but are in general a different superset than sequentially equivalent signals. These merges are also a superset of those found by combinational redundancy methods.

3.4.2 Introduction to ODC-based Simplifications

A signal s is **observable** with respect to an output o if toggling s can cause o to toggle. Clearly, for each signal s there should be an input vector and output o such that s is observable with respect to o because otherwise s would be **redundant** and can be removed from the circuit without affecting the outputs. While fully redundant signals are rare, partially redundant signals abound. Often for a signal s there exist circuit conditions under which the value of s cannot affect any circuit outputs. These conditions are known as **Observability Don't Cares** (**ODCs**), and s can be modified freely under the ODCs without affecting the circuit outputs. For correct operation, only the circuit outputs need to be preserved, and a logic synthesis

algorithm is free to modify internal (non-output) signals. By modifying s on the ODCs, significant optimizations can be made in terms are area, delay, and power.

Classical ODC algorithms attempt to derive the ODC conditions at each node [Hassoun and Sasao, 2002]. They then resynthesize that node, and proceed to calculate ODCs and resynthesize the next node in the circuit. Optimizations have been made to derive **compatible ODCs** (**CODCs**) [Savoj and Brayton, 1990] such that each node's CODC set can be precomputed before any optimization is done. The resultant optimization is then guaranteed to not change the precomputed CODC sets.

Despite the impressive body of previous work, ODCs are not currently used for the following reasons:

- It is difficult to derive the ODCs in a scalable way for large circuits. These are traditionally derived with BDDs which do not scale to large and complex circuits.
- It is difficult to store the ODC conditions. Traditionally these are stored using BDDs that may not be memory efficient.
- It is difficult to resynthesize a portion of the logic network once the ODCs are known. Traditional don't care synthesis uses BDDs and therefore has difficulty scaling to large circuits.

Recently there have been two papers that present methods to perform more scalable ODC simplifications using a SAT-based approach [Zhu *et al.*, 2006; Plaza *et al.*, 2007]. These approaches are more limited than the historical work because they only allow node merges as the underlying synthesis operation, but they are far more scalable because they do not require the set of ODCs to be explicitly derived.

Algorithm 18 outlines the main idea of ODC-based merging for combinational logic networks[4]. Two signals A and B are selected, and the algorithm is called to

[4]The specific details of Algorithm 18 differ between papers.

```
 1: function mergeODC(A, B, design)
 2:    out := ouputs of design
 3:    cut_B := cut between B and out
 4:    for all (c in cut_B) do
 5:       if (replacing B with A changes c) then
 6:          return "merge impossible"
 7:       end if
 8:    end for
 9:    replace B with A
10:    return "merge succeeded"
11: end function
```

Algorithm 18: ODC-based merging.

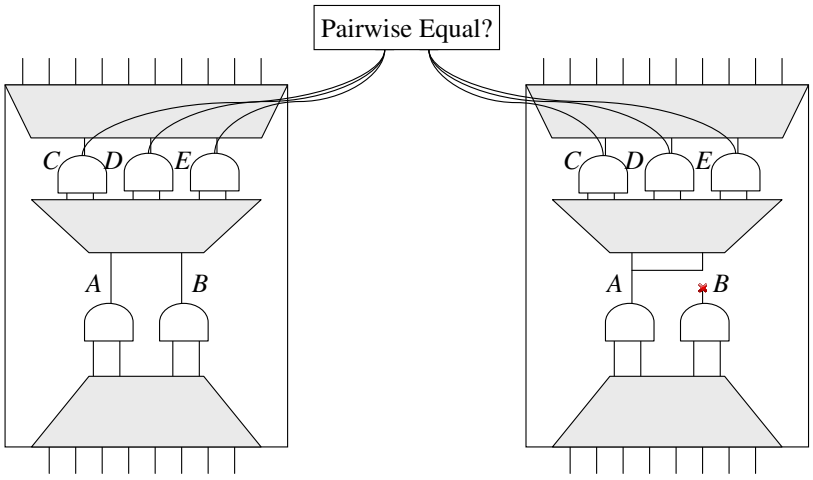


Figure 3.11: Detecting if a merge is observable in a circuit.

merge the signals if the merge does not change the circuit outputs. Note that this a **directed merge** in that we replace B with A but it may not be legal to replace A with B. In the algorithm, a set of nodes cut_B is derived such that all paths from B to an output go through a node in cut_B[5]. Then analysis is performed to see if any of the nodes in cut_B are affected by the replacing of B with A. If none of these nodes are affected then the directed merge is performed and the circuit is simplified.

The check that replacing B with A does not affect any nodes in cut_B can be

[5]This is a cut on the output side of a node. Input cuts were considered in Section 2.2.4

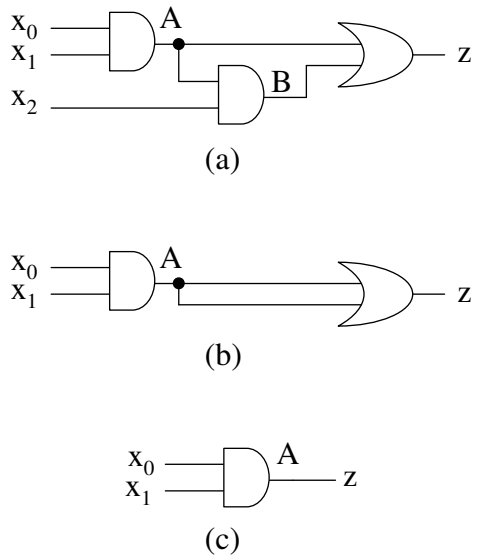


Figure 3.12: Simplification of a simple combinational logic network with ODCs.

performed using a single SAT call, as illustrated in Figure 3.11. Suppose the cut consists of the nodes C, D, E. These nodes separate A and B from all the circuit outputs[6]. To check that C, D, E are unaffected, we speculatively perform the merge operation and then compare the modified circuit against the original circuit. If the SAT solver cannot produce an input valuation that drives a pairwise difference on any of the cut signals then the merge is not observable at the cuts and therefore not observable at the outputs.

Previous approaches have been tailored to combinational optimization by failing to consider any reachable state information. Here we explore both direct and indirect sequential synthesis extensions to this basic algorithm.

Example 21. Combinational Simplification With ODCs

Consider the simple circuit shown in Figure 3.12A. This circuit can be dramatically simplified using ODC techniques as described in Algorithm 18.

Suppose we suspect that B can be replaced by A. We let cut_B *be just the single*

[6]In some literature, this is referred to as a **dominator set**.

output z and test that this merge does not affect z. This can be reasoned like so: $A = 1 \implies Z = 1$ and the value of B was not significant in deriving the value of Z. If $A = 0$ then $A = B$ and so replacing B with A does not affect the circuit.

While this reasoning was deductive and informal, such a check can be automated by speculatively simplifying the circuit and then checking that z' from the modified circuit is equivalent to z from the original circuit. This check can be performed using a single Boolean satisfiability call, similar to Figure 3.11.

Because replacing B with A does not change the circuit output, we are free to perform this replacement operation while preserving combinational equivalence with the original circuit. The resulting simplified design is shown in Figure 3.12B. Simple circuit simplifications then give the simpler circuit of Figure 3.12C.

3.4.3 Extension to Indirect Sequential Synthesis

3.4.3.1 Utilizing the Invariants

ODC-based merging can be made into an indirect sequential synthesis method by considering invariants. Invariants provide an over-approximation to the set of reachable states, and this over-approximation can be used to constrain the states searched while trying to satisfy SAT problem such as Figure 3.11.

To constrain a SAT problem, we augment the problem with the invariants that were proved. Consider Figure 3.13. The left circuit copy represents the original design, and the right copy represents a design that has been optimized with a merge. For a given output cut, we check that all cut nodes are pairwise equal. If a set of invariants has been proved, we can strengthen this by requiring all invariants to hold in both the left and right copies.

If the SAT problem of Figure 3.13 has a counterexample then the ODC-based

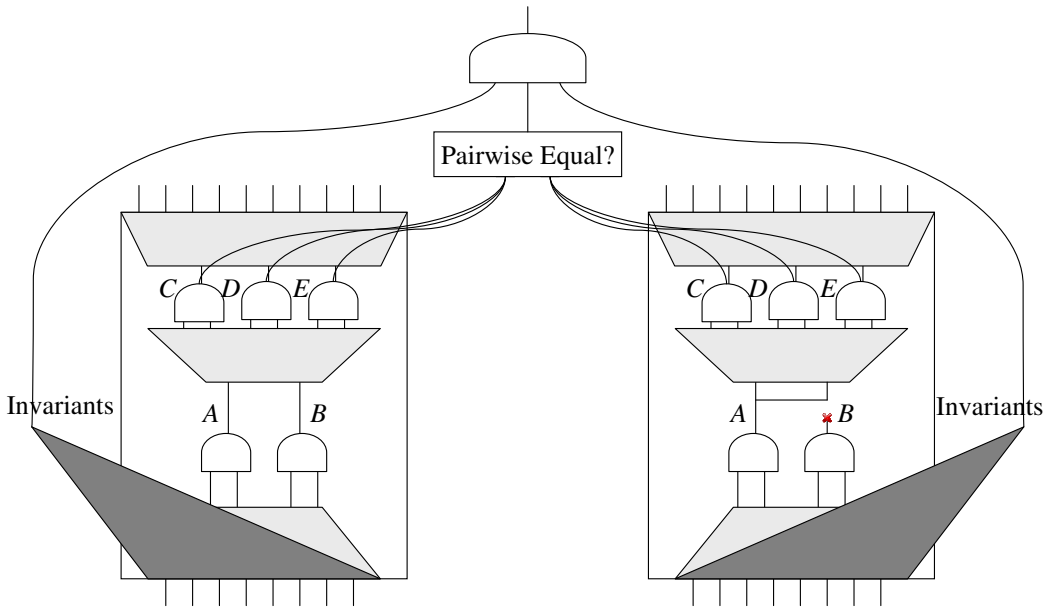


Figure 3.13: Extension of Figure 3.11 to indirect sequential synthesis.

merge is not valid (with respect to the given cut). [Zhu *et al.*, 2006; Plaza *et al.*, 2007] propose ways to either 1) derive a new cut that the merge can be tested against, or 2) conclude that the merge is not valid for any cut. By using the invariants to constrain the SAT problem, the number of satisfying assignments decreases because each satisfying assignment must satisfy all invariants. This decreases the likelihood of the problem being satisfiable, and therefore increases the chance that the merge is valid.

The changes to the SAT problem can also be viewed from a state space perspective. Suppose the unconstrained SAT problem of Figure 3.11 is satisfiable and the constrained version of Figure 3.13 is not. This means that the invariants have blocked the satisfying assignments. The invariant logic evaluates to 0 only on unreachable states, and this means that the counterexample found in Figure 3.11 was an unreachable state. By utilizing the invariants we allow merges to proceed if they only change the behavior of the cut on the subset of the unreachable states expressed by the

Table 3.8: Performance On A Set of IBM Synthesis Benchmarks

	Original		Combinational Observability		Indirect Sequential Observability		
Design	Latch	Ands	Latch	Ands	Proved Invariants	Latch	Ands
ibm4	255	1845	255	1520	60	255	1515
ibm5	93	925	93	759	5269	93	753
ibm6	151	811	142	661	2884	142	657
ibm7	428	3173	428	2710	60	428	2710
ibm8	207	3031	207	3028	1889	205	3001
ibm9	354	3896	353	3569	7695	353	3567
			99.01%	97.90%		98.85%	97.47%

invariants. Therefore the strengthened ODC-based merge algorithm is a sequential synthesis algorithm.

3.4.3.2 Experimental Results: Synthesis Benchmarks

The effect of leveraging invariants in ODC-based merging was first evaluated on 6 synthesis benchmarks taken from an IBM microprocessor design. The results are shown in Table 3.8.

The designs were first processed with combinational ODCs, similar to [Zhu *et al.*, 2006; Plaza *et al.*, 2007]. This optimization reduced the number of registers by 1% and reduced the number ANDs in the AIG by 2%. The register reductions were due to a COI reduction being run on the simplified logic cones. In some cases, registers would fall out of the COI and subsequently be removed from the design. Therefore, despite this synthesis algorithm being a purely combinational method, it can cause registers to be removed.

In a second pass, the original designs were processed with indirect sequential synthesis. An invariant discovery routine, implemented using the ideas of Chapter 2, was run for 10 minutes, and the resulting invariants were leveraged in the ODC analysis. With this additional information about the reachable state space, the ODC routine was able to eliminate 1.15% of the registers and 2.5% of the AND gates.

Clearly, invariants are able to help ODC analysis, but the improvements due to invariants are very modest.

3.4.3.3 Experimental Results: Verification Benchmarks

To further examine the effect of invariants on ODC analysis, we turned to verification benchmarks. The IBM formal verification tool *SixthSense* is used to perform unbounded property checking on IBM microprocessor designs, and many synthesis algorithms are used inside *SixthSense* to reduce the size of the model being verified. ODC simplifications, based on [Zhu *et al.*, 2006; Plaza *et al.*, 2007], is implemented inside *SixthSense*.

83 designs, each with a single safety property, were identified such that this property is impossible to prove using any of the algorithms implemented in *SixthSense*. Of these 83 properties, the reductions offered by the existing ODC simplifications allow 4 of the properties to be solved using interpolation.

We now ask the questions: do invariants help ODC simplifications in this context, and do these simplifications increase the number of properties that can be solved with interpolation?

Table 3.9 examines indirect sequential synthesis on these benchmarks. An invariant discovery routine was given 10 minutes to find invariants, and the proved invariants were leveraged in the ODC simplifications. The addition of these invariants did not noticeably slow the ODC algorithm, but it did improve the results. Relative to ODC simplification that did not use invariants, the invariants enabled a further reduction of 1.5% in the number of ANDs and 0.7% in the number of registers.

Although the design size reductions seem modest, they successfully enable verification to be completed on the design. Recall that the combinational ODC simplifications allow 4 of the 83 difficult properties to be verified using interpolation. Strengthening the ODC method with invariants allows an additional 5 of the 83

Table 3.9: Performance On A Set of 83 IBM Verification Benchmarks

	Original		Combinational Observability		Indirect Sequential Observability	
Design	Ands	Registers	Ands	Registers	Ands	Registers
odcSynth_3	1452	184	1443	184	1408	184
odcSynth_4	2553	271	2542	271	2486	271
odcSynth_5	1452	184	1443	184	1408	184
odcSynth_6	2553	271	2542	271	2486	271
odcSynth_11	2421	146	2183	146	2337	146
odcSynth_17	429	94	429	94	427	94
...	...	...	...	...	...	...
odcSynth_84	7547	735	3948	735	3922	734
odcSynth_85	7555	737	4026	737	4049	737
odcSynth_87	2403	533	2376	532	2228	532
odcSynth_88	2411	534	2377	533	2228	532
	100.00%	100.00%	94.09%	99.61%	92.69%	98.94%

properties to be verified. This improves the verification tool and allows it to solve a property that was previously impossible.

3.4.4 Extension to Direct Sequential Synthesis

Here we consider an adaption of Algorithm 18 to direct sequential synthesis [Case *et al.*, 2008a]. We wish to find merges that preserve all circuit output values in all reachable states. Unlike the combinational ODC case, merges that may cause outputs to differ on unreachable states are also considered. This increases the optimization potential of the ODC method.

This work is limited in the sense that it will always preserve the values of the next state functions on the reachable states. Viewed from a state transition graph (STG) perspective, our method would preserve the reachable part of the STG exactly, unlike some sequential synthesis methods (eg. retiming [Leiserson and Saxe, 1991]). In this work we limit ourselves for computational reasons and cannot claim to utilize full sequential observability, just a limited form of it. This restriction may hurt our results but introduces several nice synthesis properties. Namely, the scan chain is preserved

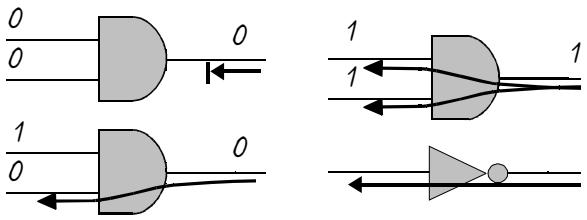


Figure 3.14: Propagating the controlling path backward through an AIG.

and post-silicon debugging is not complicated by a change in latch functions.

The **combinational outputs** (**COs**) is a set of signals comprised of the outputs and next state functions. This is sequential synthesis that will always preserve the values of the COs in all reachable states. The check that no COs are modified by the merge in any reachable states is performed directly using inductive reasoning.

3.4.4.1 Finding Merge Candidates

In the first step of our algorithm we would like to quickly find a set of candidate merges that is a superset of the CO-preserving merges. That is, we would like to find ordered pairs of signals (A, B) such that it is likely that for all reachable states S that one of the following is true:

- $A = B$ in state S.
- Merging A and B will not produce a difference at a CO in S.

In the language of [Zhu *et al.*, 2006; Plaza *et al.*, 2007] this is: for all reachable states, either $A = B$ or B is not observable. The merge operation is free to replace A with B because B being unobservable implies that the difference will not be visible at a CO.

To quickly find a set of candidate merges we use a slightly weaker notion than observability: whether or not a signal lies on a CO's controlling path. If an AIG has been simulated for a single concrete input then the controlling path can be propagated backwards through the AIG as shown in Figure 3.14. Note that in the $0 \cdot 0 = 0$ case we

choose to conservatively conclude that neither of the inputs are observable, making the marked controlling paths smaller but guaranteeing that all marked signals have the ability to influence the value at a CO. Toggling any marked controlling path signal will cause at least one CO to toggle. The controlling signals are therefore observable at the COs.

```
1: // Given "design" and a number of reachable input "states"
2: sim = simulateDesign(design, states);
3: controlling = extractControllingPaths(design, sim);
4: for all pairs of signals (A, B) do
5:    if (sim.A == sim.B) || ! controlling.B then
6:       (A, B) is a candidate merge;
7: end for
```

Algorithm 19: Extracting candidate merges from simulation.

Our method to find merge candidates is shown in the pseudocode of Algorithm 19. We simulate the circuit with a number of known-reachable input vectors and extract the controlling paths for each simulation vector. Then we use the simulation and controlling path information to check for each ordered pair of signals (A, B) if $A = B$ or B is not controlling. For simplicity, an $O(n^2)$ algorithm is presented here, but an improvement is discussed in Section 3.4.5.

Example 22. Extracting Direct ODC Candidates

An example of this method for extracting candidate merges is illustrated in Figure 3.15. It can be proved that if the initial state is $BC = 00$ *then it is possible to merge such that* AB *is replaced with* $\overline{A \oplus C}$*:*

- $BC = 10$ *is unreachable.*
- $\overline{A \oplus C} + AB = \overline{A \oplus C}$ *for every reachable states*
- *It is safe to replace* AB *with* $\overline{A \oplus C}$ *at Z's OR gate.*

This merge cannot be an equivalence because the signals have differing support. It is not a combinationally observable merge because we need to know that $BC = 10$ *is not reachable. This merge can only be found using sequential observability.*

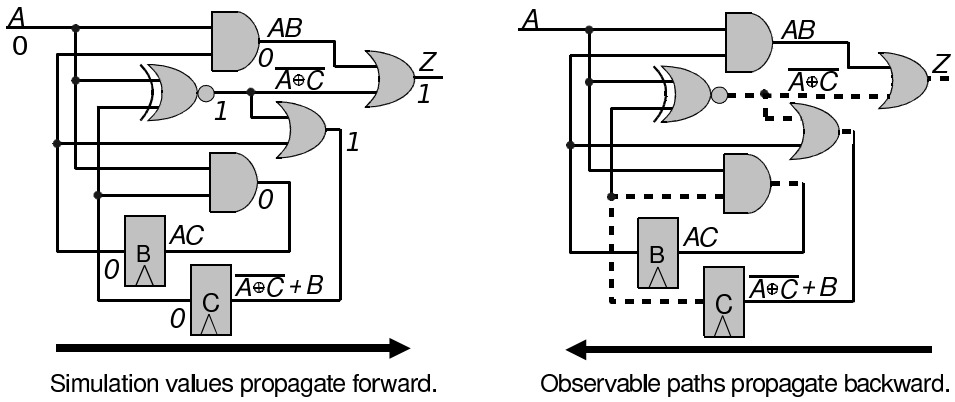


Figure 3.15: Discovering sequential observable merge candidates. (While we use AIGs, rich gate types are shown for compactness.)

We can algorithmically discover this candidate merge by simulating the initial state $BC = 00$ along with a randomly selected value for the input A. Then highlight the controlling paths that lead up to each CO. According to the simulation information $AB \neq \overline{A \oplus C}$, but AB is not on the controlling path to any CO. Therefore the ordered pair of signals $(AB, \overline{A \oplus C})$ becomes a candidate for merging. Altering the value of the AB signal will not produce a difference at any CO.[7]

3.4.4.2 Proving Candidates

In order for our synthesis algorithm to be sound, we must prove that for all reachable states, modifying the circuit with the candidate merges will not produce a difference at a CO. This will be formulated as a set of properties that are then proved by induction on the state space.

The correctness and compatibility of a set of candidate merges can be checked by building a miter circuit similar to Figure 3.16. Two copies of the circuit are compared: the original circuit and a copy that has been simplified with the candidate

[7]The sequentially observable merge $(Z, \overline{A \oplus C})$ is equivalent.

```
1: // Given "candidates" merges on "design"
2: while (1) do
3:    mod = mergeCandidates(design, candidates);
4:    miter = compareOutputs(design, mod);
5:    if (checkInduction(miter)) then
6:       break ;
7:    else
8:       pruneCandidates(design, candidates, getCex());
9:    end if
10: end while
```

Algorithm 20: Proving merges valid.

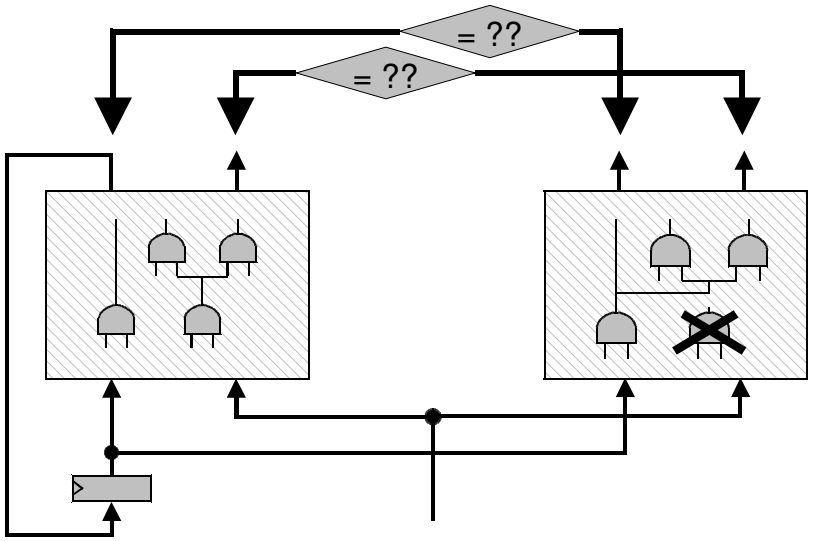


Figure 3.16: Proving that a set of merges does not change any CO.

merges. These two circuit copies are driven by the same inputs and state, and logic is synthesized to check that the COs are the same.

```
1: // Given "counterexample", "candidates", and "design"
2:
3: // fast but incomplete method (checks correctness)
4: Simulate counterexample and extract controlling paths;
5: for all (A, B) ∈ candidates do
6:    if (sim.A ≠ sim.B) && controlling.B) then
7:       discard (A, B);
8: end for
9: if (∃ discarded candidate) then return ; // Skip slow step.
10:
11: // slow but complete method (checks compatibility)
12: stack = (candidates);
13: goodCands = nil;
14: while (!stack.empty) do
15:    currCands = stack.pop();
16:    mod = design with merged goodCands and currCands;
17:    Simulate design and mod with counterexample;
18:    if (∃ output o s.t. sim.design.o ≠ sim.mod.o) then
19:       if (currCands.size() == 1) then
20:          discard currCands; // Drop single culprit.
21:       else
22:          cand1, cand2 = divideInHalf(currCands);
23:          stack.push(cand1, cand2); // Divide and conquer.
24:       end if
25:    else
26:       goodCands += currCands;
27:    end if
28: end while
```

Algorithm 21: Discarding bad merge candidates.

Note that constraining the merges such that the next state is not altered greatly simplifies the miter circuit. Without this constraint each circuit in Figure 3.16 would have an independent state, doubling the number of latches in the miter. Here we choose to simplify our miter and subsequent proof, possibly at the expense of the

optimization results. This constraint also implies that the design's reachable STG will not be altered.

The miter circuit can be viewed as a single sequential machine with a set of properties to be checked, one for each CO equivalence. We prove these properties using 1-step induction [Bjesse and Claessen, 2000]. This is an incomplete unbounded verification technique that can prove some of the properties true for all reachable states.

Algorithm 20 illustrates how induction is used to find a subset of CO-preserving candidate merges. Counterexample states are produced where the two circuits in the miter are not equivalent. By removing the merges that caused the miscompare and trying induction again, a greatest fixed point algorithm is developed that will produce a set of merges that when applied will yield an simplified equivalent circuit.

One significant challenge is to determine which candidate merges are faulty given the fact that the simplified circuit produces a differing output. This task is performed by the `pruneCandidates()` function, outlined in Algorithm 21. Two methods are utilized to find the merges responsible for the output mismatch.

The first (lines 3-9) is a fast method that only checks that each merge is individually correct, similar to Section 3.4.4.1. This method is fast but incomplete because merges can interact with each other, and a set of merges being individually correct is no guarantee they are compatible as a group. This method is used as a fast filter before calling the second, more expensive, method.

The next method (lines 11-28) will check the compatibility. It does this by performing a binary search over subsets of the candidates until it finds individual candidate merges that cause a local reduced model to have a differing circuit output. It is able to check compatibility but is slow because for each subset it must simulate a reduced circuit model. The fast method is used whenever possible, and the more expensive method is only used when the first method fails to find a guilty candidate.

Table 3.10: Candidate Merge Statistics

Design	Merges	Nodes	Redund. Nodes	Choices / Redund.	Avg. Level For Red.	Rep.
ibm4	3215	1498	616	5.1	7%	21%
ibm5	2469	715	409	4.9	11%	27%
ibm6	2167	673	381	4.7	12%	17%
ibm7	4323	2668	705	3.7	12%	37%
ibm9	8688	3478	1593	4.1	11%	29%
			0.45	4.5	10%	26%

These two methods together will greedily isolate the first compatible set of merges that preserves the COs under the given counterexample. There are often multiple compatible subsets which can be exploited by applying our proposed synthesis algorithm multiple times.

3.4.4.3 Using the Candidates

We often need to simplify a circuit using a set of candidate merges. This is done both in checking the merges as in Figure 3.16 and also in constructing the final simplified circuit.

Producing a simplified circuit from a set of candidate merges is not simple. Table 3.10 gives statistics on the set of candidate merges as examined periodically throughout a run of our tool. Each candidate merge is an ordered pair of signals (*redundant*, *replacement*) where *redundant* will be replaced with *replacement*. On average, the candidate merges suggest replacements for approximately 45% of the design nodes, and for each redundant node there are an average of 4.5 candidate replacements. Selection of the replacement to use has a large impact on ability of our algorithm to minimize the AIG size, and heuristics that select the replacement to use are important.

In selecting a replacement for a redundant node, it is very important to not

introduce a combinational cycle. Table 3.10 shows that on average for a circuit with N levels, the redundant node is at level $10\% \cdot N$, and the replacement is node is at level $26\% \cdot N$. Since a node will often be replaced with one of a higher level, we can expect combinational cycles to abound. This also means that the upper 74% of the circuit usually has few candidate merges, a potential basis for future heuristics research.

Experimentally we found that the best performance could be obtained not by replacing the entire signal *redundant* with *replacement* but by selectively replacing each of the fanouts of *redundant*. This introduces the flexibility to replace each fanout with a different signal, thereby improving the optimization potential. It also provides an easy way to handle combinational cycles; if doing replacing *redundant* with *replacement* would introduce a cycle and *redundant* is a multi-fanout net then the replacement can still occur on the subset of the fanouts not involved in the cycle. This generalization greatly improved the quality of our synthesized designs.

The heuristic used in this work is shown in Algorithm 22. Each candidate merge is passed to this routine to incrementally simplify the network. The general strategy used is to 1) avoid cycles, 2) enable constant propagation, 3) remove fanouts from low-fanout nodes in the hope that they can be removed after simplification, 4) greedily enable the maximum amount of structural hashing benefits, 5) as a tie breaker, remove fanouts from nodes that have a low ID.

3.4.5 Enhancing Scalability: Avoiding $O(n^2)$ Candidate Merge Checks

In the method to find candidate merges given in Section 3.4.4.1, each pair of signals in the design is examined. The resulting $O(n^2)$ complexity is a major problem on designs with a large number of signals n.

```
1: // Given a merge with signals (redundant, replacement)
2: for all fanouts out of redundant do
3:    if (replacement ∈ trans_fanout_cone(out)) then
4:       Drive out with redundant     // Avoid a cyclic circuit
5:    else if (redundant or replacement is constant) then
6:       Drive out with (constant)       // Propagate constants
7:    else if (redundant or replacement has ≤ 2 fanouts) then
8:       Drive out with (higher fanout) // Remove low-fanouts
9:    else if (struct_hash_gain(replacement → out) ≠
               struct_hash_gain(redundant → out)) then
10:      Drive out with (better gain)   // Maximize hash gain
11:   else if (num_fanouts(replacement) ≠
               num_fanouts(redundant)) then
12:      Drive out with (higher fanout) // Remove low-fanouts
13:   else
14:      Drive out with (higher node ID)    // Remove low-IDs
15:   end if
16: end for
```

Algorithm 22: Using a merge to simplify the logic network.

Table 3.11: Candidates In The IBM Benchmarks

Design	Sig. Pairs Tested (Using BK-Trees)
ibm4	19.29%
ibm5	42.03%
ibm6	27.40%
ibm7	24.25%
ibm9	28.50%

Papers discussing combinational observability merges [Zhu *et al.*, 2006; Plaza *et al.*, 2007] have proposed heuristics to reduce the average complexity of this step, but these heuristics are inadequate because they fail to handle a large number of simulation vectors. In this section, we discuss a method to reduce this $O(n^2)$ complexity while handling large amounts of simulation data resulting from our semi-formal analysis.

Burkhard-Keller (BK) Trees are an algorithm and datastructure used to quickly find points in a space that are "close" to a given reference point [Burkhard and Keller, 1973]. More formally, given a set of points, a metric $d(\cdot, \cdot)$, a point A, and a constant ϕ, BK-Trees can be used to find points B such that $d(A, B) \leq \phi$. BK-Trees exploit the triangle inequality to bound the search for feasible B's and only need to touch a small part of the search space to answer the query.

BK-Trees can be utilized to speed up the search for candidate merges. Given design signals A and B with simulation vectors $sim.A$ and $sim.B$, define the pseudometric $d(A, B)$ as $d(A, B) = ||sim.A \oplus sim.B||$ where $||\cdot||$ denotes the number of ones in a vector. This is not a true metric because it is not positive definite (referred to as a pseudometric [Steen and Seebach, 1970]), but BK-Trees do not rely on positive definiteness and this pseudometric can be safely used. With formalism, we fix node B and utilize $controlling.B$, a mask expressing for which simulation vectors B is controlling, to find the A's such that $d(A, B) \leq ||!controlling.B||$. The set of satisfying A's is a superset of the set of A's satisfying $||(sim.A \oplus sim.B) \cdot controlling.B|| = 0$ and therefore a superset of the A's that can be safely merged with B.

In this way, BK-Trees are used to narrow the search for B's that can be merged with a given node A. The effectiveness of this approach is shown in Table 3.11 where the percent of the search space that was examined is shown in the column "Signal Pairs Tested." While this doesn't reduce the search space exponentially, it is effective in improving the runtime in practice. Furthermore, BK-Trees are easy to implement

Table 3.12: Performance Across Benchmark Suites

Benchmark Suite				Preproc.		Sequential Observability		Combinational Observability	
Suite	Designs	Latches	Ands	Latches	Ands	Latches	Ands	Latches	Ands
IBM	5	248	2384	0.98	0.82	0.99	0.96	1.00	1.00
ISCAS89	28	109.39	751.64	0.95	0.72	1.00	0.90	1.00	1.00
PicoJava	64	627.83	2943.55	0.87	0.69	0.92	0.95	0.92	1.00

and a query over a BK-Tree takes almost constant time.

3.4.5.1 Enhancing Scalability: Selection of Proof Technique

In Section 3.4.4.2, the candidate merges are checked with a miter circuit and a series of sequential properties that can be proved using any unbounded technique. In this work we have chosen to use induction because it scales well, but it is well known that induction is an incomplete technique that is not able to prove all true properties [Prasad *et al.*, 2005]. In a verification domain this behavior is not desirable because of the requirement that all properties be proved. In a synthesis domain, this incomplete behavior is an acceptable trade-off for the scalability of induction. Merges disproved by induction are dropped, and because of the incompleteness some correct merges might be dropped along with the incorrect ones, reducing synthesis results.

We experimented with stronger induction formulations, namely K-step induction with unique state constraints [Bjesse and Claessen, 2000]. Increasing K rarely improved our optimized AIG node count, and increasing K beyond 1 significantly hurt the runtime of our method. Because of the near-independence of the results from the complexity of the induction formulation, in our implementation we always use simple or $K = 1$ induction.

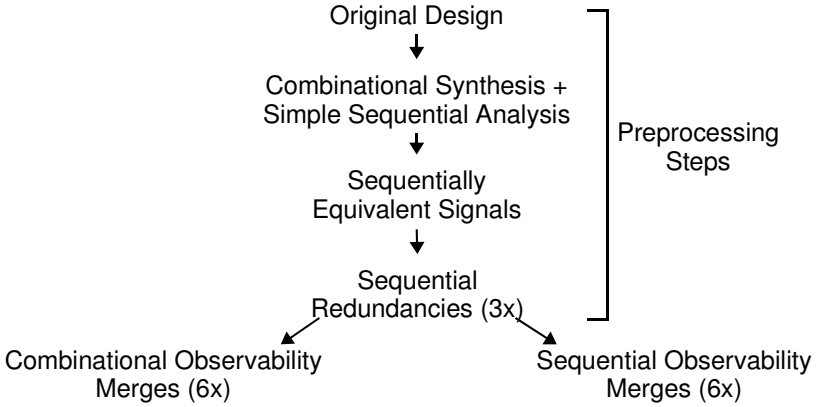


Figure 3.17: Algorithmic flow used in our experiments.

Table 3.13: Performance On A Set of IBM Benchmarks

	Original		Preprocessing			Sequential Observability			Combinational Observability		
Design	Latch	Ands	Time	Latch	Ands	Time	Latch	Ands	Time	Latch	Ands
ibm4	255	1845	49.36	253	1539	136.2	252	1453	23.64	253	1539
ibm5	93	925	7.72	93	738	42.51	93	679	5.4	93	738
ibm6	151	811	8.24	142	657	30.35	140	651	7.49	142	657
ibm7	428	3173	57.34	426	2684	302.15	422	2628	169.13	426	2684
ibm9	354	3896	21.9	353	3482	167.49	352	3447	103.35	353	3470
	1.00	1.00		0.98	0.82		0.99	0.96		1.00	1.00

3.4.5.2 Experimental Results

Sequential observability and all other algorithms in Figure 3.10 were implemented in C++ inside of a synthesis and verification environment. The environment uses ABC [Synthesis and Group, 2008] for combinational synthesis and MiniSat [Een and Sorensson, 2008] for SAT solving.

Combinational observability merges as presented in [Zhu *et al.*, 2006; Plaza *et al.*, 2007] is the best merge-based synthesis algorithm in the literature to date. We will compare our work to combinational observability on 3 benchmark suites: a set of processor blocks from IBM Corporation, the ISCAS '89 benchmarks, and a selection of blocks from the Sun PicoJava processor [Microsystems, 2008].

To fairly compare combinational and sequential observability, each design was first heavily synthesized as shown in Figure 3.17. Combinational synthesis, including SAT sweeping and rewriting [Mishchenko *et al.*, 2006] along with simple techniques to find structurally equivalent latches and sequentially constant latches [Bjesse and Kukula, 2005] was first applied. This was followed by processing of sequential equivalences and sequential redundancies. The sequential redundancy algorithm was run 3 times in order to build a high quality set of simulation vectors as discussed in Section 2.3. The designs at this point are labeled as "Preprocessed" in Tables 3.13 and 3.12. Finally, on two separate runs the preprocessed design was optimized using either combinational observable merges or sequentially observable merges. The observability algorithms were run 6 times to allow iterative circuit gain to saturate so that the maximum cumulative gain could be measured.

Results on the two sets of industrial benchmarks and the one set of academic benchmarks are shown in Table 3.12. For the columns labeled "Preprocessing" the numbers of AIG nodes and latches are given relative to the original design. In the sequential observability and combinational observability columns, the AIG nodes and

latches are given relative to the preprocessed design. Sequential observability is able to reduce the node counts of the preprocessed designs by about 6% and the latch counts by about 3%. The node count reduction is dramatically more for sequential observability than it is for combinational observability, and this indicates that reachable states are a very important degree of freedom to consider in an observability-based algorithm. It is also interesting that while this technique does not directly target latch reductions, occasionally all fanouts are removed from a latch, causing the latch to be removed.

Detailed runtimes for the IBM designs are shown in Table 3.13. Sequential observability is slower than combinational observability, but this slowdown is expected because we must check the property inductively across two time frames instead of combinationally in just one time frame. The slowdown is not severe, and the sequential case is still scalable and not likely to substantially increase the total runtime of industrial tools.

In our experiments, the combinational observability method did very little on average. We are starting from a heavily synthesized design point, and combinational observability is not able to improve upon this design point further. This indicates that the types of optimizations done by combinational observability are contained in all of the preprocessing that has been done (and the preprocessing was much cheaper). This is not true of sequential observability. Our method was able to improve upon the preprocessed designs, sometimes significantly.

3.4.6 Comparing Direct and Indirect Synthesis

The above text described two extensions to ODC-based merging. In the first extension, invariants were found. This represents an over-approximation to the set of reachable states, and this over-approximation was leveraged to strengthen ODC-based

merging. This forms an indirect sequential synthesis algorithm. In the second extension, the proof technique used in ODC-based merging was modified to use induction rather than simple combinational analysis. This forms a direct sequential synthesis algorithm.

Tables 3.8 and 3.13 describe the performance of these algorithms on the same set of IBM synthesis benchmarks. Although the two sequential synthesis algorithms were implemented at different times and in different tools, we can compare them based on their performance on this common benchmark set.

Table 3.8 shows that the indirect sequential synthesis version of ODC-based merging is slightly better than its combinational counterpart. Specifically, it was able to reduce registers by an additional 0.2% and ANDs by 0.4%.

This can be compared to Table 3.13 which compares direct sequential synthesis against combinational synthesis. Direct sequential synthesis is able to reduce registers by an additional 1% and ANDs by 4%.

We can conclude that in this case direct sequential synthesis is significantly stronger than indirect sequential synthesis. Although it was not directly examined here, we suspect that the direct and indirect sequential synthesis methods might have orthogonal strengths. Similar to the findings in Section 3.3.5, we suspect that a hybrid of these two approaches might be stronger than either of the two synthesis algorithms examined here. We leave that experiment for our future work.

3.5 Conclusions

In this chapter we introduced the notions of direct and indirect sequential synthesis. Direct sequential synthesis incorporates state space exploration within the core of the synthesis algorithm. In contrast, indirect sequential synthesis algorithms are combinational algorithms that have been strengthened only preserving the design behavior

on a pre-determined over-approximation to the reachable states. In this work, the over-approximation is developed through the automated discovery of invariants.

To explore the relationship between direct and indirect sequential synthesis, two cutting-edge synthesis algorithms were explored in detail: resubstitution and ODC-based merging. Direct and indirect sequential synthesis versions of each algorithm were presented, and experimental results were examined at length.

In general, the results show direct sequential synthesis methods to be stronger than indirect methods. It seems that while invariants are able to capture state-space properties that synthesis can leverage, the properties captured are weaker than what can be developed by using a stronger proof technique within the synthesis algorithm itself. This led to the reductions offered by direct sequential synthesis being better than the reductions offered by the corresponding indirect sequential synthesis algorithm.

Despite the poorer results, indirect methods do have other benefits. The reachability approximation that is developed through the discovery of invariants need only be derived once. This approximation can then be used for a sequence of indirect sequential synthesis methods, and by amortizing the runtime in this way the sequence of indirect sequential synthesis methods will run much more quickly than a sequence of direct methods.

Additionally, we found direct and indirect sequential synthesis methods to offer largely orthogonal results. In the case of resubstitution, the two methods were combined, and experimental results show that this hybrid is stronger than either of the two synthesis methods alone.

Chapter 4

Applying Invariants to Verification

4.1 Motivation

Unbounded verification of safety properties is a very challenging problem. Explicit exploration of every reachable state is impractical for most industrial designs, and this forces researchers to turn to indirect state exploration techniques such as induction (Section 14) and interpolation (Section 15).

Both induction and interpolation reason about the reachable state space in approximate ways. Unreachable states could be mistaken for reachable states, and this is the source of spurious counterexamples in these algorithms. While the algorithms work well for some types of properties, application of these algorithms may result in many spurious counterexamples and thus an incomplete verification.

Invariants present a possible solution to this spurious counterexamples problem. The conjunction of all proved invariants is an approximation to the set of reachable states, and while this too is an over-approximation that may include unreachable states, it may exclude the states that would be spurious counterexamples in induction or interpolation.

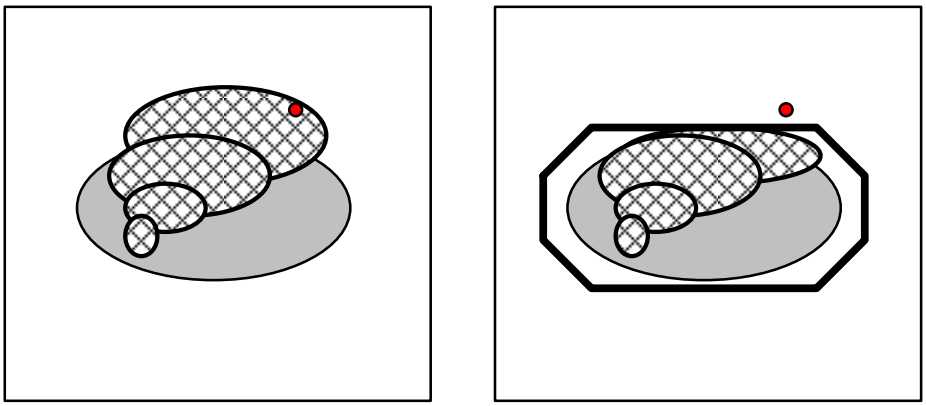


Figure 4.1: Interpolation with (left) and without (right) a state-space over-approximation.

In the extreme case, all states violating the property may lie outside of the invariants' set of approximately reachable states. The invariants alone are sufficient to prove that these bad states are unreachable and therefore the property is verified. However, this happens only very rarely, and more powerful solutions are needed for the general case.

Induction and interpolation can be strengthened by constraining them to only explore states that lie inside the reachability approximation provided by the invariants. This provides a general purpose method that explores a tighter set of states than either induction, interpolation, or invariants would explore on their own, and this is successful in verifying some difficult properties. In this chapter we explore methods by which this can be done and provide some experimental results.

4.2 Strengthening Interpolation

4.2.1 Introduction

The interpolation method is an abstraction of explicit state traversal. Instead of explicitly computing the image of a set of states under the transition relation, it finds

an over-approximation to this set. This over-approximation allows it to proceed more quickly through the reachable state set.

Proving invariants is a competing method for finding the set of reachable states. In this method, the state graph is not traversed. Instead, an invariant over the set of states is proved inductively. The proof is done so that every reachable state satisfies the invariant, and therefore the image of the invariant is an over-approximation to the set of reachable states.

Consider a hybrid of both reachability techniques – a reachability approximation framework that is more powerful than either of its two constituents [Case and Lwin, 2006]. This concept is illustrated in Figure 4.1. The search starts in the set of initial states, and the set of explored states is allowed to grow. The search may erroneously leave the reachable state space (gray) and hit an unreachable bad state (red). If the search is confined to an over-approximation of the reachable state space then this will not happen.

4.2.2 Interpolation Basics

Suppose we have a design we wish to verify, and we do a bounded model check (BMC) on that design. If no bugs are found then the underlying SAT problem appears unsatisfiable. In this case, we can extract a Boolean expression called the interpolant from the proof of unsatisfiability. The interpolant is an over-approximation of the states reachable in 1 step from the initial state used in the BMC.

Algorithm 23 outlines the interpolation algorithm. The interpolant is used to over-approximate the image of a set of states, and this over-approximated image is used in a manner similar to what is done in explicit state traversal (Section 13). If a fixed point is reached, then every reachable state has been explored and we know that the design is verified.

```
1: for (bmcDepth = 2; true; bmcDepth += 10) do
2:   startStates := initialStates
3:   while (true) do
4:     // if the model looks broken
5:     if (! bmc(startStates)) then
6:       //  if the counterexample is real
7:       if (startStates == initialStates) then
8:         return "counterexample"
9:       else
10:        // counterexample may be spurious
11:        break
12:      end if
13:    else
14:      startStates += getInterpolant()
15:      if (fixedPoint(startStates)) then
16:        return "verified"
17:      end if
18:    end if
19:  end while
20: end for
```

Algorithm 23: Basic sketch of the interpolation unbounded verification routine.

If a bad state is reached, the counterexample is only considered valid if the bounded model checker started at the true set of initial states. If it started from an interpolation result then this interpolant could contain an unreachable state. The fact that a bad state is reachable in `bmcDepth` transitions from an unreachable state is not interesting, and to avoid this the bounded model checking depth is increased and the proof begins anew in an attempt to find a concrete counterexample.

Interpolation is a unbounded model checking algorithm that is both complete and sound. It is the most robust verification method known to date [Amla *et al.*, 2005]. However, it is not without its flaws. The main defect is that if an interpolant includes an unreachable state then the algorithm will begin exploring traces starting at this unreachable state. These traces add to the overall runtime unnecessarily, and reducing these erroneous and extraneous traces would directly improve the efficiency of interpolation.

4.2.3 Strengthening Interpolation With Invariants

```
1: // Design preprocessing
2: invariants := ∅
3: for (i = 0; i ¡ N; i++) do
4:    invariants += findInvariants()
5:    indirectSequentialSynthesis(invariants)
6: end for
7:
8: // Model checking
9: interpolate(invariants)
```

Algorithm 24: Basic sketch of the interpolation unbounded verification routine.

Consider Algorithm 24 which combines the strengths of both the interpolation and invariant methods. In this procedure, the design is preprocessed with indirect sequential synthesis before it is model checked with interpolate.

Each preprocessing step helps later procedures in the following ways:

Indirect Sequential resynthesis is preformed in each step. This step helps to reduce the size of the design and therefore make interpolation more scalable.

More invariants are found. Each iteration computes more invariants, and the conjunction of these invariants gives a reachability approximation that gets tighter with every pre-processing iteration.

Orthogonal invariants are found. Each call to findInvariants() processes a slightly different design due to the synthesis efforts, and therefore the different sets of invariants are found in each iteration.

After the preprocessing, every bad state that violates the safety property might lie outside the reachability over-approximation. In this case, the design is verified and we terminate. If a single bad state satisfies our over-approximation invariant then we call a modified version of interpolation that does not explore states violating the invariant. The interpolation should be dramatically sped-up because it is presented both with a smaller problem and also a useful reachability over-approximation.

4.2.4 Experimental Results: Property Checking

The implication procedure, the interpolation procedure, and all intermediate tools necessary to form a hybrid solver were implemented in C++ in the ABC Logic Synthesis and Verification framework [Synthesis and Group, 2008]. Two sets of benchmarks were then run through the system: a set of sequential equivalence checking problems over the ISCAS89 benchmarks, and a set of safety property benchmarks from [Chalmers, 2007]. Safety benchmark contains examples from Cadence SMV, CMU SMV, SMV case studies, NuSMV, VIS, Texas97, ISCAS89 and IRST model checking group. A 3 GHz Pentium 4 machine was used to generate all the results.

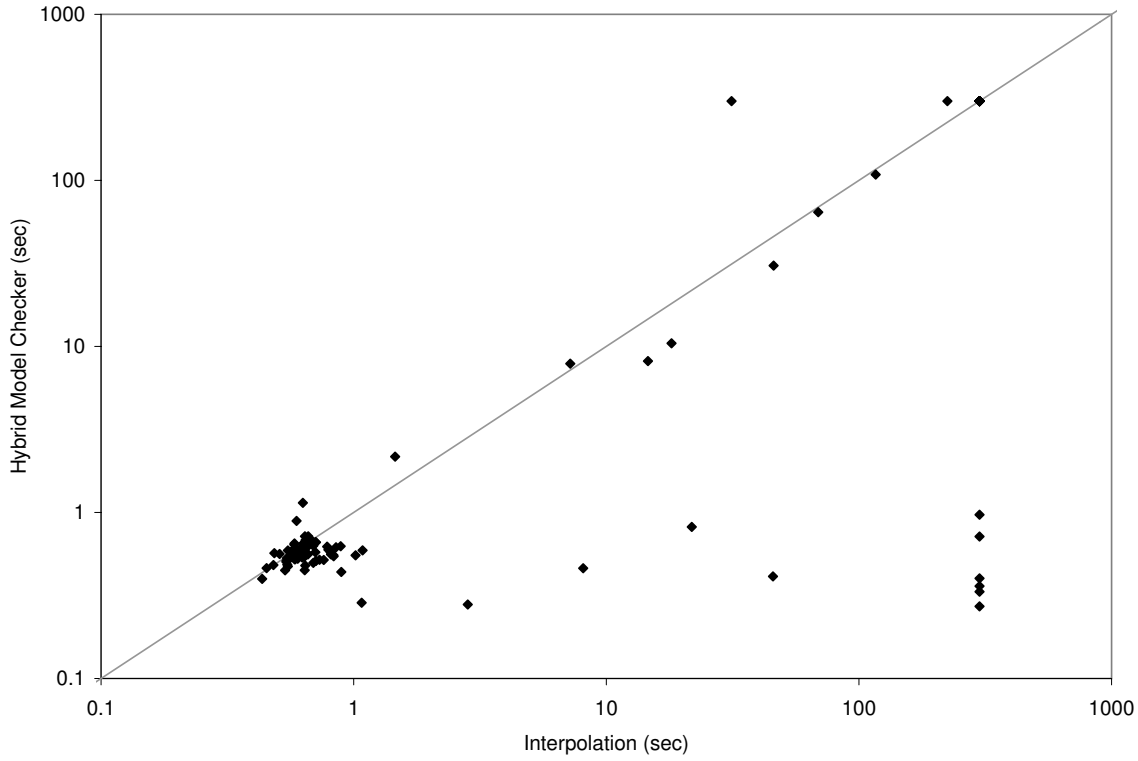


Figure 4.2: Runtimes for a selection of safety property benchmarks. Shown are 74 passing properties, 15 failing properties, and 14 undecidable properties on designs ranging from 10 to 506 latches. Model checking was run with a timeout of 300 seconds.

We preprocessed the design 5 times. Each preprocessing step takes 5 seconds and computes an invariant which over-approximates the set of reachable states. This over-approximation gets tighter as more preprocessing steps are run and the apparent set of reachable states shrinks. In practice we find that we can shrink the over-approximation to an arbitrarily small size by running an appropriate number of preprocessing steps.

After the invariant is derived, we run interpolation with this invariant. In this configuration we have a hybrid solver that benefits from the strength of both invariants (implications) and interpolation.

In the first experiment, we took 103 verification benchmarks from various checkers [Chalmers, 2007]. These benchmarks each have one safety property to be checked. We ran interpolation and the hybrid model checker on these designs, and the results are shown in Figure 4.2.

This empirical evidence shows that our hybrid technique dramatically speeds up interpolation.

4.2.5 Experimental Results: Sequential Equivalence Checking

To further explore the properties of our hybrid solver, we created a set of synthetic verification benchmarks from the ISCAS89 test suite. We retimed each design and then built a product machine and miter to verify that the outputs of the original machine match those of its retimed counterpart. If the retiming is done correctly, the output of the miter will be 0 in every reachable state. The synthetic benchmarks were chosen because they represent hard verification problems with many latches, and as such they stress the interpolation method.

Table 4.1 shows the results of the sequential equivalence checker (SEC) benchmarks. We first ran interpolation on these designs and got the mix of passing, failing,

Table 4.1: The ISCAS89 SEC benchmarks. Successive iterations of invariant discovery derive ever-tighter reachability approximations, and this approximation benefits interpolation.

Circuit Statistics			Reachable States (%) After Preproc. Step					Verif.	Runtime (sec)[3]		Verif.
Design	AND	Latch	#1	#2	#3	#4	#5	Done[2]	Interp.	Hybrid	Result
s27	17	6	12.50	12.50	-	12.50	12.50	yes	0.97	0.43	pass
s208	99	16	0.39	100.00	0.39	-	0.39	yes	1200.00	0.49	pass
s298	187	45	1.66	1.66	0.10	0.10	0.05	no	456.60	257.09	pass
s344	235	56	3.44	0.18	0.01	0.02	0.00	no	1200.00	1200.00	-
s349	235	56	3.71	1.25	0.22	0.01	0.01	no	1200.00	1200.00	-
s382	223	54	3.07	0.34	2.17	0.04	0.04	no	1200.00	1200.00	-
s400	232	56	9.06	0.42	0.10	0.01	0.00	no	1200.00	1200.00	-
s444	233	53	2.48	0.73	0.73	0.29	0.23	no	1200.00	1200.00	-
s641	301	39	7.98	0.92	0.00	0.00	0.00	yes	57.69	0.66	pass
s713	303	39	31.25	8.59	1.66	0.59	-	no	40.86	9.26	fail
s420	229	39	39.06	39.06	13.06	20.62	9.84	no	1200.00	1200.00	-
s386	268	28	0.01	0.00	0.01	-	0.00	yes	30.80	0.76	pass
s526n	267	59	42.40	2.94	0.93	0.05	0.01	no	1200.00	1200.00	-
s526	268	67	1.05	1.05	1.05	0.01	0.00	no	1200.00	1200.00	-
s510	434	31	5.13	2.79	0.57	1.27	0.34	no	110.56	91.30	pass
s820	570	19	61.68	36.74	20.13	23.47	23.47	no	140.73	184.03	pass
s838	442	72	56.25	56.25	31.64	28.13	14.94	no	1200.00	1200.00	-
s832	577	22	73.33	42.77	26.88	8.86	5.01	no	179.89	157.05	pass
s953	466	39	84.38	35.01	19.54	7.40	2.84	yes	0.80	0.82	pass
s1423	896	163	100.00	53.58	100.00	20.59	-	no	1200.00	1200.00	fail
s1196	685	37	40.63	40.63	40.63	20.31	13.49	no	3.38	2.70	pass
s1238	784	37	43.75	23.44	12.89	23.44	3.55	no	5.08	3.27	pass
s1488	1217	54	47.03	8.76	1.10	1.10	0.24	no	360.64	262.92	pass
s1494	1236	23	54.27	19.09	14.87	11.91	10.25	no	114.10	161.11	pass
s5378	1963	421	1.93	11.06	1.93	1.93	2.64	no	21.51	1.40	pass
s9234	2675	497	82.03	41.02	-	100.00	-	no	7.97	0.25	fail
s13207.1	4321	1606	25.36	25.36	25.36	-	100.00	no	22.00	20.31	fail
s13207	3660	1831	2.81	2.81	2.81	2.81	2.81	no	2.97	2.80	pass
s15850	5331	1612	21.88	21.88	21.88	21.88	21.88	no	3.60	3.40	pass
s38417	16684	4041	-	100.00	100.00	100.00	100.00	no	88.01	0.24	fail
s38584.1	18705	4672	100.00	99.95	99.90	28.80	99.90	no	109.79	103.69	fail
s38584	20276	4747	100.00	100.00	100.00	100.00	100.00	no	16.89	31.35	fail

[1] Circuit statistics represent the product machine. Nodes reported are the number of and nodes in an and-inverter graph.
[2] Verification using only the invariant.
[3] 1200 second timeout.

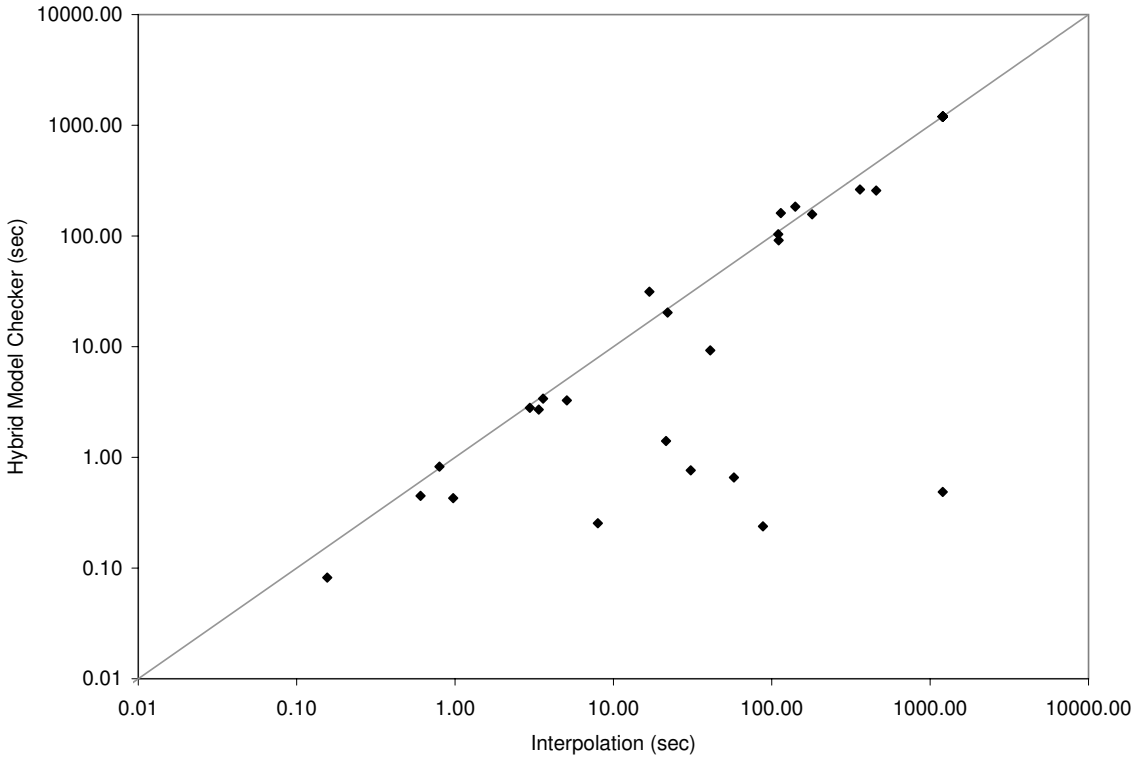


Figure 4.3: Scatter plot of the runtimes for the sequential equivalence problems of Table 4.1. Model checking was run with a timeout of 1200 seconds.

and undecidable properties shown in the table. Note that the retiming method we use does have bugs. Next we applied our hybrid model checker.

If the interpolation proves the property, and if the proof of unsatisfiability of the bounded model checking problem does not depend on the initial state, then we know that from any state, no bad states are reachable. This means that the invariant reduced the search space so much that model checking was trivial. In fact, the invariant is telling us that every bad state is unreachable. This phenomena is documented in the "Verification Complete" column located after the preprocessor columns in the table.

We see from the table that the use of the invariants significantly speeds up the interpolation. The runtimes from Table 4.1 are represented in the scatter plot shown in Figure 4.3. It is important to note here that the hybrid solver runtimes do not include the 25 second overhead introduced by preprocessing. This is a constant-time overhead that will be insignificant when this method is applied to industrial benchmarks. This preprocessing time was omitted from the table because we choose to focus on the runtime improvements within the interpolation procedure.

4.2.6 Experimental Results: Property Checking in an Industrial Environment

The final experiment in strengthening interpolation with invariants comes from industrial experience. The IBM formal verification tool *SixthSense* is used to prove safety properties on IBM microprocessor designs. While this tool successfully completes proofs on many designer-specified properties, the difficulty of some properties exceeds the capacity of the tool.

The invariant discovery techniques described in Chapter 2 were implemented inside *SixthSense*, and these invariants can be leveraged to strengthen interpolation.

To gauge the effectiveness of this setup, 91 challenging IBM designs were selected from across various microprocessor projects. Each design has a single property to be checked, and some quick analysis was done to ensure that the properties are not easily provable or falsified.

Steps were taken to further refine the 91 properties by discarding those properties that are solvable by existing techniques implemented in *SixthSense*. Bounded model checking (BMC) was applied to the designs, and this algorithm was given a 1 hour time limit. BMC was able to find counterexample traces for 8 of the 91 designs. Next, induction and interpolation were each applied to the remaining designs. The two algorithms were run one after the other, and each was given a 1 hour time limit[1]. Induction and interpolation were able to prove that 5 of the remaining properties are unsatisfiable. The remaining $91 - 8 - 5 = 78$ designs are unsolvable using the techniques currently implemented in *SixthSense*.

In an attempt to solve these remaining properties, invariant-strengthening interpolation was applied. An invariant discovery routine was run for 10 minutes, and the invariants that were found were then used to strengthen interpolation. The strengthened interpolation was again given a 1 hour time limit, and it was able to solve 4 (5.1%) of the outstanding properties.

This ability to solve previously-unsolvable properties demonstrates the value of invariant-strengthened interpolation. This technique extends the capacity of the *SixthSense* and increases its usefulness in the IBM microprocessor designs.

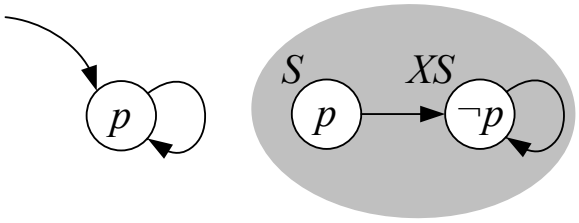


Figure 4.4: The shaded states are unreachable but will make an inductive proof of p impossible.

4.3 Strengthening Induction

4.3.1 Motivation

Induction (Section 14) is a method of proving that a property holds for all reachable states. It is easy to formulate and often executes quickly. It is an incomplete method but can be strengthened by the introduction of inductive invariants.

The technique is incomplete since there are properties that hold in every reachable state which simple induction will fail to prove. For example, Figure 4.4 shows a state transition graph on which a proof of p will fail. The shaded states are unreachable, but because of these unreachable states $\exists\, S$ and XS such that $S \models p$ and $XS \not\models p$. The inductive step fails.

Induction can be strengthened with knowledge about the reachable state set. If a reachability approximation is available that refutes the reachability of S or XS then induction will not hit this spurious counterexample, and the probability of proving the property with induction increases.

Here we consider invariants as a source of the reachability approximation necessary to strengthen induction. It is assumed that invariants have been discovered before induction is run, and with some minor modifications induction can be made to take

[1]The induction implemented in *SixthSense* will increase the induction k until either the property is found to be unsatisfiable or the time limit is reached.

advantage of these invariants.

4.3.2 Strengthening the Base Case

The base case of the inductive proof explores states reachable in a bounded number of steps from the design's initial state(s). By definition, all of these states are reachable and no reachability approximation can strengthen this computation. For this reason invariants cannot bound the set of states searched by the SAT solver during this computation.

4.3.3 Strengthening the Inductive Step

Invariants have a much stronger role in the inductive step of the inductive proof. The inductive step has the ability to explore unreachable states, so by using the proved invariants we reduce the state space that is explored in the inductive step.

The modified inductive step is shown in Algorithm 25. Similar to the base case, `checkProperty`(*node*, *cycle*, sym) defines the instance of *node* in a circuit initialized with a symbolic state and unrolled to depth *cycle*. This is used to construct the *invariantsHold* which is 1 if and only if all invariants hold in all time steps considered in the inductive step.

The signal *invariantsHold* is then conjoined with each satisfiability problem. This restricts the solver to only find counterexamples where *invariantsHold*. That is, each time step of the counterexample must contain a state in the approximately reachable states that are given by the set of proved invariants. This helps to dramatically reduce the number of spurious counterexamples.

```
1: function checkInductiveStep(design, candidates, inductionK)
2:   // check that all invariants hold in all cycles
3:   invariantsHold := 1
4:   for (cycle = 0 to inductionK) do
5:     for all (i in proved invariants) do
6:       invariantsHold := invariantsHold · checkProperty(i, cycle, sym)
7:     end for
8:   end for
9:
10:  // Iteratively weaken the inductive hypothesis until a fixed point is reached
11:  while (1) do
12:    inductiveHypothesis := 1
13:    for (cycle = 0 to inductionK - 1) do
14:      for all (c in candidates) do
15:        inductiveHypothesis := inductiveHypothesis · checkProperty(c, cycle,
           sym)
16:      end for
17:    end for
18:
19:    // Check that the candidates hold under this inductive hypothesis
20:    candidatesHold := 1
21:    for all (c in candidates) do
22:      if (satisfiable(¬checkProperty(c, inductionK, sym)· inductiveHypothesis ·
         invariantsHold) then
23:        candidates := candidates / {c}
24:        candidatesHold := 0
25:      end if
26:    end for
27:    if (candidatesHold) then
28:      break
29:    end if
30:  end while
31:
32:  return candidates
33: end function
```

Algorithm 25: Invariant-strengthened inductive step.

4.3.4 Experimental Results: Property Checking in an Industrial Environment

The ability of invariants to strengthen induction was evaluated using a setup identical to Section 4.2.6. 91 challenging IBM designs, each with a single property to be proved, were identified from a variety of microprocessor projects. These 91 properties were reduced to 78 by discarding properties that are solvable using existing techniques implemented in the IBM formal verification tool *SixthSense*. Each existing technique was given 1 hour, and the properties that remain unsolved after this rigorous setup represent properties that are currently impossible to prove using *SixthSense*.

The 78 remaining properties were then given to our implementation of invariant-strengthened induction. An invariant discovery routine, implemented using ideas from Chapter 2, was run for 10 minutes, and the resultant invariants were used to strengthen induction. This induction was given a 1 hour time limit and will continuously increase the induction k until either the property is solved or the time limit is reached.

Invariant-strengthened induction was able to solve 1 of the remaining 78 properties, or 1.3%. This represents an increase of the capacity of the *SixthSense* tool, and while the ability to solve any previously unsatisfiable problems is a noteworthy achievement, this number is a bit lower than expected. In comparison to Section 4.2.6, it seems that invariants are able to strengthen interpolation better than they are able to strengthen induction.

We anticipated more success with invariant-strengthened induction, and we suppose that perhaps we were not finding the right invariants necessary to help induction. Future research will include a wider variety of invariant families (Section 2.2) to find invariants more suited to helping induction.

4.4 Targeted Invariants

4.4.1 Motivation

Invariants are able to benefit verification by limiting the state space that must be explored. Verification on this reduced search space can be exponentially faster than the general case, and this enables verification to complete on some difficult safety problems that normally would not be possible to verify.

Unfortunately, while the discovery of invariants (Section 2.1.3) can be done in a resource bounded way, often discovering sufficient invariants that adequately help verification can take a very long time. Therefore the total code being executed, invariant discovery followed by verification, can be just as slow as the original unaided verification in the general case[2].

One possible solution to this problem is to generate only the invariants that are helpful to verification and to skip the others that a typical invariant discovery algorithm may find which may not be helpful to the verification at hand. This approach requires knowledge of the problem being verified, but this can be provided if the verification algorithm and invariant discovery algorithm are integrated. Such an approach can then find **targeted invariants** that provide key facts about the state space that are immediately helpful to the verification effort.

To harness this idea, we have built a tool that is able to show that a single, user-specified state is unreachable [Case *et al.*, 2007]. It does this by finding and proving inductive invariants. If for some reason it is unable to complete the proof of the invariants, it will find and prove other, secondary invariants that enable the first proof to proceed. This method gives a hierarchy of proofs that when complete will yield a set of inductive invariants $\{P\}$ with the following properties:

[2]Note that the invariants discovered can be re-used for multiple properties in the same design.

- $\bigwedge_{p \in \{P\}} p$ implies that the user-specified state is unreachable.
- $\bigwedge_{p \in \{P\}} p$ can be proved with simple induction

Such a tool can be easily integrated with a verification framework to provide targeted invariants.

This chapter provides the theory behind this invariant generation method and explores how targeted invariants can help interpolation.

4.4.2 Using Targeted Invariants in Verification

Induction and interpolation are two verification algorithms that can benefit from invariants (Sections 4.3 and 4.2). Here we examine these algorithms in more detail to identify the nature of the invariants that are most beneficial.

4.4.2.1 Simple Induction

It is well known that induction is an incomplete method in that there exist properties p that hold in all reachable states but are not provable with induction (Section 4.3.1).

```
1: // Let p be the property to be proved.
2: if (∃ initial state i, i ⊭ p) then
3:    return "falsified"
4: end if
5: if (∃ possibly reachable S, XS s.t. S ⊨ p, XS ⊭ p) then
6:    if (can prove S or XS unreachable) then
7:       record new invariants, goto 3 // See Section 4.4.3.3
8:    end if
9:    return "inconclusive"
10: end if
11: return "verified"
```

Algorithm 26: Modified simple induction.

To prove p in Figure 4.4, we may try k-step induction as described in [Bjesse and Claessen, 2000], but in [Case *et al.*, 2006b] we found this to be a very expensive

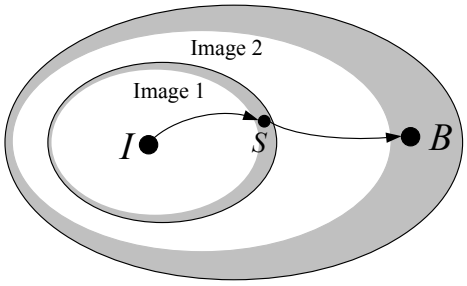


Figure 4.5: Interpolation has erroneously reached a bad state.

solution. Instead, suppose we are able to find an inductive invariant that shows that either S or XS in Figure 4.4 is unreachable. If known-unreachable states are disallowed from entering the inductive step (with an extra constraint on the SAT solver), then simple induction will be able to prove p. This is demonstrated in Algorithm 26 where the standard simple induction algorithm is improved by the addition of lines 4 and 5. What is needed is a tool that will generate specific inductive invariants that can demonstrate the unreachability of S or XS. Such a tool is described below.

4.4.2.2 Interpolation

Interpolation (Section 15 is a method that has been found useful for unbounded verification of safety properties. It works by using an over-approximation to the image operation. By applying this image operator iteratively starting from the initial state, either a fixed point is reached or a bad state is encountered. If a fixed point is reached, it is an over-approximation to the set of reachable states such that no bad states are contained in this approximation. The design is verified.

If an approximate image contains a bad state, interpolation may have found a counterexample. However there is in general no way to know if this is a real counterexample. Take for example Figure 4.5. Two image operations are applied to the initial state I, and a bad state B is found in the second image. The shaded region

represents the over-approximation inherent in the image operation, and the white shows the true image of the reachable states. We do not know which states of the image lie in the over-approximation, nor do we know if the over-approximated states are reachable. In Figure 4.5 we have the error trace $I \rightarrow S \rightarrow B$, but if either S or B lie in the over-approximation and is unreachable then the counterexample is spurious.

```
 1: // Let p be the property to be proved.
 2: set parameters for over-approximate image operator
 3: {S} := I
 4: while (1) do
 5:    {S}' := {S} ∪ approxImage({S})
 6:    if {S}' == {S} then return "verified"
 7:    if ∃ a bad state "near" {S} then
 8:       if {S} == I then return "falsified"
 9:       if can prove s unreachable then // See Section 4.4.3
10:          record new invariants, goto 7 // See Section 4.4.3.3
11:       tighten over-approximation parameters, goto 3
12:    end if
13:    {S} := {S}'
14: end while
```

Algorithm 27: Modified interpolation.

Unless specific conditions are met (line 8 of Algorithm 27), the interpolation algorithm has no way of knowing if a counterexample is spurious or true. Therefore, it discards all work up to that point and begins anew with a tighter approximation to the image operator. This restart is costly, and it is a major hot-spot in the performance of the algorithm.

In this work, we have augmented interpolation to call our targeted invariant tool as shown in Algorithm 27. On lines 9 and 10, it will find specific inductive invariants that imply that a state along the error trace is unreachable. If invariants are found, the error trace must be spurious, and interpolation is free to proceed without the costly restart on line 11.

4.4.3 The Proof Graph

4.4.3.1 Basic Definitions

Here we describe the basics of our tool to automatically find and prove useful inductive invariants. A graph structure called the **proof graph** is the core of our method.

The proof graph is a bipartite directed graph with the following node types:

- States in the sequential design. In practice this is a cube of states, but to simplify this discussion, consider only a single state. This constraint can be relaxed as discussed in [Case *et al.*, 2007].

- Sets of candidate invariants. These candidates are yet to be proved, but if we can prove them then they are invariants.

The root of the graph is a single state node. This corresponds to the user-specified state that should be proved unreachable. This root node comes from an outside source – in this work it is a state along the error trace in interpolation. The leaves, i.e. the nodes without outgoing edges, are candidate sets.

The meaning of the graph lies in its connectivity, specifically in the meanings of edges from states to candidates and from candidates to states. Being a bipartite graph, there are no other edge types.

Let a directed edge from a state S to a set of candidate invariants $\{C\}$ mean that:

$$\forall\, c \in \{C\}, \quad S \not\models c$$

That is, all candidates $\{C\}$ fail to hold in S.

The candidates $\{C\}$ may or may not hold in all reachable states, but if any such $c \in \{C\}$ can be proved then S is unreachable. We refer to $\{C\}$ as a set of **covering candidate invariants** for S.

Theorem 4.4.1 (Proving a State Unreachable)**.** *Let a candidate c fail in a state S ($S \not\models c$). If c is proved to hold in every reachable state then S is unreachable.*

Therefore the structure $S \rightarrow \{C\}$ provides a method to show S unreachable.

Let a directed edge from a set of candidates $\{C\}$ to a state S mean that:

$$\forall\, c \in \{C\}, \quad \exists \text{ a successor state } XS \text{ such that } (c \models S) \wedge (c \not\models XS)$$

That is, all candidates hold in S but fail in a successor state of S.

In the structure $\{C\} \rightarrow S$, S is the reason that the inductive proof of $\{C\}$ was not successful. In fact, S is the counterexample to the inductive hypothesis of the proof.

Proving S to be unreachable is a necessary but not sufficient condition for proving a $c \in \{C\}$. Clearly, the proof of $c \in \{C\}$ cannot succeed if S may be reachable. Conversely, if S is known to be unreachable, we have no evidence that a proof of $c \in \{C\}$ will fail. However, another counterexample S' may exist.

Example 23. An Example Proof Graph

Figure 4.6 shows an example of a proof graph and how it might evolve over time as our algorithm is run. A sample execution is given here.

1. *Suppose interpolation reaches a bad state and S_0 is a state on the error trace. We would like to show that S_0 is unreachable. (See Section 4.4.2.2.)*
2. *Our tool is called to prove that S_0 is unreachable. We set S_0 to be the root of our graph, and through simulation we generate the covering candidate invariants $\{C_0\}$. In this simulation, we select candidates that appear to hold in every reachable state but fail in S_0. This gives us Graph (1) in Figure 4.6.*

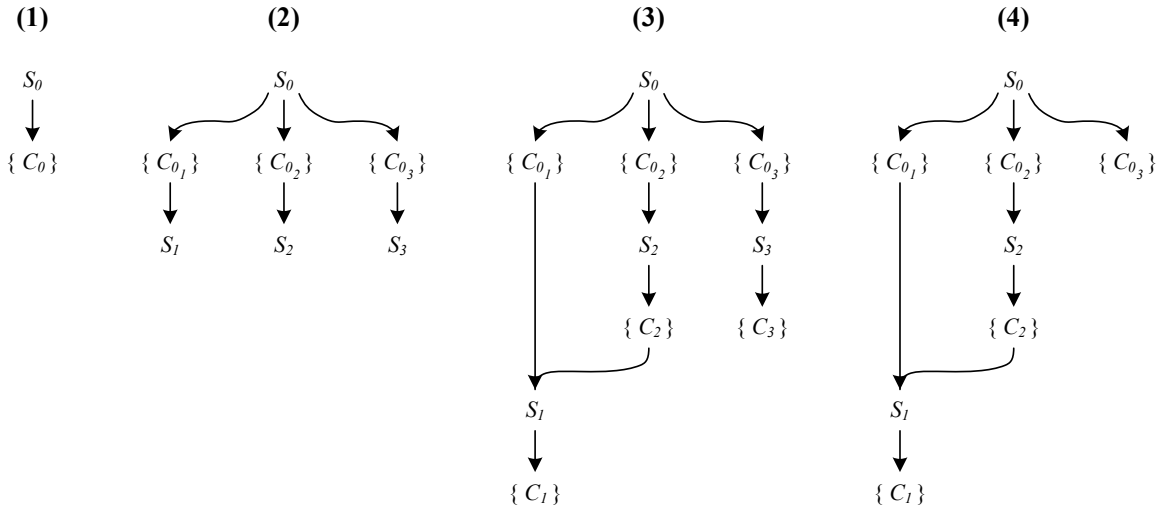


Figure 4.6: Sample evolution of a proof graph over time.

3. *We attempt to prove the candidates* $\{C_0\}$ *by simple induction. Suppose that this proof fails, and there are three counterexamples* S_1*,* S_2*, and* S_3 *in the inductive hypothesis. Each counterexample is responsible for disproving a subset of* $\{C_0\}$*, and the proof technique as implemented in [Case* et al.*, 2006b] will result in these subsets being pair-wise disjoint. We therefore split* $\{C_0\}$ *into* $\{C_{0_1}\}$*,* $\{C_{0_2}\}$*, and* $\{C_{0_3}\}$ *such that:*

   - $\{C_{0_1}\} \cap \{C_{0_2}\} = \emptyset$, $\{C_{0_1}\} \cap \{C_{0_3}\} = \emptyset$, $\{C_{0_2}\} \cap \{C_{0_3}\} = \emptyset$
   - $\{C_{0_1}\} \cup \{C_{0_2}\} \cup \{C_{0_3}\} = \{C_0\}$
   - $\forall\, j \in \{1, 2, 3\}$, S_j *causes the inductive proof of* $\{C_{0_j}\}$ *to fail.*

   Recording this information in the proof graph gives us Graph (2) in the figure. The existence of these counterexamples does not imply that the candidates $\{C\}$ *are not true but instead that we need more invariants to prove them. (See Sections 4.4.2.1 and 4.4.3.4.)*

4. *Next, cover* S_1*,* S_2*, and* S_3 *with candidates, giving us* $\{C_1\}$*,* $\{C_2\}$*, and*

$\{C_3\}$ *respectively. These candidates provide a way to show that the new states are unreachable. (This is similar to Step 2.)*

5. *By simulating each new state, we can check if it might be responsible for a future failure to prove any of the candidate sets. Suppose we find that* S_1 *is a counterexample in the inductive proof of every* $c_2 \in \{C_2\}$*. This induces the edge* $\{C_2\} \rightarrow S_1$*, and the result is Graph (3) in the figure.*

6. *We attempt to prove the candidates* $\{C_3\}$*. Suppose we find at least one property to be true for all reachable states. Now* S_3 *is known to be unreachable, and it can be removed from the proof graph. This gives Graph 4. (See Section 4.4.3.3.)*

7. *Attempt the proof of* $\{C_{0_3}\}$ *again. This proof was first attempted in Step 3, but now the reason that the original proof failed,* S_3*, is gone and we may re-attempt the proof. Suppose that this time we find that at least one candidate invariant holds for all reachable states. This implies that* S_0*, the root node, is unreachable. At this point, we have achieved our objective and we may return the new invariants to the calling routine, interpolation in this case. (See Section 4.4.3.5.)*

4.4.3.2 Selecting Which Candidate Invariants to Prove

The proof graph in general contains several candidate set nodes, and the tool must pick one single node to attempt as the next proof. Selecting that node is fairly simple once some basic properties of the proof graph are explored.

Theorem 4.4.2 (Proofs on Leaves Only)**.** *Given the sets of candidates* $\{C_0\}$ *and* $\{C_1\}$*, a state* S*, and the graph structure* $\{C_0\} \rightarrow S \rightarrow \{C_1\}$*. If no* $c_1 \in \{C_1\}$ *are proved to hold in every reachable state then* $\forall\, c_0 \in \{C_0\}$*, it is not possible to prove* c_0 *by simple induction.*

Proof. If $\exists\, c_1 \in \{C_1\}$ that has been proved, then that would be a guarantee that S is unreachable. However, because no such proved invariants exist, the reachability of S is unknown. To be conservative, we must allow S to be a counterexample in the inductive step of the proof of the candidates $\{C_0\}$. Therefore, the proof will fail $\forall\, c_0 \in \{C_0\}$. □

The above theorem defines an order in which the proofs must be attempted. Specifically, if a candidate node has an outgoing edge to a state then any proof attempt is in vain. In a chain of the graph, only the leaves (nodes without outgoing edges) may be considered as for a proof attempt. The situation is a bit more complex for a cycle however.

Theorem 4.4.3 (Cycles in The Graph)**.** *Suppose there are candidate invariant sets* $\{C_0\}, \ldots, \{C_n\}$*, unique states* $S_0, \ldots, S_n$*, and the cyclic graph structure* $\{C_0\} \rightarrow S_0 \rightarrow \cdots \rightarrow \{C_n\} \rightarrow S_n \rightarrow \{C_0\}$*.*

If $\exists\, j \in \{0, \ldots, n\}$ *such that* $\forall\, c_j \in \{C_j\}$, c_j *cannot be proved by simple induction then* $\forall\, k \in \{0, \ldots, n\}, \forall\, c_k \in \{C_k\}$, c_k *cannot be proved by simple induction*

Proof. The failure to prove $\{C_j\}$ results in not being able to prove $\{C_{(j-1 \mod n)}\}$ by Theorem 4.4.2. This establishes a base case of the inductive proof of this theorem.

Now let $k \in \{0, \ldots, n\}$ and suppose $\{C_{(k+1 \mod n)}\}$ cannot be proved. By Theorem 4.4.2, $\{C_k\}$ cannot be proved. Theorem 4.4.3 is now proved by induction. □

Theorem 4.4.3 says that in a cycle with n candidate set nodes, we must attempt to prove the union of all the candidate sets at the same time. This simple induction will either successfully prove $\geq n$ properties or 0 properties because if any candidates hold for all reachable states then at least one candidate invariant in each set must be true.

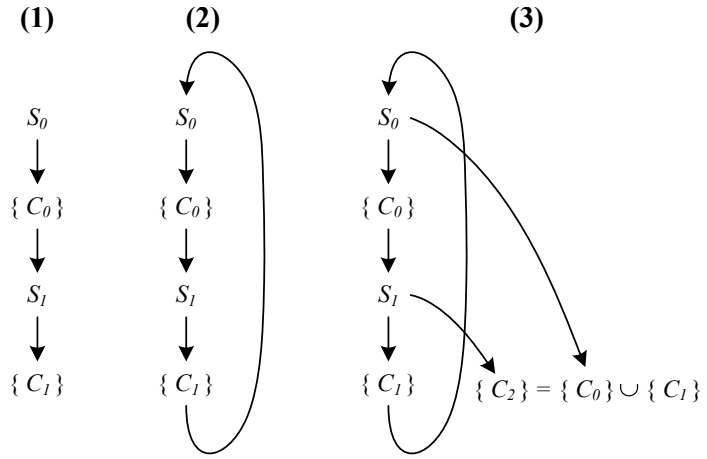


Figure 4.7: A cycle has developed in the proof graph.

Cycles must be treated differently from leaves in that the union of the cycle nodes must be proved simultaneously. However, this can be generalized as illustrated in the following example:

Example 24. Cycles in the Proof Graph

1. *Suppose the current proof graph is that shown in Graph (1) of Figure 4.7.*
2. *Suppose we find that S_0 can act as the counterexample in the inductive hypothesis for all $c_1 \in \{C_1\}$. This induces a cycle in the graph as shown in Graph (2).*
3. *Now create a new leaf node $\{C_2\} = \{C_0\} \cup \{C_1\}$ and insert it into the proof graph. This records the following information:*
   - *Both $\{C_0\}$ and $\{C_1\}$ must be proved at the same time.*
   - *A successful proof will imply that both S_0 and S_1 are unreachable.*

   The updated proof graph is shown in Graph (3).

If cycles are abstracted as illustrated in Figure 4.7 then a proof of the union of the candidate sets in the cycle is equivalent to a proof of the new leaf node. After this

generalization is made, only the leaves in the proof graph are eligible for an inductive proof.

Suppose a proof graph has multiple leaves, and one leaf must be selected for the simple induction prover. The unique leaf to be given to the induction engine is selected as follows:

- Let $d(\cdot)$ denote the distance from the root node to a candidate invariant set node. That is, $d(\{C\})$ is the number of edges in the shortest directed path from the root node to $\{C\}$. Given this metric, the candidate set $\{C\}$ which minimizes $d(\{C\})$ should be selected because it requires fewer inductive proofs to achieve the overall goal, to prove that the root node is unreachable.

- Ties should be broken by selecting the $\{C\}$ with the greatest cardinality. In the absence of other information about the design, this property set has the greatest chance of having at least one property successfully proved.

4.4.3.3 Upon a Successful Proof

Suppose a candidate invariant set $\{C\}$ in the proof graph has been selected, given to the simple induction prover, and a candidate $c \in \{C\}$ has been proved (and is therefore an invariant). This proof is independent from any assumptions, and the proved invariant is guaranteed to hold for all reachable states. The property is therefore used to simplify all future problems, both simple induction and interpolation. Enforcement of the invariant can be accomplished by the addition of constraint clauses in each respective SAT instance (Sections 4.2 and 4.3). This extra clause is maintained throughout the remainder of the execution, effectively utilizing the new invariant in all future problems.

This successful proof also allows the proof graph to be pruned. Theorem 4.4.1 implies that all state node parents of $\{C\}$ are now known to be unreachable. These

can be removed from the proof graph, along with any dangling nodes that result. This can create new candidate set leaves in the graph, enabling the proofs of some candidate sets to be re-attempted. This happened in Step 6 of the example shown in Example 23.

4.4.3.4 Upon an Unsuccessful Proof

Suppose in attempting to prove $\{C\}$, the simple induction engine fails to prove any of the candidates. From Section 4.4.2.1 we know that we can help simple induction by proving that the counterexample state that satisfies the inductive hypothesis is unreachable.

In failing to prove a set of candidates, simple induction will produce a set of counterexamples $\{S_0, S_1, \ldots, S_n\}$. In this case, for each $c \in \{C\}$, $\exists\, S_j$ such that S_j is the reason the inductive proof of c failed.

To accurately record the relationship between the properties $\{C\}$ and the states $\{S_0, S_1, \ldots, S_n\}$, $\{C\}$ must be split into n subsets, one for each of $\{S_0, S_1, \ldots, S_n\}$. We modify the proof graph by splitting $\{C\}$, adding $\{S_0, S_1, \ldots, S_n\}$ along with edges to demonstrate the failed proofs, and lastly we find covering candidate invariants $\{C_j\}$ for each new S_j. This is illustrated in Steps 3 and 4 of the example in Example 23.

One may view each new structure $S_j \rightarrow \{C_j\}$ as a subgraph rooted at S_j, the state that the induction engine wants to have shown unreachable. In this way, proving unreachable states for use in the simple induction solver is a sub-problem in the task of proving unreachable states for interpolation.

4.4.3.5 Termination Conditions

The above process describes a proof graph that grows as counterexamples are discovered in inductive proofs and shrinks as candidate invariants are successfully proved.

The proof graph oscillates in size until one of two termination conditions are satisfied:

- If we run out of proof candidates, the overall proof is impossible. This can happen if at some point there are no more leaves in the proof tree. In practice, this means that either the root state was in fact reachable or our candidate invariants did not provide sufficient information to show this. Absence of leaves is not a guarantee that the root state is reachable.
- If a covering candidate invariant of the root node is proved to hold for all reachable states then the proof graph algorithm will remove the root from the graph. If the root has been deleted, we can conclude that the root has been proved unreachable and we may stop.

```
1: // Let S_0 be the state to prove unreachable
2: root := S_0
3: cover root with properties
4: while (1) do
5:   if (root == ∅) then return "root unreachable"
6:   if (no leaves) then return "root may be reachable"
7:   {P} := selectBestLeaf()
8:   (proved, {S}) := simpleInductionProve({P})
9:   if proved then
10:    delete parents of {P}
11:  else
12:    for all counterexamples s ∈ {S} do
13:      make new proof graph node for s
14:      cover s with a new set of properties {P}
15:      Simulate S, try to break proofs of all property sets
16:      update proof graph
17:    end for
18:  end if
19: end while
```

Algorithm 28: The proof graph algorithm.

Table 4.2: Performance On A Sampling of Hard Academic Problems

Design Properties				Standard Interpolation			Interpolation + Proof Graph				
Design	ANDs	Latches	Property	MB	Sec.	Refines	MB	Sec.	Refines	Candidates	Invariants
cmu_dme1_B	236	61	?	2484	7200	5	2487	7200	5	390	0
cmu_dme2_B	296	63	?	2507	7200	7	2674	7200	7	604	0
eijk_S1423_S	902	159	True	2481	7200	1	139	77.93	0	10078	2400
eijk_S208_S	109	22	True	2451	7200	4	38	60.62	0	1668	454
eijk_S208c_S	111	23	True	2469	7200	7	30	59.04	0	1864	660
eijk_S382_S	230	57	True	2480	7200	5	228	102.17	0	23144	4176
eijk_S420_S	243	50	True	2500	7200	7	148	191.27	0	11000	2250
eijk_S444_S	240	57	True	2491	7200	5	224	507.37	0	38530	28972
eijk_S838_S	480	106	True	2570	7200	2	1199	370.63	0	152734	27308
eijk_bs1512_S	866	158	?	2471	7200	0	2475	7200	0	46108	60
eijk_bs3271_S	1841	305	?	1822	7200	1	2514	7200	1	6544	772
irst_dme4_B	593	124	?	2515	7200	4	2558	7200	4	1894	0
irst_dme5_B	790	165	?	2562	7200	5	528	7200	4	16590	0
irst_dme6_B	1181	245	?	2564	7200	5	440	7200	0	126850	0
nusmv_brp_B	375	52	?	2467	7200	1	805	7200	1	9140	1128
nusmv_queue_B	1310	84	True	2459	7200	1	95	151.78	0	3690	480
nusmv_reactor_6_C	903	76	True	2478	7200	7	52	140.66	0	6228	482
vis_bakery_E	284	25	?	2454	7200	3	2582	7203.06	5	4018	626

4.4.4 Experimental Results

For this work, we implemented two C++ plugins for the ABC Logic Synthesis and Verification System [Synthesis and Group, 2008]. The first plugin implements the interpolation algorithm as described in [McMillan, 2003], and the second plugin implements the invariant discovery method proposed in this paper. The plugins are interfaced as described in Algorithm 27 to provide inductive invariants for aiding interpolation.

We experimented with a suite of 154 academic designs that had been annotated with safety properties [Chalmers, 2007]. The designs ranged in size from 10 to 689 latches. After the designs were combinationally synthesized into And-Inverter Graphs, they had between 43 and 3716 And nodes. Each design in this benchmark suite contains a single safety property, which include 95 true properties, 34 false properties, and 25 properties of unknown nature.

The technique described in this paper can greatly speed-up model checking, but also it imposes some overhead to find and prove inductive invariants. The algorithm is best suited to run as an option in a verification package that can be invoked after more conventional methods have been exhausted. To emulate this type of flow, we attempted to verify all 154 benchmarks with standard interpolation. 132 finished in

less than 10 minutes, and 18 failed to verify in 2 hours. It is on these 18 that we then applied our method.

Table 4.2 shows these 18 benchmarks on which interpolation times-out after 2 hours. As discussed in Section 4.4.2.2, the most expensive part of the interpolation algorithm is the model refinement. The number of refinements done in the standard interpolation algorithm is shown in the table, and in some cases this number is quite high.

If specific inductive invariants can be found, then refinement can be avoided. In the last part of the table, we show the statistics for an implementation of interpolation that utilizes the proof graph. Whenever interpolation reaches a bad state, it will find and prove appropriate inductive invariants. In half of the designs, proving a small number of candidate invariants was sufficient to allow all model refinement steps to be skipped. Runtime was dramatically improved in those cases, and the inductive invariants proved to be the difference between a time-out and a successful verification run.

Interestingly, no false properties are present in Table 4.2. The technique presented here favors true properties because for these, any trace into a bad state must contain unreachable states and so there is an opportunity to find invariants to cover those states. With a falsifiable property, the error trace will only contain reachable states and the proof graph method will waste resources attempting to show that these states are unreachable. Consequently, if there are indeed false properties in this subset of the benchmark suite, these probably appear as time-out cases.

Further experimentation was done using 43 industrial designs. Each contains multiple properties which are to be proved one-by-one. These designs were chosen because they are the hardest available, each having runtimes in excess of 8 minutes when processed with standard interpolation.

Figure 4.8 shows a scatter plot comparison of standard interpolation versus inter-

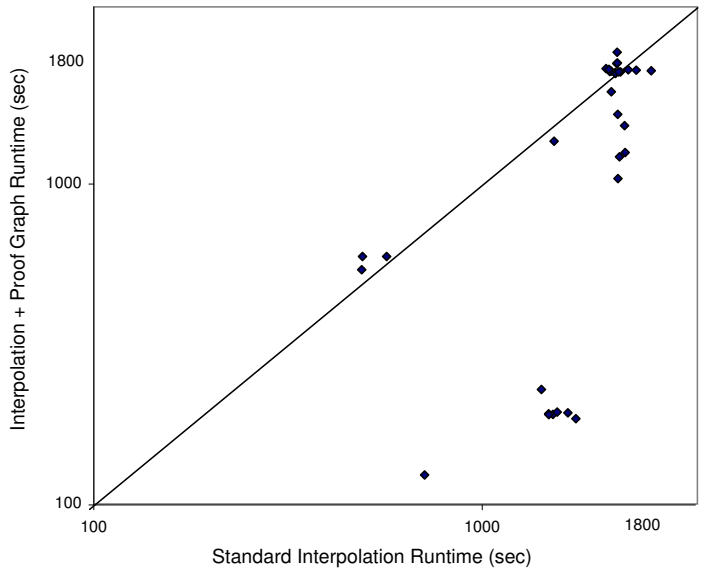


Figure 4.8: Performance On A Sampling of Hard Industrial Problems

polation aided with the proof graph. Each verification run was given a time-out of 30 minutes, and the proof graph successfully prevented 5 of the designs from exceeding the time limit. Even on designs that did not time-out with standard interpolation, using the proof graph significantly helped the runtime.

4.4.5 Conclusions

The use of targeted invariants with the proof graph is appealing because it enables the large space of candidate invariants to be reduced to only those candidates that immediately help the verification effort. This improves the runtime of the invariant discovery pass, making the combined runtime of invariant discovery and verification significantly faster than unaided verification in isolation. Additionally, because there are fewer candidate invariants and thus fewer proved invariants, memory structures needed to handle a large number of invariants (Section 2.6) can be simplified.

The work so far in targeted invariants is a good start, and in future work we would

like to apply the concept to many other domains besides just interpolation.

4.5 Verification of Sequential Synthesis

Invariants can be used to approximate the set of reachable states, and often this reachability approximation is very intricate. There exist states that invariants are able to show unreachable that other reachability approximations have difficulty in excluding from the set of reachable states. For this reason, if the set of reachable states is being approximated by some technique other than invariants then invariants should be used to complement this technique because invariants can tighten the reachability approximation in new ways.

The intricacy of an invariant-derived reachability approximation is problematic if one is trying to verify the results of sequential synthesis. Suppose invariants have been derived and used for indirect sequential synthesis. The synthesis algorithm(s) may use the intricate reachability information to perform design optimizations, and when performing a sequential equivalence check between the pre- and post-synthesis designs similar reachability information is probably needed to verify that the designs do not differ on a reachable state. Because there are states that are excluded from the invariant-derived reachability approximation, if these states were leveraged for sequential synthesis then verifying the sequential equivalence will be difficult since it will be difficult to verify that these states are unreachable.

The solution is to use invariants to strengthen the verification. The intricate reachability relationships that were leveraged in synthesis can be independently re-discovered in verification and used to complete the SEC proof. In this way, while the use of invariants for indirect sequential synthesis can make verification difficult, applying invariants to verification can reduce the difficulty of SEC to its original level.

Therefore, the use of invariants for verification can be thought of as dual for using

invariants in synthesis.

4.6 Conclusion

Invariants have been shown to be very useful in verification. In this chapter, the unbounded verification methods induction and interpolation were explored. Each can take advantage of invariants to prune the state space that is searched. We discussed two major advantages of this approach:

- Verification algorithms can be sped up considerably if invariants are considered. The reason is that the search space is reduced, and more importantly, the number of spurious counterexamples is reduced.
- Verification on some difficult properties fails to converge within reasonable time limits unless invariants are utilized. In this way, invariants are enabling these difficult properties to be verified and extending the scalability of formal verification.

Although invariants can help verification algorithms, the discovery of invariants can be quite expensive. For this reason, we also proposed a method (Section 4.4) to find a small number of invariants that are immediately helpful to the verification problem. This allows the invariant discovery routine to focus on the most valuable invariants and ignore invariants that consume runtime and are not ultimately useful.

Chapter 5

Conclusion

5.1 Summary of Ideas Discussed

This thesis focused on invariants, small properties that can be automatically formulated and proved. The methods by which invariants can be derived and efficiently proved to hold on all reachable states was discussed in Chapter 2. Invariants have two major applications: they provide a set of sequential don't cares that can be used for synthesis, and they provide an approximation to the set of reachable states that can be exploited for verification.

Chapter 3 discussed the synthesis applications of invariants. If an invariant expresses that a node is constant in all reachable states or that two nodes are equal in all reachable states then the design can be directly simplified. Otherwise, the invariants can be used as a source of sequential don't cares in indirect sequential synthesis. This type of synthesis gives more reductions than its combinational synthesis cousin. Additionally, invariants can be used to strengthen an existing sequential synthesis, giving rise to a synthesis algorithms have more power to reduce the size of the design than any other algorithm in its family.

Chapter 4 discussed the verification applications of invariants. Invariants can be used to form a reachability approximation, and verification algorithms can be augmented to only explore states that lie inside this reachability approximation. In the case of induction and interpolation, this means that fewer spurious counterexamples are explored, and this enables the algorithms to either terminate faster or, in the case of extremely challenging verification problems, complete the verification effort before exhausting computational resources, thus expanding the scalability of formal verification.

5.2 Weaknesses of Invariants and Their Solutions

Clearly invariants have many benefits. However, there are several deficiencies that this thesis has attempted to address.

Problem: The number of candidate invariants can be quite large, and their corresponding proof can be quite slow.

Solution: Chapter 2 presented numerous techniques such as narrowing the candidate invariants to those that are most interesting and then selecting a fast-running proof technique. Much effort was spent in developing simulation methods to efficiently reason about a large number of candidate invariants.

Problem: Invariants can be expensive to compute, and an invariant discovery + indirect sequential synthesis setup can be too slow.

Solution: Chapter 3 described the benefits of indirect sequential synthesis (and invariant-enhanced direct sequential synthesis) and demonstrated that although the runtime is increased, these methods offer significant reductions in the size of the design post-synthesis. Additionally, a single invariant discovery pass can

be used for multiple indirect sequential synthesis algorithms, and therefore the cost of the invariant discovery can be amortized.

Problem: While invariants can be used to reason that certain states are unreachable, the set of states shown to be unreachable may not be the right set that enables verification to complete quickly.

Solution: Chapter 4 discusses techniques to not only strengthen verification using invariants but also how to limit the invariants that are found to only those that can immediately help the current verification problem. This successfully overcomes the problem of generating valid invariants that do not successfully help verification.

5.3 An Ideal Invariant-Strengthened Verification Recipe

By considering the strengths and weaknesses of invariants, a robust invariant-strengthened verification scheme can be developed. **Transformational Based Verification** (**TBV**) [Kuehlmann and Baumgartner, 2001] is currently the best known way to verify complex safety properties in industrial gate-level designs. TBV is successful because it combines synthesis and verification in a way that simplifies the design as much as possible before attempting to verify the property. This allows verification to interact with a smaller design and thus increases the probability that the verification will come to a conclusion before computational resources are exhausted. By adding invariants into TBV, using ideas presented in this thesis, an even more robust verification framework can be developed.

Consider the invariant-strengthened TBV framework shown in Algorithm 29. First, the design is synthesized as much as possible. Next, heavy-weight verification

```
 1: function verifySafetyProperty(design, property)
 2:   if (property = constant 1) then
 3:     return "verified"
 4:   else if (property = constant 0) then
 5:     return "falsified"
 6:   end if
 7:
 8:   // Synthesize the design, leveraging invariants
 9:   invariants := ∅
10:   while (synthesis reduces design size) do
11:     invariants += findInvariants(design, invariants)
12:     design := synthesis(design, invariants)
13:     if (property = constant 1) then
14:       return "verified"
15:     else if (property = constant 0) then
16:       return "falsified"
17:     end if
18:   end while
19:
20:   // Attempt traditional verification, leveraging invariants
21:   while (1) do
22:     { spuriousCounterexamples, result } := verify(design, property, invariants)
23:     if (result = "timeout") then
24:       for all (cex in spuriousCounterexamples) do
25:         invariants += findTargettedInvariants(design, cex, invariants)
26:       end for
27:     else
28:       return result
29:     end if
30:   end while
31: end function
```

Algorithm 29: Invariant-centric verification flow.

algorithms are called to prove the property. Invariants are integrated throughout this flow.

In synthesis: Invariants are discovered and used for both indirect sequential synthesis and invariant-enhanced direct sequential synthesis. Additionally, this invariant + synthesis sequence can be repeated until there are no further reductions in the design size. Each call to the invariant discovery routine 1) sees a different version of the design (due to the synthesis that has occurred) and so will generate different invariants, 2) will be able to prove more invariants because the internal proof of the candidate invariants can be strengthened with all the previously found invariants, and 3) can be made fast by only considering candidate invariants that will refine the reachability approximation supplied by the previously proved invariants. In practice, this invariant + synthesis loop can dramatically reduce the size of the design being verified.

In verification: The invariants found for synthesis are re-purposed and leveraged a second time for verification. This allows verification to run more quickly than it otherwise would, and it increases the probability that verification will reach a conclusive result before the time limit expires. If the invariants are not strong enough and the method does timeout then a list of spurious counterexamples that were seen in the course of verification can be obtained. Invariants can be found that target these spurious counterexamples, and in this way the reachability approximation can be refined in a way that is beneficial to verification. Verification is then called again, and this loop should result in reachability being approximated to the degree necessary to verify the property quickly.

Clearly, Algorithm 29 is similar to traditional TBV, but invariants have been integrated throughout. While this idea is untested at this time, I suspect that invariants will make TBV significantly stronger and allow it to complete verification both on

harder properties and in larger designs. This will increase the industrial usage of formal verification.

5.4 Future Work

Many ideas pertaining to invariants, synthesis, and verification were explored in this thesis, but this should be viewed as the beginning of research into invariants and not the end. I have several ideas for future work involving invariants.

- I wish to implement the invariant-strengthened TBV algorithm described in Algorithm 29.
- I plan to investigate other invariant families in order to see if there exists a family that 1) approximates reachability well, and 2) is easy to prove the candidate invariants.
- I plan to apply invariants to strengthen even more synthesis and verification algorithms.

Bibliography

[Amla *et al.*, 2005] N. Amla, X. Du, A. Kuehlmann, R.P. Kurshan, and K. McMillan. An analysis of sat-based model checking techniques in an industrial environment. *CHARME*, 2005.

[Baumgartner and Kuehlmann, 2001] J. Baumgartner and A. Kuehlmann. Min-area retiming on flexible circuit structures. *ICCAD*, 2001.

[Baumgartner, 2006] J. Baumgartner. Integrating fv into main-stream verification: The ibm experience. *FMCAD*, 2006.

[Bjesse and Claessen, 2000] P. Bjesse and K. Claessen. Sat-based verification without state space traversal. *FMCAD*, 2000.

[Bjesse and Kukula, 2005] P. Bjesse and J. Kukula. Automatic generalized phase abstraction for formal verification. *ICCAD*, 2005.

[Brand, 1993] D. Brand. Verification of large synthesized designs. *ICCAD*, 1993.

[Brand, 2007] Dan Brand. Conversations with dan brand, July 2007.

[Brayton *et al.*, 1987] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. Mis: A multiple-level logic optimization system. *TCAD*, 1987.

[Bryant, 1992] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 1992.

[Burkhard and Keller, 1973] W.A. Burkhard and R.M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 1973.

[Cabodi *et al.*, 2006] G. Cabodi, M. Murciano, S. Nocco, , and S. Quer. Stepping forward with interpolants in unbounded model checking. *ICCAD*, 2006.

[Case and Brayton, 2007] M.L. Case and R.K. Brayton. Maintaining a minimum equivalent graph in the presence of graph connectivity changes. Technical report, UC Berkeley, 2007.

[Case and Lwin, 2006] M.L. Case and K. Lwin. A hybrid model checker. Technical report, UC Berkeley, 2006.

[Case *et al.*, 2006a] M.L. Case, R.K. Brayton, and A. Mishchenko. A practical way to maintain a transitive reduction. Technical report, UC Berkeley, 2006.

[Case *et al.*, 2006b] M.L. Case, A. Mishchenko, and R.K. Brayton. Inductively finding a reachable state space over-approximation. *IWLS*, 2006.

[Case *et al.*, 2007] M.L. Case, A. Mishchenko, and R.K. Brayton. Automated extraction of inductive invariants to aid model checking. *FMCAD*, 2007.

[Case *et al.*, 2008a] M.L. Case, V.N. Kravets, A. Mishchenko, and R.K. Brayton. Merging nodes under sequential observability. *DAC*, 2008.

[Case *et al.*, 2008b] M.L. Case, A. Mishchenko, and R.K. Brayton. Cut-based inductive invariant computation. *IWLS*, 2008.

[Case *et al.*, 2008c] M.L. Case, A. Mishchenko, R.K. Brayton, J. Baumgartner, and H. Mony. Invariant-strengthened elimination of dependent state elements. *FMCAD*, 2008.

[Chalmers, 2007] Chalmers. *Property Checking Benchmark Suite*, 2007. `http://www.cs.chalmers.se/~een/Tip/`.

[Craig, 1957] W. Craig. Linear reasoning: A new form of the herbrand-gentzen theorem. *J. Symbolic Logic*, 1957.

[Devadas *et al.*, 1994] S. Devadas, A. Ghosh, and K. Keutzer. *Logic Synthesis*. McGraw-Hill, 1994.

[Een and Sorensson, 2008] Niklas Een and Niklas Sorensson. *MiniSat*, 2008. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat`.

[Goldberg *et al.*, 2001] E. Goldberg, M.R. Prasad, and R.K. Brayton. Using sat in combinational equivalence checking. *DATE*, 2001.

[Hassoun and Sasao, 2002] S. Hassoun and T. Sasao. *Logic Synthesis and Verification*. Kluwer Academic Publishers, 2002.

[IEEE, 2008] IEEE. *IEEE Standard Verilog Hardware Description Language*, 2008. `http://www.verilog.com/IEEEVerilog.html`.

[Intel Corporation, 2001] Intel Corporation. Intel pentium 3 tualatin in 0.18 μm, 2001. `http://www.sandpile.org/impl/pics/intel/p3/die_018.jpg`.

[Jiang and Brayton, 2004] J.H. Jiang and R.K. Brayton. Functional dependency for verification reduction. *CAV*, 2004.

[Kroening and Strichman, 2003] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. *VMCAI*, 2003.

[Kuehlmann and Baumgartner, 2001] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. *CAV*, 2001.

[Kuehlmann and Krohm, 1997] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. *DAC*, 1997.

[Kunz *et al.*, 1997] W. Kunz, D. Stoffel, and P. Menon. Logic optimization and equivalence checking by implication analysis. *ICCAD*, 1997.

[Kunz, 1993] W. Kunz. Hannibal: An efficient tool for logic verification based on recursive learning. *ICCAD*, 1993.

[Lee *et al.*, 2007] C. Lee, J. Jiang, C. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental sat solving. *ICCAD*, 2007.

[Leiserson and Saxe, 1991] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 1991.

[Lu *et al.*, 2003] F. Lu, L.C. Wang, K.T. Cheng, and R.C.Y. Huang. A circuit sat solver with signal correlation guided learning. *DATE*, 2003.

[McMillan, 2003] K.L. McMillan. Interpolation and sat-based model checking. *CAV*, 2003.

[Microsystems, 2008] Sun Microsystems. *Processor Technology Resources - picoJava*, 2008. `http://www.sun.com/software/communitysource/processors/picojava.xml`.

[Mishchenko *et al.*, 2006] A. Mishchenko, S. Chatterjee, and R.K. Brayton. Dag-aware aig rewriting: A fresh look at combinational logic synthesis. *DAC*, 2006.

[Mony *et al.*, 2004] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. *FMCAD*, 2004.

[Mony *et al.*, 2005] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. Exploiting suspected redundancy without proving it. *Design Automation Conference*, 2005.

[Nicely, 2008] Thomas R. Nicely. Pentium fdiv flaw faq. Technical report, Lynchburg College, February 2008. http://www.trnicely.net/pentbug/pentbug.html.

[OSCI, 2008] OSCI. Open systemc initiative, 2008. http://www.systemc.org/home.

[Plaza *et al.*, 2007] S. Plaza, K.H. Chang, I. Markov, and V. Bertacco. Node mergers in the presence of don't cares. *ASPDAC*, 2007.

[Prasad *et al.*, 2005] M.R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *Software Tools for Technology Transfer*, 2005.

[Pudlák, 1997] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 1997.

[Savoj and Brayton, 1990] H. Savoj and R.K. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. *Design Automation Conference*, 1990.

[Schuppan and Biere, 2006] V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 2006.

[Somenzi, 2005] Fabio Somenzi. *CUDD: CU Decision Diagram Package, Release 2.4.1.* University of Colorado at Boulder, May 2005. http://vlsi.colorado.edu/~fabio/CUDD.

[Steen and Seebach, 1970] L.A. Steen and J.A. Seebach. *Counterexamples in Topology*. Courier Dover Publications, 1970.

[Synthesis and Group, 2008] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*, 2008. http://www.eecs.berkeley.edu/~alanmi/abc.

[van Eijk, 2000] C.A.J. van Eijk. Sequential equivalence checking based on structural similarities. *TCAD*, 2000.

[Wedler *et al.*, 2004] M. Wedler, D. Stoffel, and W. Kunz. Exploiting state encoding for invariant generation in induction-based property checking. *ASP-DAC*, 2004.

[Wikipedia, 2008] Wikipedia. *Age of the Universe*, October 2008. http://en.wikipedia.org/wiki/Age_of_the_universe.

[Wikipedia, 2009] Wikipedia. *VHSIC hardware description language*, 2009. http://en.wikipedia.org/wiki/VHDL.

[Zhang and Malik, 2003] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: practical implementations and other applications. *DATE*, 2003.

[Zhu *et al.*, 2006] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli. Sat sweeping with local observability don't cares. *DAC*, 2006.

Index