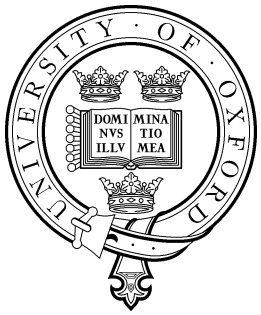
Abstraction Discovery and Refinement for Model Checking by Symbolic Trajectory Evaluation



Sara Elisabeth Adams

Exeter College

University of Oxford

A dissertation submitted for the degree of

Doctor of Philosophy

Trinity Term 2013

Preface

This dissertation is submitted for the degree of Doctor of Philosophy at the University of Oxford. The research presented herein was conducted under the supervision of Professor Thomas F. Melham in the Department of Computer Science, University of Oxford.

To the best of my knowledge this work is original, except where acknowledgements and references are made to previous work. Furthermore, the results presented in this dissertation were in parts developed jointly with Magnus Björk, researcher at Jasper Design Automation, Thomas F. Melham, professor of Computer Science at Oxford University, and Carl Seger, Senior Principal Engineer at Intel®.

In particular, I closely collaborated with Magnus Björk and Carl Seger in 2006–2007, when both researchers were visiting Oxford University for a year, as a Research Associate and a Visiting Professor respectively.

Our joint paper nicely summarises the results obtained during that period [1]. The contributions are split as follows. Carl Seger and I developed the automatic abstraction discovery algorithms, both the basic and improved version as presented in Chapters 3 and 4. Magnus Björk contributed the reindexing optimisations described in Section 4.7. Carl Seger originally wrote the code that implements the algorithms, which I later reworked and significantly extended. I proved all correctness results, in particular Theorems 3.1, 3.5 and 4.1. Finally, Carl Seger wrote the code that generates the hardware designs we verified in Chapters 5 and 8.

All this work profited from the many fruitful discussions Magnus Björk, Tom Melham, Carl Seger, and I had throughout the year of close collaboration.

I worked on the automatic abstraction refinement presented in Chapters 6 and 7 by myself. This includes both the theory presented, as well as the code that implements the proposed approach, and delivers the experimental results given in Chapter 8.

Acknowledgements

Anyone who has completed their doctorate knows it is a labour of love. Or, as my supervisor liked to put it, "Getting your PhD requires 10% intelligence and 90% persistence.", a nice twist of the quote "Genius is 1% inspiration and 99% perspiration" by Thomas Edison.

But while the research presented here was much solitary work, it would not have been possible without several people. First and foremost, I would like to thank Professor Tom Melham for being a superb supervisor. He provided me with invaluable feedback both while researching, as well as in the crucial phase of writing this dissertation. He was supportive throughout, and displayed admirable patience when finishing my dissertation was delayed by the joyful arrival of my son, Felix.

Special thanks also goes to Dr. Carl Seger, with whom I had an enjoyable and productive first year of research in my D.Phil. His hands-on approach, which is surely a result of his extensive industry experience, was very refreshing and complemented my more theoretical approach wonderfully. Similarly, working with Dr. Magnus Björk in that first year was a great experience, which I am thankful for.

Without doubt, family and friends have also been a big part of finishing my doctorate. Special thanks go to my mother and grandmother, who have been strong women throughout my life; to Dieter, who never gave up asking me about the status of my D.Phil.; and to Ulf, my husband, who has been my most important supporter hands-down. I cannot thank him enough especially for his help in the last phase of writing this dissertation. While I worked in the very early mornings to avoid distractions, he took care of our baby son whenever he woke up and needed an extra cuddle before falling asleep again.

Finally, I would like to thank the Engineering and Physical Sciences Research Council (EPSRC) for funding the first phase of my D.Phil. under Research Grant EP/E026745. Supporting research is a great investment into the future of society, and it was a privilege to profit from it. I hope the research I have done and the work I will continue to contribute constitutes a nice return.

Abstract

This dissertation documents two contributions to automating the formal verification of hardware – particularly memory-intensive circuits – by Symbolic Trajectory Evaluation (STE), a model checking technique based on symbolic simulation over abstract sets of states. The contributions focus on improvements to the use of BDD-based STE, which uses binary decision diagrams internally.

We introduce a solution to one of the major hurdles in using STE: finding suitable abstractions. Our work has produced the first known algorithm that addresses this problem by automatically discovering good, non-trivial abstractions. These abstractions are computed from the specification, and essentially encode partial input combinations sufficient for determining the specification's output value. They can then be used to verify whether the hardware model meets its specification using a technique based on and significantly extending previous work by Melham and Jones [2]. Moreover, we prove that our algorithm delivers correct results by construction. We demonstrate that the abstractions received by our algorithm can greatly reduce verification costs with three example hardware designs, typical of the kind of problems faced by the semiconductor design industry.

We further propose a refinement method for abstraction schemes when over-abstraction occurs, i.e., when the abstraction hides too much information of the original design to determine whether it meets its specification. The refinement algorithm we present is based on previous work by Chockler et al. [3], which selects refinement candidates by approximating which abstracted input is likely the biggest cause of the abstraction being unsuitable. We extend this work substantially, concentrating on three aspects. First, we suggest how the approach can also work for much more general abstraction schemes. This enables refining any abstraction allowed in STE, rather than just a subset. Second, Chockler et al. describe how to refine an abstraction once a refinement candidate has been identified. We present three additional variants of refining the abstraction. Third, the refinement at its core depends on evaluating circuit logic gates. The previous work offered solutions for NOT- and AND-gates. We propose a general approach to evaluating arbitrary logic gates, which improves the selection process of refinement candidates. We show the effectiveness of our work by automatically refining an abstraction for a content-addressable memory that exhibits over-abstraction, and by evaluating some common logic gates.

These two contributions can be used independently to help automate the hardware verification by STE, but they also complement each other. To show this, we combine both algorithms to create a fully automatic abstraction discovery and refinement loop. The only inputs required are the hardware design and the specification, which the design should meet. While only small circuits could be verified completely automatically, it clearly shows that our two contributions allow the construction of a verification framework that does not require any user interaction.

Contents

1 Introduction **13**
- 1.1 Formal Verification . . . 14
  - 1.1.1 Simulation . . . 15
  - 1.1.2 Deductive Verification . . . 15
  - 1.1.3 Model Checking . . . 16
- 1.2 Symbolic Trajectory Evaluation . . . 17
- 1.3 Contributions . . . 17
  - 1.3.1 Automatic Abstraction Discovery . . . 18
  - 1.3.2 Automatic Abstraction Refinement . . . 19
  - 1.3.3 Case Studies . . . 20
- 1.4 Outline . . . 20
- 1.5 Notation Overview . . . 23

2 Symbolic Trajectory Evaluation **24**
- 2.1 Circuit Model . . . 24
- 2.2 Simulation . . . 29
  - 2.2.1 Binary Simulation . . . 29
  - 2.2.2 Ternary Simulation . . . 30
  - 2.2.3 Galois Connection . . . 32
- 2.3 Trajectories . . . 37
- 2.4 Trajectory Evaluation Logic . . . 38
- 2.5 Verification by STE . . . 42
- 2.6 Symbolic Indexing . . . 44
- 2.7 Representation . . . 46
  - 2.7.1 Binary Decision Diagrams . . . 47
  - 2.7.2 SAT-based STE . . . 50
- 2.8 History of STE . . . 51
  - 2.8.1 Other Abstraction Frameworks . . . 54
- 2.9 Reindexing in STE . . . 54

3 Abstraction Discovery **59**
- 3.1 Abstraction through Reindexing . . . 61
- 3.2 Basic Algorithm . . . 63
  - 3.2.1 Form of Specifications . . . 64
  - 3.2.2 Shape of SIR Relations . . . 65

3.2.3 Computing an SIR Relation . . . 66
3.3 Correctness Statements . . . 72
3.3.1 Completeness . . . 72
3.3.2 Soundness . . . 73
3.4 Summary . . . 80

4 Abstraction Discovery Improvements 82
4.1 Directed Acyclic Graphs . . . 83
4.1.1 Handling and Correctness . . . 83
4.1.2 Efficient Handling . . . 85
4.2 Multiple Outputs . . . 88
4.3 Symbolic Constants . . . 88
4.4 Additional Constructors . . . 90
4.4.1 Multiple-Input AND Constructors . . . 90
4.4.2 OR Constructors . . . 94
4.4.3 XNOR Constructors . . . 95
4.5 Algorithm . . . 97
4.6 Correctness . . . 98
4.7 Variable Reuse . . . 106
4.8 Reindexing Optimisations . . . 110
4.9 Summary . . . 112

5 Experimental Results for Abstraction Discovery 113
5.1 Content-Addressable Memory . . . 113
5.1.1 Hardware Model . . . 114
5.1.2 SIR Relation . . . 116
5.1.3 Observed Execution Times . . . 119
5.2 Memory . . . 122
5.2.1 Hardware Model . . . 122
5.2.2 SIR Relation . . . 125
5.2.3 Observed Execution Times . . . 127
5.3 Scheduler . . . 128
5.3.1 Hardware Model . . . 129
5.3.2 SIR Relation . . . 129
5.3.3 Observed Execution Times . . . 134
5.4 Summary and Conclusions . . . 136

6 Abstraction Refinement with Symbolic Indexing 138
6.1 CEGAR using Degree of Responsibility . . . 140
6.1.1 Degree of Responsibility . . . 142
6.2 Extension to Symbolic Indexing . . . 148
6.2.1 Evaluation of Symbolic Values . . . 149
6.2.2 Refinement of Abstraction Schemes . . . 153
6.2.3 Example . . . 158
6.3 Summary . . . 161

7 Abstraction Refinement with Arbitrary Gates **163**
7.1 The Cost of Having a High Output 165
7.2 The Cost of Having a Low Output 168
7.3 Determination of the Approximate Degree of Responsibility 170
7.3.1 Set of All Deciding Input Combinations 171
7.3.2 Average of Multiple Deciding Scenarios 173
7.3.3 Approximate Degree of Responsibility for Multiple Fanins . . 173
7.4 Summary 179

8 Experimental Results for Abstraction Refinement **181**
8.1 Abstraction Refinement for the CAM 181
8.2 Approximate Degree of Responsibility for Arbitrary Gates 185
8.2.1 3-Input AND 186
8.2.2 Multiple-Input OR 188
8.2.3 NAND 190
8.2.4 NOR 191
8.2.5 XOR 192
8.2.6 XNOR 193
8.2.7 MUX 194
8.3 Summary and Conclusions 198

9 Conclusion **200**
9.1 Future Work 202

References **205**

Chapter 1

Introduction

With computer technology appearing in almost every area of our life, it is getting more and more imperative to validate the correctness of its implementation. This enhances the safety and security of a system in the spirit of professional engineering. Both aspects are of high importance in life-critical systems, such as space-related or medical equipment, and in computer-dependent businesses. Moreover, the increasing network connectivity magnifies the relevance of professional engineering for commercial applications.

Still, computer systems fail frequently. Often this does not cause serious problems. For example, a crash of your desktop computer is inconvenient, but probably no more than that. But other failures can have disastrous consequences.

In 1994 the Pentium FDIV bug made headlines. A design error caused inaccuracies when dividing certain floating point numbers [4]. This could have been detected when verifying the design. Instead Intel® not only lost about 450 million dollars, but also experienced reputational damage.

On 4 June 1996, the European Ariane 5 was launched using a new expendable launch system. Needless to say, this project had a lengthy preparation phase and cost many millions of dollars. Nonetheless, a design error led to incorrect results when calculating the rocket's movements, thus causing a catastrophic failure of the project: the rocket exploded only seconds after launching [5, 6].

On 14 August 2003, a massive power outage occurred in Northeast America. The trigger for this disaster was the shutdown of one generating plant in Ohio amid high electrical demand. By itself this is not an unusual event, and it should have initiated an alarm so that further actions could be taken. But a race condition caused the alarm system to fail, leading to over 100 further power plants having to shut down [7]. This had an immediate impact on around 50 million people, the largest blackout in North

American history to date.

The list of computer failures with devastating consequences goes on, and reaches into the present. RISKS Digest, a forum on risks to the public in computers and related systems, has devoted itself to regularly publishing newly discovered issues [8]. Ever since computers were invented, bugs in both software and hardware have been a problem, leading to grave economical damage, and sometimes even costing lives. So the need for reliable and robust computer systems is evident.

One aspect of checking a computer system's reliability is ensuring its design fulfils the cause it was created for, i.e., the implementation meets its functional specification. We can ascertain this using *formal verification*.

1.1 Formal Verification

An obvious way of examining whether a computer system works correctly is *testing* it. In testing, we run the system in specific settings, i.e., we define the values of the inputs of the system, and then check whether the outputs have the values we expected. But with testing, we gain certainty only for those cases that we run a test for. To guarantee complete correctness we need to check every possible input combination. Unfortunately, this attempt is infeasible even for relatively small computer systems. To visualise, verifying the correctness of a 64-bit by 64-bit floating point multiplier would need over $3.4 \cdot 10^{38}$ test runs. Even when processing 10^{18} runs every second, this would take more than 10^{13} years, i.e. longer than the universe has likely existed.

Although testing extensively is impracticable, it does ensure correctness. It is a verification technique: it *proves* – or disproves – that the system works as intended. In *formal verification* we use formal mathematics to prove that a design meets its specification. With this we already imply that we have two components at hand: first, a description, or *model*, of the system, and second a *specification* of the desired behaviour of the system. The verification task then is to demonstrate that in every possible case the system modelled behaves as specified.

Much research has concentrated on developing practicable approaches for verifying computer systems. These use different representations of the design to verify, and different methods for checking whether the design satisfies the specification. The different techniques can be divided into three main categories: *simulation*, *deductive verification*, and *model checking*.

1.1.1 Simulation

In hardware verification, tests can be carried out by *simulating* the circuit with software. As discussed above, we can verify only very small systems using this technique. In particular, we need to ensure that all input scenarios are covered. This is usually prohibitively costly. Nonetheless, simulation is often used in industry for finding bugs. Although it is only possible to guarantee correctness by testing exhaustively, simulating just a subset of all input scenarios can locate errors. Nowadays the most basic simulation techniques can hence be seen as a method for improving the quality of the design, rather than ensuring its full correctness. If tests are performed on the actual fabricated hardware it can inspect whether the design was manufactured correctly [9].

Simulation can become a powerful verification method by introducing extensions and modifications. The most prominent example is *symbolic simulation*. Instead of simulating the system with concrete (Boolean) inputs, symbolic values are used. Although computational limits are still given, this methodology has solved hard verification tasks at an industrial scale [10].

1.1.2 Deductive Verification

In *deductive verification* the model of the system and its specification are expressed in higher order logic [11–13]. By applying axioms and rules to these statements correctness is proved or disproved. In the early stages of deductive verification proofs were derived manually. Tools, such as interactive theorem provers, were then developed to ensure that all axioms and rules are applied correctly. As a further step software was built that could partly automate these proofs. But no fully automatic approach is available as of today. Indeed, due to restrictions given by the theory of computability, it is impossible to fully automate this process for the complete logic. However, it is a method that allows verifying very large, or even infinite state systems.

Good results have been achieved using deductive verification. Some notable examples include verifying against IEEE specifications: Miner and Leathrum verified a general class of IEEE-compliant subtractive division algorithms in PVS [14]; Moore et al. verified the floating point division of the AMD K5 processor in ACL2 [15]; and Harrison verified the IA-64 floating point and integer division algorithms in HOL [16].

1.1.3 Model Checking

In *model checking* the model of the computer system is given as a *finite state* machine, and a statement in a *temporal logic* describes the desired behaviour of the system over time. Verification then consists of an exhaustive search through the state space. In contrast to deductive verification, the temporal logic used is weak enough for decidability of the verification problem. Hence model checking algorithms can be automated, thus – in principle – allowing verification without need for human interaction. In the beginning model checking was explicit, that is all reachable states were enumerated [17]. This quickly led to the *state explosion* problem, the combinational blow up of the state-space. This is similar to the restrictions given for simulation.

A cornerstone in model checking was laid in 1987 when *symbolic representations* of states were introduced [18]. The state graph is not built explicitly, but instead Boolean formulae are used to represent sets and relations. Using reduced ordered binary decision diagrams (ROBDD) for the formulae, this allows the verification of systems with many more orders of magnitude.

In 1999 Biere et al. proposed using a satisfiability solver for bounded symbolic model checking, rather than BDDs [19]. Here, the finite state machine can be unrolled for a fixed number of steps, and checking a property of the restricted model is solved with Boolean decision procedures. Aside from avoiding the potential space blowup of BDDs, SAT-based model checking also produces small counter-examples quickly.

By *abstracting* the system to check further improvements can be achieved: a smaller, simpler version of the model is constructed; its states represent multiple states of the initial model. Proving or disproving properties of the abstraction then allows conclusions about the original system. Different abstraction techniques have been explored [20,21], but all of them have two problems in common: first, how do we create an abstraction; and second, how do we refine an abstraction if needed? Here two basic strategies can be distinguished: we can start with a very detailed model, and then refine it, so it gets more abstract; or we can start with an abstraction that hides most details, and refine it by adding details back in. Ideally, of course, the initial abstraction needs no, or only very little refinement.

One successful method for refining a too information-sparse abstraction is counter-example guided abstraction refinement (CEGAR) [22]. First, assume we have an abstraction that allows no *false positives*. That is, the verification of the abstraction succeeds only if the verification of the model itself would succeed. Given this abstraction we now try to verify the desired property. Two outcomes are possible. Either the verification succeeds, and we are done. Or the verification fails with a

counter-example. In the latter case the failure can have two causes: the model has a bug; or the counter-example is spurious, because the abstraction retained too little information. Then the given counter-example can be used to alter the abstraction such that it will not occur in subsequent verification runs again.

CEGAR is critically dependent on extracting a counter-example when the verification run fails, detecting whether it is spurious, and especially refining the abstraction to exclude the counter-example's occurrence. A prominent example where CEGAR has been fully automated was introduced by Clarke et al. [22], which allows specifications in CTL* and abstractions that partition the state space into clusters. But for many abstraction approaches, complete solutions have not yet been developed.

1.2 Symbolic Trajectory Evaluation

Symbolic Trajectory Evaluation (STE) is a model checking technique that is based on symbolic simulation over abstract sets of states [23]. Families of these abstractions can be encoded by binary formulae, or *symbolically indexed*, thus enabling the verification of multiple abstraction cases in a single model-checking run. This has proved valuable especially in memory verification [24–26].

In contrast to most symbolic model checking techniques, STE does not concentrate on calculating all reachable states. Instead, no assumptions are made in the initial states, and bounded model checking that starts in free, unrestricted states is performed.

As already seen for model checking, two main problems need to be addressed when verifying by STE: how do we create an abstraction, and how can it be refined if needed? Creating abstractions translates to finding good symbolic indexings. To date, users have had to create these manually, which is a hard task. No automatic solution has previously been published. In the area of refinement, it is unclear how to extract concrete counter-examples on *over-abstraction*, i.e., when the abstraction used hides too much information. Some refinement techniques have, however, recently been published that do not require a concrete counter-example, but still work in a way similar to CEGAR [1, 3, 27, 28].

1.3 Contributions

This dissertation reports on successful research into a fully automated verification process by Symbolic Trajectory Evaluation, in the form of an automatic abstraction

discovery and refinement loop. This greatly simplifies the use of STE, and enables more circuits to be verified, and with fewer resources – particularly less expert user interaction, memory, and time.

At our framework's core stand two algorithms, which solve the problem of finding good abstractions for STE and of refining abstractions when over-abstraction occurs. Below we briefly summarise these contributions.

1.3.1 Automatic Abstraction Discovery

We propose a novel abstraction framework that processes the specification of a system to compute an STE statement which expresses the system's formal correctness [1] and encodes an abstraction suited to the system. Thus, the design meets its specification exactly if it satisfies the STE statement constructed. Our technique for constructing the STE statement is based on previous work by Melham and Jones [2]. This requires a relation that encodes an abstraction scheme, which is then applied to an auxiliary STE statement to receive the statement that can be used for verifying correctness. In contrast to Melham and Jones' approach, the method reported in this dissertation does not require the user to provide the auxiliary statement. This constitutes a fundamental advance over the previous technique. We also prove that our approach delivers correct verification results, provided the relation satisfies certain side conditions.

We introduce an algorithm that computes such relations using a novel approach. It processes the specification only. This has two big advantages: the specification is usually small compared to the implementation, which reduces the time needed; and the abstraction scheme computed is optimised for the verification task at hand.

The computed abstraction relations encode abstractions that hide as much information as possible. They essentially enumerate multiple partial input combinations, i.e., settings where the value of some inputs are specified and the value of all remaining inputs is unknown. Irrespective of which values the unspecified inputs have, the output will be the same. Furthermore, these partial input combinations are minimal in that removing any of the specified input assignments would result in an indeterminate output of the specification, and thus also a correct implementation. The abstraction scheme covers all possible input combinations. This is a side condition the relation must satisfy to guarantee correct results. We prove that our algorithm produces relations that satisfy this condition by construction. A weaker version of this side condition was also required by Melham and Jones, and they commented that checking the property can create costs that nullify the gain of abstraction [2].

The relations we generate provably satisfy the side conditions by construction, which eliminates this risk.

All this delivers the first automatic abstraction mechanism known for Symbolic Trajectory Evaluation. In the past, a significant bottleneck of STE was the necessity of manually finding abstraction schemes, which is a hard task [2, 24]. The proposed framework, together with the algorithm, is a solution that overcomes this. In particular, it constitutes an efficient, automatic framework with which industrial-sized circuits can be verified as long as no over-abstraction occurs.

1.3.2 Automatic Abstraction Refinement

Our second major contribution is to propose a refinement mechanism to adjust abstraction schemes when over-abstraction occurs, i.e., when too much abstraction was applied in an STE run. At its core stands an algorithm that selects refinement candidates. These are inputs that are indeterminate in at least some cases, and likely need to be determinate to compute the implementation's output value. Our approach is a direct but significant extension of previous work by Chockler et al. [3]. This previous approach is augmented in three essential aspects. First, we fundamentally broaden the core algorithm to support the analysis of arbitrary STE properties, and in particular those that use non-trivial symbolic indexing. Previously, only a very specific subset could be handled, which restricted the applicability greatly. Second, we propose a general approach for determining the approximate degree of responsibility for arbitrary logic gates. The approximate degree of responsibility stands at the core of Chockler et al.'s refinement candidate selection process. Chockler et al. outlined approximations for NOT- and AND-gates only. While this is sufficient to express the behaviour of any other gate, the error of the approximations add up. Our general approach computes approximations of arbitrary gates directly, thus avoiding cumulative errors. Consequently, it enables identifying better refinement candidates. Finally, Chockler et al. suggested one option for refining abstractions once refinement candidates are identified. We examine three additional approaches to incorporating the algorithm results, and introduce a heuristic for deciding which one to best apply. These three enhancements both broaden the applicability of the abstraction refinement, as well as produce refined abstractions, which potentially lead to lower verification costs.

1.3.3 Case Studies

We implemented the automatic abstraction discovery algorithm and show its performance on three types of circuits. The first is a content-addressable memory (CAM), the second a memory, and the third a scheduler. The abstraction scheme automatically computed for the CAM corresponds to one presented as a research result by Pandey [24]. The scheduler is a design that was previously not verified by STE, as no good abstraction scheme had been proposed yet. We describe the idea behind all abstraction schemes our algorithm suggests, and show that it makes the verification of the designs feasible.

Each circuit type is verified in varying sizes, showing the scalability of the algorithm. The scheduler highlights the power of automation in enabling the verification of circuits by STE that were not obtainable before.

We also present how some common gates can be analysed as part of the refinement algorithm to produce better approximations of which indeterminate input is presumably most responsible for the output being indeterminate when over-approximation occurs. Finally, we implemented the abstraction refinement algorithm and set up a fully automatic abstraction discovery and refinement loop to show how our abstraction discovery and our abstraction refinement complement each other, and can be combined to formally verify designs without the need for any human interaction.

1.4 Outline

The dissertation includes contributions in two main areas, abstraction discovery and abstraction refinement. This is also reflected in the structure of the dissertation.

The first major contribution is concerned with the automatic construction of good abstraction schemes. In Chapter 2 we introduce Symbolic Trajectory Evaluation in more detail. We also summarise previous work done by Melham and Jones in the area of reindexing in STE [2], which we build upon. Chapter 3 extends this work to then suggest a fully automatic abstraction discovery framework for STE. At its core stands an algorithm, auto_abstract, that computes relations by analysing the specification the circuit to verify shall meet. We prove our framework correct in two main theorems. The first shows the framework delivers meaningful verification results, provided the relation used meets certain side conditions; the second that the relations our algorithm produces satisfy these side conditions by construction. Thereafter, Chapter 4 extends and improves various aspects of the auto_abstract algorithm described in Chapter

3. These changes are diverse, ranging from allowing more general specifications, suggesting more efficient encodings in the relation, and optimising computations in the final STE run based on the shape of the abstraction relation we automatically generate. These improvements are also proved correct. Our proposed automatic abstraction is put to the test in Chapter 5, where we verify three circuits: a content-addressable memory, a memory, and an oldest-ready scheduler. For each of these examples we first introduce the circuit and its specification, go on to explain what abstraction is automatically computed by our auto_abstract algorithm, to finally give run times of verifying the circuits at increasing sizes. This shows that the work presented in the previous chapters is powerful and allows verifying realistically-sized circuits. In particular, it highlights that automation allows the verification of circuits for which no good abstractions are known yet.

The second main contribution introduces an automatic approach for addressing over-abstraction, which occurs when abstraction schemes are too information-sparse to formally verify a circuit. In Chapter 6 first prior work by Chockler et al. is reviewed [3]. We then build upon their work to extend it in two crucial ways, namely a more general evaluation for finding refinement candidates, and a more prudent refinement step once refinement candidates have been identified. These extensions are non-trivial, while delivering the same results in the special cases handled by Chockler et al. In Chapter 7 we further generalise the abstraction refinement by improving the calculations required for identifying refinement candidates. We present a methodology and provide algorithms which allow the evaluation of arbitrary logic gates when approximating which inputs of a circuit are "most responsible" for the indeterminate output value of that circuit. In Chapter 8 we finally show the effectiveness of our adapted, automatic refinement by verifying one of the circuits previously verified in Chapter 5. In Chapter 5 we utilised user-provided "symbolic constants", which essentially express which inputs are so important that they shall always have a symbolic value, rather than sometimes being indeterminate. With the automatic refinement proposed in this dissertation these symbolic constants are automatically computed, or the initial relation is refined by driving the refinement candidates with symbolic values. In Chapter 8 we furthermore examine several common gates using our general approach to calculating the approximate degree of responsibility, which stands at the core of our abstraction refinement. We demonstrate that these calculations are superior to those received by the previous approach.

The abstraction refinement proposed nicely complements our solution to abstraction discovery. Both can be used independently to help automate the formal verifica-

tion of hardware designs by Symbolic Trajectory Evaluation. They can additionally be combined to deliver a fully automatic verification framework for STE. To the best of our knowledge, this is the first such fully automatic approach for verification by STE. We present experimental results in Chapter 8 as a proof of concept. The dissertation concludes with Chapter 9, which also points to interesting areas for future work.

1.5 Notation Overview

Throughout the dissertation we use the notation provided in Figure 1.1. Especially the proofs of our theorems, which are concerned with the correctness of our results, rely on a precise, mathematical notation. We hope this overview guides the reader whenever necessary.

$f : A \to B, a \mapsto b(a)$	a function f with domain A and co-domain B, where an element $a \in A$ is mapped to $b(a) \in B$
$t[j]$	the j^{th} entry of the tuple $t = (t_0, \ldots, t_{k-1})$ where $0 \leq j < k$
tt	the Boolean value true
ff	the Boolean value false
v	a binary variable
$\mathcal{V}$	a set of binary variables $\mathcal{V} = \{i : v_i\}$
$\mathcal{V} \dot\cup \mathcal{W}$	the disjoint union of the sets $\mathcal{V}$ and $\mathcal{W}$, i.e., $\mathcal{V} \cap \mathcal{W} = \emptyset$
F	a binary expression
$F[\mathcal{V}]$	a binary expression where all variables in $\mathcal{V}$ are free; there might be further free variables in F
$F[v]$	the short form for $F[\{v\}]$
$F[\mathcal{V}, \mathcal{W}]$	an expression where all variables in $\mathcal{V} \dot\cup \mathcal{W}$ are free
$\overleftarrow{v}$	an assignment for the variable v
$F(\overleftarrow{v})$	an expression where the free variable v is replaced by the value specified in the assignment $\overleftarrow{v}$
$\overleftarrow{\mathcal{V}}$	an assignment for all variables in the set $\mathcal{V}$
$F(\overleftarrow{\mathcal{V}})$	an expression where the free variables $\mathcal{V}$ are replaced by the values specified in the assignment $\overleftarrow{\mathcal{V}}$
$\textsc{spec}[\mathcal{V}]$	a bexpr-tree on the set of variables $\mathcal{V}$ (also see page 64)
$\textsc{spec}(\overleftarrow{\mathcal{V}})$	the output of SPEC when its inputs $\mathcal{V}$ are assigned the values specified in the assignment $\overleftarrow{\mathcal{V}}$

Figure 1.1: Notation overview

Chapter 2

Symbolic Trajectory Evaluation

This chapter first explains the foundations needed for understanding Symbolic Trajectory Evaluation (STE). We then introduce STE itself, describe how it can be used to verify circuits, and highlight some of its key aspects – namely the abstraction mechanism used and its computational representation. We conclude by giving a short history of STE and summarising previous work on *reindexing* in STE, a concept that our work heavily builds upon and extends.

2.1 Circuit Model

When formally verifying hardware, no actual hardware is involved. Instead, the design of the hardware is captured in a *model*. Hardware can be modelled through different representations and at different levels of detail. In this dissertation, we concentrate on gate-level verification, where designs are represented as networks of logic gates. Hence the circuits to be verified are described by their *netlist* models.

Definition 2.1 (Gate)**.** *A gate g is given by a tuple $(name_g, args_g, e_g)$, where $name_g$ is the name of the gate, $args_g \in \mathbb{N}$ specifies the number of inputs the gate has, and the excitation function e_g describes the behaviour of the gate as a function*

1. $\mathbb{B}^{args_g} \to \mathbb{B}$ *for combinational gates, and*

2. $\mathbb{B}^{args_g} \times \mathbb{B} \to \mathbb{B}$ *for state-holding gates.*

Definition 2.2 (Netlist)**.** *The netlist representation of a circuit is a pair $(\mathcal{G}, \mathcal{N})$, where $\mathcal{G} = \mathcal{G}_C \dot\cup \mathcal{G}_S$ is the finite set of combinational and state-holding gates, and $\mathcal{N} \subset \mathcal{G}_\perp \times \mathcal{G}_\perp \times \mathbb{N}$ is the relation of connections between such gates. Here $\mathcal{G}_\perp = \mathcal{G} \dot\cup \{\perp\}$, where $\perp$ is used to indicate a blank entry.*

The elements of the relation represent the wires between gates, and are referred to as the *nodes* of the circuit. Given a node (g_1, g_2, i), g_1 defines which gate produces the value on it, g_2 which gate uses this value as an input, and i which input of that gate it is. Nodes that have a blank first entry, $\perp$, are primary circuit inputs; those that have a blank second entry are outputs. By convention, the third component of an output has the value 0. All other nodes are internal nodes.

We now define which properties netlists need to satisfy to qualify as well-formed.

Definition 2.3 (Well-Formed Netlist)**.** *We call a netlist* $(\mathcal{G}, \mathcal{N})$ *well-formed if the following properties hold:*

1. *Each input of a gate is driven by at most one node:*
   $\forall (g_1, h_1, i), (g_2, h_2, j) \in \mathcal{N} : h_1 = h_2 \wedge i = j \rightarrow g_1 = g_2$

2. *Each input of a gate is driven by a node:*
   $\forall g \in \mathcal{G}.|\{(h, i) : \ (h, g, i) \in \mathcal{N}\}| = args_g$

3. *Each output of a gate drives some node:*
   $\forall g \in \mathcal{G} \exists h \in \mathcal{G}.\exists i. \leq args_h \wedge (g, h, i) \in \mathcal{N}$

4. *There is at least one initial input:* $\exists (g, h, i) \in \mathcal{N} : g = \perp$

5. *There is at least one output:* $\exists (g, h, 0) \in \mathcal{N} : h = \perp$

6. *Each gate lies on a path that leads from an initial input to an output, i.e., for each* $g \in \mathcal{G}$ *there exists a sequence of nodes* $(g_j, h_j, i_j)_{0 \leq j \leq n} \subset \mathcal{N}^{n+1}$ *of length* $n + 1$ *such that:*

   - *it starts in an initial input,* $g_0 = \perp$
   - *it ends in an output,* $h_n = \perp$
   - g *is part of the sequence,* $\exists 0 < k \leq n : \ g_k = g$
   - *consecutive nodes are connected,* $\forall 0 \leq j < n : \ h_j \neq \perp \wedge g_{j+1} = h_j$

7. *All loops contain at least one state-holding element, i.e., there exists no sequence of nodes* $(g_j, h_j, i_j)_{0<=j<=n} \subset \mathcal{N}^{n+1}$ *of length* $n + 1$ *such that*

   - *consecutive nodes are connected,* $\forall 0 \leq j < n : h_{j+1} = g_j$
   - *the last node is connected to the first one,* $h_{n+1} = g_0$
   - *none of the gates are state-holding,* $\forall 0 \leq j \leq n : \ g \in \mathcal{G}_C$

Requirement 6 ensures that there are no orphan gates. The definition allows a netlist to consist of several connected components. Each of the components has initial inputs and outputs and are a netlist in themselves. Usually a netlist consists of a single connected component, but the algorithms presented in this dissertation do not require this.

When verifying hardware with STE, specification statements usually put a constraint on the initial inputs and then require the outputs to satisfy desired properties. No assumptions are made about the initial values of state-holding gates, i.e. their values are X. This means that STE specifications usually define only the input-output-behaviour of a circuit, irrespective of both the initial state and the algorithm used to determine that behaviour.

If a netlist has several connected components it may be beneficial to run the verification on each netlist separately. For each component, all constraints on the initial inputs and outputs of other components can be omitted. This can reduce the verification costs of each individual run.

Netlists are commonly given in text form or as a diagram. In this dissertation, preference is given to the diagrammatic representation, for better readability. It is routine to represent such diagrams formally.

Example: Three representations of the same netlist
We give three different representations of the same netlist. It uses three gates, whose excitation functions are

$$\begin{array}{ll} \text{NOT}: & \{0 \mapsto 1,\ 1 \mapsto 0\} \\ \text{AND}: & \{(0,0) \mapsto 0,\ (0,1) \mapsto 0,\ (1,0) \mapsto 0,\ (1,1) \mapsto 1\} \\ \text{DELAY}: & \{(0,0) \mapsto 0,\ (0,1) \mapsto 0,\ (1,0) \mapsto 1,\ (1,1) \mapsto 1\} \end{array}$$

The first two gates are combinational, whereas the delay is a state-holding element. The first entry of a tuple for the DELAY gate gives the value of the input and the second the value of the state. A delay gate ignores the previous state and simply outputs the input at the next point in time.

- Mathematical representation

$$\begin{array}{ll} \mathcal{G} = & \{NOT, AND, DELAY\} \\ \mathcal{N} = & \{(\bot, AND, 0), (NOT, AND, 1), (AND, DELAY, 0) \\ & (DELAY, NOT, 0), (DELAY, \bot, 0)\} \end{array}$$

- Diagrammatic representation
  Figure 2.1 shows a diagrammatic representation. The small triangle in the delay gate represents the dependence on the global clock, which determines when the next point in time is reached. Every state-holding element has such a dependence on the clock of the circuit.

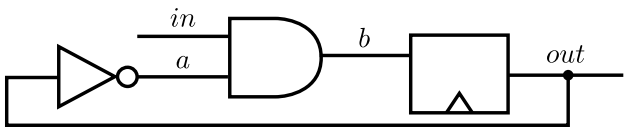


Figure 2.1: Diagrammatic representation of a circuit netlist

- VHSIC hardware description language (VHDL)
  Figure 2.2 provides example VHDL code for the netlist.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity CIRCUIT is
   port (
      data_in    : in std_logic;
      clock      : in std_logic;
      data_out   : out std_logic
   );
end CIRCUIT;

architecture BEHAVIOUR of CIRCUIT is
signal prev : std_logic;
begin
   process(data_in, prev, clock)
   begin
      if rising_edge(clock) then
         data_out <= data_in and not prev;
         prev <= data_in and not prev;
      end if;
   end process;
end BEHAVIOUR;
```

Figure 2.2: VHDL code for a netlist

★

2.2 Simulation

Symbolic Trajectory Evaluation is based on simulation over a ternary domain of circuit values. In the following we briefly describe binary and ternary simulation, as well as present the theory of Galois connections [29, 30], which establishes a link between binary and ternary simulation. This then provides us with the basis upon which STE is built.

2.2.1 Binary Simulation

Binary simulation is a method for completely emulating the behaviour of a circuit M. It requires a state of the circuit, i.e., a mapping from node names to Boolean values that assigns values to all initial inputs and all state-holding elements. Simulation then computes the values of all other nodes and finds the values of the state-holding elements at the next point in time. By fixing an order for the nodes, we can encode the states as tuples of binary values. Binary simulation thus provides the means of computing the next state function in the binary domain, $Y_M^{\mathbb{B}} : \mathbb{B}^{i+l} \to \mathbb{B}^l$. Here i denotes the number of initial inputs of M and l the number of state-holding elements, or *latches*. It computes this function as follows:

1. First, topologically sort the gates of the netlist. Outputs of state-holding elements are treated like initial inputs. This delivers a directed, acyclic graph, because valid netlists must not contain loops that do not include a state-holding element.

2. Next, assign values to the nodes for initial inputs and latches as given in $b \in \mathbb{B}^{i+l}$.

3. Now handle each gate in the order of the topological sort as follows. Apply its excitation function using the values of its inputs, thus delivering a value for the output node of that gate. Note that the topological sort guarantees that all input nodes for each gate will already have been computed.

4. The newly computed inputs to the state-holding elements become their outputs at the next point in time, thus delivering $Y_M^{\mathbb{B}}(b)$.

When simulating a circuit over several clock cycles, we require a series of assignments to the initial inputs and a starting configuration for the state-holding elements. The $Y_M^{\mathbb{B}}$ function is then computed for each time step, where the values of the latches,

i.e., their output values, are determined by the result of the $Y_M^{\mathbb{B}}$ function for the previous time step.

Example: Binary simulation
The gates of the netlist shown in Figure 2.1 have the topological sort NOT < AND < DELAY. Suppose we are given three consecutive values of the input, $in = 0, 1, 1$, and the initial state of the DELAY-gate, $out = 0$. Then the binary simulation of that netlist computes the values shown in Figure 2.3. ★

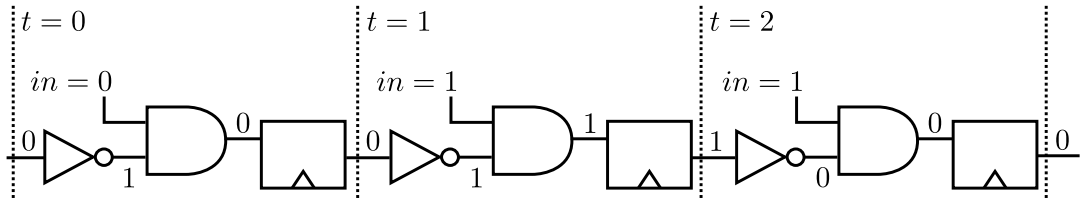


Figure 2.3: Binary simulation of the circuit depicted in Figure 2.1 for three cycles

Note that the behaviour of the next state function Y_M of a circuit M depends on the function it computes. In this dissertation the subscript specifies which circuit the next state function is for. If it is clear which circuit we are reasoning about, the subscript may be omitted. We may also extend the next state function to handle sets of states:

$$Y_M^{\mathbb{B}} : \mathscr{P}(\mathbb{B}^{i+l}) \to \mathscr{P}(\mathbb{B}^l),\ Y_M^{\mathbb{B}}(A) = \{Y_M^{\mathbb{B}}(a) :\ a \in A\}$$

Here $\mathscr{P}(\mathbb{B}^n)$ denotes the power set of $\mathbb{B}^n$.

2.2.2 Ternary Simulation

Ternary simulation was first introduced by Bryant in 1986 [31]. Here a circuit M is emulated over a ternary domain $\mathbb{T} = \{0, 1, \mathrm{X}\}$. In addition to allowing nodes to have Boolean values, an unknown value called X is introduced. As seen in binary simulation, the values of state-holding elements in the next time step are computed using the values of the initial inputs and the state-holding elements in the current time step. To cater for indeterminate node values, we need to expand the excitation functions of the gates. Every logic function can be expressed using conjunction and negation. So it is sufficient to give truth tables for these two functions and a simple delay, as seen in Figure 2.4.

AND	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

NOT	
0	1
1	0
X	X

	state		
DELAY	0	1	X
input 0	0	0	0
1	1	1	1
X	X	X	X

Figure 2.4: Excitation functions for conjunction, inversion, and delaying over a ternary domain.

Accordingly, a next state function for a circuit M can also be defined on the ternary domain, $Y_M^{\mathbb{T}} : \mathbb{T}^{i+l} \to \mathbb{T}^l$. Later we will see that it is sometimes convenient for the domain and co-domain to be the same. We can achieve this by prefixing the result with i indeterminate values:

$$Y_M^{\mathbb{T}} : \mathbb{T}^{i+l} \to \mathbb{T}^{i+l}, \forall 0 \leq j < i : \; Y_M^{\mathbb{T}}(t)[j] = \mathrm{X}$$

Here $Y_M^{\mathbb{T}}(t)[j]$ denotes the j^{th} tuple entry of $Y_M^{\mathbb{T}}(t)$. The computation of the remaining values is as before. Intuitively, you could say that we do not know the values of the inputs in the next cycle, and hence the values of these are indeterminate.

Example: Ternary simulation
The gates of the netlist shown in Figure 2.1 have the topological sort NOT < AND < DELAY. Suppose we are given three consecutive values of the input, $in = 0, 1, \mathrm{X}$, and the initial state of the DELAY-gate, $out = \mathrm{X}$. Then the ternary simulation of that netlist computes the values shown in Figure 2.5. ★

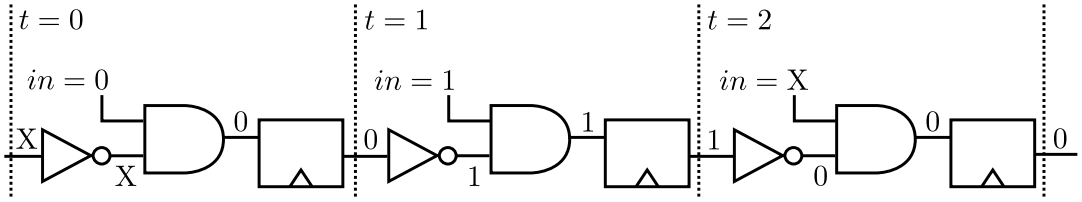


Figure 2.5: Ternary simulation of the circuit depicted in Figure 2.1 for three cycles

In the above example, not all values are Boolean. In particular, for the cycle starting at $t = 2$ the input in is indeterminate, but the output observed at $t = 3$ can still be calculated. This means that the binary simulation with $in = 0$, and with $in = 1$ respectively, both yield the same result $out = 0$. So ternary simulation has the potential to calculate the result of multiple binary simulation runs in one.

2.2.3 Galois Connection

We can formally verify that a circuit meets its specification by checking whether all observable states are included in the set of allowed states.

Definition 2.4 (Antecedent, Consequent, and Satisfaction)**.** *Let M be a circuit and $M(A) \subseteq \mathbb{B}^l$ denote the set of states that the model can exhibit assuming the precondition, or* antecedent*, $A \subseteq \mathbb{B}^{i+l}$. Let $C \subseteq \mathbb{B}^l$ be the set of states that are allowed by the specification of M; C is called a* consequent*. We then say M satisfies its specification assuming A if $M(A) \subseteq C$, and denote this by $M \models A \Rightarrow C$.*

If some of the model states are not included in the consequent, then the circuit does not meet its specification, and we write $M \not\models A \Rightarrow C$.

Using simulation, we can determine which states a model can exhibit given a precondition. In the following it will be advantageous to use ternary simulation to reason about the binary model. So it is necessary to formalise the connection between binary and ternary simulation, and characterise the conclusions that are sound. The theory of Galois connections provides a mathematical foundation for this [29, 30].

Definition 2.5 (Galois Connection)**.** *A Galois Connection is a pair of monotonic functions $(\alpha : \mathcal{C} \to \mathcal{A}, \gamma : \mathcal{A} \to \mathcal{C})$ on the partially ordered sets $(\mathcal{C}, \subseteq)$ and $(\mathcal{A}, \preceq)$ such that*

$$\forall C \in \mathcal{C}, a \in \mathcal{A} : \quad \alpha(C) \preceq a \Leftrightarrow C \subseteq \gamma(a)$$

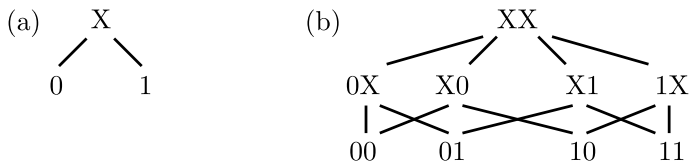


Figure 2.6: Hasse diagram for partial order based on level of abstraction in (a) $\mathbb{T}$ and (b) $\mathbb{T}^2$

In our context, $\mathcal{A} = \mathbb{T}^{i+l}$ and $\mathcal{C} = \mathscr{P}(\mathbb{B}^{i+l})$. Here, $\mathcal{A}$ represents the set of abstract states that are defined by the values of inputs and latches at a specific point in time. As seen in Figure 2.6, we define a partial order on this set based on the level of abstraction. That is, for each node, X is more abstract than 0 or 1, so $0 \preceq$ X and $1 \preceq$ X. On the other hand, neither 0 or 1 are more abstract than the other, and

therefore no order is defined between them. By extending this order pointwise to tuples of nodes, we define a partial order on $\mathbb{T}^{i+l}$.

Notice that $(\mathbb{T}^{i+l}, \preceq)$ is not a lattice. While it is a partially ordered set, it lacks a least element. But this is required for a lattice, as any two elements need to have a supremum and infimum. By introducing a new element *bottom*, $\bot$, and extending the order such that $\forall a \in \mathbb{T}^{i+l} : \bot \preceq a$ we receive a complete lattice. So not just any two elements have a supremum and infimum, but this also holds for any subset [29]. The bottom element essentially captures the case where we have contradicting data, thus identifying an invalid state. While this state cannot be observed in practice, introducing it allows us to take advantage of the properties of complete lattices when reasoning about properties. Hence, in the following, we use $\mathcal{A} = \mathbb{T}^{i+l} \cup \{\bot\}$.

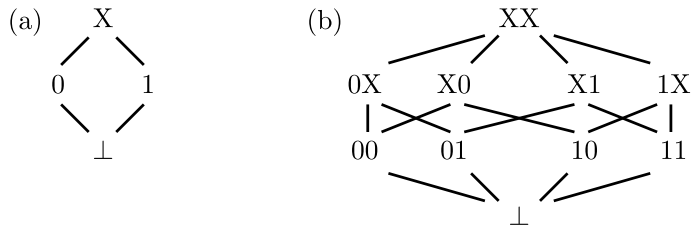


Figure 2.7: Introduction of $\bot$ to establish complete lattices (a) $\mathbb{T} \cup \{\bot\}$ and (b) $\mathbb{T}^2 \cup \{\bot\}$

For $\mathcal{C}$ we use the power set of $\mathbb{B}^{i+l}$ with the subset ordering $\subseteq$, which forms a complete lattice. The function α is then an abstraction function that maps a set of states $C \in \mathcal{C}$ in the binary domain to an abstract state in the ternary domain. The abstract state $\alpha(C)$ is chosen so that it is the smallest state that can represent all given states in the set C. Note that the smallest state retains as much binary information as possible. As $\mathbb{B}^{i+l}$ is a subset of $\mathbb{T}^{i+l}$, every element $C \in \mathcal{C}$ is a subset of $\mathcal{A}$, and therefore every $c \in C$ is also an element of $\mathcal{A}$. Thus, more formally, the smallest state is defined by

$$\forall c \in C : \; c \preceq \alpha(C) \quad \text{and} \quad \forall a \in \mathcal{A} : \; (\forall c \in C : c \preceq a) \Rightarrow \alpha(C) \preceq a.$$

In other words, the abstraction function maps sets of concrete states to their least upper bound, $\alpha(C) = \bigsqcup_{c \in C} c$. Conversely, the concretisation function γ maps an abstract state to the maximum set of states that the abstract state can represent. The theory of Galois Connections says that if both $\mathcal{A}$ and $\mathcal{C}$ are complete lattices, then defining either of the functions α or γ uniquely defines the other function. For

clarity, we give a concrete definition of both functions:

$$
\begin{aligned}
\gamma: \quad \mathbb{T}^{i+l} \cup \{\bot\} &\rightarrow \mathscr{P}(\mathbb{B}^{i+l}), \text{ where} \\
\bot &\mapsto \emptyset, \text{ and} \\
a &\mapsto \{c \in \mathbb{B}^k : \ c \preceq a\} \\
\alpha: \quad \mathscr{P}(\mathbb{B}^{i+l}) &\rightarrow \mathbb{T}^{i+l} \cup \{\bot\}, \text{ where} \\
\emptyset &\mapsto \bot, \text{ and} \\
\{c_j : \ j \in J\} &\mapsto a, \text{ such that for all } 0 \leq k < i+l \\
a[k] &= \begin{cases} \text{X} & \exists m, n \in J : \ c_m[k] \neq c_n[k] \\ c_j[i] \text{ for some } j \in J & \text{otherwise} \end{cases}
\end{aligned}
$$

Here $a[k]$ denotes the k^{th} entry of the tuple $a \in \mathbb{T}^{i+l}$. Notice that $\gamma \circ \alpha : \mathcal{C} \rightarrow \mathcal{C}$ is not an identity function. For example,

$$\gamma(\alpha(\{011, 101\})) = \gamma(XX1) = \{001, 011, 101, 111\}$$

This corresponds to the loss of information seen when using abstraction; once we abstract the real value of nodes, we cannot reproduce the concrete values later.

As first stated by Ching-Tsun Chou in [32], Theorem 2.6 establishes that this Galois connection allows properties for binary circuits to be proved in the ternary domain. Here we use the notation introduced in Definition 2.4.

Theorem 2.6. *Let M be a circuit, and let $C \in \mathscr{P}(\mathbb{B}^l)$ be the set of states that satisfy a desired consequent. Further, let $M(A) \in \mathscr{P}(\mathbb{B}^l)$ be the set of circuit states in the next time step assuming the antecedent $A \in \mathscr{P}(\mathbb{B}^{i+l})$, i.e., $M(A) = Y^{\mathbb{B}}(A)$.*

1. *If $\alpha(M(A)) \preceq \alpha(C)$, then $M(A) \subseteq \gamma(\alpha(A))$, so $M \models A \Rightarrow \gamma(\alpha(C))$.*
2. *If $\alpha(M(A)) \not\preceq \alpha(C)$, then $M(A) \not\subseteq C$, so $M(A) \not\models A \Rightarrow C$.*

Proof. This is a direct conclusion from the defining property of Galois connections:

1. $\alpha(M(A)) \preceq \alpha(C)$
   $\Leftrightarrow$ $M(A) \subseteq \gamma(\alpha(C))$
2. $\alpha(M(A)) \not\preceq \alpha(C)$
   $\Leftrightarrow$ $M(A) \not\subseteq \gamma(\alpha(C))$
   $\Rightarrow$ $M(A) \not\subseteq C$

□

Note, however, that Theorem 2.6 cannot directly be applied to ternary and binary simulation. Suppose we determine $M(A)$ via simulation, i.e., $M(A) = Y^{\mathbb{B}}(A)$. Then the relationship between $\alpha(Y^{\mathbb{B}}(A))$ and $\alpha(C)$ determines whether $M(A) \subseteq \gamma(\alpha(C))$ or not. But ternary simulation computes $Y^{\mathbb{T}}(\alpha(A))$, rather than $\alpha(Y^{\mathbb{B}}(A))$.

There is a relationship between these values, though:

$$\forall A \in \mathscr{P}(\mathbb{B}^{i+l}) : \ \alpha(Y^{\mathbb{B}}(A)) \preceq Y^{\mathbb{T}}(\alpha(A)) \tag{2.2.1}$$

This is a direct conclusion from the monotonicity of α, $Y^{\mathbb{B}}$ and $Y^{\mathbb{T}}$. We can easily see that α is monotone, because it maps elements to their least upper bound. The binary next state function $Y^{\mathbb{B}}$ on sets of states is monotone by definition: $A \mapsto \{Y^{\mathbb{B}}(a) : \ a \in A\}$. Finally, the ternary next state function $Y^{\mathbb{T}}$ is monotone, because it inherits the monotonicity of the excitation functions given in Figure 2.4.

Using observation 2.2.1, Theorem 2.6 gives us the following relationship between binary and ternary simulation:

1. If $Y^{\mathbb{T}}(\alpha(A)) \preceq \alpha(C)$, then $M(A) \subseteq \gamma(\alpha(C))$, so $M \models A \Rightarrow \gamma(\alpha(C))$
   This means that when simulating the behaviour of a circuit on any states that satisfy a specific antecedent A, then the resulting set of states also satisfies the consequent. In short, we say M satisfies C assuming A. The above implication follows directly from Theorem 2.6 and the fact that $\alpha(Y^{\mathbb{B}}(A)) \preceq Y^{\mathbb{T}}(\alpha(A))$.

2. If $Y^{\mathbb{T}}(\alpha(A)) \not\preceq \alpha(C)$

   - If $\exists j : \ Y^{\mathbb{T}}(\alpha(A))[j] \in \mathbb{B} \wedge Y^{\mathbb{T}}(\alpha(A))[j] \not\preceq \alpha(C)[j]$, then $M(A) \not\subseteq C$
     This means that M does not satisfy C assuming A, $M \not\models A \Rightarrow C$. The implication follows directly from Theorem 2.6, and the following observation. $\alpha(Y^{\mathbb{B}}(A)) \preceq Y^{\mathbb{T}}(\alpha(A))$ implies that any binary assignment in $Y^{\mathbb{T}}(\alpha(A))$ must have the same binary assignment in $\alpha(Y^{\mathbb{T}}(A))$. So $\alpha(Y^{\mathbb{T}}(A)) \not\preceq \alpha(C)$ even if only one such assignment exists.
   - If $\exists j : \ (Y^{\mathbb{T}}(\alpha(A))[j] = \mathrm{X}) \wedge (\alpha(C)[j] \in \mathbb{B})$, then we cannot say whether $M(A) \subseteq C$ or not.
     In this case we cannot say whether M satisfies C assuming A. We call this case a *weak disagreement*. There are cases where such a result occurs when the circuit satisfies the consequent C assuming the antecedent A, but also cases when it violates C. Hence the only conclusion that can be drawn is that the simulation run was not appropriate for the stated problem. As

this case cannot occur with binary simulation, its cause is a too heavy use of abstraction, called *over-abstraction* in this dissertation.

Example:
Consider the circuit modelled in Figure 2.8. Using the ordering $i_1 < i_2 < i_3 < o_1 < o_2$, suppose $A = \{01100, 10100\}$. Then $\alpha(A) = \text{XX100}$, $Y^{\mathbb{T}}(\alpha(A)) = \text{X1}$, and $\alpha(Y^{\mathbb{B}}(A)) = \alpha(11) = 11$. Observe that abstracting the result of binary simulation delivers a less abstract result than simulating the abstracted antecedent: $\alpha(Y^{\mathbb{B}}(A)) = 11 \preceq \text{X1} = Y^{\mathbb{T}}(\alpha(A))$. As noted above, in some cases we can still draw a conclusion from the ternary simulation run result, in others we cannot:

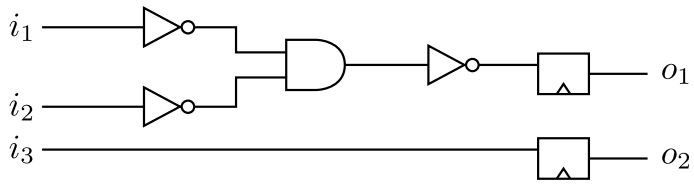


Figure 2.8: A simple circuit for which over-abstraction can be observed.

1. Suppose $C = \{01, 11\}$. Then $Y^{\mathbb{T}}(\alpha(A)) = \text{X1} \preceq \text{X1} = \alpha(C)$, and thus we can conclude that $M \models A \Rightarrow \gamma(\alpha(C))$

2. Suppose $C = \{00\}$. Then $Y^{\mathbb{T}}(\alpha(A)) = \text{X1} \neq 00 = \alpha(C)$. But there exists a circuit node that has a binary assignment and violates the consequent: $Y^{\mathbb{T}}(\alpha(A))[2] = 1 \not\preceq 0$. Thus we may conclude that $M \not\models A \Rightarrow C$.

3. Finally, there are two options for a weak disagreement – the case where a property does hold, and that in which it does not hold. We cannot tell which of these cases holds when encountering such a result. Hence we cannot determine whether the circuit satisfies the property or not.

   - Suppose $C = \{11\}$. The value of o_1 is indeterminate, but the property requires it to be false. All other requirements, namely $o_2 = 1$, are met. Hence we cannot conclude that the model does not satisfy the property. Indeed, the binary run shows that the circuit does meet the specification.
   - Suppose $C = \{01\}$. As above, we cannot conclude whether the model does not satisfy the property. In this case the circuit does not meet the specification, but as seen above it might have.

The key here is that the result of the ternary simulation does not always provide all the necessary information for determining whether a circuit satisfies the desired property. In this case the verification neither passes, nor fails. Instead it signals that $\alpha(A)$ did not provide enough information to determine the result.

★

In the following sections we introduce Symbolic Trajectory Evaluation, including the logic it is based on, Trajectory Evaluation Logic. It allows only statements for which $\gamma(\alpha(C)) = C$, and thus Theorem 2.6 is applicable. As captured in Theorem 2.11 this allows the verification or disproof of properties of a binary circuit by simulating it in the ternary domain. First, however, we need to define *trajectories*, which express runs of ternary simulation.

2.3 Trajectories

A trajectory is an infinite sequence of states in the ternary domain that the circuit M could actually exhibit. For each cycle, it gives the ternary values of the state-holding elements observable assuming the circuit inputs are driven with specific values. As already described in Section 2.2.2 we employ the next state function to formalise this concept. For trajectories we need to apply the next state function repeatedly. Formalising this is easier when the domain and co-domain are the same. Thus, as we suggested previously, we prefix the sequence of values of the state-holding elements with the sequence of values of the circuit inputs, which are all unknown and so have the value X.

The ultimate goal is to use trajectories for formal verification, so the ordering on ternary states is essential. Previously, we introduced $\preceq$. However, to be consistent with conventional STE presentations, we now switch to a different ordering $\sqsubseteq$ on ternary states. It is an information order, and the reverse of the abstraction order $\preceq$ introduced in Section 2.2.3: $a \sqsubseteq b \;:\Leftrightarrow\; b \preceq a$. So $\sqsubseteq$ pointwise extend the partial order where X $\sqsubseteq$ 0 and X $\sqsubseteq$ 1. X provides less concrete information about a node than 0 or 1, thus motivating the name "information order". Moreover, it is partial, because we cannot order 0 below or above 1. Augmented with a greatest element $\top$, this partial order gives us a complete lattice $(\mathbb{T}^{i+l} \cup \{\top\}, \sqsubseteq)$. See Figure 2.9 for two examples. Notice that the greatest element $\top$ corresponds to the least element $\bot$ introduced in Section 2.2.3.

Using this new ordering we can define a *trajectory* in terms of the monotonic next

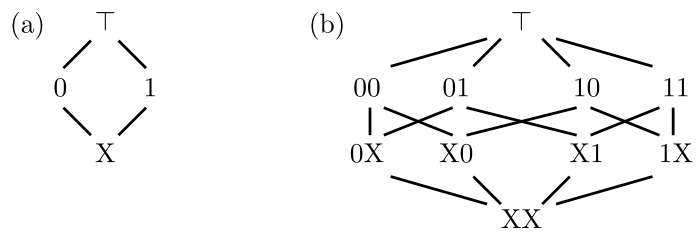


Figure 2.9: Hasse diagram for the complete lattices (a) $(\mathbb{T} \cup \{\top\}, \sqsubseteq)$ and (b) $(\mathbb{T}^2 \cup \{\top\}, \sqsubseteq)$

state function $Y^{\mathbb{T}}$.

Definition 2.7 (Trajectory)**.** *A trajectory* $\sigma : \mathbb{N} \to \mathbb{T}^{i+l}$ *of a circuit* M *is an infinite sequence of states such that*

$$\forall t \in \mathbb{N} : \; Y_M(\sigma(t)) \sqsubseteq \sigma(t+1),$$

where i *denotes the number of circuit inputs, and* l *the number of state-holding elements in* M*.*

Definition 2.8 (Suffix of a Trajectory)**.** *The* i^{th} suffix σ^i *of a trajectory* σ *is a trajectory such that*

$$\forall j \in \mathbb{N} : \; \sigma^i(j) = \sigma(i+j).$$

Next we introduce a logic, which allows us to express which behaviour we expect a circuit to have.

2.4 Trajectory Evaluation Logic

The logic used for expressing properties of circuits in Symbolic Trajectory Evaluation is called *Trajectory Evaluation Logic*, in short TEL [2]. It can be used to stipulate the values that nodes must have at specified points within a bounded period of time. The full syntax is given in Figure 2.10. Trajectory Evaluation Logic is quite restrictive. Conjunction of formulae is allowed, but negation and disjunction are not. Somewhat more expressive statements than might at first appear possible can be phrased with *symbolic indexing*, which we discuss in more detail in Section 2.6.

A guarded formulae $G \rightarrowtriangle f_1$ expresses that the formula f_1 needs only be asserted if a propositional formula G, called *guard*, is satisfied. Whenever guards are used, the

$f_1, f_2 :=$	n is 0	the value of node $n \in \mathcal{N}$ is 0
$\mid$	n is 1	the value of node $n \in \mathcal{N}$ is 1
$\mid$	f_1 and f_2	conjunction of formulae
$\mid$	$\mathsf{N}f_1$	f_1 holds in the next time step
$\mid$	$G \rightarrowtriangle f_1$	assert f_1 only if the propositional formula G holds

Figure 2.10: Syntax of Trajectory Evaluation Logic given a circuit with nodes $\mathcal{N}$.

TEL formula can have a set of free variables $\mathcal{V}$, which the guards depend on. Usually, different guards will have some variables in common. A very simple example is the TEL formula

$$(v_i \rightarrowtriangle n_i \text{ is } 1) \text{ and } (\overline{v_i} \rightarrowtriangle n_i \text{ is } 0).$$

This requires the node n_i to have the "same" value as the variable v_i. When v_i evaluates to true, then the TEL formula n_i is 1 needs to be asserted, when v_i evaluates to false, i.e., $\overline{v_i}$ holds, then the TEL formula n_i is 0 needs to be asserted. Two guards are used, v_i and $\overline{v_i}$, which both depend on the same variable v_i. We will later see that such a TEL formula can be used for saying that the node n_i has a symbolic value v_i. Throughout this dissertation, we will see many more complex constructs that make heavy use of guards.

Next we define when a trajectory σ satisfies a TEL formula f under an assignment $\overleftarrow{\mathcal{V}}$ to the free variables used in f, more specifically in its guards. Given a guard G, we may document the variables it contains by giving a set of variables in square brackets: $G[\mathcal{V}]$. $G(\overleftarrow{\mathcal{V}})$ then denotes the formula where the free variables $\mathcal{V}$ of G are replaced by the values specified in the assignment $\overleftarrow{\mathcal{V}}$. We can then determine the formula's Boolean value and, for simplicity, write $G(\overleftarrow{\mathcal{V}}) \in \{\mathsf{tt}, \mathsf{ff}\}$. As all guards evaluate to either true or false when applying the assignment $\overleftarrow{\mathcal{V}}$, f can then be simplified to a formula without guards. We will later see that in STE satisfaction of a formula by a circuit model means satisfying the different TEL formulae that arise across all possible assignments $\overleftarrow{\mathcal{V}}$, not just a single one.

The definition of satisfaction of a TEL formula f is given in Figure 2.11. If f simply states that a node n_i must have a concrete value, e.g. 0, then a sequence σ satisfies f if and only if the corresponding node is assigned that value: $\sigma(0)[i] = 0$. Remember that $\sigma(0)[i]$ denotes the i^{th} entry of the tuple $\sigma(0)$, which corresponds to the value of the i^{th} node of the circuit at time 0. If f is a conjunction of two

other TEL formulae, then the sequence must satisfy both of these formulae. A TEL formula $\mathsf{N}f$ states that at the next point in time the formula f needs to be satisfied; so σ satisfies the formula $\mathsf{N}f$ exactly if its suffix σ^1 satisfies f. Finally, for a guarded formula $G \rightarrowtriangle f$, σ needs to satisfy the formula f only when $G(\overleftarrow{\mathcal{V}})$ evaluates to true.

$$
\begin{array}{lcl}
\overleftarrow{\mathcal{V}}, \sigma \models n_i \text{ is } 0 & :\Leftrightarrow & (\sigma(0) = \top) \text{ or } (\sigma(0)[i] = 0) \\
\overleftarrow{\mathcal{V}}, \sigma \models n_i \text{ is } 1 & :\Leftrightarrow & (\sigma(0) = \top) \text{ or } (\sigma(0)[i] = 1) \\
\overleftarrow{\mathcal{V}}, \sigma \models f_1 \text{ and } f_2 & :\Leftrightarrow & (\overleftarrow{\mathcal{V}}, \sigma \models f_1) \text{ and } (\overleftarrow{\mathcal{V}}, \sigma \models f_2) \\
\overleftarrow{\mathcal{V}}, \sigma \models \mathsf{N}f & :\Leftrightarrow & \overleftarrow{\mathcal{V}}, \sigma^1 \models f \\
\overleftarrow{\mathcal{V}}, \sigma \models G \rightarrowtriangle f & :\Leftrightarrow & G(\overleftarrow{\mathcal{V}}) \text{ implies } (\overleftarrow{\mathcal{V}}, \sigma \models f)
\end{array}
$$

Figure 2.11: Satisfaction of TEL formulae

Every trajectory formula f has the useful property that given $\overleftarrow{\mathcal{V}}$ it has a unique weakest sequence $[f]^{\overleftarrow{\mathcal{V}}}$ which satisfies it. Note that when we say it is the *weakest* sequence we express that any other sequence that also satisfies the same TEL formula is ordered below it using the ordering $\sqsubseteq$:

$$\sigma_1 \sqsubseteq \sigma_2 \;:\Leftrightarrow\; \forall t \in \mathbb{N} : \sigma_1(t) \sqsubseteq \sigma_2(t)$$

Note that existence of a weakest sequence is ensured, because the sequence $\forall t : \sigma(t) = \top$ satisfies all TEL formulae. There being a unique weakest sequence is guaranteed, because our domain is a complete lattice, and so every subset of elements has a least upper bound. As the sequence is uniquely defined, we call it *the defining sequence of* f.

Definition 2.9 (Defining Sequence)**.** *The* defining sequence $[f]^{\overleftarrow{\mathcal{V}}}$ *of a TEL formula* f *is the weakest sequence that satisfies* f *given the assignment* $\overleftarrow{\mathcal{V}}$ *to its free variables:*

$$\forall \overleftarrow{\mathcal{V}}. \forall \sigma.\ \overleftarrow{\mathcal{V}}, \sigma \models f \Leftrightarrow [f]^{\overleftarrow{\mathcal{V}}} \sqsubseteq \sigma$$

The defining sequence allows an easy way to check whether another sequence σ satisfies a formula. Rather than using the original definition of satisfaction, we can check whether σ is ordered below $[f]^{\overleftarrow{\mathcal{V}}}$ with respect to $\sqsubseteq$:

$$\overleftarrow{\mathcal{V}}, \sigma \models f \;\Leftrightarrow\; [f]^{\overleftarrow{\mathcal{V}}} \sqsubseteq \sigma$$

Explicitly stating the defining sequence is straightforward, as summarised in Figure 2.12. In essence, for each point in time the defining sequence constrains a node to

have a concrete value if and only if the TEL formula requires this. For conjunctions, the least upper bound of the defining sequences of each of the subformulae needs to be taken, so that both formulae are satisfied simultaneously. We achieve this by determining the join $\sigma_1 \sqcup \sigma_2$ of two sequences σ_1 and σ_2:

$$\sigma_1(t) \sqcup \sigma_2(t) = \begin{cases} \top & \text{if } \exists j.\sigma_1(t)[j] \not\sqsubseteq \sigma_2(t)[j] \wedge \sigma_2(t)[j] \not\sqsubseteq \sigma_1(t)[j] \\ j \mapsto & \begin{cases} \sigma_1(t)[j] & \text{if } \sigma_2(t)[j] \sqsubseteq \sigma_1(t)[j] \\ \sigma_2(t)[j] & \text{if } \sigma_1(t)[j] \sqsubseteq \sigma_2(t)[j] \end{cases} & \text{otherwise} \end{cases}$$

In particular, if a TEL formula requires contradicting values, e.g. $(n_i \text{ is } 0) \text{ and } (n_i \text{ is } 1)$, then the join of the defining sequences of each of the subformulae is $\top$, as $1 \not\sqsubseteq 0$ and $0 \not\sqsubseteq 1$. Thus, $\top$ corresponds to an overconstrained sequence.

$$\begin{aligned}
[n_i \text{ is } 0]^{\overleftarrow{\mathcal{V}}}(0) &= j \mapsto \begin{cases} 0 & \text{if } j = i \\ \mathrm{X} & \text{otherwise} \end{cases} \\
[n_i \text{ is } 0]^{\overleftarrow{\mathcal{V}}}(t+1) &= j \mapsto \mathrm{X} \\
[n_i \text{ is } 1]^{\overleftarrow{\mathcal{V}}}(0) &= j \mapsto \begin{cases} 1 & \text{if } j = i \\ \mathrm{X} & \text{otherwise} \end{cases} \\
[n_i \text{ is } 1]^{\overleftarrow{\mathcal{V}}}(t+1) &= j \mapsto \mathrm{X} \\
[f_1 \text{ and } f_2]^{\overleftarrow{\mathcal{V}}}(t) &= [f_1]^{\overleftarrow{\mathcal{V}}}(t) \sqcup [f_2]^{\overleftarrow{\mathcal{V}}}(t) \\
[G \rightarrowtriangle f]^{\overleftarrow{\mathcal{V}}}(t) &= j \mapsto \begin{cases} [f]^{\overleftarrow{\mathcal{V}}}[j] & \text{if } G(\overleftarrow{\mathcal{V}}) = \mathsf{tt} \\ \mathrm{X} & \text{otherwise} \end{cases} \\
[\mathsf{N}f]^{\overleftarrow{\mathcal{V}}}(0) &= j \mapsto \mathrm{X} \\
[\mathsf{N}f]^{\overleftarrow{\mathcal{V}}}(t+1) &= [f]^{\overleftarrow{\mathcal{V}}}(t)
\end{aligned}$$

Figure 2.12: Value of the defining sequence of f under $\overleftarrow{\mathcal{V}}$

Note that the defining sequence of a TEL formula is not necessarily a trajectory with respect to a circuit M. We therefore introduce the notion of the *defining trajectory*.

Definition 2.10 (Defining Trajectory)**.** *The* defining trajectory $[\![f]\!]^{\overleftarrow{\mathcal{V}}}$ *of a TEL formula f is the weakest trajectory of a circuit M that satisfies f for the assignment $\overleftarrow{\mathcal{V}}$ to its free variables.*

The defining trajectory can be determined recursively by taking the least upper

bound of the defining sequence at that point in time, and the result of the next state function from the previous point in time, as seen in Figure 2.13. Applying the next state function to the defining sequence at the previous point in time ensures that the model can indeed exhibit the behaviour encoded in $[\![f]\!]^{\overleftarrow{\mathcal{V}}}$, and thus leads to a trajectory.

$$\begin{array}{lcl} [\![f]\!]^{\overleftarrow{\mathcal{V}}}(0) & = & [f]^{\overleftarrow{\mathcal{V}}}(0) \\ [\![f]\!]^{\overleftarrow{\mathcal{V}}}(t+1) & = & [f]^{\overleftarrow{\mathcal{V}}}(t+1) \sqcup Y_M^{\mathbb{T}}([\![f]\!]^{\overleftarrow{\mathcal{V}}}(t)) \end{array}$$

Figure 2.13: Value of the defining trajectory of f under $\overleftarrow{\mathcal{V}}$ for the circuit M

2.5 Verification by STE

Statements in Symbolic Trajectory Evaluation have the form $A \Rightarrow C$, where the antecedent A and the consequent C are TEL formulae. We say that a circuit M satisfies the property if the consequent does not require any of the nodes of the model to have a different concrete value than it actually exhibits. In other words, for each node and for each point in time, the required node value v is either the same value w as the circuit exhibits, or it is X.

Theorem 2.11. *Let M be a circuit, let A and C be TEL formulae, and let $\overleftarrow{\mathcal{V}}$ be an assignment to the free variables in A and C. If $[\![A]\!]^{\overleftarrow{\mathcal{V}}} \preceq [C]^{\overleftarrow{\mathcal{V}}}$, then*

$$\gamma([\![A]\!]^{\overleftarrow{\mathcal{V}}}) \subseteq \gamma([C]^{\overleftarrow{\mathcal{V}}}).$$

Proof. The set of binary states represented by a sequence s is $\gamma(s)$. Note that γ has to be extended to sequences for this. This is done in a straightforward fashion by applying γ to each sequence element.

By applying Theorem 2.6, we can conclude that $\gamma([\![A]\!]^{\overleftarrow{\mathcal{V}}}) \subseteq \gamma([C]^{\overleftarrow{\mathcal{V}}})$ implies that $\gamma([\![A]\!]^{\overleftarrow{\mathcal{V}}}) \subseteq \gamma(\alpha(\gamma([C]^{\overleftarrow{\mathcal{V}}})))$. But $\gamma \circ \alpha \circ \gamma = \gamma$:

I	$\forall C \in \mathcal{C}, a \in \mathcal{A}:\ \alpha(C) \preceq a \Leftrightarrow C \subseteq \gamma(a)$	Definition of Galois connection
II	$\forall a \in \mathcal{A}:\ \alpha(\gamma(a)) \preceq a$	Restrict I to $C = \gamma(a)$
III	$\forall a \in \mathcal{A}:\ \gamma(\alpha(\gamma(a))) \subseteq \gamma(a)$	Monotonicity of γ and II
IV	$\forall C \in \mathcal{C}:\ C \subseteq \gamma(\alpha(C))$	Restrict I to $a = \alpha(C)$
V	$\forall a \in \mathcal{A}:\ \gamma(a) \subseteq \gamma(\alpha(\gamma(a)))$	Restrict VI to $C = \gamma(a)$
VI	$\forall a \in \mathcal{A}:\ \gamma(\alpha(\gamma(a))) = \gamma(a)$	Apply III and V

Therefore $\gamma([\![A]\!]^{\overleftarrow{\mathcal{V}}}) \subseteq \gamma([C]^{\overleftarrow{\mathcal{V}}})$ as required. □

Theorem 2.11 thus allows us to verify a circuit using STE by verifying that $[\![A]\!]^{\overleftarrow{\mathcal{V}}} \preceq [C]^{\overleftarrow{\mathcal{V}}}$, or, using the information ordering, $[C]^{\overleftarrow{\mathcal{V}}} \sqsubseteq [\![A]\!]^{\overleftarrow{\mathcal{V}}}$. In Definition 2.12 we introduce a short form of writing $M \models [\![A]\!]^{\overleftarrow{\mathcal{V}}} \Rightarrow [C]^{\overleftarrow{\mathcal{V}}}$. It is used henceforth, and is also the notation commonly seen in other publications on STE.

Definition 2.12 (Satisfaction under an Assignment)**.** *A circuit M satisfies $A \Rightarrow C$ under the assignment $\overleftarrow{\mathcal{V}}$, written $\overleftarrow{\mathcal{V}}, M \models A \Rightarrow C$, if $[C]^{\overleftarrow{\mathcal{V}}} \sqsubseteq [\![A]\!]^{\overleftarrow{\mathcal{V}}}$.*

Note that here the set of free variables $\mathcal{V}$ includes all free variables in A and C. Some of the same variables may – and usually will – be used in the antecedent and the consequent. More details on the effect of sharing these variables is given in Section 2.6.

Also, in the beginning of Section 2.3 we introduced the information order $\sqsubseteq$ as the reverse of the order $\preceq$, which we had used when formalising the connection between binary and ternary simulation. By extending the $\preceq$ order to sequences as well, we can formulate a verification satisfaction that looks more similar to the definition given in Section 2.2.3:

$$\overleftarrow{\mathcal{V}}, M \models A \Rightarrow C \;\Leftrightarrow\; [\![A]\!]^{\overleftarrow{\mathcal{V}}} \preceq [C]^{\overleftarrow{\mathcal{V}}}$$

where

$$\sigma_1 \preceq \sigma_2 \;\Leftrightarrow\; \forall t \in \mathbb{N} : \sigma_1(t) \preceq \sigma_2(t)$$

Satisfaction is defined in terms of the information order $\sqsubseteq$, as it is the order used in most of the literature on Symbolic Trajectory Evaluation, presumably because the first publication on STE introduced this order [23].

Finally, we say that a circuit has the property $A \Rightarrow C$ if it satisfies it irrespective of the assignment to the free variables of the antecedent and consequent:

Definition 2.13. *A circuit M satisfies the STE statement $A \Rightarrow C$ if it does so for all assignments $\overleftarrow{\mathcal{V}}$ to the free variables of the TEL formulae A and C:*

$$M \models A \Rightarrow C \;:\Leftrightarrow\; \forall \overleftarrow{\mathcal{V}} :\; \overleftarrow{\mathcal{V}}, M \models A \Rightarrow C$$

If it is clear which circuit is being verified, we may write $\models A \Rightarrow C$.

2.6 Symbolic Indexing

As seen in Figure 2.10, TEL formulae may use propositional formulae to guard other TEL formulae: $G \rightarrowtriangle f$. Using such guards introduces free Boolean variables that allow us to attach symbolic values to nodes. For example, the formula $(v_i \rightarrowtriangle n_i \text{ is } 1) \text{ and } (\overline{v_i} \rightarrowtriangle n_i \text{ is } 0)$ can be interpreted as requiring the node n_i to have the same logic value as the Boolean valuation of v_i. We introduce the shorthand notation

$$n \text{ is } v_i \;\Leftrightarrow (v_i \rightarrowtriangle n \text{ is } 1) \text{ and } (\overline{v_i} \rightarrowtriangle n \text{ is } 0)$$

where n is a node and v_i is a binary variable. In this dissertation we say that n is a *symbolic constant*: it has a symbolic value, which is a simple variable, with two possible Boolean values. This notation can also be used when attaching a whole expression to n: $n \text{ is } P$, where P is a formula of logic.

We call the use of guards *symbolic indexing*. This is motivated by the fact that a TEL formula f with guards essentially encodes several formulae in one. The assignment $\overleftarrow{\mathcal{V}}$ to the free variables $\mathcal{V}$ of f then specifies which encoded formula without guards applies. So the values of the free variables in the guards of f give an indexing of all the formulae encoded by it.

Example: A TEL formula that uses guards
The TEL formula $(n_1 \text{ is } v_1) \text{ and } (n_2 \text{ is } v_2)$ encodes four TEL formulae:

1. The assignment $a \mapsto \mathsf{ff}, b \mapsto \mathsf{ff}$ delivers the TEL formula $(n_1 \text{ is } 0) \text{ and } (n_2 \text{ is } 0)$.
2. The assignment $a \mapsto \mathsf{ff}, b \mapsto \mathsf{tt}$ delivers the TEL formula $(n_1 \text{ is } 0) \text{ and } (n_2 \text{ is } 1)$.
3. The assignment $a \mapsto \mathsf{tt}, b \mapsto \mathsf{ff}$ delivers the TEL formula $(n_1 \text{ is } 1) \text{ and } (n_2 \text{ is } 0)$.
4. The assignment $a \mapsto \mathsf{tt}, b \mapsto \mathsf{tt}$ delivers the TEL formula $(n_1 \text{ is } 1) \text{ and } (n_2 \text{ is } 1)$.

★

In this example we saw how guards can be used to define symbolic constants. This does not demonstrate the full potential of guards, which we now discuss. Realising this full potential lies at the heart of the contributions of this dissertation.

Example: More elaborate symbolic indexing
Figure 2.14 shows a circuit that returns true in the next cycle if and only if any of the inputs n_2, n_3, or n_4 are equal to the input n_1 in the current cycle. Suppose we

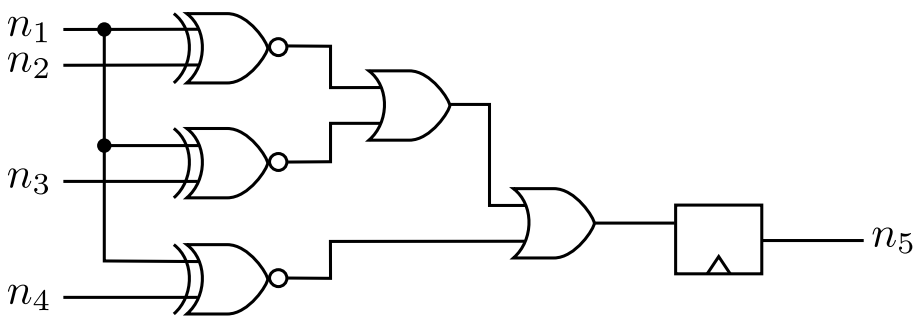


Figure 2.14: A circuit that returns true if and only if any of the inputs n_2, n_3, n_4 are equal to the input n_1

want to verify that if n_1 and n_2 have the same value, then the output n_5 is 1. Then we can choose the following antecedent A_2 and consequent C:

$$A_2 = (n_1 \text{ is } v_1) \text{ and } (n_2 \text{ is } v_1)$$
$$C = \mathsf{N}(n_5 \text{ is } 1)$$

Now suppose we want to verify that if any of the second to fourth inputs are equal to the first input, then the output n_5 will then be 1. This requires two more antecedents similar to the one above, $A_3 = (n_1 \text{ is } v_1) \text{ and } (n_3 \text{ is } v_1)$, and $A_4 = (n_1 \text{ is } v_1) \text{ and } (n_4 \text{ is } v_1)$. We can then check that $M \models A_i \Rightarrow C,\ i \in \{2, 3, 4\}$ separately. But with symbolic indexing this can also be expressed with a single STE run, as follows:

$$\begin{aligned} A = (n_1 \text{ is } v_1) &\text{ and } (\overline{v_2} \wedge \overline{v_3} \rightarrow n_2 \text{ is } v_1) \\ &\text{and } (\overline{v_2} \wedge v_3 \rightarrow n_3 \text{ is } v_1) \\ &\text{and } (v_2 \rightarrow n_4 \text{ is } v_1) \end{aligned}$$

Notice how the assignment $v_2 \mapsto \mathsf{ff}, v_3 \mapsto \mathsf{ff}$ leads to the same STE run as using A_2 as the antecedent, $v_2 \mapsto \mathsf{ff}, v_3 \mapsto \mathsf{tt}$ leads to the STE run using A_3, and $v_2 \mapsto \mathsf{tt}$ leads to the STE run using A_4 irrespective of the assignment to v_3. ★

In actual implementations of STE, all included runs are computed simultaneously, rather than separately simulating each STE run under one variable assignment. In Section 2.7 we detail how these parallel computations can be carried out. Thereafter we provide an example on how different symbolic indexings affect simulation costs.

2.7 Representation

The efficiency of computing the result of STE runs that include symbolic indexing heavily depends on how ternary values and the binary guard formulae are represented. In the following we introduce the most common representation for this, which is based on a dual rail encoding for ternary values, and binary decision diagrams for binary formulae [33]. It is also the representation we assume is being used throughout this dissertation. It is not the only representation, though. To highlight this we briefly describe a second family of representations, which encode the verification problem in non-canonical binary expressions to be examined by a satisfiability solver.

Dual Rail Encoding

A ternary elements $t \in \mathbb{T}$ can be encoded with a pair of binary values using the dual rail encoding. In this dissertation we assume the encoding shown in Figure 2.15.

$$dual\_rail : \mathbb{T} \to \mathbb{B}^2, \begin{cases} \mathrm{X} \mapsto (\mathsf{tt}, \mathsf{tt}) \\ \mathsf{tt} \mapsto (\mathsf{tt}, \mathsf{ff}) \\ \mathsf{ff} \mapsto (\mathsf{ff}, \mathsf{tt}) \end{cases}$$

Figure 2.15: Dual rail encoding of ternary values

Intuitively, the first entry of the pair signifies whether a true value is allowed, and the second whether a false one is allowed. If both values are allowed, this leads to an indeterminate value; otherwise the corresponding binary value is represented. Note that the pair $(\mathsf{ff}, \mathsf{ff})$ does not represent any ternary value. When it is necessary to represent the overconstraint value $\top$, that encoding may be used. The intuitive explanation also works here: neither Boolean value is allowed.

As seen in Figure 2.16, this encoding enables an efficient computation of all basic operations required for STE. The following example makes conjunction and disjunction of dual rail pairs more explicit.

$$
\begin{array}{ll}
\overline{(H, L)} & = (L, H) \\
(H_1, L_1) \wedge (H_2, L_2) & = (H_1 \wedge H_2, L_1 \vee L_2) \\
(H_1, L_1) \sqsubseteq (H_2, L_2) & \Leftrightarrow (H_1 \rightarrow H_2) \wedge (L_1 \rightarrow L_2) \\
(H_1, L_1) \sqcup (H_2, L_2) & = (H_1 \vee H_2, L_1 \vee L_2) \\
(H_1, L_1) \sqcap (H_2, L_2) & = (H_1 \wedge H_2, L_1 \wedge L_2)
\end{array}
$$

Figure 2.16: Computation of negation, conjunction, ordering, and upper and lower bounds using the dual rail encoding.

Example: Conjunction and disjunction of dual rail pairs

	$\wedge$	ff $(\mathsf{ff}, \mathsf{tt})$	tt $(\mathsf{tt}, \mathsf{ff})$	X $(\mathsf{tt}, \mathsf{tt})$
ff	$(\mathsf{ff}, \mathsf{tt})$	$(\mathsf{ff}, \mathsf{tt})$	$(\mathsf{ff}, \mathsf{tt})$	$(\mathsf{ff}, \mathsf{tt})$
tt	$(\mathsf{tt}, \mathsf{ff})$	$(\mathsf{ff}, \mathsf{tt})$	$(\mathsf{tt}, \mathsf{ff})$	$(\mathsf{tt}, \mathsf{tt})$
X	$(\mathsf{tt}, \mathsf{tt})$	$(\mathsf{ff}, \mathsf{tt})$	$(\mathsf{tt}, \mathsf{tt})$	$(\mathsf{tt}, \mathsf{tt})$

	$\vee$	ff $(\mathsf{ff}, \mathsf{tt})$	tt $(\mathsf{tt}, \mathsf{ff})$	X $(\mathsf{tt}, \mathsf{tt})$
ff	$(\mathsf{ff}, \mathsf{tt})$	$(\mathsf{ff}, \mathsf{tt})$	$(\mathsf{tt}, \mathsf{ff})$	$(\mathsf{tt}, \mathsf{tt})$
tt	$(\mathsf{tt}, \mathsf{ff})$	$(\mathsf{tt}, \mathsf{ff})$	$(\mathsf{tt}, \mathsf{ff})$	$(\mathsf{tt}, \mathsf{ff})$
X	$(\mathsf{tt}, \mathsf{tt})$	$(\mathsf{tt}, \mathsf{tt})$	$(\mathsf{tt}, \mathsf{ff})$	$(\mathsf{tt}, \mathsf{tt})$

★

2.7.1 Binary Decision Diagrams

It still remains to explain how binary values are represented, especially if they are symbolic. In [34] Lee introduced the notion of Binary Decision Diagrams (BDDs). These are rooted, directed, acyclic graphs, which have internal decision nodes, and leaves with the label 0 or 1. Each internal decision node is labelled with its corresponding variable name, and has two child nodes, which represent an assignment of the variable to 0 and 1 respectively. Bryant introduced efficient algorithms for BDDs in [35]. In particular, he introduced the notion of reduced, ordered BDDs (ROBDDs). A BDD is *ordered* if for all paths from the root to one of the leaves the variables appear in the same order. A BDD can be *reduced* by merging isomorphic subgraphs, and eliminating nodes when both children point to the same node. In this dissertation all BDDs are ordered and reduced.

Example: Reduced Ordered Binary Decision Diagrams
Figure 2.17 shows two reduced, ordered BDDs for $(v_1 \vee v_2) \wedge v_3 \wedge v_4$. Note how there are several BDDs that represent the same binary, symbolic value, and that depending on the variable order the BDD can have a different number of internal nodes.

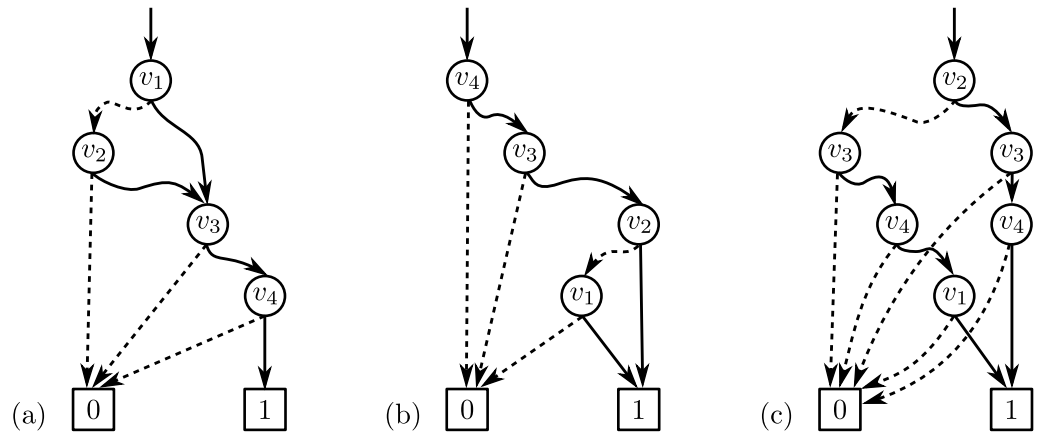


Figure 2.17: BDD for $(v_1 \vee v_2) \wedge v_3 \wedge v_4$ using the variable ordering (a) v_1, v_2, v_3, v_4, (b) v_4, v_3, v_2, v_1, and (c) v_2, v_3, v_4, v_1

First consider Figure 2.17 (a). The branches that assign a low value to the variable are marked with dashed lines, and those that assign a high value are marked with solid ones. Notice that if v_1 is true, then the value does not depend on the value of v_2 anymore. So the node for v_2 is omitted, and the high branch of v_1 goes directly to the node for v_3. Similarly, the low branch of v_3 directly leads to 0, because irrespective of the value of v_4 the value is 0. Finally, several nodes point to a single leaf for 0. This is so, because all leaves with a label 0 are isomorphic, and are hence merged. ★

As already seen in the small example above, the size of the BDD depends on the variable ordering used. Given a binary value that depends on k variables the BDD can have up to 2^k nodes, possibly only because an unsuitable ordering was chosen. Unfortunately, finding the best ordering is NP-hard [36].

In STE, ternary values are usually stored as a pair of BDDs. This allows simultaneously computing the result of STE statements that include guarded formulae, because we do not need assignments to the variables introduced by the guards of the TEL formulae. All assignments are captured by the BDDs. When choosing symbolic indexings for STE, we want them to lead to small BDDs. Then both time and memory costs decrease. In particular, using fewer variables in an indexing potentially leads

to smaller BDDs. Also, if the simulation result does not depend on all the variables specified in the indexing, the resulting BDD is often smaller. Finding such indexings is especially valuable.

Example: Reducing the size of BDDs using Symbolic Indexing
If we want to verify that the circuit depicted in Figure 2.14 (page 45) returns a high output if and only if any of the inputs n_2, n_3, n_4 matches the input n_1, then the following symbolic indexing can be used:

$$\begin{aligned} A_1 = \quad & (n_1 \text{ is } v_1) \qquad \text{and } (\overline{v_2} \wedge \overline{v_3} \rightarrowtriangle n_2 \text{ is } v_1) \\ & \qquad\qquad\quad \text{and } (\overline{v_2} \wedge v_3 \rightarrowtriangle n_3 \text{ is } v_1) \\ & \qquad\qquad\quad \text{and } (v_2 \wedge \overline{v_3} \rightarrowtriangle n_4 \text{ is } v_1) \\ & \qquad\qquad\quad \text{and } (v_2 \wedge v_3 \rightarrowtriangle (n_2 \texttt{ is } \overline{v_1}) \text{ and } (n_3 \texttt{ is } \overline{v_1}) \text{ and } (n_4 \texttt{ is } \overline{v_1})) \\ C_1 = \quad & n_5 \text{ is } \overline{v_2 \wedge v_3} \end{aligned}$$

Forte is a verification environment developed by Intel® [37]. It includes an implementation of BDD-based STE. Simulating the circuit with the antecedent A_1 in the Forte environment results in the pair $(\overline{v_2} \vee \overline{v_3}, v_2 \wedge v_3)$ on node n_5. Note that this pair represents the symbolic value $\overline{v_2 \wedge v_3}$, as seen in the consequent C_1. Indeed, only if the simulation result implies the consequent does the verification succeed. More importantly for this example, though, is that the variable v_1 used in the symbolic indexing does not appear in the simulation result. This simplification was possible, because we used v_2 and v_3 for enumerating cases, and v_1 only for comparing the first input with the other ones.

In contrast, suppose we had represented each input with an individual variable:

$$\begin{aligned} A_2 = \quad & (n_1 \text{ is } v_1) \text{ and } (n_2 \text{ is } v_2) \text{ and } (n_3 \text{ is } v_3) \text{ and } (n_4 \text{ is } v_4) \\ C_2 = \quad & n_5 \text{ is } \quad (v_1 \wedge v_2) \vee (v_1 \wedge v_3) \vee (v_1 \wedge v_4) \vee \\ & \qquad\quad (\overline{v_1} \wedge \overline{v_2}) \vee (\overline{v_1} \wedge \overline{v_3}) \vee (\overline{v_1} \wedge \overline{v_4}) \end{aligned}$$

Simulating the circuit with the antecedent A_2 in the Forte environment results in the pair (H, L) on n_5 where

$$\begin{aligned} H &= (v_1 \wedge v_2) \vee (v_1 \wedge v_3) \vee (v_1 \wedge v_4) \vee (\overline{v_1} \wedge \overline{v_2}) \vee (\overline{v_1} \wedge \overline{v_3}) \vee (\overline{v_1} \wedge \overline{v_4}) \\ L &= (v_1 \wedge \overline{v_2} \wedge \overline{v_3} \wedge \overline{v_4}) \vee (\overline{v_1} \wedge v_2 \wedge v_3 \wedge v_4). \end{aligned}$$

Figure 2.18 shows BDDs that represent the high rail of each of the results: (a) with a good encoding of the symbolic indexing, and (b) with a variable per input. In

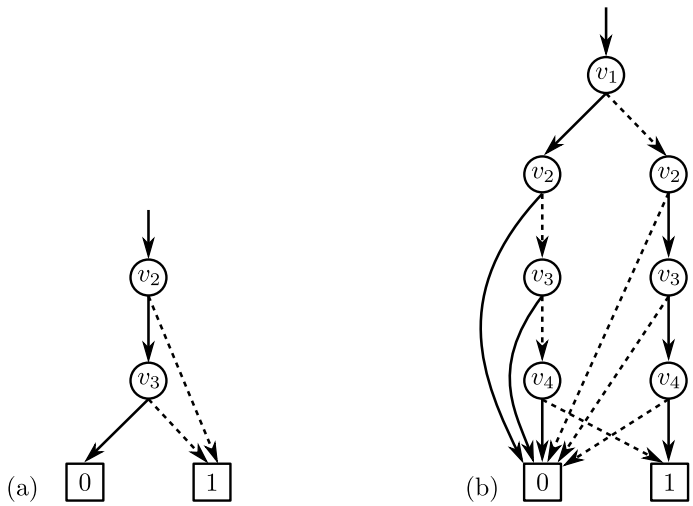


Figure 2.18: BDD for the high rail when simulating the circuit with respect to (a) A_1, and (b) A_2

our case the low rail is the exact complement, so the BDDs look almost the same – only the leaves 0 and 1 are swapped. Note that we chose an optimal ordering for the BDD returned when simulating with A_2. The ordering v_2, v_3, v_4, v_1 would result in a significantly bigger BDD. ★

Thus, two aspects influence verification costs substantially: first, which symbolic indexing is used to encode the property the circuit should meet; and second, which variable ordering is used. All experimental results collected in this dissertation do not specify the variable ordering, thus letting the implementation of STE decide which ordering to apply. Finding good orderings for ROBDDs is an area of research in itself. We concentrate on the aspect of creating good symbolic indexings.

2.7.2 SAT-based STE

Recall that when verifying $\models A \Rightarrow C$, implementations of Symbolic Trajectory Evaluation first compute the weakest trajectory that satisfies the antecedent A, and then check whether that trajectory also satisfies the consequent C. While commonly these computations are done using a dual rail encoding and BDDs to represent the values of each rail as described above, other approaches exist. Notably, Symbolic Trajectory Evaluation can also be done with a SAT-based check. Several different approaches exist that utilise a satisfiability solver.

One approach is to encode the ternary domain using two variables. The simulator then works on non-canonical binary expressions to determine an expression that

represents the weakest trajectory. Checking whether $\models A \Rightarrow C$ holds is then done by feeding a propositional formula into a SAT-solver that says there exists an assignment such that the weakest trajectory does not satisfy the consequent. Thus, if a satisfying assignment is found, the design is not correct. Bjesse et al. implemented this approach and used it to find bugs in an Alpha microprocessor [38].

A second option is to work on all trajectories that satisfy the antecedent, rather than computing the weakest one. Roorda and Claessen suggest this approach and present an algorithm, which computes a constraint that has no solution if and only if $\models A \Rightarrow C$ holds. This, they argue, delivers a representation of the verification problem that SAT-solvers can solve more efficiently, and thus results in better performance [39].

Further alternatives for using satisfiability solvers in STE have been suggested and put to the test. For example, Roorda uses a 3-valued SAT-solver, which thus does not require two variables to represent the ternary domain [40]. Grumberg et al. leverage the Circuit-SAT method, which is based on a justification algorithm that searches for partial input combinations that force a specific node to a given value, to present another variant of SAT-based STE [41].

This dissertation, however, is restricted to BDD-based Symbolic Trajectory Evaluation. In particular, all experimental results presented in Chapters 5 and 8 reflect STE runs using a dual rail encoding and BDDs. Despite extensive experimental investigation of SAT-based STE, we did not find a way to exploit good symbolic indexing encodings via SAT.

2.8 History of STE

Symbolic Trajectory Evaluation, as described above, was introduced by Bryant, Beatty, and Seger in 1991 [23, 42, 43]. These publications give a first description of STE, but the underlying theory is not explained fully.

In 1999, Chou then elaborated the theory of STE, giving both a set-theoretic and a lattice-theoretic viewpoint [32]. While the set-theoretic approach is impractical, it does provide an understandable description of STE. Circuits are seen as functions that maps sets of configurations to the next possible set of configurations. These configurations work over the Boolean domain. Assertions are described by a quintuple. The set of relevant states, a designated initial state, and a transition relation specify which trajectories are valid. The antecedent and the consequent function map each

state to a set of configurations. A circuit then satisfies an assertion if every trajectory that complies with the antecedent also fulfils the consequent.

For the lattice-theoretic description, rather than working over the Boolean domain, ternary vectors are used. This reduces the problem size considerably, but comes at the price of information loss. Hence assertions have to be formulated differently, or a family of assertions is necessary instead. The ternary model itself is shown to be an abstract interpretation of the Boolean model via a Galois connection [30]. This means properties can be proved on an abstraction to then conclude the same properties hold for the concrete model. The lattice-theoretic approach is more closely related to actual implementations of STE.

In 2006, Roorda and Claessen published the closure semantics of STE [44]. This work is motivated by the fact that the previously stated semantics is not faithful to what implementations actually do. They give a simple example where, according to the published semantics, the assertion cannot be verified, but known STE implementation succeed. The key difference between the two semantics is as follows. The previous semantics only specifies how information is propagated between nodes from one time-point to the next. The closure semantics complete this by also describing what happens inside combinational logic in between the two time points. The closure semantics introduced is faithful to the proving power of STE algorithms, which can track and specify logic implications between nodes at the same time-point.

The first implementation, called VOSS, was established by Seger [33]. A strongly typed, higher order, lazy functional programming language called FL was introduced. This allows the organisation of large proof tasks and efficient scripting. At the same time, FL can be used as an expressive specification language. In this system, STE is built in using ordered binary decision diagrams (OBDDs). The ternary domain is represented via the dual-rail encoding introduced above: the first formula specifies whether the value is true, the second whether the value is false; if both of these hold, the value is indeterminate.

VOSS was then extended to produce Intel®'s Forte environment [37], which used to be publicly available for noncommercial use. A seamless integration of both STE and theorem proving was accomplished [45]. For this, FL was lifted: it is used both as the object and the meta language. This enables both the execution of functions and reasoning about them. Essentially, any expression that evaluates to true can be transformed into a theorem. Thus, a tight integration between theorem proving and

model checking is achieved. A complete description of the Forte system, including theory, usage methodology, and case studies, was published in 2005 [46].

The first big industrial breakthrough for STE was accomplished in 1999: the complete formal verification of floating-point arithmetic hardware of the Intel® Pro processor against IEEE-level specifications [47]. This clearly showed the potential of Symbolic Trajectory Evaluation, and greatly motivated the work that followed on STE. Numerous other success-stories on industrial designs have also been published [25, 26, 45, 48–55].

In 2001, Bjesse et al. implemented Symbolic Trajectory Evaluation using a different representation. Namely, rather than using BDDs, as in VOSS, the verification problem is encoded as a satisfiability problem. If there exists an assignment to the variables for a specially constructed binary formulae, then the hardware does not meet its specification. Bjesse et al. successfully used this implementation to find bugs in an alpha microprocessor [38]. Further approaches, in which STE uses satisfiability solvers to determine the verification output have been presented since [39, 41].

More recently, work has concentrated on automation techniques for STE. Roorda and Claessen propose a SAT-based algorithm that helps users manually refine abstractions [56]. If the abstraction used in an STE run retains too little information, the proposed algorithm calculates a strengthening using SAT. Tzoref and Grumberg describe how to determine which inputs to drive using a symbolic value when over-abstraction occurs [27]. These inputs are chosen using heuristics that distinguish between control nodes and data nodes. Chockler, Grumberg, and Yadgar suggest another heuristic, which selects input to refine the abstraction by using the concept of "degree of responsibility" [3]. Notably, [56] gives guidance to the user only, while [27] and [3] propose an automatic refinement loop. However, the latter cannot handle complex encodings of abstraction families. Parts of this dissertation builds on this work, and amongst other extensions addresses how to handle such complex encodings.

Finally, *Generalised Symbolic Trajectory Evaluation* (GSTE) [57] was introduced in 2001. It extends Symbolic Trajectory Evaluation in that it has a more powerful logic to express properties. Notably, while STE only allows properties over finite time intervals, GSTE supports properties over unbounded time intervals, too. Some work has been done to clarify and further formalise GSTE [58–63], as well as suggest refinement techniques [28, 64]. In this dissertation, however, we concentrate on Symbolic Trajectory Evaluation only.

2.8.1 Other Abstraction Frameworks

Easing verification by abstraction, as well as refining these abstractions, is a common idea not just found in Symbolic Trajectory Evaluation. A wealth of research has focused on how to make feasible the verification of increasingly large and complex designs through abstraction. An extensive summary of all this work is not appropriate here, but some excellent surveys of different approaches have been published previously; we mention some of these here.

In [65] Grumberg surveys abstraction and refinement in model checking. In particular, she concentrates on existential abstraction, which over-approximates the behaviours of concrete models. The paper focuses on abstraction functions only, i.e., every concrete state may only be represented by one abstract state. In particular, this excludes Symbolic Trajectory Evaluation, which commonly covers a concrete state multiple times, by the nature of ternary simulation based on the indeterminate value X.

Predicate abstraction is a popular abstraction technique especially for software model checking [21]. Skilfully selected predicates are used to extract finite state models from infinite state systems. The same technique can also be applied to hardware designs, which are finite state, but at a very large scale. Clarke et al. describe how SAT-based predicate abstraction can used in hardware verification [66]. In particular, they also describe a counterexample-guided abstraction refinement loop, by which spurious behaviour is eliminated.

Other work concentrates on verifying hardware, where the concrete model is presented at a higher level of abstraction, thus moving away from the low-level representation of netlists. Kroening and Seshia survey such techniques in [67], presenting word-level and term-level verification with predicate abstraction and satisfiability modulo theories (SMT).

This dissertation, however, concentrates solely on verification of hardware at the level of netlists.

2.9 Reindexing in STE

In Section 2.6 we described symbolic indexing, which allows combining several TEL formulae to one by introducing propositional formulae called guards. The choice of appropriate guards can significantly reduce verification costs, which, for BDD-based STE, greatly depend on the size of the BDDs used. The size in turn is reliant on the

number of variables, their ordering, and the expressions on those variables that need to be stored. The choice of guards impact two of these three aspects, namely the set of variables, and the initial expressions, which are used in the simulation. So a good symbolic indexing keeps memory costs low, and thus potentially enables verifying circuits previously classified as infeasible.

But symbolic indexing is not used nearly as often as it is applicable [2]. Melham and Jones identify the two main reasons for this: it is hard to find good indexings, and results obtained by indexed STE runs cannot be composed. To address the second hurdle, they developed a theory of how to switch between different indexings, a procedure called *reindexing*. This paves the way for wider use of symbolic indexing by enabling compositionality.

Reindexing is motivated by the fact that there are multiple options for guards to essentially encode the same TEL formula in that they enumerate the same set of TEL formulae. For example,

$$f[\mathcal{V}] = (v_1 \wedge v_2 \rightarrow n_1 \text{ is } 1 \text{ and } n_2 \text{ is } 0) \text{ and } (\overline{v_1} \wedge \overline{v_2} \rightarrow n_1 \text{ is } 1) \text{ and } (\overline{v_1} \wedge v_2 \rightarrow n_2 \text{ is } 0)$$

and

$$f'[\mathcal{X}] = (x_1 \rightarrow n_1 \text{ is } 1) \text{ and } (\overline{x_2} \rightarrow n_2 \text{ is } 0)$$

both encode four different, simpler TEL formulae:

TEL formula	$\overleftarrow{\mathcal{V}}$	$\overleftarrow{\mathcal{X}}$
n_1 is 1 and n_2 is 0	$\{v_1 \mapsto \mathsf{tt}, v_2 \mapsto \mathsf{ff}\}$	$\{x_1 \mapsto \mathsf{tt}, x_2 \mapsto \mathsf{tt}\}$
n_1 is 1	$\{v_1 \mapsto \mathsf{tt}, v_2 \mapsto \mathsf{tt}\}$	$\{x_1 \mapsto \mathsf{ff}, x_2 \mapsto \mathsf{ff}\}$
n_2 is 0	$\{v_1 \mapsto \mathsf{ff}, v_2 \mapsto \mathsf{ff}\}$	$\{x_1 \mapsto \mathsf{ff}, x_2 \mapsto \mathsf{tt}\}$
	$\{v_1 \mapsto \mathsf{ff}, v_2 \mapsto \mathsf{tt}\}$	$\{x_1 \mapsto \mathsf{tt}, x_2 \mapsto \mathsf{ff}\}$

Melham and Jones describe how to move from one such formula $f[\mathcal{V}]$ to the other $f'[\mathcal{X}]$ using a *reindexing relation* $R[\mathcal{X}, \mathcal{V}]$. It essentially encodes which assignments $\overleftarrow{\mathcal{X}}$ correspond to the assignments $\overleftarrow{\mathcal{V}}$. So given $f[\mathcal{V}]$ and $R[\mathcal{X}, \mathcal{V}]$, they describe how to construct $f'[\mathcal{X}]$. Their main motivation was allowing compositionality of STE statement for decompositional verification of designs. In Chapter 3 we argue that reindexing can also be used to reduce the problem of finding good indexings to that of finding appropriate reindexing relations. Note that Melham and Jones do not provide guidance on how to find reindexing relations, but only how to apply them

once they have been devised. Part of the research reported in this dissertation fills exactly this major automation gap, in essence by computing candidate reindexing relations automatically.

Devising reindexing relations that introduce better symbolic indexings is hard. The verification costs of BDD-based STE highly depend on the size of the BDDs, which in turn is determined by the symbolic indexing. Finding an optimal variable ordering for BDDs is NP-hard already. So finding the reindexing relation that leads to the best symbolic indexing seems to be NP-hard, too.

Suppose we have a reindexing relation $R[\mathcal{X}, \mathcal{V}]$. Given an STE task to verify that $M \models A \Rightarrow C$, Melham and Jones describe how to replace the guards in the antecedent $A[\mathcal{V}]$ and the consequent $C[\mathcal{V}]$ by new propositional formulae over the set of variables $\mathcal{X}$. They prove that if the transformed STE run succeeds, then the original one, $\models A \Rightarrow C$, also holds. This requires the reindexing relation to have specific properties; it needs to express which assertions of the future guards handle each and every assertion of the previous guards, to ensure that all cases checked before are still covered by the new verification.

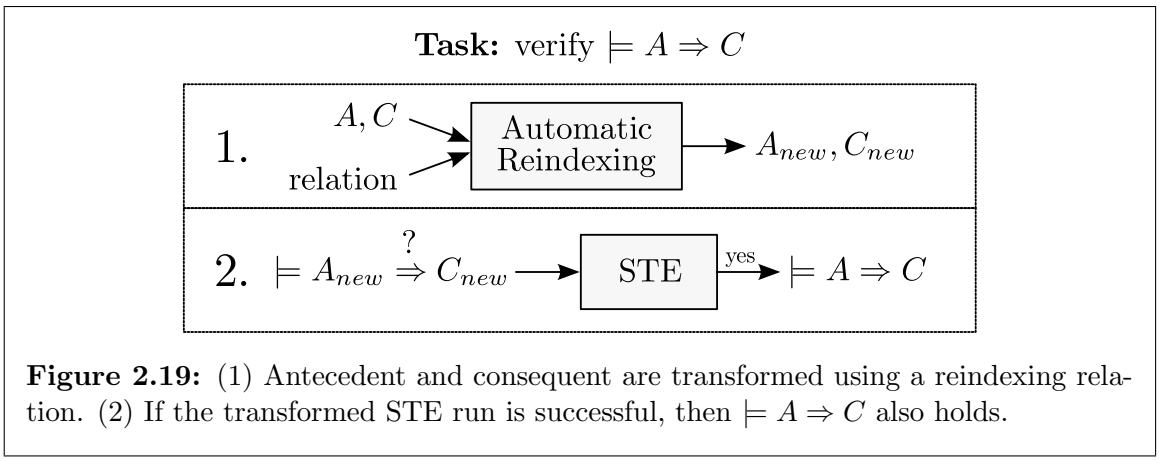


Figure 2.19: (1) Antecedent and consequent are transformed using a reindexing relation. (2) If the transformed STE run is successful, then $\models A \Rightarrow C$ also holds.

The reindexing algorithm updates both the antecedent and the consequent of the STE run using the provided relation. During this update, the verification task $M \models A \Rightarrow C$ must not lose strength, and updating the antecedent and the consequent require different operations. *Antecedent weakening*, that is relaxing the initial requirements specified by the antecedent, sustains or increases strength:

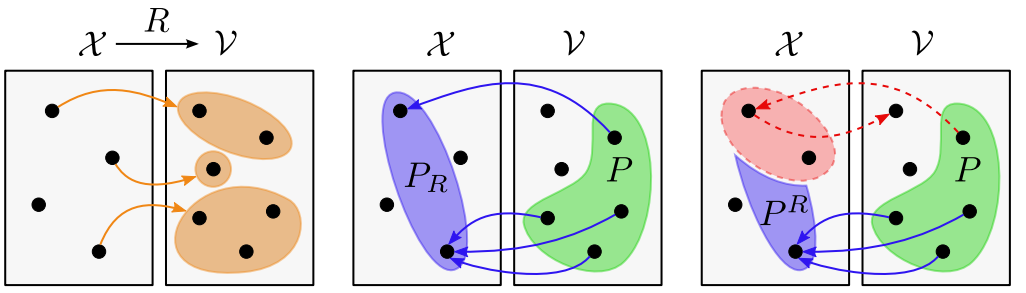
$$\text{if} \quad [A_{old}]^{\overleftarrow{\mathcal{V}}} \sqsubseteq [A_{new}]^{\overleftarrow{\mathcal{V}}} \quad \text{and} \quad \overleftarrow{\mathcal{V}} \models A_{new} \Rightarrow C \quad \text{then} \quad \overleftarrow{\mathcal{V}} \models A_{old} \Rightarrow C$$

Similarly, *consequent strengthening*, that is increasing the demands postulated by the

consequent, meets the same requirement:

$$\text{if} \quad [C_{new}]^{\overleftarrow{\mathcal{V}}} \sqsubseteq [C_{old}]^{\overleftarrow{\mathcal{V}}} \quad \text{and} \quad \overleftarrow{\mathcal{V}} \models A \Rightarrow C_{new} \quad \text{then} \quad \overleftarrow{\mathcal{V}} \models A \Rightarrow C_{old}$$

Melham and Jones weaken and strengthen STE assertions by computing certain preimages of the provided relation. The relation expresses which new indexing cases, dependent on the variables $\mathcal{X}$, correspond to the old indexing cases, which are dependent on the variables $\mathcal{V}$. The *weak preimage* P_R of a predicate P expresses which new indexing cases *can* represent old indexing cases that satisfy the predicate. The *strong preimage* P^R, on the other hand, captures which new indexing cases *must* represent old indexing cases that satisfy the predicate. So P^R is a subset of P_R that excludes all indexing cases that can also represent old indexing cases that do not satisfy the predicate.



(1) reindexing relation R (2) weak preimage P_R of P (3) strong preimage P^R of P

Figure 2.20: (1) The reindexing relation encodes which new indexing cases cover the old ones. (2) The weak preimage P_R includes all new cases that can correspond to an old case that satisfies the predicate P. (3) The strong preimage $P^R \subseteq P_R$ excludes all new cases that can correspond to old cases that do not satisfy P.

Figure 2.20 visualises the connection between weak and strong preimage. More formally, weak and strong preimage are defined by

$$P_R[\mathcal{X}] = \exists \overleftarrow{\mathcal{V}}.R(\overleftarrow{\mathcal{V}})[\mathcal{X}] \wedge P(\overleftarrow{\mathcal{V}}) \quad \text{and} \quad P^R[\mathcal{X}] = P_R[\mathcal{X}] \wedge \overline{\overline{P}_R[\mathcal{X}]}.$$

These computations have two effects. First, the updated predicate depends on a different set of variables. Second, Melham and Jones showed that computing the weak preimage leads to a weakening, and computing the strong preimage leads to a strengthening of STE assertions [2]. This facilitates their main theorem:

Theorem 2.14 (Automatic Reindexing [2])**.** *Let A and C be TEL formulae containing guards on the set of variables $\mathcal{V}$. If $R[\mathcal{X}, \mathcal{V}] \models A_R \Rightarrow C^R$ and the reindexing*

relation satisfies the coverage condition

$$\forall \overleftarrow{\mathcal{V}}.\exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}),$$

then $\models A \Rightarrow C$ *holds.*

The coverage condition $\forall \overleftarrow{\mathcal{V}}.\exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$ ensures that the new indexing scheme covers all previous indexing cases. The relation depicted in Figure 2.20 (1) meets this requirement: all cases in $\mathcal{V}$ have a preimage.

Theorem 2.9 also motivates a different intuition as to what the reindexing relation expresses. Given an assignment $\overleftarrow{\mathcal{X}}$ for the variables $\mathcal{X}$, the relation states which old indexing cases are checked by the transformed STE run:

$$\text{If} \quad \overleftarrow{\mathcal{X}} \models A_R \Rightarrow C^R \quad \text{then} \quad R(\overleftarrow{\mathcal{X}})[\mathcal{V}] \models A \Rightarrow C.$$

Summing up, applying a new indexing scheme to an STE run requires (1) a relation that encodes which new indexing cases cover the old indexing cases, and (2) the assurance that the new indexing cases cover all old indexing cases. The second requirement is equivalent to checking whether the coverage condition $\forall \overleftarrow{\mathcal{V}}.\exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$ is satisfied.

Melham and Jones formalised the process of reindexing STE assertions. Their method ensures that the transformed STE run is sufficient for proving the desired properties, given the coverage condition holds. Verifying this additional requirement is machine-checkable, which allows automation in principle. But, as mentioned, the reindexing algorithm represents only a part of what is needed for full automation in that a suitable relation has to be provided. The problem of finding good indexing schemes itself was left open by Melham and Jones. Moreover, checking whether the relation satisfies the side condition can be very expensive, since it involves existential quantification over a possibly large number of variables.

In this dissertation we propose an approach for finding good symbolic indexings by injecting the specification into auxiliary STE statements and reindexing them. In particular, our algorithm generates relations that satisfy all required side conditions by construction, thus eliminating a potentially costly check. So we provide a solution to both limiting factors just identified: finding symbolic indexings, and ensuring they are correct. In Chapter 3 we demonstrate that these symbolic indexings are good on the example of three different hardware designs.

Chapter 3

Abstraction Discovery

The characteristic power of verification by STE lies in the application of good abstraction schemes. But finding good abstractions manually is a hard task and has presented a bottleneck to effective use of abstraction in verifying circuits.

This chapter introduces a novel automatic abstraction framework for STE, the first ever solution to fully automatically verifying hardware with complex abstractions in STE. Our approach takes the specification of a circuit and analyses it to generate an STE statement that encodes an abstraction family. If the circuit meets this statement, then the design meets its specification.

The STE statement created typically makes heavy use of non-trivial symbolic indexing. This indexing is extracted from the specification in the hope that it keeps verification costs low. At its core, our approach computes a group of partial input combinations sufficient for determine the output of the specification. Each of these partial input combinations, and the value of the specification for that setting, then constitutes one of the STE statements enumerated by symbolic indexing. We gradually build a symbolic indexing to construct the STE statement, letting the specification guide us to a symbolic indexing that is likely to be advantageous.

At the core of this framework stands an approach similar to Melham and Jones' reindexing method, which we introduced in Section 2.9. Recall that they suggest changing the symbolic indexing of an STE statement by applying a reindexing relation. So they require an initial STE statement $A \Rightarrow C$ and a relation R, which has to satisfy a coverage condition, and then produce a new STE statement $A_R \Rightarrow C^R$. If the design satisfies this statement, then it also satisfies the initial statement. In contrast, our approach takes the *specification* that the circuit is supposed to satisfy. It then automatically generates an auxiliary STE statement and a relation. Finally, it applies that relation to the auxiliary STE statement, and outputs the result. If the

model satisfies the returned statement, then the design meets its specification.

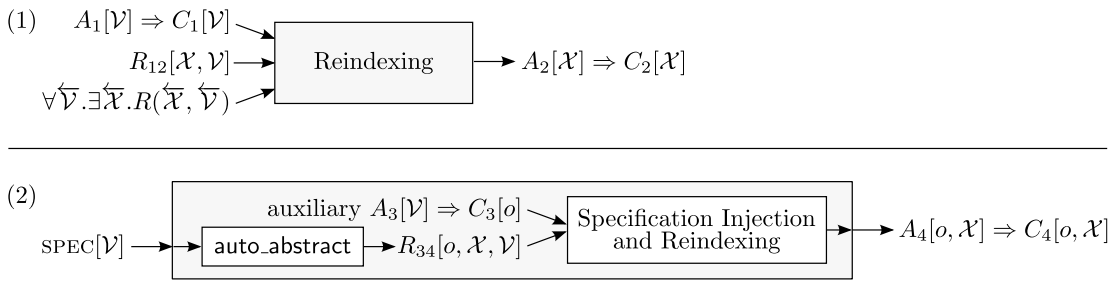


Figure 3.1: (1) Reindexing by Melham and Jones, and (2) our approach to automatic abstraction discovery

Figure 3.1 visualises the connection between the work presented in [2] and our approach. The reindexing we perform as part of our abstraction discovery framework differs significantly from the reindexing introduced by Melham and Jones. We apply the relation to an auxiliary STE statement, in which the antecedent depends on a set of variables $\mathcal{V}$ and the consequent on just a single variable o. The resulting statement then still depends on o, but all occurrences of $\mathcal{V}$ are replaced in the reindexing process. In contrast, Melham and Jones [2] eliminate all variables used in the initial STE statement. Furthermore, our initial statement is independent of the specification, so it is by itself not meaningful. Only after computing the weak and strong preimages do the antecedent and the consequent express the requirements the specification sets. So here the preimage computations change the meaning of the STE statement, whereas Melham and Jones' approach maintained the same meaning. The relation we generate is thus not strictly speaking a reindexing relation, but rather a relation that injects the specification into an auxiliary STE statement while at the same time introducing a symbolic indexing. In the following we will call the relation a *specification injection and reindexing (SIR) relation.*

Still, our approach uses reindexing as a basis. We present our modifications to Melham and Jones' method in Section 3.1. Notably, the coverage condition the relation has to satisfy is more restrictive to ensure correct verification results. In Section 3.2 we then present an automatic abstraction discovery algorithm, auto_abstract, which, given a specification, computes an SIR relation. This relation satisfies the more restrictive side condition by construction, so checking this does not impose additional costs. It also does not require user input. Finally, in Section 3.3, we prove two theorems. They state the correctness of the new approach, as well as our relation satisfying the adjusted coverage condition by construction.

The significance of our approach is that all three manual aspects of Melham and Jones' work [2] are automated away. We do not have to construct the initial STE statement anymore, we do not have to provide a relation, and we do not have to check that the relation covers all observable cases. Both providing a relation and checking its coverage were very significant intellectual and computational limiting factors in the previous approach. Our method is automatic and thus removes both these bottlenecks.

3.1 Abstraction through Reindexing

In Section 2.9 we saw that given a suitable relation, Melham and Jones' algorithm changes the indexing used by STE. In particular, a reindexing relation that encodes the transformation from full symbolic simulation to a more elaborate indexing scheme introduces abstraction. Thus, automatically computing such a relation yields an automatic abstraction mechanism for Symbolic Trajectory Evaluation.

Full symbolic simulation declares that each input node n_i has a symbolic value denoted by a corresponding binary variable v_i: $\bigwedge_i (v_i \rightarrow n_i \texttt{ is } 1) \wedge (\overline{v_i} \rightarrow n_i \texttt{ is } 0)$, or, in short, $\bigwedge_i n_i \texttt{ is } v_i$. In the following, the set of variables used for full symbolic simulation is called $\mathcal{V} = \{v_i : \ i \in I\}$. Suppose that the consequent states that the implementation output equals the value of a Boolean function of these variables, expressed as a formula of propositional logic, $\textsc{spec}[\mathcal{V}]$. Then a successful STE run verifies the full correctness of the implementation:

$$\models \bigwedge_i n_i \texttt{ is } v_i \Rightarrow \mathit{out} \texttt{ is } \textsc{spec}[\mathcal{V}] \tag{3.1.1}$$

Here $\textsc{spec}[\mathcal{V}]$ is the expression representing the specification. Its free variables, $\mathcal{V}$, are exactly those used in the antecedent. Recall that out `is` $\textsc{spec}[\mathcal{V}]$ is shorthand for $(\textsc{spec}[\mathcal{V}] \rightarrow \mathit{out}\ \mathsf{is}\ 1) \wedge (\overline{\textsc{spec}[\mathcal{V}]} \rightarrow \mathit{out}\ \mathsf{is}\ 0)$.

This full simulation of the implementation is not feasible for many circuits, even small ones, due to memory and time constraints. But it does provide a starting point for introducing abstraction via reindexing. Applying a suitable reindexing relation changes the symbolic indexing, and – as full simulation includes no abstraction – will lead to an STE statement that will drive fewer inputs with symbolic values.

Our approach modifies this general idea by reindexing a slightly different STE statement than seen in 3.1.1. This modification allows further automation. In Melham

and Jones' approach [2] it is sufficient to show that

$$\forall \overleftarrow{\mathcal{V}}.\exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{X}},\overleftarrow{\mathcal{V}}) \quad \text{and} \quad R[\mathcal{X},\mathcal{V}] \models (\bigwedge_i n_i \texttt{ is } v_i)^R \Rightarrow \Big(out \texttt{ is } \textsc{spec}[\mathcal{V}]\Big)_R$$

to conclude formal correctness

$$\models \bigwedge_i n_i \texttt{ is } v_i \Rightarrow out \texttt{ is } \textsc{spec}[\mathcal{V}].$$

Recall that $R(\overleftarrow{\mathcal{X}},\overleftarrow{\mathcal{V}})$ stands for the relation where the variables $\mathcal{X}$ are replaced by the values given in the assignment $\overleftarrow{\mathcal{X}}$ and the variables $\mathcal{V}$ are replaced by values given in the assignment $\overleftarrow{\mathcal{V}}$.

In our *specification injection and reindexing* (SIR) approach it is sufficient to show that

$$o \notin \mathcal{X} \qquad \text{and} \qquad \forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{o}.\Big(\overleftarrow{o} = \{o \mapsto \textsc{spec}(\overleftarrow{\mathcal{V}})\} \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{o},\overleftarrow{\mathcal{X}},\overleftarrow{\mathcal{V}})\Big) \tag{3.1.2}$$

and

$$R[o,\mathcal{X},\mathcal{V}] \models (\bigwedge_i n_i \texttt{ is } v_i)^R \Rightarrow (out \texttt{ is } o)_R \tag{3.1.3}$$

to conclude formal correctness.

Simply put, our relation $R[o,\mathcal{X},\mathcal{V}]$ both encompasses the specification and reindexes the full simulation to a more elaborate indexing. The STE run itself is simply:

$$\models \bigwedge_i n_i \texttt{ is } v_i \Rightarrow out \texttt{ is } o, \tag{3.1.4}$$

By itself, STE run 3.1.4 has no expressive power with respect to verifying the circuit, because it contains no data about the specification. This information is encompassed in the SIR relation $R[o,\mathcal{X},\mathcal{V}]$, and in particular the use of the variable o within it.

The algorithm we introduce in Section 3.2 creates an SIR relation. The variable o encodes whether each indexing case leads to a high or low output: the relation received by replacing o with true results in a relation that enumerates all cases where the specification is high and replacing o with false results in all cases where the specification is low. Thus, the relation includes the specification. The trivial side condition $o \notin \mathcal{X}$ ensures that o does not index cases itself, and its value can therefore be chosen

independently. Condition 3.1.2 ensures that o correctly encodes the value of $\textsc{spec}(\overleftarrow{\mathcal{V}})$. A direct consequent of the side condition is $\forall\overleftarrow{\mathcal{V}}.\exists\overleftarrow{\mathcal{X}}.R\big(\{o \mapsto \textsc{spec}(\overleftarrow{\mathcal{V}})\}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}\big)$ and subsumes the previous coverage condition, $\forall\overleftarrow{\mathcal{V}}.\exists\overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$. The fact that the relation holds if and only if this assignment is chosen captures that the relation encodes the specification accurately.

Theorem 3.1 (page 73) formalises this new approach. Provided the SIR relation R correctly and fully captures the specification, i.e., it satisfies 3.1.2, and it covers all observable cases, the STE run 3.1.3 is sufficient for proving formal correctness. This sets the foundation for creating an automatic abstraction relation. Every relation that satisfies these side conditions is suitable for introducing abstraction and at the same time encoding the specification.

3.2 Basic Algorithm

This section introduces a basic algorithm that computes an SIR relation $R[o, \mathcal{X}, \mathcal{V}]$ that satisfies the required side conditions, given by 3.1.2. The structure of the specification guides the abstraction encoded into the relation. More specifically, the abstraction achieved leads to an STE run in which each verification case uses a minimal partial input combination. Only some inputs are driven in each run, and they are minimal in that (1) they are sufficient for the specification output to be concrete, and (2) not specifying any of the inputs that are driven would result in an indeterminate output. So our automatic abstraction algorithm has the task of determining which partial input combinations are sufficient for a concrete output of the specification, and generating a relation that maps these partial input combinations to the symbolic values v_i.

We use the specification to construct a suitable symbolic indexing for verifying a design, because it is more uniform than the implementation: circuit structures designed for speed, power consumption, size, and physical limitations are not involved. Moreover, the specification often presents a simple version of the design, in which these non-functional details are elided, while exhibiting the same input-output behaviour as a correct implementation.

The core idea of our algorithm is to leverage constructs within the specification with controlling values. They are the right candidates for abstraction, because if only a subset of the inputs needs to be determinate for a concrete output, then all other inputs may be driven with the indeterminate value X without leading to over-abstraction.

Applying this idea uniformly across the specification allows us to compute all partial input combinations that lead to determinate outputs of the specification. Ideally, this abstraction also leads to determinate outputs when used to simulate the hardware to verify, although this is not always the case, i.e., over-abstraction may occur. In Chapters 4 and 6 we suggest two different approaches to address this.

We now describe how we compute the relation that stands at the core of our automatic abstraction discovery approach. As it takes the specification as an input, we first clarify which form of specifications we assume.

3.2.1 Form of Specifications

The auto_abstract algorithm, which is introduced in Section 3.2.3, computes an SIR relation by looking at the structure of the given functional specification. For simplicity, we assume the specification is a propositional expression built with three constructors:

$$spec := \text{VAR } v \quad | \quad \text{NOT } spec \quad | \quad spec \text{ AND } spec$$

VAR introduces variables, NOT inverts an expression, and AND conjuncts two expressions. In this dissertation, we call such a Boolean expression *bexpr*, and it is only used for specifications of circuits. We usually denote these by SPEC. In essence, we can use any representation that we can process structurally, and which is not canonical. This allows us to deconstruct the expression without having to execute expensive operations to re-establish the canonical form.

For simplicity, at this point we assume a tree-structure of the specification, rather than a directed graph. As the same variable can be used multiple times in the specification, this does not restrict its expressiveness. Furthermore, we will extend our approach to directed acyclic graphs Section 4.1.

The bexpr type introduced above is also known as an *and-inverter graph* (AIG) [68][1]. We do not to use this name, as we initially work on trees only, and introduce additional constructors, such as XNOR, later on in the dissertation.

[1]This early paper raised the interest in AIGs. While Hellerman calls them NAND circuits, the name AIG established itself over the years.

3.2.2 Shape of SIR Relations

The algorithm takes a specification that depends on a set of variables $\mathcal{V} = \{v_i : i \in I\}$ that correspond to the circuit's inputs. Additionally, the algorithm takes a variable o. It computes a relation of the following general shape:

$$R[o, \mathcal{X}, \mathcal{V}] = \bigwedge_i (H_i[o, \mathcal{X}] \rightarrow v_i) \wedge (L_i[o, \mathcal{X}] \rightarrow \overline{v_i})$$

This enumerates a collection of partial input combinations which lead to a determinate output of the specification. For this a set of fresh indexing variables $\mathcal{X}$ is introduced. The value of the variable o tells us whether each partial input combination leads to a high or low value of the specification.

In essence, H_i includes every indexing case in which the input node n_i needs to be driven with a high value v_i. Dually, L_i includes every indexing case in which it needs to be driven with a low value $\overline{v_i}$. Thus, for each assignment $\overleftarrow{o}$ and $\overleftarrow{\mathcal{X}}$, the relation $R(\overleftarrow{o}, \overleftarrow{\mathcal{X}})$ provides a partial input combination, which states which inputs need to be driven with a concrete value, including whether it needs to be true or false. The assignment $\overleftarrow{o}$ determines whether the output of the specification is high or low for this partial input combination.

Note that each valuation of $\overleftarrow{o}$ and $\overleftarrow{\mathcal{X}}$ leads to at least one of $H_i(\overleftarrow{o}, \overleftarrow{\mathcal{X}})$ and $L_i(\overleftarrow{o}, \overleftarrow{\mathcal{X}})$ being false, i.e., H_i and L_i are mutually exclusive. Whenever both H_i and L_i are false, this means the value of the input n_i is not specified, i.e., it is X, thus indeed leading to *partial* input combinations. Our algorithm ensures that each of these partial input combinations is minimal in that removing any of the specified inputs would lead to an indeterminate output of the specification.

Example: 3-input AND-gate
Consider a circuit consisting of a single 3-input AND-gate. An SIR relation for this circuit needs to enumerate all partial input combinations that lead to a determinate output value of its specification, as well as encode that value.

The output of a 3-input AND-gate is high if and only if all three inputs are high. Thus, it is sufficient for one of the three inputs to be low to conclude that the output is low. So we can describe the full behaviour of this gate using just four partial input combinations, as listed Figure 3.2. Notice that all eight possible concrete input combinations are covered by these four partial input combinations.

Figure 3.3 shows an SIR relation that encodes these partial input combinations. It introduces two variables x_1, x_2 and uses them together with o to enumerate the cases

n_1	n_2	n_3	out
		0	0
	0		0
0			0
1	1	1	1

Figure 3.2: The four minimal input combinations that lead to a concrete output for a 3-input AND-gate.

shown in Figure 3.2. The relation is structured so that the variable o determines whether the output is high or low in each enumerated case. In particular, the example relation satisfies the coverage condition

$$\forall \mathcal{V}. \forall \overleftarrow{o}. \overleftarrow{o} = \left\{ o \mapsto \textsc{spec}(\overleftarrow{\mathcal{V}}) \right\} \Leftrightarrow \exists \overleftarrow{\mathcal{X}}. R(\overleftarrow{o}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}).$$

Note that no relation that uses only two variables can achieve this. It is easy to see this, as the case where $\overleftarrow{o} = \{o \mapsto 0\}$ includes three different partial input combinations, and their enumeration requires at least two additional variables.

1. SIR relation

$$\begin{aligned} R = \quad & (o \rightarrow v_1) \wedge (\overline{o} \wedge x_1 \wedge x_2 \rightarrow \overline{v_1}) \wedge \\ & (o \rightarrow v_2) \wedge (\overline{o} \wedge x_1 \wedge \overline{x_2} \rightarrow \overline{v_2}) \wedge \\ & (o \rightarrow v_3) \wedge (\overline{o} \wedge \overline{x_1} \rightarrow \overline{v_3}) \end{aligned}$$

2. Partial input combinations

o	x_1	x_2	R
0	0		$\overline{v_3}$
0	1	0	$\overline{v_2}$
0	1	1	$\overline{v_1}$
1			$v_0 \wedge v_1 \wedge v_2$

Figure 3.3: (1) An SIR relation for the specification ((VAR v_1) AND (VAR v_2)) AND (VAR v_3), and (2) which partial input combinations it encodes

★

3.2.3 Computing an SIR Relation

Figure 3.4 shows a simple automatic abstraction discovery algorithm that generates relations that satisfy the SIR condition by construction. It decomposes the specification expression step by step, starting at the output and working backwards until reaching all initial inputs. In each step it propagates backwards the information it has collected thus far.

```
1   auto_abstract(SPEC, H, L) =
2     if is_VAR(SPEC) then
3       v := get_var(SPEC)
4       return ((H → v) ∧ (L → v̄))
5     elseif is_NOT(SPEC)
6       return auto_abstract(strip_NOT(SPEC),L,H)
7     else // is_AND
8       x := get_fresh_indexing_variable()
9       (SPEC1, SPEC2) := destruct_AND(SPEC)
10      rel1 := auto_abstract(SPEC1,H,L ∧ x)
11      rel2 := auto_abstract(SPEC2,H,L ∧ x̄)
12      return rel1 ∧ rel2
```

Figure 3.4: Simple back-propagation algorithm.

General Description of the Algorithm

The goal of the algorithm provided in Figure 3.4 is to construct an SIR relation with the correct properties. The algorithm takes three parameters: a bexpr-tree, and two Boolean expressions. The pair (H, L) encodes when we want the specification currently being processed by the algorithm to be true or false respectively. In the initial call the bexpr-tree passed in is the circuit's specification, and the two Boolean expressions are $H = o$ and $L = \overline{o}$. The recursive calls will work on simpler bexpr-trees, received by deconstructing the initial specification, and increasingly elaborate expressions, which constitute partial symbolic indexings.

First, if the specification is a simple variable declaration, VAR v, then it is easy to construct an SIR relation, as seen in line 4: $(H \rightarrow v) \wedge (L \rightarrow \overline{v})$.

In all other cases, we process the specification recursively. For this, we first determine the outermost constructor. When representing the specification as a tree, this corresponds to the root of the tree.

If the outermost constructor is a negation, NOT, then we return the result of running the algorithm with the negated specification while swapping the two expressions H and L, as seen in line 6. This is done because the output of a NOT-gate is high if and only if its input is low. We receive the negated specification by stripping the outermost NOT constructor, which ensures that the new run of the algorithm processes a specification with fewer constructors.

If the specification is a conjunction, i.e., the outermost constructor is an AND, then we run the algorithm on each of its operands with the two pairs of expressions

$(H, L \wedge x_i)$ and $(H, L \wedge \overline{x_i})$ respectively, as seen in lines 9–11. Here x_i is a thus far unused variable (line 8) that encodes which of the operands should be low to ensure the AND-gate's output is low. When one operand is low, the other one may have any value, i.e., be X. This is encoded by conjoining L with x_i for the first operand, and the inverse, $\overline{x_i}$, for the second operand. As in an assignment $\overleftarrow{\mathcal{X}}$ either x_i or $\overline{x_i}$ evaluates to false, at least one of the conditions, $L \wedge x_i$ and $L \wedge \overline{x_i}$ is false. This means we do not require that input to the AND-gate to be low. If we want the gate to have a high output, we must ensure both inputs are high, too. Therefore, we pass through the first expression unchanged. The algorithm returns the conjunction of the recursive call on the two operands with the modified high-low-conditions (line 12) to merge the intermediary results. Again, the recursive calls work on a specification with fewer constructors, as it is stripped of its outermost AND.

Hence, every recursive call of the algorithm satisfies the property of running on a specification with fewer constructors, which ensures that the algorithm terminates.

If specification and implementation are identical, this algorithm leads to an abstraction for 3-valued simulation which hides as much information as possible. This means that for each assignment $\overleftarrow{\mathcal{V}}$ to the indexing variables of the returned relation a minimal antecedent is delivered. It is minimal in that removing any of the specified values would result in an indeterminate output. We achieve this property by evaluating each specification constructor greedily. We modify the (H, L)-pairs in such a way that the loosest requirements is encoded. If the implementation needs no more input information than the specification, the abstraction never hides too much information. However, if the implementation requires additional data, over-abstraction is possible. Especially when arithmetic operations are involved, this can occur. A discussion of this follows in Section 4.3.

First, however, we will present our algorithm in action on a simple example while elaborating further on the algorithm's inner workings.

Example: Automatic Abstraction in Action

Figure 3.5 shows a graphical description of a simple specification $\textsc{spec}[\mathcal{V}]$. It uses four variables $\mathcal{V} = \{v_0, v_1, v_2, v_3\}$, one for each circuit input. The output of the algorithm is an SIR relation, which encodes an indexing scheme where each of these inputs are driven only sometimes, rather than always.

We want to determine which minimal input combinations lead to a true output, and which ones lead to a false output. So for each input we compute a pair (H_i, L_i)

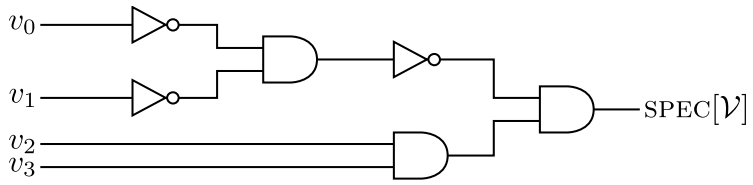


Figure 3.5: Graphical description of the specification function $\text{SPEC}[\mathcal{V}] = \Big(NOT\ \big((NOT\ v_0)\ AND\ (NOT\ v_1)\big)\Big)\ AND\ (v_2\ AND\ v_3)$

that tells us when we want the input to be true, and when we want it to be false respectively. If we do not require either of these, then the node needs not be driven, which corresponds to an indeterminate value X. We further ensure that the algorithm computes pairs where H and L are mutually exclusive, so the case $(\mathsf{tt}, \mathsf{tt})$ never occurs.

This pair encodes which value a node will be driven with, similar to the dual rail encoding introduced in Section 2.7. For every valuation $\overleftarrow{\mathcal{X} \cup o}$ of the free variables of H and L the evaluated pair $\big(H(\overleftarrow{o}, \overleftarrow{\mathcal{X}}), L(\overleftarrow{o}, \overleftarrow{\mathcal{X}})\big)$ determines the value of its associated node. If $\big(H(\overleftarrow{o}, \overleftarrow{\mathcal{X}}), L(\overleftarrow{o}, \overleftarrow{\mathcal{X}})\big) = (\mathsf{tt}, \mathsf{ff})$, then the corresponding subrelation is

$$R_i(\overleftarrow{o}, \overleftarrow{\mathcal{X}})[\mathcal{V}] = \big(H(\overleftarrow{o}, \overleftarrow{\mathcal{X}}) \to v_i\big) \wedge \big(L(\overleftarrow{o}, \overleftarrow{\mathcal{X}}) \to \overline{v_i}\big) = (\mathsf{tt} \to v_i) \wedge (\mathsf{ff} \to \overline{v_i}) = v_i$$

and thus n_i is driven with a high value; if $\big(H(\overleftarrow{o}, \overleftarrow{\mathcal{X}}), L(\overleftarrow{o}, \overleftarrow{\mathcal{X}})\big) = (\mathsf{ff}, \mathsf{tt})$ then the corresponding subrelation is $R_i = (\mathsf{ff} \to v_i) \wedge (\mathsf{tt} \to \overline{v_i}) = \overline{v_i}$ and n_i is driven with a low value. In the dual rail encoding $(\mathsf{tt}, \mathsf{ff})$ also represents a high value and $(\mathsf{ff}, \mathsf{tt})$ represents a low value. The two representations differ for indeterminate values. While the dual rail encoding for $(\mathsf{tt}, \mathsf{tt})$ represents the X value, in our case it is $(\mathsf{ff}, \mathsf{ff})$. The subrelation $R_i = (\mathsf{ff} \to v_i) \wedge (\mathsf{ff} \to \overline{v_i}) = \mathsf{tt}$ assigns no value to n_i, and it it thus implicitly X. Intuitively speaking, the dual rail encoding works on the premise of which values a node *may* take on. So if it may take on both values, this corresponds to an X value. Our pairs, on the other hand, indicate which value a node *must* have. So if neither a high or low value is required, the node is driven with X. The pair $(\mathsf{tt}, \mathsf{tt})$, on the other hand, represents the subrelation $(\mathsf{tt} \to v_i) \wedge (\mathsf{tt} \to \overline{v_i}) = v_i \wedge \overline{v_i} = \mathsf{ff}$, which encodes a non-indexing case, i.e., it is ignored. In the dual rail encoding $(\mathsf{ff}, \mathsf{ff})$ represents such overconstraint cases.

We start at the output of the specification and work through the specification backwards until we reach all initial inputs. We want to compute the input combinations for both the case when the output of the specification is true, and when it is false. We keep these cases separate by using mutually exclusive formulae in the pair (H, L). The

first pair, which is passed into the algorithm should thus also be mutually exclusive. The simplest such pair of formulae is a variable and its inverse, and in this dissertation we always use $(o, \overline{o})$. More generally, we could use other mutually exclusive formulae (H_{out}, L_{out}), as long as we also adjust the consequent of the auxiliary STE statement $\models \bigwedge_i n_i$ is $v_i \Rightarrow out$ is o to $\models \bigwedge_i n_i$ is $v_i \Rightarrow (H_{out} \rightarrowtriangle out$ is $1)$ and $(L_{out} \rightarrowtriangle out$ is $0)$.

Now we need to study the structure of the specification to determine which requirements to set on the inputs to encode all partial input combinations lead to a high when o holds, or a low output when $\overline{o}$ holds. We do so by always processing the outermost constructor of the specification, and propagating what this means for its subexpressions. In a graphical description this process corresponds to handling gates from right to left. In our example the outermost constructor is an AND. We want its value to be true when the H-condition, o, is satisfied. The output of an AND-gate is true exactly when both its inputs are true. As seen in Figure 3.6, the H-condition for both inputs is thus the same as the H-condition of the output. Similarly, we need to answer the question what conditions the inputs of the AND-gate need to satisfy to guarantee a false output. Either the first input of the gate has a low value, and it is irrelevant what value the second input has, or versed. We encode this choice by introducing a new variable x_1. Whenever x_1 holds we require the first input to be false, and whenever $\overline{x_1}$ holds we require the second input to be false. So the new L-conditions are a conjunction of $\overline{o}$, which expresses when we want to achieve a low output, and x_1 – or $\overline{x_1}$ – which expresses which input is responsible for that outcome. So while processing each gate, computing the (H, L)-pairs of its inputs depends on the (H, L)-pair of its output, as well as the type of the gate. The subexpressions that fan into the AND-gate are handled recursively.

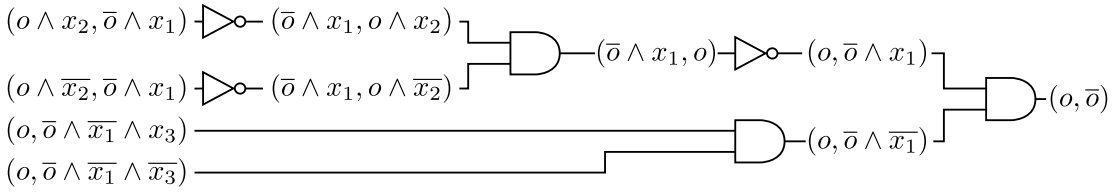


Figure 3.6: High and low expressions computed by the basic automatic abstraction algorithm provided in Figure 3.4

The first input to the AND-gate now has the pair $(o, \overline{o} \wedge x_1)$. The outermost constructor of the fanin is a negation. The output of a NOT-gate is true exactly when its input is false, and versed. Whenever we want the output to be determinate, we also need its only input to be determinate. So the (H, L)-pair for the NOT-gate

input simply swaps the values of the pair for the output, in this particular case it leads to the pair $(\overline{o} \wedge x_1, o)$.

All the other AND- and NOT-gates of the example specification are handled the same way. For each AND-gate, one new variable is introduced to encode each choice. Figure 3.6 provides the resulting (H, L) pairs. In particular, this leads to four such pairs, one for each input of the specification, and thus the circuit.

Throughout the explanation we always said that the first value of the pair encodes when the node shall be driven high, and the second value of the pair encodes when it shall be driven low. We capture this meaning by producing the subrelation $(H_i \rightarrow v_i) \wedge (L_i \rightarrow \overline{v_i})$ for each initial input. The complete relation is then the conjunction of each of these subrelations:

$$R = \bigwedge_i (H_i \rightarrow v_i) \wedge (L_i \rightarrow \overline{v_i})$$

Figure 3.7 provides the relation in full.

$$\begin{array}{rrcrcl} R = & (o \wedge x_2 \rightarrow v_0) & \wedge & (\overline{o} \wedge x_1 \rightarrow \overline{v_0}) & \wedge \\ & (o \wedge \overline{x_2} \rightarrow v_1) & \wedge & (\overline{o} \wedge x_1 \rightarrow \overline{v_1}) & \wedge \\ & (o \rightarrow v_2) & \wedge & (\overline{o} \wedge \overline{x_1} \wedge x_3 \rightarrow \overline{v_2}) & \wedge \\ & (o \rightarrow v_2) & \wedge & (\overline{o} \wedge \overline{x_1} \wedge \overline{x_3} \rightarrow \overline{v_2}) & \end{array}$$

Figure 3.7: Relation computed by the automatic abstraction algorithm for the specification given in Figure 3.5

Remember that our goal was to find a relation that tells us how to change the STE run, where each initial input is driven by a variable v_i. Only if the full relation is satisfied, i.e it evaluates to true under an assignment to all its free variables, does it lead to a case that is simulated by STE. The relation we computed expresses that whenever the condition H_i is satisfied, the corresponding initial input n_i must be driven with a high value – represented by v_i. And whenever the condition L_i is satisfied, the initial input must be driven with a low value, represented by $\overline{v_i}$. If neither condition is satisfied, then v_i can have either value without falsifying the relation, which encodes that the initial input is not driven, i.e., has the indeterminate value X. So, indeed, we have encoded a set of partial input combinations. By construction each case gives sufficiently many inputs a value to determine the value of the specification. Indeed, each of the partial input combinations is minimal, as desired.

3.3 Correctness Statements

In the previous section we proposed an algorithm that computes an SIR relation, which we want to utilise for formally verifying circuits by

$$R[o, \mathcal{X}, \mathcal{V}] \models (\bigwedge_i n_i \texttt{ is } v_i)^R \Rightarrow (out \texttt{ is } o)_R.$$

It is only meaningful if that STE run ensures formal correctness of the circuit. Hence, this section first makes some statements on the completeness of our approach, and provides two theorems stating the soundness of our approach. First, that our STE run formally verifies a circuit provided the SIR conditions

$$o \notin \mathcal{X} \qquad \text{and} \qquad \forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{o}.\Big(\overleftarrow{o} = \big\{o \mapsto \textsc{spec}(\overleftarrow{\mathcal{V}})\big\} \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{o}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})\Big)$$

hold; and, second, that the SIR relation computed by our automatic abstraction algorithm indeed satisfies these side conditions. This section shows that our work is sound: it ensures the expressiveness of the adjusted verification task, and that the relations created by our auto_abstract algorithm are sufficient by construction. In contrast to the previous work by Melham and Jones, this means no further coverage proofs need to be completed when using our automatically generated relations, which removes a potentially very expensive step.

3.3.1 Completeness

As already stated previously, using the abstractions automatically computed with our approach can lead to over-abstraction. This makes clear than our solution is not complete. However, by the nature of our algorithm, namely determining which inputs are sufficient for determining the value of the specification, we know at least one class of circuits where no over-abstraction can occur. That is, if the specification and the implementation match exactly, then our approach produces an abstraction that does not lead to over-abstraction, but produces a concrete verification result, pass or fail. In other words, the equivalence checking we perform never results in a weak disagreement if both inputs – the specification and the circuit to verify – are identical.

3.3.2 Soundness

Next we prove that any concrete verification result is meaningful. Whenever the verification has a concrete result, pass or fail, we can safely conclude that the circuit meets the specification we used to automatically compute an abstraction.

Specification Injection

In Section 3.1 we reviewed Melham and Jones' work, which shows that $R[\mathcal{X},\mathcal{V}] \models (\bigwedge_i n_i \texttt{ is } v_i)^R \Rightarrow \Big(out \texttt{ is } \textsc{spec}[\mathcal{V}]\Big)_R$ is as powerful a verification run as $\models (\bigwedge_i n_i \texttt{ is } v_i) \Rightarrow (out \texttt{ is } \textsc{spec}[\mathcal{V}])$ provided the side condition $\forall \overleftarrow{\mathcal{V}}.\exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{X}},\overleftarrow{\mathcal{V}})$ holds. In this chapter, we describe a different approach, claiming that for the SIR relations we automatically compute, the verification run

$$R[o,\mathcal{X},\mathcal{V}] \models (\bigwedge_i n_i \texttt{ is } v_i)^R \Rightarrow (out \texttt{ is } o)_R$$

is sufficient for formally verifying a circuit. The main difference is that an SIR relation $R[o,\mathcal{X},\mathcal{V}]$ encodes the specification of the circuit in addition to a reindexing. In contrast, before the specification was still part of the STE run, which was reindexed with a relation.

Theorem 3.1 formalises and proves that this new method does indeed work.

Theorem 3.1 (Sufficiency)**.** *Let c be a circuit with input nodes $\{n_i : i \in I\}$ and output node out. Let* SPEC *be a bexpr-tree, where $\mathcal{V} = \{v_i : i \in I\}$ is its set of free variables. Then the formal verification of c*

$$\models \bigwedge_i n_i \texttt{ is } v_i \Rightarrow out \texttt{ is } \textsc{spec}[\mathcal{V}] \tag{3.3.1}$$

can be computed by

$$R[o,\mathcal{X},\mathcal{V}] \models (\bigwedge_i n_i \texttt{ is } v_i)^{R[o,\mathcal{X},\mathcal{V}]} \Rightarrow (out \texttt{ is } o)_{R[o,\mathcal{X},\mathcal{V}]} \tag{3.3.2}$$

provided that

$$o \notin \mathcal{X} \qquad \textit{and} \qquad \forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{o}.\ \overleftarrow{o} = \{o \mapsto \textsc{spec}(\overleftarrow{\mathcal{V}})\} \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{o},\overleftarrow{\mathcal{X}},\overleftarrow{\mathcal{V}}). \tag{3.3.3}$$

Proof. Suppose the STE assertion 3.3.2 holds:

$$R[o, \mathcal{X}, \mathcal{V}] \models (\bigwedge_i n_i \texttt{ is } v_i)^{R[o,\mathcal{X},\mathcal{V}]} \Rightarrow (out \texttt{ is } o)_{R[o,\mathcal{X},\mathcal{V}]}$$

By Melham and Jones' Theorems 1 and 2 [2], this implies

$$R[o, \mathcal{X}, \mathcal{V}] \models \bigwedge_i n_i \texttt{ is } v_i \Rightarrow out \texttt{ is } o.$$

As $o \notin \mathcal{X}$ by assumption 3.3.3, both the antecedent ($\bigwedge_i n_i$ `is` v_i) and the consequent (out `is` o) are independent of the variables in $\mathcal{X}$. Hence we can existentially quantify over $\mathcal{X}$ in $R[o, \mathcal{X}, \mathcal{V}]$ of the assumption:

$$\exists \overleftarrow{\mathcal{X}}.R[o, \mathcal{V}](\overleftarrow{\mathcal{X}}) \models \bigwedge_i n_i \texttt{ is } v_i \Rightarrow out \texttt{ is } o$$

Using assumption 3.3.3 we know that for all $\overleftarrow{o}$ and for all $\overleftarrow{\mathcal{V}}$,

$$\overleftarrow{o} = \{o \mapsto \text{SPEC}(\overleftarrow{\mathcal{V}})\} \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{o}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}),$$

and therefore we can conclude that

$$o \equiv \text{SPEC}[\mathcal{V}] \models \bigwedge_i n_i \texttt{ is } v_i \Rightarrow out \texttt{ is } o$$

and therefore, by eliminating o, we have

$$\models \bigwedge_i n_i \texttt{ is } v_i \Rightarrow out \texttt{ is } \text{SPEC}[\mathcal{V}]$$

as desired. □

SIR Conditions

This section proves that the auto_abstract algorithm we suggest generates relations that satisfy the SIR conditions

$$o \notin \mathcal{X} \qquad \text{and} \qquad \forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{o}.\ \overleftarrow{o} = \{o \mapsto \text{SPEC}(\overleftarrow{\mathcal{V}})\} \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{o}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$$

by construction. This extremely powerful property of the automatically generated relations is captured in Theorem 3.5. In combination with Theorem 3.1, this shows

that our automatic abstraction discovery framework is sound.

The proof of Theorem 3.5 is quite elaborate, so we first prove some lemmata, which state important auxiliary results. First, Lemma 3.2 shows that our algorithm terminates for all specifications, which thus allows us to argue freely about the relations returned by the algorithm.

Lemma 3.2 (Termination)**.** *For every bexpr-tree* SPEC *and any two expressions H and L, the algorithm* auto_abstract(SPEC, H, L) *terminates.*

Proof. Every bexpr-tree has only finitely many constructors. Furthermore, whenever auto_abstract is called recursively, the bexpr it is passed was reduced by one constructor. So after finitely many recursive calls auto_abstract is passed a bexpr that consists of only one constructor. For all well-formed bexpr-tree, the innermost constructors are VAR. On these the algorithm terminates, as seen in line 4 of Figure 3.8. □

Next follow two simple lemmata used in the proof of Theorem 3.5, and also later in the proof of Theorem 4.1, which shows correctness of an advanced version of the auto_abstract algorithm. These make statements about the relative strength of relations, and thus can be used to establish that if one relation is satisfied, so are all weaker ones. These weaker relations play a role in specific cases encountered in the proof of the main theorems.

Lemma 3.3 (Relation weakening)**.** *Let* SPEC *be a bexpr-tree and $\mathcal{V}$ be its set of free variables. Let $R_{\text{SPEC},H,L}$ be the relation that is returned by* auto_abstract(SPEC, H, L)*, where H and L are any two expressions. For all expressions O, P, Q such that they have no free variables in $\mathcal{V}$ and such that $O \to P$,*

$$R_{\text{SPEC},Q,P} \to R_{\text{SPEC},Q,O} \quad \textit{and} \quad R_{\text{SPEC},P,Q} \to R_{\text{SPEC},O,Q},$$

i.e., the relation is monotonic in both parameters H and L.

Proof. Let SPEC and O, P, Q have the required properties. As seen in line 4 of the back propagation algorithm presented in Figure 3.8, in the relation returned by auto_abstract(SPEC, H, L) variables in $\mathcal{V}$ occur on the right-hand side of the implications only:

$$R_{\text{SPEC},H,L} = \bigwedge_i (H_i \to v_i) \wedge (L_i \to \overline{v_i}).$$

More precisely, $H_i = E_H \wedge F_H$ and $L_i = E_L \wedge F_L$ where $E_H, E_L \in \{H, L\}$ are one of the initial expressions passed to the algorithm. This can be seen in lines 10–11. Note

that in line 6 the high and low arguments are swapped, so both H and L are options for E_H and E_L.

Now suppose H is Q and L is P. As Q and P have no free variables in $\mathcal{V}$, this means that Q and P occur only on the left-hand side of the implications. Similarly, this holds if L is O. Using $O \to P$ we can conclude that $(P \wedge F) \to G$ implies $(O \wedge F) \to G$ for all expressions F, G. Thus, $R_{\text{SPEC},Q,P} \to R_{\text{SPEC},Q,O}$ as desired.

Similarly, $R_{\text{SPEC},P,Q} \to R_{\text{SPEC},O,Q}$. □

Lemma 3.4 (Tautological relation)**.** *For all bexpr-trees* SPEC *the relation returned by* ***auto_abstract***(SPEC, ***ff***, ***ff***) *is a tautology.*

Proof. As seen in the proof of Lemma 3.3, there exist expressions F_H, F_L such that $R_{\text{SPEC,ff,ff}} = \bigwedge_i ((\mathsf{ff} \wedge F_H) \to v_i) \wedge ((\mathsf{ff} \wedge F_L) \to \overline{v_i}) = \mathsf{tt}$. □

Finally, we prove correctness of the algorithm presented in Figure 3.8. For presentational purposes we assume that the input specification and the output relation have the same type, i.e., the relation is also a bexpr-tree. This does not change the proof, but eases notation when comparing bexpr-trees with expressions. For better readability the notation for 'F AND G' is '$\text{F} \wedge \text{G}$', the notation for 'NOT F' is '$\overline{\text{F}}$', and the notation for 'VAR v' is simply 'v'. Finally, $R_{\text{SPEC},H,L}$ denotes the relation that is returned by auto_abstract(SPEC, H, L).

```
 1  auto_abstract(SPEC, H, L) =
 2    if is_VAR(SPEC) then
 3      v := get_var(SPEC)
 4      return ((H → v) ∧ (L → v̄))
 5    elseif is_NOT(SPEC)
 6      return auto_abstract(strip_NOT(SPEC),L,H)
 7    else // is_AND
 8      x := get_fresh_indexing_variable()
 9      (SPEC1, SPEC2) := destruct_AND(SPEC)
10      rel1 := auto_abstract(SPEC1,H,L ∧ x)
11      rel2 := auto_abstract(SPEC2,H,L ∧ x̄)
12      return rel1 ∧ rel2
```

Figure 3.8: Algorithm as first seen in Figure 3.4 (page 67).

Theorem 3.5 (Correctness of the Basic Algorithm)**.** *Let* SPEC *be an expression represented as a bexpr-tree with free variables* $\mathcal{V}$ *and let* o *be a variable such that* $o \notin \mathcal{V}$*.*

Furthermore, let $R_{\text{SPEC},o,\overline{o}}[o, \mathcal{X}, \mathcal{V}]$ *be the result of* **auto_abstract**$(\text{SPEC}[\mathcal{V}], o, \overline{o})$ *as presented in Figure 3.4. Then*

$$o \notin \mathcal{X} \tag{3.3.4}$$

and

$$\forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{o}.\Big(\overleftarrow{o} = \{o \mapsto \text{SPEC}(\overleftarrow{\mathcal{V}})\} \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R_{\text{SPEC},o,\overline{o}}(\overleftarrow{o}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})\Big) \tag{3.3.5}$$

Proof. Claim 3.3.4 follows, because the only line of the algorithm in which indexing variables are introduced is line 8. But in line 8 a fresh, thus far unused variable is inserted. So o is never reused, and is therefore independent of both $\mathcal{X}$ and $\mathcal{V}$.

For claim 3.3.5 an induction on k, the depth of the bexpr-tree $\text{SPEC}[\mathcal{V}]$, follows. We show that for all expressions E such that $\mathcal{E} \cap (\mathcal{X} \dot{\cup} \mathcal{V}) = \emptyset$, where $\mathcal{E} = \textsf{free\_vars}(E)$:

$$\forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{\mathcal{E}}.\Big(E(\overleftarrow{\mathcal{E}}) = \text{SPEC}(\overleftarrow{\mathcal{V}}) \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R_{\text{SPEC},E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})\Big).$$

By choosing $E = o$ the claim then follows.

Induction start: $k = 1$

The only bexpr-tree of depth 1 is $\text{SPEC}[\mathcal{V}] = v$. As seen in line 4, $R_{v,E,\overline{E}}[\mathcal{E}, \mathcal{X}, \mathcal{V}] = (E \to v) \wedge (\overline{E} \to \overline{v})$. In particular, $\mathcal{X} = \emptyset$ and the relation is true if and only if $E \equiv v$ as required.

Induction hypothesis: For all bexpr-trees $\text{SPEC}[\mathcal{V}]$ of maximum depth k and for all expressions E such that $\mathcal{E} \cap (\mathcal{X} \dot{\cup} \mathcal{V}) = \emptyset$, where $\mathcal{E} = \textsf{free\_vars}(E)$, assume the following holds:

$$\forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{\mathcal{E}}.E(\overleftarrow{\mathcal{E}}) = \text{SPEC}(\overleftarrow{\mathcal{V}}) \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R_{\text{SPEC},E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$$

Induction step: We show that under the induction hypothesis the claim also holds for all bexpr-trees of depth $k + 1$.

Let E be an arbitrary expression of depth $k+1$ such that $\mathcal{E} \cap (\mathcal{X} \dot{\cup} \mathcal{V}) = \emptyset$, and let $\overleftarrow{\mathcal{V}}$ and $\overleftarrow{\mathcal{E}}$ be arbitrary assignments to the variables in $\mathcal{V}$ and $\mathcal{E}$ respectively.

(a) Case $\text{SPEC} = \overline{\text{G}}$.

As seen in line 6,

$$R_{\overline{\text{G}},E,\overline{E}}[\mathcal{E}, \mathcal{X}, \mathcal{V}] \equiv R_{\text{G},\overline{E},E}[\mathcal{E}, \mathcal{X}, \mathcal{V}] \tag{3.3.6}$$

(a i) Need to show: $E(\overleftarrow{\mathcal{E}}) = \overline{\mathrm{G}}(\overleftarrow{\mathcal{V}}) \Rightarrow \exists \overleftarrow{\mathcal{X}}.\ R_{\overline{\mathrm{G}},E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$.

By induction hypothesis, $\overline{E}(\overleftarrow{\mathcal{E}}) = \mathrm{G}(\overleftarrow{\mathcal{V}}) \Rightarrow \exists \overleftarrow{\mathcal{X}}.R_{\mathrm{G},\overline{E},\overline{\overline{E}}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$. But $\overline{E}(\overleftarrow{\mathcal{E}}) = \mathrm{G}(\overleftarrow{\mathcal{V}})$ is equivalent to $E(\overleftarrow{\mathcal{E}}) = \overline{\mathrm{G}}(\overleftarrow{\mathcal{V}})$, and $\overline{\overline{E}}(\overleftarrow{\mathcal{E}}) = E(\overleftarrow{\mathcal{E}})$. Thus, $E(\overleftarrow{\mathcal{E}}) = \overline{\mathrm{G}}(\overleftarrow{\mathcal{V}}) \Rightarrow \exists \overleftarrow{\mathcal{X}}.R_{\mathrm{G},\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$, and by applying observation 3.3.6 the claim follows.

(a ii) Need to show: $\exists \overleftarrow{\mathcal{X}}.R_{\overline{\mathrm{G}},E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) \Rightarrow E(\overleftarrow{\mathcal{E}}) = \overline{\mathrm{G}}(\overleftarrow{\mathcal{V}})$.

By induction hypothesis, $\exists \overleftarrow{\mathcal{X}}.R_{\mathrm{G},\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$ implies that $\overline{E}(\overleftarrow{\mathcal{E}}) = \mathrm{G}(\overleftarrow{\mathcal{V}})$, and thus $E(\overleftarrow{\mathcal{E}}) = \overline{\mathrm{G}}(\overleftarrow{\mathcal{V}})$ as required.

(b) Case $\textsc{spec} = \mathrm{G}_1 \wedge \mathrm{G}_2$.

As seen in line 9–12,

$$R_{\mathrm{G}_1 \wedge \mathrm{G}_2,E,\overline{E}}[\mathcal{E}, \mathcal{X}, \mathcal{V}] = R_{\mathrm{G}_1,E,\overline{E} \wedge y}[\mathcal{E}, \mathcal{X}_1 \dot\cup \{y\}, \mathcal{V}] \wedge R_{\mathrm{G}_2,E,\overline{E} \wedge \overline{y}}[\mathcal{E}, \mathcal{X}_2 \dot\cup \{y\}, \mathcal{V}] \tag{3.3.7}$$

(b i) Need to show: $E(\overleftarrow{\mathcal{E}}) = \mathrm{G}_1(\overleftarrow{\mathcal{V}}) \wedge \mathrm{G}_2(\overleftarrow{\mathcal{V}}) \Rightarrow \exists \overleftarrow{\mathcal{X}}.R_{\mathrm{G}_1 \wedge \mathrm{G}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$.

By induction hypothesis, $E(\overleftarrow{\mathcal{E}}) = \mathrm{G}_1(\overleftarrow{\mathcal{V}}) \Rightarrow \exists \overleftarrow{\mathcal{X}_1}.R_{\mathrm{G}_1,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$ and $E(\overleftarrow{\mathcal{E}}) = \mathrm{G}_2(\overleftarrow{\mathcal{V}}) \Rightarrow \exists \overleftarrow{\mathcal{X}_2}.R_{\mathrm{G}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$. The indexing variables introduced in line 8 are never reused, so $\mathcal{X}_1$ and $\mathcal{X}_2$ are disjoint, and $y \notin \mathcal{X}_1 \dot\cup \mathcal{X}_2$. So we can choose a variable assignment for y, $\mathcal{X}_1$ and $\mathcal{X}_2$ independently. We show that $\exists \overleftarrow{y}.\exists \overleftarrow{\mathcal{X}_1}.\exists \overleftarrow{\mathcal{X}_2}.\ R_{\textsc{spec},E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1 \dot\cup \mathcal{X}_2 \dot\cup \{y\}}, \overleftarrow{\mathcal{V}})$ by choosing $\overleftarrow{y} = \mathrm{G}_2(\overleftarrow{\mathcal{V}})$. The claim then follows by observation 3.3.7.

(b i α) Suppose $y = \mathsf{tt}$.

Then $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathrm{G}_1(\overleftarrow{\mathcal{V}}) \wedge \mathsf{tt} = \mathrm{G}_1(\overleftarrow{\mathcal{V}})$. Furthermore,

$$R_{\mathrm{G}_1 \wedge \mathrm{G}_2,E,\overline{E}} = R_{\mathrm{G}_1,E,\overline{E} \wedge \mathsf{tt}} \wedge R_{\mathrm{G}_2,E,\overline{E} \wedge \mathsf{ff}} = R_{\mathrm{G}_1,E,\overline{E}} \wedge R_{\mathrm{G}_2,E,\mathsf{ff}}.$$

The induction hypothesis for G_1 delivers that

$$E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}}) \Rightarrow \exists \overleftarrow{\mathcal{X}_1}.R_{\mathrm{G}_1,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}}).$$

It remains to show that

$$E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}}) \Rightarrow \exists \mathcal{X}_2.R_{\mathrm{G}_2,E,\mathsf{ff}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}}).$$

If $\mathrm{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt} = \mathrm{G}_2(\overleftarrow{\mathcal{V}})$, then the induction hypothesis for G_2 and Lemma

3.3 deliver this. If, on the other hand, $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \mathsf{ff} = E(\overleftarrow{\mathcal{E}})$, then $R_{\textsc{g}_2,E,\mathsf{ff}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{V}})[\mathcal{X}_2] = R_{\textsc{g}_2,\mathsf{ff},\mathsf{ff}}(\overleftarrow{\mathcal{V}})[\mathcal{X}_2]$ and by Lemma 3.4 $\overleftarrow{\mathcal{X}_2}$ can be chosen randomly.

(b i β) Suppose $y = \mathsf{ff}$.

Then $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \textsc{g}_1(\overleftarrow{\mathcal{V}}) \wedge \mathsf{ff} = \mathsf{ff} = \textsc{g}_2(\overleftarrow{\mathcal{V}})$. Furthermore,

$$R_{\textsc{g}_1 \wedge \textsc{g}_2,E,\overline{E}} = R_{\textsc{g}_1,E,\overline{E}\wedge\mathsf{ff}} \wedge R_{\textsc{g}_2,E,\overline{E}\wedge\mathsf{tt}} = R_{\textsc{g}_1,E,\mathsf{ff}} \wedge R_{\textsc{g}_2,E,\overline{E}}.$$

Then the induction hypothesis for $\textsc{g}_2$ delivers that

$$E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}}) \Rightarrow \exists\overleftarrow{\mathcal{X}_2}.R_{\textsc{g}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}}).$$

It remains to show that

$$E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{ff} \Rightarrow \exists\overleftarrow{\mathcal{X}_1}.R_{\textsc{g}_1,E,\mathsf{ff}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}}).$$

But if $E(\overleftarrow{\mathcal{E}}) = \mathsf{ff}$ Lemma 3.4 can be applied, and thus $\overleftarrow{\mathcal{X}_1}$ can be chosen randomly.

(b ii) Need to show: $\exists\overleftarrow{\mathcal{X}}.R_{\textsc{g}_1\wedge\textsc{g}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) \Rightarrow E(\overleftarrow{\mathcal{E}}) = \textsc{g}_1(\overleftarrow{\mathcal{V}}) \wedge \textsc{g}_2(\overleftarrow{\mathcal{V}})$.

By induction hypothesis, $\exists\overleftarrow{\mathcal{X}_1}.R_{\textsc{g}_1,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) \Rightarrow E(\overleftarrow{\mathcal{E}}) = \textsc{g}_1(\overleftarrow{\mathcal{V}})$ and $\exists\mathcal{X}_2.R_{\textsc{g}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) \Rightarrow E(\overleftarrow{\mathcal{E}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}})$. Now suppose $\exists\overleftarrow{\mathcal{X}} = \overleftarrow{\mathcal{X}_1 \dot\cup \mathcal{X}_2 \dot\cup \{y\}}.R_{\textsc{g}_1\wedge\textsc{g}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$.

If $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}})$, then – independent of the assignment for y – $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \textsc{g}_1(\overleftarrow{\mathcal{V}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}})$. Then the induction hypothesis delivers $E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}})$: for $y = \mathsf{tt}$ use the induction hypothesis for $\textsc{g}_1$, and for $y = \mathsf{ff}$ use the induction hypothesis for $\textsc{g}_2$.

Next assume that $\textsc{g}_1(\overleftarrow{\mathcal{V}}) \neq \textsc{g}_2(\overleftarrow{\mathcal{V}})$, without loss of generality $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$. Then we need to show that $E(\overleftarrow{\mathcal{E}}) = \textsc{g}_1(\overleftarrow{\mathcal{V}}) = \textsc{spec}(\overleftarrow{\mathcal{V}})$, so it is sufficient to prove that $y = \mathsf{tt}$: then the induction hypothesis for $R_{\textsc{g}_1,E,\overline{E}}$ delivers the result. So assume for contradiction that $y = \mathsf{ff}$. Then $R_{\textsc{g}_2,E,\overline{E}\wedge\overline{y}} = R_{\textsc{g}_2,E,\overline{E}}$ and the induction hypothesis delivers $E(\overleftarrow{\mathcal{E}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$. Thus, $R_{\textsc{g}_1,E,\overline{E}\wedge y} = R_{\textsc{g}_1,\mathsf{tt},\mathsf{ff}}$. By induction hypothesis this implies that $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, which is a contradiction to the assumption. So $y = \mathsf{tt}$ as required.

This completes the proof that the coverage condition holds if and only if o is equivalent to $\textsc{spec}[\mathcal{V}]$. □

3.4 Summary

This chapter introduced the first solution for fully automatically verifying circuits by STE. Given a circuit's specification, it computes an STE statement. The circuit meets its specification only if it satisfies this statement. Importantly, the statement includes non-trivial symbolic indexing, which aims at reducing verification costs considerably compared to a full symbolic simulation. Our approach achieves this symbolic indexing in two steps. First, our auto_abstract algorithm computes a relation, which enumerates the partial input combinations which are sufficient for determining the output value of the specification. Second, we apply the relation to an auxiliary STE statement using preimage computations. These operations both inject the specification, thus making the STE statement meaningful, as well as introduce an intricate abstraction scheme.

Here, we assume the specification is given as a Boolean formula, and our verification in essence does equivalence checking the circuit and the specification. Note that the abstraction computed may lead to over-abstraction. In Chapters 6 and 7 we propose one method of eliminating over-abstraction when it does occur.

Note also that our work uses the specification to automatically compute an abstraction. In that, the specification drives the resulting abstraction in a predictable, controllable way. Therefore, the user can change the computed abstraction by providing the specification in a different form. Used like this, our approach can also be seen as a tool to automatically compute sophisticated symbolic indexing schemes when the user provides the general direction by the shape of the specification.

Our automatic abstraction discovery algorithm is based on previous work by Melham and Jones [2], but significantly recasts it to allow full automation. We address two barriers of their approach. First, we give an algorithm that automatically generates relations that encode suitable symbolic indexings for verification by STE. Melham and Jones had shown that reindexing relations can be used to change STE statements without losing expressiveness, but they had not provided a way of generating these relations. Arguably, manually devising such relations is as hard as writing the symbolically indexed properties in the first place. Second, we proved that the relations generated by our auto_abstract algorithm by construction always satisfy the side conditions required for correctness. In our approach, the side conditions are stronger than those Melham and Jones identified. Melham and Jones note that checking that a reindexing relation satisfies the coverage condition can be expensive, and in the worst case outweigh the work saved in the modified STE run. So the correctness by

construction of our SIR relations as stated and proved in Theorem 3.5 makes our approach very powerful and promising.

Chapter 4

Abstraction Discovery Improvements

In the previous chapter we introduced an algorithm that automatically computes SIR relations given a specification. This chapter describes some key improvements to that algorithm. These enhancements both increase the flexibility of the algorithm and deliver relations that further decrease verification costs.

First, we address how to handle specifications that are directed acyclic graphs, rather than just tree structures. We then explain how to verify circuits with more than one output. Then we add support for specifying symbolic constants, which are inputs that shall always be driven with a symbolic value. Later, in Chapter 6, we suggest an automatic approach of selecting such symbolic constants. In this chapter we assume they are provided by the user.

We further improve the auto_abstract algorithm by recognising multiple-input AND-gates, OR-gates, and XNOR-gates. Our special handling results in better SIR relations that more concisely express minimal partial input combinations required to determine output values. We go on to prove that the algorithm with these adjustments still generates SIR relations that deliver correct verification results.

Next we discuss how indexing variables can be reused even more aggressively, which has the potential of reducing verification costs further. Finally, we optimise the weak and strong preimage calculations performed on the antecedent and consequent by leveraging the specific shape of the SIR relations we generate. These optimisations were largely developed by Magnus Björk.

Thus, this chapter takes the work introduced in Chapter 3, and transforms a proof-of-concept solution into a powerful method of verifying circuits. We put this solution to the test in Chapter 5, in which we verify different circuits using our method and

discuss the strength of the observed results.

4.1 Directed Acyclic Graphs

In the previous chapter we assumed that we are given a bexpr-tree. We now describe how the auto_abstract algorithm can be extended to handle directed acyclic graphs (DAGs). We first see how in principle the current algorithm can indeed already handle such structures, and then introduce a direct, more efficient variant of generating abstractions for DAGs.

This direct handling of DAGs is extremely valuable, as DAGs are much more compact than trees that express the same specification. More importantly, though, a DAG more directly encodes matching values, and thus allows our algorithm to pick up this information. This, in turn, leads to fewer indexing variables being introduced and better sharing of variables. Thus, extending our algorithm to analysing bexpr-DAGs constitutes an important improvement.

4.1.1 Handling and Correctness

Suppose we are given a specification that is an acyclic directed graph. This means identical subexpressions are shared, rather than duplicated. Then some nodes fan out to several gates. Functionally, this is equivalent to looking at the corresponding tree for this DAG by duplicating each shared structure. This can lead to an exponential increase in size, and we later explain how to avoid this. First we discuss how our algorithm can process such a tree, which has the same input-output behaviour as the DAG, and is in that respect suitable for determining an abstraction scheme using our original auto_abstract algorithm.

Duplicating subexpressions leads to inputs appearing multiple times in the tree, for an example see Figure 4.1(b). This raises the question whether our algorithm still works correctly when inputs occur more than once. Indeed, Theorem 3.5 makes no assumptions on how often inputs are encountered by the auto_abstract algorithm. So the coverage proof, and thus correctness, still hold. Roughly speaking, Proof 3.5 shows that for each possible input scenario there exists an assignment to the indexing variables that corresponds to that scenario. When an input occurs twice in the specification, we can see this as two separate inputs i and i'. Correctness follows by showing that for each input scenario where i and i' have the same value there exists a corresponding assignment to the indexing variables. The set of such scenarios is a

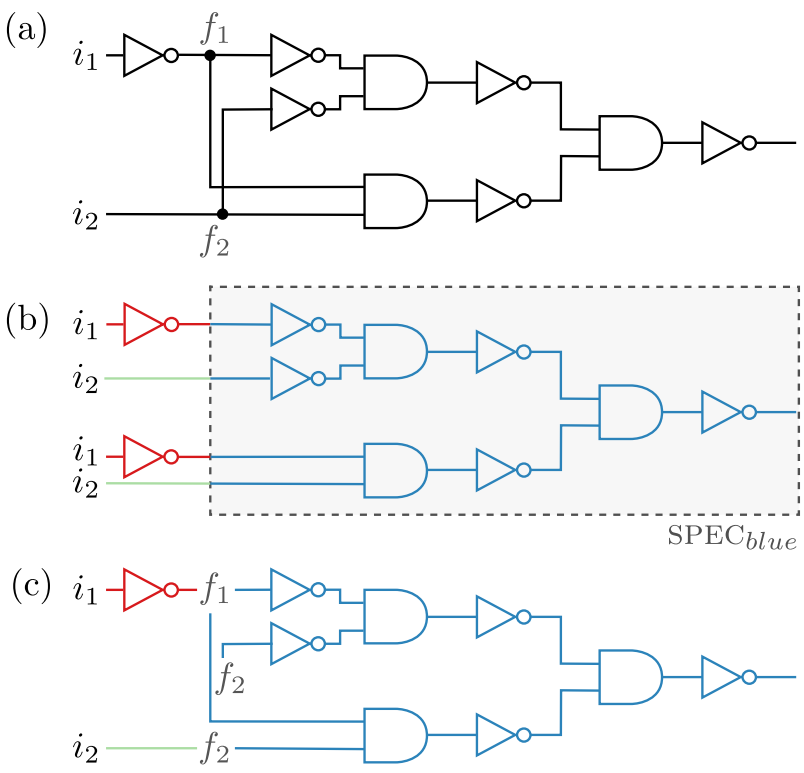


Figure 4.1: A (a) directed acyclic graph with two fanout points f_1 and f_2, (b) the corresponding tree after duplicating the fanins of the fanout points, and (c) the corresponding collection of trees, as retrieved after splitting on fanout points.

subset of all scenarios – we are ignoring those where i and i' have conflicting values – and thus the proof is more than sufficient.

Also note that the relation returned by the algorithm when running it on a specification where the input i occurs twice can be written as a conjunction $R \wedge (H_1 \rightarrow i) \wedge (L_1 \rightarrow \bar{i}) \wedge (H_2 \rightarrow i) \wedge (L_2 \rightarrow \bar{i})$. This is equivalent to $R \wedge ((H_1 \vee H_2) \rightarrow i) \wedge ((L_1 \vee L_2) \rightarrow \bar{i})$. Intuitively, this means that if the value of an input is required by one part of the circuit, then it is driven with a Boolean value – irrespective of whether the other part of the circuit requires it.

As noted above, some assignments to the indexing variables may lead to conflicts, namely when both $H_1 \vee H_2$ and $L_1 \vee L_2$ evaluate to true. In this case the relation evaluates to false, which STE handles by skipping these runs. Importantly, Theorem 3.5 guaranteed that at least some assignment to the indexing variables exists where such conflicts do not occur.

While handling DAGs in this fashion is correct, it is not very efficient. For one, we need to duplicate parts of the specification DAG to turn it into a tree, which can lead

to an exponential increase in size, and the algorithm then has to work on that enlarged structure. Additionally, our algorithm introduces new indexing variables for each of the duplicated structures. This not only leads to higher time and memory costs for the auto_abstract algorithm, but also delivers an abstraction scheme that could be indexed much more efficiently. Therefore we introduce a more suitable approach for DAGs. It is correct by the same explanation as before, in particular with respect to (H, L)-pairs that are not mutually exclusive.

4.1.2 Efficient Handling

We want to avoid duplicating subexpressions of a DAG to then work on its corresponding tree. Instead we want to process each shared subexpression once only, and then merge the intermediary results to get the indexing relation.

This already suggests an approach. We can split a directed acyclic graph into a set of trees by cutting it at every fanout point. The inputs and outputs of the subexpressions that are created by such a cut are then labelled with unique names, in the following called f_i. Then we run the auto_abstract algorithm on each of the subexpressions. Note that only one tree includes the ultimate output of the initial bexpr-DAG, and all other trees have outputs that previously were fanout points. For each of these trees we pass different (H, L)-pairs to the algorithm. We process the tree that includes the ultimate output with $H = o$ and $L = \overline{o}$, for all other trees we use distinct indexing variables, $H = h_f$ and $L = l_f$. We thus receive several relations, some of which specify when a fanout point shall be driven with a Boolean value, $(H_f \rightarrow f) \wedge (L_f \rightarrow \overline{f})$. This is so, because f is not just an output for one of the trees received by cutting the DAG in the fanout points, but it is also an input to at least two trees. So there are several relations which specify when to drive the same fanout point f, one for each branch of the fanout. We merge these by taking the disjunction of the different H- and L-conditions determined for f. In the following we detail the merge process in more detail.

For ease of notation, rather than using the full relations, we refer to the relations as a set of tuples (i, H, L). Each such tuple then corresponds to the subrelation $(H \rightarrow v_i) \wedge (L \rightarrow \overline{v_i})$, and the full relation is a conjunction of all those subrelations, $\bigwedge_i (H_i \rightarrow v_i) \wedge (L_i \rightarrow \overline{v_i})$.

Now let $R_{out=f}$ be the set of tuples computed by auto_abstract for a subexpression with fanout output f. Let $R_{in=f}$ be the set of tuples where the first entry of the tuples is f. $R_{in=f}$ includes subrelations from all auto_abstract runs that process a

subexpression that has an input labelled f. We can then merge the relations as follows.

Let $hi(f) = \bigvee_{(f,H,L)\in R_{in=f}} H$, and $lo(f) = \bigvee_{(f,H,L)\in R_{in=f}} L$. For each $(i, L_i, H_i) \in R_{out=f}$ replace h_f by $hi(f)$, and l_f by $lo(f)$. This means that the variables h_f and l_f are eliminated from the relation in the merging step. After merging all relations we thus receive a set of tuples where no such variables exist anymore, the tuples only determine when ultimate inputs of the initial bexpr-DAG shall be driven, and there is only one such tuple for each input.

Example: Splitting DAGs on fanout points
Consider the directed acyclic graph shown in Figure 4.1(a) (page 84). It has two fanout points f_1 and f_2. Figure 4.1(c) shows the subexpressions we get when splitting the DAG at its fanout points. The blue subexpression, SPEC_{blue}, includes the ultimate output, and we thus run auto_abstract($\text{SPEC}_{blue}, o, \overline{o}$) for it. The red subexpression, NOT VAR i_1, is processed by auto_abstract(NOT VAR i_1, h_{f_1}, l_{f_1}), and the green one, VAR i_2, by auto_abstract(VAR i_2, h_{f_2}, l_{f_2}). This leads to three calls of auto_abstract:

1. The blue subexpression, which ends in the ultimate output, is processed with the run auto_abstract($\text{SPEC}_{blue}, o, \overline{o}$). This delivers the tuples $(f_1, \overline{o} \wedge y, o \wedge x)$, $(f_1, o \wedge \overline{x}, \overline{o} \wedge z)$, $(f_2, \overline{o} \wedge \overline{y}, o \wedge x)$, $(f_2, o \wedge \overline{x}, \overline{o} \wedge \overline{z})$. See Figure 4.5 (page 95) for details.

2. The red subexpression, which ends in the fanout point f_1, is processed with the run auto_abstract(NOT VAR i_1, h_{f_1}, l_{f_1}). This delivers the tuple (i_1, l_{f_1}, h_{f_1}).

3. The green subexpression, which ends in the fanout point f_2, is processed with the run auto_abstract(VAR i_2, h_{f_2}, l_{f_2}). This delivers the tuple (i_2, h_{f_2}, l_{f_2}).

Now we need to merge these tuples. The set of tuples computed for the fanout output f_1 corresponds to the result of the run on the red subexpression, so $R_{out=f_1} = \{(i_1, l_{f_1}, h_{f_1})\}$. The set of tuples that f_1 drives is a subset of the results of the run on the blue subexpression, namely $R_{in=f_1} = \{(f_1, \overline{o} \wedge y, o \wedge x), (f_1, o \wedge \overline{x}, \overline{o} \wedge z)\}$. So $hi(f_1) = \overline{o} \wedge y \vee o \wedge \overline{x}$ and $lo(f_1) = o \wedge x \vee \overline{o} \wedge z$. By replacing h_{f_1} with $hi(f_1)$ and l_{f_1} with $lo(f_1)$ the merging step thus delivers the tuple $(i_1, \overline{o} \wedge y \vee o \wedge \overline{x}, o \wedge x \vee \overline{o} \wedge z)$. Similarly, the merging step for f_2 delivers the tuple $(i_2, \overline{o} \wedge \overline{y} \vee o \wedge \overline{x}, o \wedge x \vee \overline{o} \wedge \overline{z})$.

So the relation computed for this instance is

$$\begin{array}{lllll}\Big((\overline{o} \wedge y \quad \vee \quad o \wedge \overline{x}) \rightarrow i_1\Big) & \wedge & \Big((o \wedge x \quad \vee \quad \overline{o} \wedge z) \rightarrow \overline{i_1}\Big) & \wedge \\ \Big((\overline{o} \wedge \overline{y} \quad \vee \quad o \wedge \overline{x}) \rightarrow i_2\Big) & \wedge & \Big((o \wedge x \quad \vee \quad \overline{o} \wedge \overline{z}) \rightarrow \overline{i_2}\Big). & \end{array}$$

★

Notice that the condition under which an input shall be driven with a high value, and the condition under which it shall be driven with a low value are not necessarily mutually exclusive anymore. In the above example the assignments

$$\overleftarrow{o} = \{o \mapsto \mathsf{ff}\}, \quad \text{and} \quad \overleftarrow{\mathcal{X}} = \{x \mapsto \mathsf{tt}, y \mapsto \mathsf{tt}, z \mapsto \mathsf{tt}\}$$

deliver

$$R(\overleftarrow{o}, \overleftarrow{\mathcal{X}})[\mathcal{V}] = (\mathsf{tt} \rightarrow i_1) \wedge (\mathsf{tt} \rightarrow \overline{i_1}) \wedge (\mathsf{ff} \rightarrow i_2) \wedge (\mathsf{ff} \rightarrow \overline{i_2}) = i_1 \wedge \overline{i_1} = \mathsf{ff},$$

so the relation is unsatisfiable. More generally, whenever an assignment leads to conflicting requirements the SIR relation evaluates to false. As detailed in Theorem 3 by Melham and Jones ([2]), this means that the STE run that is created through our automatic abstraction framework does not process the setting. These passes are not a problem, as the coverage proof guarantees that an assignment exists that does not lead to such conflicts.In essence, the conflicts are non-indexing assignments to the indexing variables, and are recognised as such. While conflicts do not change the correctness of the result, they do show that the cases to verify could have been encoded with fewer variables, or with simpler expressions.

Still, this section described a way of considerably reducing the number of variables introduced when handling DAGs. Furthermore, as we do not duplicate the analysis of the specification at fanout points, we avoid unnecessarily increasing the complexity of the conditions H_i and L_i for the input n_i. Another side-effect is a decreased number of conflicting, non-indexing cases. This has a significant effect, because BDD operations are usually faster if fewer variables are involved, and if expressions are simpler. As we assume that STE internally uses binary decision diagrams to represent the node values, splitting DAGs at fanout points to generate an SIR relation has an immediate effect on the verification costs.

In Section 4.7 we discuss how to reuse variables even further, thus reducing the number of variables required. The experimental results provided in Chapter 5 assess what effect this has on the execution times of verification tasks. In particular, we

observe that merely reducing the number of indexing variables without paying attention to how much this increases the complexity of the expressions can backfire. It may indeed increase verification costs rather than reduce them.

4.2 Multiple Outputs

The discussion so far has assumed that the specification has a single output only. Of course this is an unrealistic restriction, which we remove at this point. We can handle multiple outputs in a way similar to DAGs. The most obvious extension is to run the algorithm for each output independently, and use a unique indexing variable, o_i, for each run. Furthermore, indexing variables must not be shared in the relations produced by the separate runs. This ensures independence of the results and allows us to form the conjunction of the relations, where coverage follows trivially.

Alternatively, we can run STE independently for each output. This can reduce verification time in some instances.

In both variants of handling multiple outputs the specification is analysed several times. But when the bexpr is a DAG – which is essentially always the case for multiple outputs – this process can be sped up. When running the basic algorithm on the trees received after cutting the DAG at its fanout points we need only replace the intermediary variables h_f and l_f by the correct H and L values. If we want to construct one STE run for all outputs at once, we additionally have to substitute the introduced variables for fresh ones to make them unique. This avoids running the auto_abstract algorithm multiple times on the same expressions, which can save time especially when the specification is large. With respect to correctness it is equivalent to running the algorithm several times, so coverage is guaranteed without the need for further proof.

4.3 Symbolic Constants

Recall that we construct symbolically indexed STE runs by computing a relation that transforms the full symbolic simulation to a simulation with more elaborate indexing. Full symbolic simulation gives every input a symbolic value, n_i `is` v_i. As this means that the input has a Boolean value for every evaluation of the indexing variables, we call this a *symbolic constant*. The more elaborate abstraction scheme, on the other hand, gives the inputs a symbolic value only in particular cases, namely when the specification needs it to be determinate. But sometimes it might be desirable or

necessary to give inputs a symbolic value even if the specification does not suggest this.

For example, consider the circuit shown in Figure 4.2. It essentially implements "if i_1 then i_2 else i_2". While we need not know the value of input i_1 to determine the value of the specification, VAR i_2, simulation does require it. If we simulate the circuit with $i_1 = \mathrm{X}$, then the output is indeterminate regardless of i_2. This is a consequence of the abstraction framework in STE not being able to capture certain relationships. Only when specifying i_1 does the simulator know which of its inputs the multiplexer would output – irrespective of whether both inputs have the same value. Of course this example is contrived, but in practice we often see that the implementation needs more information to get a determinate value than the specification suggests. This is the reason why over-abstraction can occur.

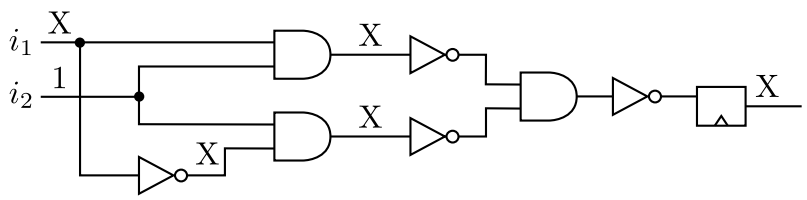


Figure 4.2: Circuit with a specification of VAR i_2, computed by implementing a multiplexer "if i_1 then i_2 else i_2"; ternary simulation with $i_1 = \mathrm{X}$ and $i_2 = 1$ leads to an indeterminate output.

We give an example of such an occurrence in Chapter 8. The content-addressable memory we want to verify needs additional input information, because some arithmetic operations are performed. While such computations can sometimes be logically simplified in the specification, the implementation always requires the values of the nodes involved in the arithmetic operations.

When verifying a circuit we may know up-front that specific inputs must be driven to get a determinate output. In such cases we may want to signal that these inputs always be driven with a symbolic value, even if the specification does not require it. In this dissertation we call such inputs *symbolic constants*. This name is motivated by the fact that the Boolean value is always known during simulation time, and thus the inputs can be processed in a way similar to the handling of constant values.

It is easy to modify the auto_abstract algorithm to support symbolic constants: as we are constructing a relation that tells us how to reindex a run which runs all inputs with a variable, i.e., all inputs are symbolic constants, we only need to ensure that the declared inputs must not be reindexed in the process.

Recall that the relation we construct is a conjunction of implications,

$$\bigwedge_i (H_i \to v_i) \wedge (L_i \to \overline{v_i}).$$

This expresses that the input n_i needs to have a high value only when H is satisfied, and a low value when L is satisfied. If we always want n_i to have a determinate value, then choosing $H = v_i$ and $L = \overline{v_i}$ is adequate. But then, of course, the implications $(v_i \to v_i) \wedge (\overline{v_i} \to \overline{v_i})$ are tautologies and can be omitted. In essence, this means that the algorithm can terminate on symbolic constants without extending the relation. Thus, correctness follows, as seen in Theorem 3.5.

4.4 Additional Constructors

The basic auto_abstract algorithm assumes that the specification uses the constructors VAR, NOT, and AND. We can extend the algorithm to accept other specification constructors as well. This can express more information about the task of parts of the circuit, and thus helps us construct better encodings of the partial input combinations we determine.

4.4.1 Multiple-Input AND Constructors

We can extend the allowed constructors to include multiple-input AND expressions. A multiple-input AND constructor is a gate which has multiple inputs and one output. If any one of its inputs is low, then the output is low. Only if all of its inputs are high, is the output high.

Figure 4.3(a) shows which relation, expressed in tuples, the auto_abstract algorithm returns when running it on the specification

$$((i_1 \text{ AND } i_2) \text{ AND } (i_3 \text{ AND } i_4)) \text{ AND } ((i_5 \text{ AND } i_6) \text{ AND } (i_7 \text{ AND } i_8)).$$

The SIR relation encodes that all eight inputs need to be high if the output is to be high, that is, when o evaluates to true; and only one input needs to be low if the output is to be false, i.e., when o evaluates to false. We use seven variables, which are introduced while handling each AND-constructor.

But three variables are sufficient for enumerating eight cases. For example, the relation shown in Figure 4.3(b) encodes the same behaviour for driving inputs. So if we know that we want to encode the partial input combinations for an 8-input AND-

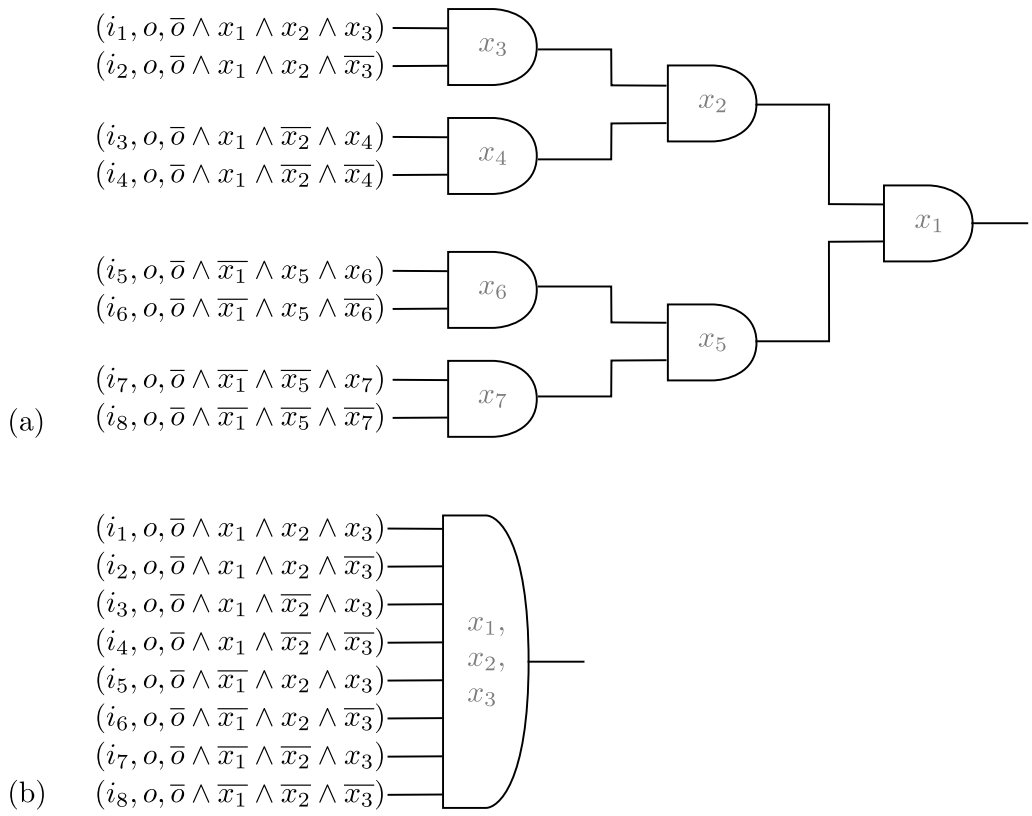


Figure 4.3: Relation tuples returned when running auto_abstract (a) on seven AND constructors, and the suggested handling of (b) an 8-input AND constructor. Each AND-gate is labelled with the variables introduced by the algorithm when processing it.

gate, rather than handling a tree of seven 2-input AND constructors, we can choose that better encoding. Especially when reducing the number of variables does not lead to more complex expressions, this can lead to drastically decreased verification costs when BDDs are used. This different approach corresponds to using a binary encoding when deciding which input must be low to force a low output, whereas before we used a unary encoding.

More generally, we can handle n-input AND constructors as follows:

$$\mathsf{auto\_abstract}(\mathrm{AND}_{i \in I} \mathrm{SPEC}_i, H, L) = \bigwedge_{i \in I} \mathsf{auto\_abstract}(\mathrm{SPEC}_i, H, L \wedge case_i),$$

where $\{case_i \mid i \in I\}$ is a set of mutually exclusive expressions that use $\lceil \log_2 i \rceil$ fresh indexing variables. For example, for a 5-input AND-gate three variables are required,

and a possible set of case expressions is

$$\{\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3}, \quad \overline{x_1} \wedge \overline{x_2} \wedge x_3, \quad \overline{x_1} \wedge x_2 \wedge \overline{x_3}, \quad \overline{x_1} \wedge x_2 \wedge x_3, \quad x_1\}.$$

This optimisation can also be applied if several 2-input AND constructors are used successively in the specification. Multiple-input AND-gates can be detected greedily. If the outer-most constructor is a 2-input AND, then also examine the two subexpressions, which are inputs to the AND. If either also have an AND as their outer-most constructor, use those subexpressions as inputs instead. Continue processing subexpressions until none of the subexpressions have an AND as their outer-most constructor anymore.

Example: Finding multiple-input AND-gates
Suppose we are given the specification

(i_1 AND i_2) AND (i_3 AND (NOT i_4 AND NOT (i_5 AND i_6)) AND (NOT i_7 AND i_8).

Previously we would have simply processed the outer-most AND constructor, and then recursively called auto_abstract on

(i_1 AND i_2) AND (i_3 AND (NOT i_4 AND NOT (i_5 AND i_6))

and

(NOT i_7 AND i_8).

With the above modification, instead we process a 7-input AND with the inputs i_1, i_2, i_3, NOT i_4, *NOT* (i_5 AND i_6), NOT i_7, i_8. See Figure 4.4 for a graphical representation of the above specification. Note that the VAR constructors are omitted, and the AND constructors that can be combined to a single multiple-input AND are depicted in bold font. ★

Theorem 4.1 (page 98) shows that handling multiple-input AND constructors this way is correct.

Optimisation for Symbolic Constant Inputs

In Section 4.3 we introduced the notion of symbolic constants. These are inputs that are always driven with a Boolean value. Now suppose one of the inputs of an AND-gate is a symbolic constant, without loss of generality i_1. As this input is driven with

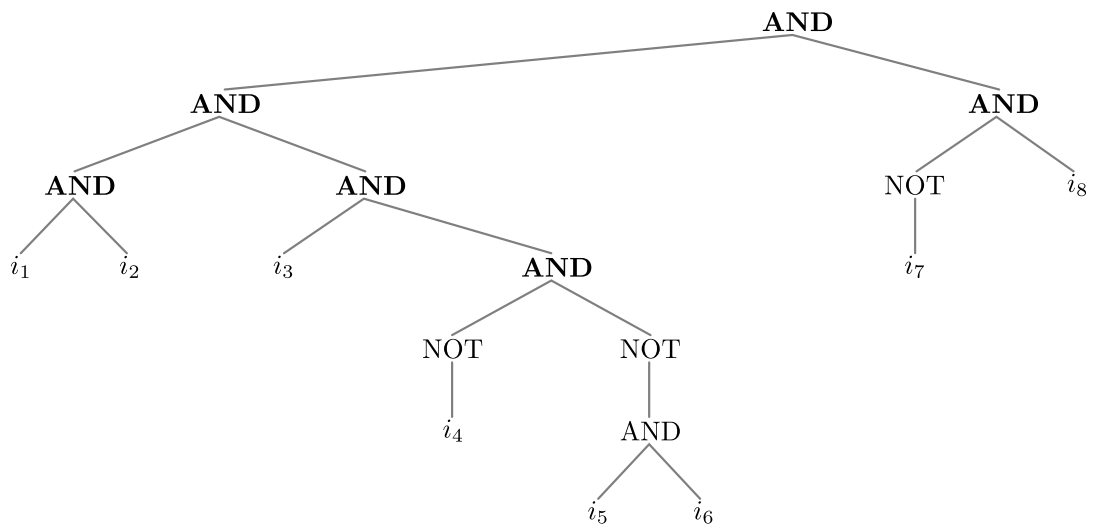


Figure 4.4: Graphical representation of the specification (i_1 AND i_2) AND (i_3 AND (NOT i_4 AND NOT (i_5 AND i_6)) AND (NOT i_7 AND i_8).

a Boolean value in every case, it is not necessary for the algorithm to analyse whether it actually needs to be driven. So we do not have to index this case, and potentially use fewer indexing variables.

Example: 2-input AND with Symbolic Constants
Suppose one of the inputs, i_1, of a normal, 2-input AND is a symbolic constant. We then do not need to introduce any indexing variable:

$$\mathsf{auto\_abstract}(i_1 \text{ AND } i_2, H, L) = \mathsf{auto\_abstract}(i_2, H, L \wedge i_1)$$

Essentially, before we used an indexing variable to encode whether the first input or the second input needed to be false to lead to a false output. But if the first input always has a Boolean value, then we need only drive the second input with a Boolean value if the first one is not driven with a false value, i.e., whenever i_1 is true. This corresponds to conjoining the L-condition with i_1. ★

More generally, we can apply this simplification as soon as an input of an AND-gate depends only on symbolic constants, as it then always has a Boolean, non-X value. Furthermore, several inputs may qualify for this simplification. This means that we can partition the set of inputs for a multiple-input AND-gate into those inputs that depend only on symbolic constants, called $I_{\mathcal{C}}$, and those that do not, $I \backslash I_{\mathcal{C}}$. We can

then handle multiple-input AND-gates as follows:

$$\mathsf{auto\_abstract}(\mathrm{AND}_{i\in I}\ i, H, L) = \bigwedge_{i\in I\setminus I_{\mathcal{C}}} \mathsf{auto\_abstract}(i, H, L \wedge \mathsf{tt}_{I_{\mathcal{C}}} \wedge case_i),$$

where $\{case_i \mid i \in I\setminus I_{\mathcal{C}}\}$ is a set of mutually exclusive expressions that depend on $\lceil \log_2 |I\setminus I_{\mathcal{C}}| \rceil$ fresh indexing variables, and where $\mathsf{tt}_{I_{\mathcal{C}}} = \bigwedge_{j\in I_{\mathcal{C}}} j$ encodes when all inputs that depend only on symbolic constants are driven with a true value. Theorem 4.1 shows that this adjusted handling still leads to correct verification results.

4.4.2 OR Constructors

Similarly, we can allow OR constructors. For these the abstraction scheme should capture that we only need one of the inputs to be high for a high output, and all inputs to be low for a low output. That is,

$$\begin{aligned}\mathsf{auto\_abstract}(\mathrm{SPEC}_1\ \mathrm{OR}\ \mathrm{SPEC}_2, H, L) = \quad & \mathsf{auto\_abstract}(\mathrm{SPEC}_1, H \wedge x, L) \wedge \\ & \mathsf{auto\_abstract}(\mathrm{SPEC}_2, H \wedge \overline{x}, L),\end{aligned}$$

where x is a fresh indexing variable. Indeed, when running the auto_abstract algorithm on the bexpr NOT (NOT a AND NOT b), which expresses the same behaviour, then we receive the exact same relation. This is so, because the algorithm only swaps the high condition, H, and the low condition, L, on a NOT constructor, rather than having to introduce new indexing variables.

So we can extend the algorithm by an OR constructor, but the proof of correctness is already supplied by the ones for the NOT and AND constructors. If we recognise multiple-input OR constructors, then this is possible with the same reasoning as seen for multiple-input AND constructors:

$$\mathsf{auto\_abstract}(\mathrm{OR}_{i\in I}\ \mathrm{SPEC}_i, H, L) = \bigwedge_{i\in I} \mathsf{auto\_abstract}(\mathrm{SPEC}_i, H \wedge case_i, L),$$

where $\{case_i \mid i \in I\}$ is a set of mutually exclusive expressions that use $\lceil \log_2 i \rceil$ fresh indexing variables. The proof of correctness corresponds to that of the multiple-input AND one provided in Theorem 4.1, and the basic handling of NOT gates using the equivalence $\mathrm{OR}_{i\in I}\ i \equiv \mathrm{NOT}\ (\mathrm{AND}_{i\in I}\ (\mathrm{NOT}\ i))$.

Optimisation for Symbolic Constant Inputs

As seen with multiple-input AND-gates, optimisations are also possible for multiple-input OR gates when some of the inputs depend only on symbolic constants:

$$\mathsf{auto\_abstract}(\mathrm{OR}_{i \in I}\ i, H, L) = \bigwedge_{i \in I \setminus I_C} \mathsf{auto\_abstract}(i, H \wedge \mathsf{ff}_{I_C} \wedge case_i, L),$$

where $\{case_i \mid i \in I \setminus I_C\}$ is a set of mutually exclusive expressions that depend on $\lceil \log_2 |I \setminus I_C| \rceil$ fresh indexing variables, and where $\mathsf{ff}_{I_C} = \bigwedge_{j \in I_C} \overline{j}$ encodes when all inputs that depend only on symbolic constants are driven with a false value.

4.4.3 XNOR Constructors

Another constructor to consider is XNOR, which returns a high value if both of its inputs have the same value. Figure 4.5 shown a bexpr that captures this behaviour, as well as the relation tuples that are returned by the auto_abstract algorithm. Notice

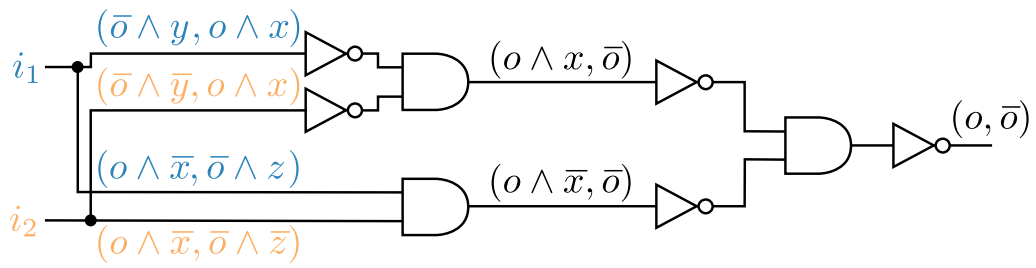


Figure 4.5: Result of running auto_abstract on an XNOR gate expressed in NOT- and AND-gates

that three indexing variables are introduced, and the following assignments lead to relations that evaluate to true:

o	x	y	z	i_1	i_2
1	0			1	1
1	1			0	0
0		0	1	0	1
0		1	0	1	0

In particular, any assignment where the indexing variables y and z are given the same value leads to a false relation, thus resulting in a pass in the final STE run.

Indeed, the following handling of XNOR constructors needs one variable only:

$$\begin{aligned}&\mathsf{auto\_abstract}(\text{SPEC}_1 \text{ XNOR } \text{SPEC}_2, H, L) = \\ &\quad \mathsf{auto\_abstract}(\text{SPEC}_1, H \wedge x \vee L \wedge x, H \wedge \overline{x} \vee L \wedge \overline{x}) \wedge \\ &\quad \mathsf{auto\_abstract}(\text{SPEC}_2, H \wedge x \vee L \wedge \overline{x}, H \wedge \overline{x} \vee L \wedge x)\end{aligned}$$

For example, assume that an adjusted version of auto_abstract is run on the simple specification v_1 XNOR v_2. The algorithm then returns the relation

$$\begin{aligned}&((o \wedge x \vee \overline{o} \wedge x \rightarrow v_1) \quad \wedge \\ &((o \wedge \overline{x} \vee \overline{o} \wedge \overline{x}) \rightarrow \overline{v_1}) \quad \wedge \\ &((o \wedge x \vee \overline{o} \wedge \overline{x}) \rightarrow v_2) \quad \wedge \\ &((o \wedge \overline{x} \vee \overline{o} \wedge x) \rightarrow \overline{v_2})\end{aligned}$$

The basic idea behind this encoding is as follows. We drive the input i_1 with the value of the indexing variable x, and then – depending on the value of o – drive i_2 with the same, or inverted value of x. In particular, if we look at the relation where o is assigned a Boolean value, then we receive:

$$\begin{aligned}R(\{o \mapsto \mathsf{tt}\})[\mathcal{X}, \mathcal{V}] = &\quad (x \rightarrow v_1) \wedge (\overline{x} \rightarrow \overline{v_1}) \wedge (x \rightarrow v_2) \wedge (\overline{x} \rightarrow \overline{v_2}) \\ R(\{o \mapsto \mathsf{ff}\})[\mathcal{X}, \mathcal{V}] = &\quad (x \rightarrow v_1) \wedge (\overline{x} \rightarrow \overline{v_1}) \wedge (\overline{x} \rightarrow v_2) \wedge (x \rightarrow \overline{v_2})\end{aligned}$$

Theorem 4.1 (page 98) shows that this handling still ensures correctness, while using two variables fewer. Furthermore, notice that now no assignments falsify the relation anymore.

Optimisation for Symbolic Constant Inputs

Suppose one of the inputs of an XNOR gate is a symbolic constant, without loss of generality i_1. Then we do not need to introduce any indexing variables at all. Instead we can use the Boolean value of i_1 to determine when to drive i_2:

$$\mathsf{auto\_abstract}(i_1 \text{XNOR} i_2, H, L) = \mathsf{auto\_abstract}(i_2, H \wedge i_1 \vee L \wedge \overline{i_1}, H \wedge \overline{i_1} \vee L \wedge i_1)$$

Previously, i_1 was driven with the value of x and i_2 with the value of x or $\overline{x}$, depending on whether we wanted a high or low output, respectively. But if i_1 is a symbolic constant, we can use it instead of x. We then drive i_2 with the Boolean value of i_1 or $\overline{i_1}$, depending on whether we want a high or low output respectively. More generally, as soon as one of the inputs of the XNOR depends only on symbolic constants it

always has a Boolean value, i.e., is never X. In all such cases this simplification can be applied. Again, this optimisation is shown to be correct in Theorem 4.1.

4.5 Algorithm

```
 1  auto_abstract(C, SPEC, H, L, name) =
 2    if free_vars(SPEC) ⊆ C or is_VAR(SPEC) then
 3      return {(SPEC, H, L)}
 4    elseif is_XNOR(SPEC) then
 5      (SPEC1, SPEC2) := destruct_XNOR(SPEC)
 6      if free_vars(SPEC1) ⊆ C then
 7        return auto_abstract(C, SPEC2, H ∧ SPEC1 ∨ L ∧ ¬SPEC1,
                                          H ∧ ¬SPEC1 ∨ L ∧ SPEC1, name)
 8      else
 9        {X1, X2} := get_case_expressions(name, 2)
10        {name1, name2} := get_unique_names(name, 2)
11        return auto_abstract(C, SPEC1, H ∧ X1 ∨ L ∧ X1,
                                          H ∧ X2 ∨ L ∧ X2, name1)
                 ∪ auto_abstract(C, SPEC2, H ∧ X1 ∨ L ∧ X2,
                                          H ∧ X2 ∨ L ∧ X1, name2)
12    elseif is_NOT(SPEC) then
13      return auto_abstract(C, strip_NOT(SPEC), L, H, name)
14    else // is_AND
15      (cinps, oinps) := find_max_AND(C, SPEC)
16      if cinps ≠ ∅ then
17      c := ⋀_{ci∈cinps} ci
18        rel := {(c, H, ff)}
19      else
20        c := tt
21        rel := ∅
22      cases := get_case_expressions(name, |oinps|)
23      if H = ff then
24        names := get_same_names(name, |oinps|)
25      else
26        names := get_unique_names(name, |oinps|)
27      for i from 1 to |oinps|
28        rel := rel ∪ auto_abstract(C, oinps_i, H, L ∧ cases_i ∧ c, names_i)
29      return rel
```

Figure 4.6: Advanced back-propagation algorithm.

Figure 4.6 shows an auto_abstract algorithm, which can handle symbolic constants,

multiple-input AND-gates, XNOR gates, and has a slightly more relaxed approach to introducing indexing variables. In particular, destruct_XNOR as seen in line 5 checks whether the outermost gates have the same input-output behaviour as an XNOR-gate. It furthermore orders the two inputs of the gate, such that if one of its fanins only depends on symbolic constants, it is the first input. This avoids duplicating the code seen in lines 6–7. The call to find_max_AND as seen in line 15 determines the largest possible multiple-input AND-gate, as already described in Section 4.4.1, and also splits the inputs into those that only depend on symbolic constants, *cinps*, and the remaining inputs, *oinps*.

In Section 4.7 we discuss the requirements that the get_case_expressions (lines 9 and 22) function needs to satisfy. For the correctness proof of Theorem 4.1 we assume that get_case_expressions introduces new indexing variables, and returns a set of mutually exclusive expressions that depend only on those variables. The more flexible, and thus more powerful approach described in Section 4.7 is accompanied by notes on how the proof can be adjusted to also guarantee correctness by construction. Finally, recall that we handle DAG specifications by cutting them in their fanout points to receive tree structures, which can be analysed by the improved auto_abstract algorithm separately, and then merged. This can be seen as a preprocessing step, and is thus not captured in the algorithm or proof to help simplify the presentation. The correctness of handling DAGs is a simple extension of Theorem 4.1, as already argued in Section 4.1 for Theorem 3.5.

4.6 Correctness

Theorem 3.5 proved that, by construction, the basic version of auto_abstract generates SIR relations that cover all observable simulation cases, and thus can be applied to the auxiliary STE run, $R \models (\bigwedge n_i \texttt{ is } v_i)_R \Rightarrow (out \texttt{ is } o)^R$, to formally verify that a circuit meets its specification. As checking that the SIR conditions hold can be very expensive, we want to be able to make the same statement of correctness by construction for our advanced version of the automatic abstraction discovery algorithm. This is captured in Corollary 4.2, which directly follows from Theorem 4.1.

Theorem 4.1 (Correctness of the Advanced Algorithm)**.** *Given a binary expression represented as a bexpr-tree* SPEC$[\mathcal{V}]$*, define*

$$R[\mathcal{E}, \mathcal{X}, \mathcal{V}] = \textit{auto\_abstract}(\mathcal{C}, \text{SPEC}, E, \overline{E}, name)$$

as the result of the algorithm in Figure 4.6. Here, $\mathcal{E}$ is the set of free variables used in E, $\mathcal{V}$ is the set of variables used in SPEC*, and $\mathcal{X}$ is the set of variables introduced by the algorithm. Assuming the choice of name ensures that none of the generated variables $\mathcal{X}$ are in the set $\mathcal{E}$ or $\mathcal{V}$, the following is valid:*

$$\forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{\mathcal{E}}.E(\overleftarrow{\mathcal{E}}) = \text{SPEC}(\overleftarrow{\mathcal{V}}) \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) \tag{4.6.1}$$

Proof. As already seen in Theorem 3.5, we use the shorthand $R_{\text{G},H,L}$ when reasoning about the result of running $\mathsf{auto\_abstract}(\mathcal{C}, \text{G}, H, L, name)$. The set of symbolic constants, $\mathcal{C}$, is never changed and can safely be omitted in this notation. We also omit *name* for better readability, but we go into the importance of *name* where adequate.

For Claim 3.3.5, first assume that $\overleftarrow{\mathcal{V}}$ and $\overleftarrow{\mathcal{E}}$ are arbitrary, but fixed. An induction on k, the depth of the binary expression, follows. The induction start, hypothesis, and step are the same as in the proof of Theorem 3.5. The handling of NOT-gates also has not changed. It remains to analyse the handling of XNOR, and to generalise the 2-input AND to a multiple-input AND.
(a) $\text{SPEC} = \text{G}_1$ XNOR G_2 and $\mathsf{free\_vars}(\text{G}_1) \subseteq \mathcal{C}$

As seen in line 6,

$$\begin{aligned} R[\mathcal{E}, \mathcal{X}, \mathcal{V}] &= \mathsf{auto\_abstract}(\mathcal{C}, \text{SPEC}, E, \overline{E}, name) \\ &= \mathsf{auto\_abstract}(\mathcal{C}, \text{G}_1 \text{ XNOR } \text{G}_2, E, \overline{E}, name) \\ &= \mathsf{auto\_abstract}(\mathcal{C}, \text{G}_2, \quad E \wedge \text{G}_1 \vee \overline{E} \wedge \overline{\text{G}_1}, \\ &\qquad\qquad\qquad\qquad E \wedge \overline{\text{G}_1} \vee \overline{E} \wedge \text{G}_1, name) \end{aligned}$$

As $\mathsf{free\_vars}(\text{G}_1) \subseteq \mathcal{C}$, we know that during simulation the value of $\text{G}_1(\overleftarrow{\mathcal{V}})$ is always determinate. Hence, we can argue using the concrete value of $\text{G}_1(\overleftarrow{\mathcal{V}})$.
(a i) $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$

We need to show that $E(\overleftarrow{\mathcal{E}}) = \text{SPEC}(\overleftarrow{\mathcal{V}}) \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$. But if $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, then $\text{SPEC}(\overleftarrow{\mathcal{V}}) = \text{G}_1(\overleftarrow{\mathcal{V}})$ XNOR $\text{G}_2(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ XNOR $\text{G}_2(\overleftarrow{\mathcal{V}}) = \text{G}_2(\overleftarrow{\mathcal{V}})$. Furthermore, $R_{\text{SPEC},E,\overline{E}} = R_{\text{G}_2,E\vee\mathsf{ff},\mathsf{ff}\vee\overline{E}} = R_{\text{G}_2,E,\overline{E}}$. So by induction hypothesis
$E(\overleftarrow{\mathcal{E}}) = \text{SPEC}(\overleftarrow{\mathcal{V}}) = \text{G}_2(\overleftarrow{\mathcal{V}}) \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$

(a ii) $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$

We need to show that $E(\overleftarrow{\mathcal{E}}) = \text{SPEC}(\overleftarrow{\mathcal{V}}) \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$. But if $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$, then $\text{SPEC}(\overleftarrow{\mathcal{V}}) = \text{G}_1(\overleftarrow{\mathcal{V}})$ XNOR $\text{G}_2(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ XNOR $\text{G}_2(\overleftarrow{\mathcal{V}}) = \overline{\text{G}_2(\overleftarrow{\mathcal{V}})}$. Further-

more, $R_{\text{SPEC},E,\overline{E}} = R_{\text{G}_2,\mathsf{ff}\vee\overline{E},E\vee\mathsf{ff}} = R_{\text{G}_2,\overline{E},E}$. So, as seen for NOT-gates,
$E(\overleftarrow{\mathcal{E}}) = \text{SPEC}(\overleftarrow{\mathcal{V}}) = \overline{\text{G}_2(\overleftarrow{\mathcal{V}})} \Leftrightarrow \exists\overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) = R_{\overline{\text{G}_2},\overline{E},E}(\overleftarrow{\text{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$.

(b) SPEC = G_1 XNOR G_2 and $\mathsf{free\_vars}(\text{G}_1) \not\subseteq \mathcal{C}$

As seen in line 11,

$$\begin{aligned}
R[\mathcal{E},\mathcal{X},\mathcal{V}] &= \mathsf{auto_abstract}(\mathcal{C}, \text{SPEC}, E, \overline{E}, name) \\
&= \mathsf{auto_abstract}(\mathcal{C}, \text{G}_1 \text{ XNOR } \text{G}_2, E, \overline{E}, name) \\
&= \mathsf{auto_abstract}(\mathcal{C}, \text{G}_1, \quad E \wedge case_1 \vee \overline{E} \wedge case_1, \\
&\qquad\qquad\qquad\qquad E \wedge case_2 \vee \overline{E} \wedge case_2, name_1) \wedge \\
&\qquad \mathsf{auto_abstract}(\mathcal{C}, \text{G}_2, \quad E \wedge case_1 \vee \overline{E} \wedge case_2, \\
&\qquad\qquad\qquad\qquad E \wedge case_2 \vee \overline{E} \wedge case_1, name_2)
\end{aligned}$$

(b i) Need to show: $E(\overleftarrow{\mathcal{E}}) = \text{SPEC}(\overleftarrow{\mathcal{V}}) \Rightarrow \exists\overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$

By induction hypothesis, $E(\overleftarrow{\mathcal{E}}) = \text{G}_i(\overleftarrow{\mathcal{V}}) \Rightarrow \exists\overleftarrow{\mathcal{X}_i}.R_{\text{G}_i,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_i}, \overleftarrow{\mathcal{V}})$.
(b i α) Case $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$

Let $\mathcal{X}_s$ be the set of fresh variables introduced in line 9,

$$\mathcal{X}_s = \mathsf{free\_vars}(case_1) = \mathsf{free\_vars}(case_2).$$

Then by construction of the case expressions $case_1$ and $case_2$, there exists an assignment $\overleftarrow{\mathcal{X}_s}$ such that $case_1(\overleftarrow{\mathcal{X}_s}) = \mathsf{tt}$, and thus $case_2(\overleftarrow{\mathcal{X}_s}) = \mathsf{ff}$. Then $\text{SPEC}(\overleftarrow{\mathcal{V}}) = \text{G}_2(\overleftarrow{\mathcal{V}})$, and $R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{V}})[\mathcal{X}] = R_{\text{G}_1,\mathsf{tt},\mathsf{ff}}(\overleftarrow{\mathcal{V}})[\mathcal{X}_1] \wedge R_{\text{G}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{V}})[\mathcal{X}_2]$. By induction hypothesis, $E(\overleftarrow{\mathcal{E}}) = \text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt} \Rightarrow \exists\overleftarrow{\mathcal{X}_1}.R_{\overleftarrow{g}_1,\mathsf{tt},\mathsf{ff}}(\overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$, and $E(\overleftarrow{\mathcal{E}}) = \text{G}_2(\overleftarrow{\mathcal{V}}) \Rightarrow \exists\overleftarrow{\mathcal{X}_2}.R_{\text{G}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$. But by choice of *name* it is guaranteed that $\mathcal{X}_1$, $\mathcal{X}_2$, and $\mathcal{X}_s$ are disjoint. So the three assignments can be combined without conflict. Thus, $\exists\overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{V})$.

(b i β) Case $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$

Then by construction of the case expressions $case_1$ and $case_2$, there exists an assignment $\overleftarrow{\mathcal{X}_s}$ such that $case_2(\overleftarrow{\mathcal{X}_s}) = \mathsf{tt}$, and thus $case_1(\overleftarrow{\mathcal{X}_s}) = \mathsf{ff}$. Then $\text{SPEC}(\overleftarrow{\mathcal{V}}) = \overline{\text{G}_2(\overleftarrow{\mathcal{V}})}$, and
$R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{V}})[\mathcal{X}] = R_{\text{G}_1,\mathsf{ff},\mathsf{tt}}(\overleftarrow{\mathcal{V}})[\mathcal{X}_1] \wedge R_{\text{G}_2,\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{V}})[\mathcal{X}_2]$. By induction hypothesis, $E(\overleftarrow{\mathcal{E}}) = \overline{\text{G}_1(\overleftarrow{\mathcal{V}})} = \mathsf{ff} \Rightarrow \exists\overleftarrow{\mathcal{X}_1}.R_{\text{G}_1,\mathsf{ff},\mathsf{tt}}(\overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$, and
$E(\overleftarrow{\mathcal{E}}) = \overline{\text{G}_2(\overleftarrow{\mathcal{V}})} \Rightarrow \exists\overleftarrow{\mathcal{X}_2}.R_{\text{G}_2,\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$. But by choice of *name* it is guaranteed that $\mathcal{X}_1$, $\mathcal{X}_2$, and $\mathcal{X}_s$ are disjoint. So the three assignments can be combined without conflict. Thus, $\exists\overleftarrow{\mathcal{X}}.R(\overleftarrow{o}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$.

(b ii) Need to show: $\exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{o}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) \Rightarrow E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}})$

First observe that because both $\overleftarrow{\mathcal{V}}$ and $\overleftarrow{\mathcal{E}}$ are fixed, we know whether $E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}})$ holds or not. If it does, nothing needs to be shown. So it is sufficient to show that

$$E(\overleftarrow{\mathcal{E}}) \neq \textsc{spec}(\overleftarrow{\mathcal{V}}) \Rightarrow \nexists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}),$$

or, by applying the equality seen in line 11 of the algorithm,

$$\begin{aligned} &E(\overleftarrow{\mathcal{E}}) \neq (\textsc{g}_1 \text{ XNOR } \textsc{g}_2)(\overleftarrow{\mathcal{V}}) \Rightarrow \\ &\quad \nexists \overleftarrow{\mathcal{X}_1 \dot\cup \mathcal{X}_2 \dot\cup \mathcal{X}_s}. \quad R_{\textsc{g}_1, case_1, case_2}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1 \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) \wedge \\ &\qquad\qquad\qquad R_{\textsc{g}_2, E\wedge case_1 \vee \overline{E} \wedge case_2, E \wedge case_2 \vee \overline{E} \wedge case_1}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2 \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) \end{aligned}$$

We show this by first reasoning there is no such assignment when $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{ff}$, and then when $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{tt}$. In both cases we examine the possible assignments to the variables, $\mathcal{X}_s$, introduced for the XNOR.

(b ii α) $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{ff}$, $\overleftarrow{\mathcal{X}_s}$ such that $case_1 = \mathsf{tt}$ and $case_2 = \mathsf{ff}$

As $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, this means that $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}})$ must hold. If $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, then the induction hypothesis delivers that $\mathsf{ff} = E(\overleftarrow{\mathcal{E}}) \neq \textsc{g}_2(\overleftarrow{\mathcal{V}}) \Rightarrow \nexists \overleftarrow{\mathcal{X}_2}.R_{\textsc{g}_2, E, \overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$. But when $case_1 = \mathsf{tt}$ and $case_2 = \mathsf{ff}$ we know that $R_{\textsc{g}_2, E\wedge case_1 \vee \overline{E} \wedge case_2, E \wedge case_2 \vee \overline{E} \wedge case_1}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2 \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) = R_{\textsc{g}_2, E, \overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$. But if no assignment exists that delivers $R_{\textsc{g}_2, E, \overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}}) = \mathsf{tt}$, then the conjunction of this relation with any other expression is also false, as desired.
If, on the other hand, $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$, then $R_{\textsc{g}_1, case_1, case_2}[\mathcal{X}_1 \dot\cup \mathcal{X}_s] = R_{\textsc{g}_1, \mathsf{tt}, \mathsf{ff}}[\mathcal{X}_1]$. Suppose for contradiction that $\exists \mathcal{X}_1.R_{\textsc{g}_1, \mathsf{tt}, \mathsf{ff}}(\overleftarrow{\mathcal{X}_1}) = \mathsf{tt}$. Then by induction hypothesis with $E = \mathsf{tt}$ we can conclude that $\mathsf{tt} = E = \textsc{g}_1(\overleftarrow{\mathcal{V}})$, which is a contradiction to the assumption that $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$. So no such assignment can exist, as desired.

(b ii β) $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{ff}$, $\overleftarrow{\mathcal{X}_s}$ such that $case_1 = \mathsf{ff}$ and $case_2 = \mathsf{tt}$

As $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, this means that $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}})$ must hold. If $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \textsc{g}_2(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, then $R_{\textsc{g}_1, case_1, case_2}[\mathcal{X}_1 \dot\cup \mathcal{X}_s] = R_{\textsc{g}_1, \mathsf{ff}, \mathsf{tt}}[\mathcal{X}_1]$. Now suppose for contradiction that $\exists \overleftarrow{\mathcal{X}_1}.R_{\textsc{g}_1, \mathsf{ff}, \mathsf{tt}}(\overleftarrow{\mathcal{X}_1})$. Then by the induction hypothesis with $E = \mathsf{ff}$ we can conclude that $\mathsf{ff} = E = \textsc{g}_1(\overleftarrow{\mathcal{V}})$, which is a contradiction to the assumption that $\textsc{g}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$. So no such assignment can exist, as desired.

If, on the other hand, $\text{G}_1(\overleftarrow{\mathcal{V}}) = \text{G}_2(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$, then the induction hypothesis delivers that $\mathsf{tt} = \overline{E}(\overleftarrow{\mathcal{E}}) \neq \text{G}_2(\overleftarrow{\mathcal{V}}) \Rightarrow \nexists \overleftarrow{\mathcal{X}_2}.R_{\text{G}_2,\overline{E},\overline{\overline{E}}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}_2, \overleftarrow{\mathcal{V}})$. But $R_{\text{G}_2,\overline{E},\overline{\overline{E}}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}_2, \overleftarrow{\mathcal{V}}) = R_{\text{G}_2,\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}_2, \overleftarrow{\mathcal{V}})$ and also $R_{\text{G}_2,E\wedge case_1\vee\overline{E}\wedge case_2,E\wedge case_2\vee\overline{E}\wedge case_1}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}_2, \overleftarrow{\mathcal{V}}) = R_{\text{G}_2,\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}_2, \overleftarrow{\mathcal{V}})$. But if no assignment exists that delivers $R_{\text{G}_2,\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}}) = \mathsf{tt}$, then the conjunction of this relation with any other expression is also false, as desired.

(b ii γ) $\text{SPEC}(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{tt}$, $\overleftarrow{\mathcal{X}_s}$ such that $case_1 = \mathsf{tt}$ and $case_2 = \mathsf{ff}$

As $\text{SPEC}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, this means that $\text{G}_1(\overleftarrow{\mathcal{V}}) = \overline{\text{G}_2(\overleftarrow{\mathcal{V}})}$ must hold. If $\text{G}_2(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$, then the induction hypothesis delivers that $\mathsf{tt} = E(\overleftarrow{\mathcal{E}}) \neq \text{G}_2(\overleftarrow{\mathcal{V}}) \Rightarrow \nexists \overleftarrow{\mathcal{X}_2}.R_{\text{G}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$. But if no assignment exists that delivers $R_{\text{G}_2,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}}) = \mathsf{tt}$, then the conjunction of this relation with any other expression is also false, as required.

If, on the other hand, $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$, then assume for contradiction that $\exists \overleftarrow{\mathcal{X}_1}.R_{\text{G}_1,\mathsf{tt},\mathsf{ff}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$. Then by the induction hypothesis with $E = \mathsf{tt}$ we can conclude that $\mathsf{ff} = E \neq \text{G}_1(\overleftarrow{\mathcal{V}})$, which is a contradiction to the assumption that $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$. So no such assignment can exist. But $R_{\text{G}_1,case_1,case_2}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1 \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) = R_{\text{G}_1,\mathsf{tt},\mathsf{ff}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$, so we showed no fitting assignment exists, as desired.

(b ii δ) $\text{SPEC}(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{tt}$, $\overleftarrow{\mathcal{X}_s}$ such that $case_1 = \mathsf{ff}$ and $case_2 = \mathsf{tt}$

As $\text{SPEC}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, this means that $\text{G}_1(\overleftarrow{\mathcal{V}}) = \overline{\text{G}_2(\overleftarrow{\mathcal{V}})}$ must hold. If $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, then suppose for contradiction that $\exists \overleftarrow{\mathcal{X}_1}.R_{\text{G}_1,\mathsf{tt},\mathsf{ff}}$. Then by the induction hypothesis with $E = \mathsf{ff}$ we can conclude that $\mathsf{ff} = E = \text{G}_1(\overleftarrow{\mathcal{V}})$, which is a contradiction to the assumption that $\text{G}_1(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$. So no such assignment can exist. But $R_{\text{G}_1,case_1,case_2}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1 \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) = R_{\text{G}_1,\mathsf{ff},true}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$, so we showed no fitting assignment exists, as desired.

If, on the other hand, $\text{G}_2(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$, then the induction hypothesis delivers that $\mathsf{ff} = \overline{E}(\overleftarrow{\mathcal{E}}) \neq \text{G}_2(\overleftarrow{\mathcal{V}}) \Rightarrow \nexists \overleftarrow{\mathcal{X}_2}.R_{\text{G}_2,\overline{E},\overline{\overline{E}}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$. But $R_{\text{G}_2,\overline{E},\overline{\overline{E}}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}}) = R_{\text{G}_2,\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$ and $R_{\text{G}_2,E\wedge case_1\vee\overline{E}\wedge case_2,E\wedge case_2\vee\overline{E}\wedge case_1}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2 \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) = R_{\text{G}_2,\overline{E},E}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_2}, \overleftarrow{\mathcal{V}})$, so we showed no fitting assignment exists, as desired.

(c) $\text{SPEC} = \text{AND}_i \text{G}_i$

First, assume that $E \neq \mathsf{ff}$ and $cinps = \emptyset$, and let $\mathcal{X}_s$ be the set of fresh variables

introduced for *cases*. Then, as seen in lines 27–28,

$$\begin{aligned} R[\mathcal{E},\mathcal{X},\mathcal{V}] &= \mathsf{auto\_abstract}(\mathcal{C}, \textsc{spec}, E, \overline{E}, name) \\ &= \mathsf{auto\_abstract}(\mathcal{C}, \mathrm{A}ND_i\mathrm{G}_i, E, \overline{E}, name) \\ &= \textstyle\bigwedge_i \mathsf{auto\_abstract}(\mathcal{C}, \mathrm{G}_i, E, \overline{E} \wedge cases_i \wedge \mathsf{tt}, names_i) \end{aligned}$$

(c i) Need to show: $E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}}) \Rightarrow \exists\overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$

(c i α) Case $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$

As $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ we also know that $\mathrm{G}_i(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ for all i, and thus in particular for $i = 1$. By construction of *cases*, we also know $\exists\overleftarrow{\mathcal{X}_s}.cases_1(\overleftarrow{\mathcal{X}_s}) = \mathsf{tt}$, and thus $cases_i(\overleftarrow{\mathcal{X}_s}) = \mathsf{ff}$ for all $i \neq 1$. But by the induction hypothesis, $E(\overleftarrow{\mathcal{E}}) = \mathrm{G}_i(\overleftarrow{\mathcal{V}}) \Rightarrow \exists\overleftarrow{\mathcal{X}_i}.R_{\mathrm{G}_i,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_i}, \overleftarrow{V})$. So $\exists\overleftarrow{\mathcal{X}_1}.R_{\mathrm{G}_1,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$, and $R_{\mathrm{G}_1,E,\overline{E}\wedge cases_1}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1 \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) = R_{\mathrm{G}_1,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$. Also, by Lemma 3.3 with $\mathsf{ff} \rightarrow \overline{E}$ we can thus conclude that $E(\overleftarrow{\mathcal{E}}) = \mathrm{G}_i(\overleftarrow{\mathcal{V}}) \Rightarrow \exists\overleftarrow{\mathcal{X}_i}.R_{\mathrm{G}_j,E,\mathsf{ff}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_i}, \overleftarrow{\mathcal{V}})$. But by construction, $cases_i(\overleftarrow{\mathcal{X}_s}) = \mathsf{ff}$ for all $i \neq 1$, and so $\overline{E} \wedge cases_i = \mathsf{ff}$. Finally, as for each recursive call a different base name for new variables, $names_i$, is used, it is guaranteed that $\mathcal{X}_i$ are pairwise disjoint. Thus, the assignments $\overleftarrow{\mathcal{X}_i}$ and $\overleftarrow{\mathcal{X}_s}$ can be combined without conflict:

$$\bigwedge_i E(\overleftarrow{\mathcal{E}}) = \mathrm{G}_i(\overleftarrow{\mathcal{V}}) \Rightarrow \exists\overleftarrow{\mathcal{X}_s}\exists_i\overleftarrow{\mathcal{X}_i}. \bigwedge_i R_{\mathrm{G}_i,E,\overline{E}\wedge cases_i}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_i \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}})$$

But $\bigwedge_i R_{\mathrm{G}_i,E,\overline{E}\wedge cases_i}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_i \dot\cup \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) = R_{\textsc{spec},E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$ with $\mathcal{X} = \mathcal{X}_s \dot\cup \bigcup_i \mathcal{X}_i$, so we have found an assignment.

(c i β) Case $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$

As $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ we can conclude that there exists a set $J \subseteq I$ such that $\mathrm{G}_j(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ for all $j \in J$. By construction of *cases*, we also know $\exists\overleftarrow{\mathcal{X}_s}.cases_j(\overleftarrow{\mathcal{X}_s}) = \mathsf{tt}$ for every $j \in J$. Without loss of generality assume $1 \in J$ and choose $\overleftarrow{\mathcal{X}_s}$ such that $cases_1(\overleftarrow{\mathcal{X}_s}) = \mathsf{tt}$. With the same reasoning as in (b i α), we can thus conclude that $\bigwedge_{j\in J} E(\overleftarrow{\mathcal{E}}) = \mathrm{G}_j(\overleftarrow{\mathcal{V}}) \Rightarrow \exists_{j\in J}\overleftarrow{\mathcal{X}_j}.R_{\mathrm{G}_j,E,\mathsf{ff}} \wedge R_{\mathrm{G}_1,E,\overline{E}}$. So it remains to show that $\exists_{i\in I}\overleftarrow{\mathcal{X}_i}.R_{\mathrm{G}_i,E,\mathsf{ff}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_i}, \overleftarrow{\mathcal{V}})$. But for the fixed $\overleftarrow{\mathcal{E}}$ we are examining, $E(\overleftarrow{\mathcal{E}}) = \mathsf{ff}$, and by Lemma 3.4 $R_{\mathrm{G}_i,\mathsf{ff},\mathsf{ff}}$ is a tautology and we can thus choose $\overleftarrow{\mathcal{X}_i}$ for $i \in I$ randomly receive an assignment $\overleftarrow{\mathcal{X}}$ as desired.

(c ii) Need to show: $\exists\overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) \Rightarrow E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}})$

First observe that because both $\overleftarrow{\mathcal{V}}$ and $\overleftarrow{\mathcal{E}}$ are fixed, we know whether $E(\overleftarrow{\mathcal{E}}) = \textsc{spec}(\overleftarrow{\mathcal{V}})$ holds or not. If it does, nothing needs to be shown. So it is sufficient to show that

$$E(\overleftarrow{\mathcal{E}}) \neq \textsc{spec}(\overleftarrow{\mathcal{V}}) \Rightarrow \nexists \overleftarrow{\mathcal{X}}.R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})$$

We show this by first reasoning there is no such assignment when $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{ff}$, and then when $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{tt}$. In both cases we examine the possible assignments to the variables, $\mathcal{X}_s$, introduced for the multiple-input AND.

(c ii α) $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{ff}$

From $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ we can conclude that $\textsc{g}_j(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ for all i, too. By induction hypothesis for all i it holds that $E(\overleftarrow{\mathcal{E}}) \neq \textsc{g}_i(\overleftarrow{\mathcal{V}}) \Rightarrow \nexists \overleftarrow{\mathcal{X}_i}.R_{\textsc{g}_i,E,\overline{E}}$. We can now choose $\overleftarrow{\mathcal{X}_s}$ such that $cases_1(\overleftarrow{\mathcal{X}_s}) = \mathsf{tt}$, and thus $R_{\textsc{g}_1,E,\overline{E} \wedge cases_1}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1 \dot{\cup} \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) = R_{\textsc{g}_1,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$. But if no assignment exists for this subrelation of $R[\mathcal{E}, \mathcal{X}, \mathcal{V}]$, then no assignment $\overleftarrow{\mathcal{X}}$ can exist that ensures $R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) = \mathsf{tt}$, as desired.

(c ii β) $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ and $E(\overleftarrow{\mathcal{E}}) = \mathsf{tt}$

From $\textsc{spec}(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$ we can conclude there exists an i such that $\textsc{g}_i(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$, without loss of generality $i = 1$. We can now choose $\overleftarrow{\mathcal{X}_s}$ such that $cases_1(\overleftarrow{\mathcal{X}_s}) = \mathsf{tt}$, and thus $R_{\textsc{g}_1,E,\overline{E} \wedge cases_1}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1 \dot{\cup} \mathcal{X}_s}, \overleftarrow{\mathcal{V}}) = R_{\textsc{g}_1,E,\overline{E}}(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}_1}, \overleftarrow{\mathcal{V}})$. But by induction hypothesis $E(\overleftarrow{\mathcal{E}}) \neq \textsc{g}_1(\overleftarrow{\mathcal{V}}) \Rightarrow \nexists \overleftarrow{\mathcal{X}_1}.R_{\textsc{g}_1,E,\overline{E}}$. Finally, if no assignment exists for this subrelation of $R[\mathcal{E}, \mathcal{X}, \mathcal{V}]$, then no assignment $\overleftarrow{\mathcal{X}}$ can exist that ensures $R(\overleftarrow{\mathcal{E}}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}}) = \mathsf{tt}$, as desired.

(d) $\textsc{spec} = \text{AND}_i \textsc{g}_i$, continued

Recall that in (c) we first assumed that $E \neq \mathsf{ff}$, and $cinps = \emptyset$. It remains to show that even when $E = \mathsf{ff}$, or when $cinps \neq \emptyset$ coverage is guaranteed, too.

(d i) $E = \mathsf{ff}$

The main difference in this case is visible in lines 23–24. Namely, in the recursive calls of auto_abstract we use the same base names, rather than unique ones. This has the consequence that in the subrelations computed by

$$\mathsf{auto\_abstract}(\mathcal{C}, \textsc{g}_i, H, L \wedge cases_i, names_i)$$

it may, and usually will, happen that indexing variables with the same names

are used: $\mathcal{X}_i \cap \mathcal{X}_j \neq \emptyset$. But recall that in the proof of (c) we depended on being able to choose assignments $\overleftarrow{\mathcal{X}_i}$ independently and then combine them to an assignment $\overleftarrow{\mathcal{X}}$. This is, in general, only possible with pairwise disjoint $\mathcal{X}_i$. In the specific case where $E = \mathsf{ff}$, though, we can argue that an assignment $\overleftarrow{\mathcal{X}}$ is still possible with a very similar reasoning. Recall that $cases_i(\overleftarrow{\mathcal{X}_s})$ is true for exactly one i, and false for all other $j \neq i$. So

$$\forall \overleftarrow{\mathcal{X}_s}.\ \bigwedge_j \mathsf{auto\_abstract}(\mathcal{C}, \mathrm{G}_j, \mathsf{ff}, \overline{E} \wedge cases_j, name) = \\ \mathsf{auto\_abstract}(\mathcal{C}, \mathrm{G}_i, \mathsf{ff}, \mathsf{tt}, name) \wedge \bigwedge_{j \neq i} \mathsf{auto\_abstract}(\mathcal{C}, \mathrm{G}_j, \mathsf{ff}, \mathsf{ff}, name)$$

But by Lemma 3.4 we know that $\bigwedge_{j \neq i} \mathsf{auto\_abstract}(\mathcal{C}, \mathrm{G}_j, \mathsf{ff}, \mathsf{ff}, name)$ is a tautology, and thus it does not matter which assignments $\overleftarrow{\mathcal{X}_j}$ are chosen. In particular, an assignment $\overleftarrow{\mathcal{X}_i}$ which includes assignments to variables in $\mathcal{X}_j$, delivers a partial assignment for the variables in $\mathcal{X}_j$, which still suits our cause. Thus, we can find an assignment $\overleftarrow{\mathcal{X}}$ by finding an assignment $\overleftarrow{\mathcal{X}_i \dot{\cup} \mathcal{X}_s}$ and the remaining assignments can be chosen randomly.

(d ii) $cinps \neq \emptyset$

First, note that *cinps* are all those G_i whose set of free variables is fully included in $\mathcal{C}$, the set of symbolic constant:

$$cinps = \{\mathrm{G}_i : \ \mathsf{free\_vars}(\mathrm{G}_i) \subseteq \mathcal{C}\}.$$

If at least one such G_i exists, two differences can be observed to the prior case. First, the subrelation $E \rightarrow c$, where $c = \wedge_{\mathrm{G}_i \in cinps} \mathrm{G}_i$ is added. This ensures that the *H*-component, which computes the guard that shall guarantee a low output for $\mathrm{G}_j \notin cinps$, is correct. And second, in the recursive calls $\mathsf{auto\_abstract}(\mathcal{C}, \mathrm{G}_j, E, \overline{E} \wedge cases_i \wedge c, names_j)$ that same c is conjoined on the *L*-component, which computes the guard that shall guarantee a low output to $\mathrm{G}_j \notin cinps$.

In the first case the subrelation described by the tuple (c, E, ff), which encodes $E \rightarrow c$, ensures that whenever the guard for the *H*-component of the recursive calls is satisfied, we also force the symbolic constants to a high value. This is required, so that the conjunction of all G_i – both those in *cinps* and those that are not in *cinps* – is still true:

$$\text{SPEC} = \bigwedge_{\mathrm{G}_i \in cinps} \mathrm{G}_i \wedge \bigwedge_{\mathrm{G}_j \notin cinps} \mathrm{G}_j = \mathsf{tt} \wedge \bigwedge_{\mathrm{G}_j \notin cinps} \mathrm{G}_j.$$

If any of the $\text{G}_i \in \mathit{cinps}$ evaluated to false, then the value of SPEC would also evaluate to false, which is something we need to prevent. This modification, on the other hand, right away shows that the proof from before, restricted to $\text{G}_j \notin \mathit{cinps}$ is sufficient with respect to the H-component.
For the L-component, we simply conjunct $c = \bigwedge_{\text{G}_i \in \mathit{cinps}} \text{G}_i$ onto $\overline{E} \wedge \mathit{cases}_j$. If $c(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$ this means our previous proof works exactly as before. If $c(\overleftarrow{\mathcal{V}}) = \mathsf{ff}$, on the other hand, we can treat the one case where $\mathit{cases}_i(\overleftarrow{\mathcal{X}_s}) = \mathsf{tt}$ holds just like those where $\mathit{cases}_j(\overleftarrow{\mathcal{X}_s}) = \mathsf{ff}$, i.e., it then does not need special handling anymore.
Thus, the proof seen in (c) is sufficient even when $\mathit{cinps} \neq \emptyset$. We merely assumed $\mathit{cinps} = \emptyset$ to split off some complexity and ease the reasoning in the already quite elaborate section of the proof.

□

Corollary 4.2. *Let c be a circuit with input nodes $\{n_i : i \in I\}$ and output node out. Let* SPEC *be a bexpr-tree, where $\mathcal{V} = \{v_i : i \in I\}$ is its set of free variables. Then the formal verification of c*

$$\models \bigwedge_{i \in I} n_i \;\mathsf{is}\; v_i \Rightarrow \mathit{out} \;\mathsf{is}\; \text{SPEC}[\mathcal{V}]$$

can be computed by

$$R[o, \mathcal{X}, \mathcal{V}] \models (\bigwedge_{i \in I} n_i \;\mathsf{is}\; v_i)^{R[o,\mathcal{X},\mathcal{V}]} \Rightarrow (\mathit{out} \;\mathsf{is}\; o)_{R[o,\mathcal{X},\mathcal{V}]}$$

where $R[o, \mathcal{X}, \mathcal{V}] = \mathsf{auto\_abstract}(\mathcal{C}, \text{SPEC}[\mathcal{V}], o, \overline{o}, \mathit{name})$ is the result of the algorithm in Figure 4.6, and the choice of name guarantees that $o \notin \mathcal{X}$.

Proof. This directly follows from Theorem 3.1, and Theorem 4.1 with $E = o$. □

4.7 Variable Reuse

In the proof of Theorem 4.1 we assumed that get_case_expressions, as seen in lines 9 and 22 of the advanced auto_abstract algorithm provided in Figure 4.6, introduces new indexing variables, and returns a set of mutually exclusive expressions that depend only on those variables. More precisely, the indexing variables are not reused thereafter, with one exception. Namely, the code in lines 24–25 cause some of the

recursive calls to use the same base name for new indexing variables. This leads to indexing variables being reused after analysing multiple-input AND-gates while the H-component equates to false. This reuse was proved correct in Theorem 4.1.

But indeed, this reuse of indexing variables is much more careful than necessary from a correctness point of view. Instead of introducing new indexing variables in most cases, we can maintain a table that captures when previously introduced indexing variables were used. When requiring new case expressions we can then check whether any of the old indexing variables have been used in the current setting. If not, then we can reuse the variables. Only if no reuse is possible do we need to introduce new indexing variables. Note that with this approach the base names passed into the recursive calls of auto_abstract are not necessary anymore and are ignored. This, as a consequence, also does not profit from the observations made concerning reuse in the multiple-input AND-gate whenever $H = \mathsf{ff}$. Even though this reuse of variables is thus eliminated, the general approach indeed potentially introduces that reuse, in addition to other reuse. This becomes evident once we have explained how to determine whether reusing indexing variables is safe or not.

It remains to argue in more detail when indexing variables have already been utilised, and when it is safe to reuse them. For this we need to recall how exactly indexing variables are leveraged in the auto_abstract algorithm. Importantly, we always conjunct $case_i$, which depends on indexing variables introduced or reused for that step, with some expression E, which depends on previously introduced indexing variables. For XNOR-gates, $case_i$ is conjoined with multiple expressions (as seen in line 11), which we can combine in a disjunction to capture all scenarios in which $case_i$ is used in a single expression E.

For each variable x_j that is included in the free variables of $case_i$ we can thus store the expression E. It expresses that for every assignment in which E holds the variable has already been used and must not be utilised again. Now assume that at a later point we require new case expressions. Further assume these are then conjoined with the expression F. If $E \wedge F = \mathsf{ff}$ irrespective of the assignment to the free variables of E and F, then it is safe to reuse any indexing variable, which has so far only been utilised whenever E holds. Thereafter, we need to update our list of when indexing variables have already been used by adding F, i.e., if x_j was previously used whenever E holds, it is now used whenever $E \vee F$ holds.

Example: Keeping Track of Usage of Indexing Variables
Suppose we run the auto_abstract algorithm and we need to introduce indexing vari-

ables for the first time. For example, assume we encounter an 8-input AND, and thus require eight case expressions. As we previously did not introduce any indexing variables yet, the table that keeps track of when indexing variables have already been used is still empty. Further assume we need to conjunct the cases with $L = \overline{b}$ (as seen in line 28 of Figure 4.6). Then we update the table as follows:

variable	already used when
x_1	$\overline{b}$
x_2	$\overline{b}$
x_3	$\overline{b}$

Next, suppose we require four case expression, which are used only when $a \wedge b$. By checking the table, we find that both x_1 and x_2 can be reused, as $\overline{b} \wedge (a \wedge b) = \mathsf{ff}$. We thus update the table as follows:

variable	already used when
x_1	$\overline{b} \vee (a \wedge b)$
x_2	$\overline{b} \vee (a \wedge b)$
x_3	$\overline{b}$

Finally, suppose we require four further case expressions, which are used only when b. We thus discover that x_1 and x_2 cannot be used, as $(\overline{b} \vee a \wedge b) \wedge b = a \wedge b \neq \mathsf{ff}$. On the other hand, x_3 can be reused, and so we only need to introduce one further indexing variable, x_4, to construct four cases. Thus, we receive the following table:

variable	already used when
x_1	$\overline{b} \vee (a \wedge b)$
x_2	$\overline{b} \vee (a \wedge b)$
x_3	tt
x_4	b

In particular, the indexing variable x_3 has now been utilised to the maximum. It cannot be reused, as the check $\mathsf{tt} \wedge E = \mathsf{ff}$ does not pass for any expression $E \neq \mathsf{ff}$. Thus, such entries can indeed be dropped from the table to reduce its size. ★

With this adjustment the correctness proof works just as before, with the same reasoning as provided for those cases where $case_i(\overleftarrow{\mathcal{X}_s}) = \mathsf{ff}$, or as seen in part (d) of the proof, which argues why the same base names can be reused whenever $H = \mathsf{ff}$ (as seen in lines 23–24 of the advanced auto_abstract algorithm provided in Figure 4.6).

Also observe that for XNOR-gates the case expressions are conjoined with both H and L. If H and L are each other's complement, then we have to use fresh indexing variables, and these are fully utilised, so that they cannot be reused for other cases. In general, this is not the case, but the indexing variables used for the case expressions are facilitated more strongly. While for AND-gates the variables are only used whenever L, for XNOR-gates, they are used whenever $H \vee L$.

Note that we could allow even more extreme reuse of variables by not just selecting variables to reuse, and then constructing mutually exclusive case expressions on those, but by storing even more detailed information. We could additionally store which case expressions were already used for specific indexing variables. In particular, when requiring n case expressions where n is not a power of two we could keep the variables unused for specific formulae. Namely, before we assumed that $\bigvee_i cases_i = \mathsf{tt}$, and with this more aggressive approach we would receive $\bigvee_i cases_i = U$, where U is an expression on the indexing variables used in $cases_i$. Then $\overline{U}$ exactly expresses when those variables have not been utilised yet, even if the expression E, with which $cases_i$ is conjoined, does not hold. While this approach maximises reuse, it also requires elaborate tracking of variable use. Furthermore, this extreme sharing of variables can lead to undesirably complex case expressions, which can in turn increase, rather than decrease computation costs when verifying circuits.

On a similar note, even the more modest sharing we first suggested can be disadvantageous. Two adjustments can help decrease this risk. First, if utilisation expressions get too complex, we can decide to drop the corresponding entries from the table, just as we suggested in the example when an indexing variable has already been used maximally. Second, ideally, we want to share variables especially then when the different cases exhibit a specific symmetry, as this reduces the cost of internal handling using BDDs. Thus, one can imagine adding some commentary to the list that captures when indexing variables are already used. A simple, yet potentially effective commentary is provided by the node names. For example, assume we are analysing an array of data with entries $d[i]$. Then this suggests there is some symmetry for the different values of i. A possible heuristic could be to store these node names, and when searching for indexing variables to reuse to prefer those which were already used for similar node names. Similarity could, for example, be defined as equality of node names after removing all numbers.

Finally, we could also go to the other extreme and decide to not reuse any indexing variables. This translates to using the auto_abstract algorithm as shown in Figure 4.6,

except that the *name* parameter is ignored. Thus, get_case_expressions always creates new indexing variables, even when the same parameters are passed into two different recursive runs of auto_abstract, as is done for multiple-input AND-gates whenever $H = \mathsf{ff}$.

In Chapter 5 we provide experimental results for verifying circuits using our advanced auto_abstract algorithm. There we show execution times for four reuse patterns: never reusing, as just suggested; reusing as shown in the auto_abstract algorithm, and finally the two variants based on mutual exclusivity – the moderate version first introduced in this section, and then the adapted version, which also uses the simple heuristic on which indexing variables to prefer by analysing similar node names. We observe that reusing variables aggressively and merely based on whether it still leads to correct verification results is generally bad, sometimes to the extreme of making verification infeasible again. This shows how fragile modifications to the sharing of variables is.

4.8 Reindexing Optimisations

The auto_abstract algorithm we suggest produces an abstraction relation, which is used in the STE run

$$R[o, \mathcal{X}, \mathcal{V}] \models (\bigwedge_i n_i \text{ is } v_i)^R \Rightarrow (out \text{ is } o)_R.$$

In joint work largely done by Magnus Björk, we developed optimisations for the preimage calculations. These can be applied due to the specific shape of the SIR relation computed in our approach. In particular, the relation's shape allows an early existential quantification to be used in the preimage computations. These improvements are based on the work described in [69], but do even better by exploiting the special form that our relations have by virtue of how they are generated:

$$R[o, \mathcal{X}, \mathcal{V}] = \bigwedge_i (H_i[o, \mathcal{X}] \rightarrow v_i) \wedge (L_i[o, \mathcal{X}] \rightarrow \overline{v_i})$$

In particular, to facilitate the optimisations we first separate the relation into two parts:

$$R[o, \mathcal{X}, \mathcal{V}] = S[o, \mathcal{X}] \wedge T[o, \mathcal{X}, \mathcal{V}]$$

such that $S[o, \mathcal{X}]$ does not depend on any of the variables in $\mathcal{V}$. Notably, all terms in $S[o, \mathcal{X}]$ are generated by line 3 of Figure 4.6, and thus this division of the relation can also be achieved directly when generating the relation, rather than after the fact. Additionally, $T[o, \mathcal{X}, \mathcal{V}]$ has the shape

$$\bigwedge_i (H_i[o, \mathcal{X}] \to v_i) \wedge (L_i[o, \mathcal{X}] \to \overline{v_i})$$

where H_i and L_i are independent of $\mathcal{V} = \{v_i : \ i \in I\}$. We can then further divide the SIR relation by defining

$$T_i[o, \mathcal{X}, v_i] = (H_i[o, \mathcal{X}] \to v_i) \wedge (L_i[o, \mathcal{X}] \to \overline{v_i}),$$

and thus $T[o, \mathcal{X}, \mathcal{V}] = \bigwedge_i T_i[o, \mathcal{X}, v_i]$. Next, given a guard P, $R \downarrow P$ is defined as the part of R that mentions the target variables $\mathcal{V}$ in P:

$$R \downarrow P = \bigwedge_{v_i \in \mathsf{free\_vars}(P)} T_i[o, \mathcal{X}, v_i]$$

Then, using the definition of the weak preimage, $P_R[\mathcal{X}] = \exists \overleftarrow{\mathcal{V}}.R(\overleftarrow{\mathcal{V}})[\mathcal{X}] \wedge P(\overleftarrow{\mathcal{V}})$ (see page 57), and with $\mathcal{P}_1 = \mathsf{free\_vars}(P) \cap \mathcal{V}$ and $\mathcal{P}_2 = \Big(\mathsf{free\_vars}(P) \cup \mathcal{V}\Big) \backslash \mathcal{P}_1$

$$P_{R\downarrow P}[\mathcal{P}_2] = \exists \overleftarrow{\mathcal{P}_1}.R \downarrow P(\overleftarrow{\mathcal{P}_1}) \wedge P(\overleftarrow{\mathcal{P}_1}).$$

Having introduced this notation, Björk proved the following optimisations can be applied, where $D = S \wedge \bigwedge_i \overline{H_i \wedge L_i}$:

$$P_R = \begin{cases} D \wedge P & \text{if } \mathsf{free\_vars}(P) \cap \mathcal{V} = \emptyset \\ D \wedge \overline{L_i} & \text{if } P = v_i \\ D \wedge \overline{H_i} & \text{if } P = \overline{v_i} \\ D \wedge P_{R\downarrow P} & \text{otherwise} \end{cases}$$

and

$$P^R = D \wedge \overline{\overline{P}_R}.$$

As D does not depend on P, it can be precomputed when the relation R is created and used for all guards in the trajectory assertion to be transformed. By building hash

tables that map v_i to H_i and L_i, the two middle cases can be computed very quickly. The fourth cases, seen only rarely in practise, can be optimised using quantification scheduling.

Finally, factoring in that the SIR relations we compute provably satisfy the coverage condition

$$\forall \overleftarrow{\mathcal{V}}.\forall \overleftarrow{o}.\Big(\overleftarrow{o} = \{o \mapsto \textsc{spec}(\overleftarrow{\mathcal{V}})\} \Leftrightarrow \exists \overleftarrow{\mathcal{X}}.R(\overleftarrow{o}, \overleftarrow{\mathcal{X}}, \overleftarrow{\mathcal{V}})\Big)$$

we can conclude that $D(\overleftarrow{o}, \overleftarrow{\mathcal{X}}) = \mathsf{tt}$ in all cases, and can thus be safely removed from all formulae. This significantly reduces the complexity of the preimage operations.

A full proof for the correctness of this optimisation, and further details are given in our joint paper with Björk [1].

4.9 Summary

In this chapter we saw several improvements and extensions to the basic automatic abstraction algorithm introduced in Chapter 3. We extended the types of specifications we could work on by allowing directed acyclic graphs, rather than just trees, by allowing further constructors, and by offering a solution for designs with multiple outputs. We also improved the auto_abstract algorithm by making it recognise more elaborate constructs in the specification, namely constructs equivalent to XNOR-gates, and multiple-input AND-gates, so that their indexing cases can be encoded more efficiently. Finally, we discussed different options for reusing indexing variables when constructing the SIR relation through our automatic abstraction algorithm, as well as possible optimisations in computing the strong and weak preimage required in the STE run $R \models A^R \Rightarrow C_R$ using a partitioned version of the SIR relation our algorithm produces.

While some of these changes simply make our work applicable to more designs, the majority are concerned with improving the efficiency of our approach throughout. These improvements turn our prototype, as presented in Chapter 3, into a powerful verification tool, which can be used to show formal correctness of realistically-sized circuits. This we demonstrate in Chapter 5, where we verify three designs, a content-addressable memory, a memory, and a scheduler, of increasing sizes.

Chapter 5

Experimental Results for Abstraction Discovery

In this chapter we verify three example circuits, namely a content-addressable memory, a memory, and a scheduler, to provide experimental assessment of the effectiveness of our algorithm for calculating automatic abstraction schemes. Each example is presented in three main parts. First, we describe the hardware model and how we generate it. Second, we determine the SIR relation produced by running the advanced auto_abstract algorithm and discuss the symbolic indexing it encodes. Finally, we run STE using that relation to show each design meets its specification. We provide execution times for each phase, and different variants of reusing indexing variables. The execution times provided where collected on a 2GB memory machine with an Intel® Core 2 Duo CPU with 3.06GHz.

We additionally vary the sizes of all three designs to show how well our method scales. The results show that we can successfully verify circuits of industrial size, thus addressing the state explosion problem.

5.1 Content-Addressable Memory

Our first example is a content-addressable memory (CAM). Pandey et al. pioneered the verification of a CAM using symbolic indexing in [24], and it has evolved to a classic example since. As illustrated in Figure 5.1, the simple CAM we verified takes a *key* as an input, and returns a bit that indicates whether the key is contained in its internal memory. More complex CAMs return a list of addresses that store the *key*. In our example we simply return a Boolean value, *hit*, indicating whether the *key* was found or not.

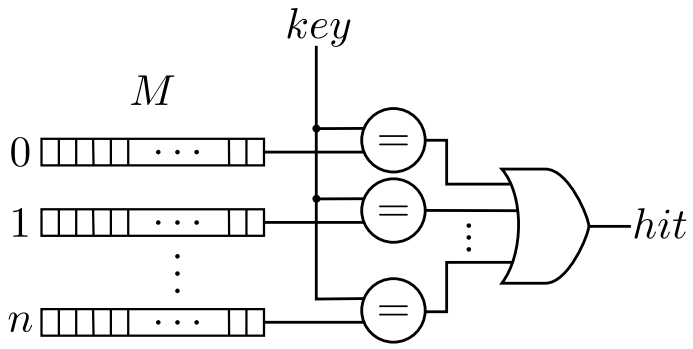


Figure 5.1: High-level model of the content-addressable memory verified

Additionally, you can write to the internal memory of the CAM by providing a write address and the data to be stored to that location. However, for this experiment we verify only the correctness of the *hit* signal, and thus omit the write behaviour from Figure 5.1 to avoid clutter.

5.1.1 Hardware Model

Figure 5.2 shows the FL-code, the functional language used in the Forte verification environment, that we used to generate the design of CAMs in varying sizes. It defines all input and outputs, as well as a circuit's gates, each given by defining truth tables. In our figures we omit the truth tables, instead using descriptive names, such as mk_mux for a multiplexer, or mk_re_ff for a rising edge flip-flop.

In lines 5–8 the signal vectors are defined. The *key* signal is the input for which we want to determine whether it is stored in one of the entries. The *din* signal is only used when writing data to one of the entries. It then provides which data to write to an entry. The data is written to the entry with the address provided in the *waddr* signal. The write signal, *we*, consists of a single bit, and thus does not have to be listed explicitly in these lines. It is used in the subsequent code, though. Finally, *lhit* and *hits* are auxiliary vectors. The vector *lhit* stores in its j^{th} component whether the j^{th} bit of an entry matched with the j^{th} bit of the key. Similarly, *hits* stores in its i^{th} component whether the i^{th} CAM entry matches with the key.

Lines 14–20 generate a single CAM cell, which is essentially the heart of the CAM. Provided the write signal is high, it writes a single bit of data into the memory entry (lines 16–17). It then compares a single bit of the key with a single bit of the stored entry. The definition of a CAM cell is followed by that of a full CAM line (lines 22–27), which combines several CAM cells to receive the conjunction of each

```
1  let mk_CAM entries word_size =
2    let es = entries-1 in
3    let ks = word_size-1 in
4    let name = sprintf "CAM_%d_entries_%d_bits" (entries,word_size) in
5    // Short hand for signals
6    let key = sprintf "key[%d:0]" ks in
7    let din = sprintf "din[%d:0]" ks in
8    let waddr = sprintf "waddr[%d:0]" es in
9    let lhit = sprintf "lhit[%d:0]" ks in
10   let hits = ev (sprintf "hits[%d:0]" es) in
11   // Real body
12   mk_module name ["we", "clk", key, din, waddr] ["hit"] (
13     // CAM cell
14     mk_module "cam_cell" ["clk", "write", "ki", "di"] ["lhi"] (
15       // Multiplexer in front of rising edge flip-flop
16       mk_mux "write" "di" "oi" "ti" #
17       mk_re_ff "clk" "ti" "oi" #
18       // Comparator
19       mk_xnor2 ["oi","ki"] "lhi"
20     ) #
21     // Line of CAM cells
22     mk_module "cam_line" ["clk", "we", "wi", key, din] ["hit"] (
23       mk_and ["we", "wi"] "write" #
24       (forall [(i,ki,di,lhi) | zip4 (ks downto 0)
25         (ev key)
26         (ev din)
27         (ev lhit)].
28       mk_instance "cam_cell" (sprintf "cc%d" i)
29         ["clk", "write", ki, di] [lhi]
30       ) #
31       mk_and (ev lhit) "hit"
32     ) #
33     //
34     (forall [(i,wi,hi) | zip3 (es downto 0) (ev waddr) hits ].
35       mk_instance "cam_line" (sprintf "cl%d" i)
36         ["clk","we",wi,key,din] [hi]
37     ) #
38     mk_or hits "hit"
39   )
40 ;
```

Figure 5.2: The code used to generate a CAM

of the CAM cell's outputs. Thus, a CAM line returns whether the *key* is stored in that memory location. Finally, the disjunction of the output of several CAM lines (lines 34–38) delivers the *hit* signal we are interested in.

```
38        // Simulated error correction
39        (mk_constant 1 (ev one)) #
40        (mk_zero_extend (ev key) (ev ext_key)) #
41        (mk_adder (ev ext_key) (ev one) (ev keyp1)) #
42        (mk_subtractor (ev keyp1) (ev one) (ev keyp1m1)) #
43        (mk_truncator (ev keyp1m1) (ev key')) #
44        (forall [(i,wi,hi) | zip3 (es downto 0) (ev waddr) hits ].
45          mk_instance "cam_line" (sprintf "cl%d" i)
46            ["clk","we",wi,key',din] [hi]
47        ) #
48        mk_or hits "hit"
49      )
50  ;
```

Figure 5.3: Adjusted CAM generation code to simulate error correction, lines 1–37 are as in Figure 5.2

This hardware model is an almost one-to-one translation of the specification we introduce in the next part. However, in real designs often some more complex calculations are executed for speedup reasons, reducing the physical size of a design, or for increasing robustness (e.g. fault tolerance). We hence augment the model as shown in Figure 5.3. This code first adds one to the *key* and then subtracts it again. It simulates a possible error-correction, in which the key and entries would have redundant bits that are used for a checksum. The computation we added of course does not implement such a functionality, but merely ensures that the CAM includes some more elaborate calculations, which simulate the kinds of computations that could be observed in actual industry designs.

5.1.2 SIR Relation

Our method of automatically computing indexing schemes works on the specification of the model to be verified. The input-output behaviour of a CAM provides a very obvious specification function for the CAM. The *hit* signal, whose value is represented

by o, is high exactly if at least one of the entries coincides with the key provided:

$$o = \bigvee_i (M[i] = key)$$

So the output is the disjunction of equalities, one per entry of the CAM's internal memory.

We give this specification to our automatic abstraction algorithm, additionally declaring the *key* inputs to be symbolic constants. We declare the bits of the key as symbolic constants because we perform an arithmetic operation, namely adding and subtracting one, on the key. As mentioned in Section 4.3, such inputs are good candidates for symbolic constants. Later, in Chapter 8 we demonstrate how these inputs can also be automatically detected as good candidates for symbolic constants.

Before we describe in detail which indexing is computed, recall that our auto_abstract algorithm computes tuples of values of the form (n_i, H_i, L_i). These are short-hand for the actual relation $R = \bigwedge_i (H_i \to v_i) \wedge (L_i \to \overline{v_i})$. The first component of the tuple signifies which node we are driving, the second when to drive it with a high value, and the third when to drive it with a low value. Here node n_i is an input of the hardware model, which is represented by v_i in the circuit's specification.

For a model that has *entries* number of entries, and where each entry has *bits* number of bits, running

$$\mathsf{auto\_abstract}(\{key[j] : j = 0..(bits-1)\}, \bigvee_{i=0..(entries-1)} (M[i] = key), o, \overline{o}, \text{“”})$$

generates tuples of the shape

$$\begin{aligned}(M[i][j], \quad & key[j] \wedge entry_i \wedge o \vee \overline{key[j]} \wedge bit_{ij} \wedge \overline{o}, \\ & \overline{key[j]} \wedge entry_i \wedge o \vee key[j] \wedge bit_{ij} \wedge \overline{o}).\end{aligned}$$

Here, $entry_i$ is an expression that requires $\log_2 entries$ variables and encodes which CAM entry matches with the key. The expressions bit_{ij} require $\log_2 bits$ variables for each entry of the CAM and specify which bit of the i^{th} CAM entry does not match with the key. This means in total $entries \cdot \log_2 bits$ variables are used when verifying that the output is low whenever none of the entries match with the key. In total, $\log_2 entries + entries \cdot \log_2 bits$ indexing variables are introduced.

Note that the variable o encodes whether we are driving values with the goal of verifying a high or low output. When expecting a high output the SIR relation drives

exactly one CAM entry with the same values as the bits of the key – it is the entry that ensures that the *hit* signal is high. When expecting a low output, on the other hand, it drives exactly one bit of each CAM entry with the opposite value of the key in that bit. To clarify, we give the full set of tuples for $entries = 2$ and $bits = 2$.

Example: SIR relation for a 2×2 CAM

$$\begin{array}{llll}
(M[0][0], & key[0] \wedge a \wedge o \vee \overline{key[0]} \wedge b \wedge \overline{o}, & \overline{key[0]} \wedge a \wedge o \vee key[0] \wedge b \wedge \overline{o}) \\
(M[0][1], & key[1] \wedge a \wedge o \vee \overline{key[1]} \wedge \overline{b} \wedge \overline{o}, & \overline{key[1]} \wedge a \wedge o \vee key[1] \wedge \overline{b} \wedge \overline{o}) \\
(M[1][0], & key[0] \wedge \overline{a} \wedge o \vee \overline{key[0]} \wedge c \wedge \overline{o}, & \overline{key[0]} \wedge \overline{a} \wedge o \vee key[0] \wedge c \wedge \overline{o}) \\
(M[1][1], & key[1] \wedge \overline{a} \wedge o \vee \overline{key[1]} \wedge \overline{c} \wedge \overline{o}, & \overline{key[1]} \wedge \overline{a} \wedge o \vee key[1] \wedge \overline{c} \wedge \overline{o})
\end{array}$$

In this concrete example only one variable, a, is required for the expressions $entry_i$. This variable is relevant only when o evaluates to true, i.e., when verifying the hit case. For the miss case, two variables are required, the variable b for selecting the bit that does not match in the first CAM entry, and the variable c for selecting the bit that does not match in the second CAM entry. These need to be different variables, so that each bit can be selected independently.

The variables $key[0]$ and $key[1]$ are symbolic constants. They encode which value the i^{th} bit of the key has in each run. Hence, if $key[0]$ is used in the H component of a $(M[i][0], H, L)$ tuple, this means the memory cell has the same value as the key in the first bit. On the other hand, if $key[0]$ is used in the L component of that same tuple, this means that the memory cell has the opposite value as the key in the first bit: while the key bit is high, it leads to a low bit in the memory. Accordingly, $\overline{key[0]}$ signifies differing values in the H-component, and matching values in the L-component. ★

It is noteworthy that the indexing computed is the same one as the manually devised abstraction scheme proposed in [24]. When verifying the CAM's behaviour when it contains the key, we first decide which CAM entry is the matching one, and the drive it with the same values as the key. Thus, if the model is correct, a high *hit* signal should be observed. When verifying the CAM's behaviour when it does not contain the key, on the other hand, for each CAM entry we drive one bit with exactly the opposite value as that the key has in that bit. Thus, a correct model should yield a low output.

So we automatically compute an indexing for the CAM that previously had to be developed with careful reasoning. Pandey et al.'s previous result was significant, because this design could not be verified for larger sizes without using symbolic indexing. For BDD-based verification memory was the bottleneck, for SAT-based verification it

was time. In particular, we tried to verify the CAM presented here without symbolic indexing, which could only be successfully completed for the CAM with 64 entries, each having four bits (in 20.59 seconds). Increasing the number of bits to six was already too costly to complete with our machine.

Finally, it is worth mentioning that if we do not declare any symbolic constants when computing the abstraction scheme we obtain a much finer-grained symbolic indexing scheme. For example, one case included in this finer indexing family covers the case in which bit j of the key is different from bit j in every entry in the CAM. This, of course, is sufficient information to decide that the *hit* signal should be low – from a purely theoretical point of view. But this extremely detailed indexing is not necessarily desirable. In particular, it failed on our augmented design due to over-abstraction. Rather than observing a low output, we obtained an X-value. In Chapters 6 and 7 we address the problem of over-abstraction by suggesting automatic refinement strategies for symbolic indexing schemes. At this point, however, we simply stated the *key* bits to be symbolic constants.

5.1.3 Observed Execution Times

Now that we have described both the model to verify, as well as an SIR relation to speed up the process, we can run STE to determine whether the CAM meets its specification. Figure 5.4 shows the execution times we collected for CAMs of increasing sizes, all of which passed the verification. In particular, we timed the three different phases of the verification individually. First, we created the hardware model M with the code provided in Figure 5.3 by executing mk_CAM *entries bits*. Then we computed our SIR relation R on the specification $hit = \bigvee_i (M[i] = key)$ while declaring all of the bits of the key as symbolic constants:

$$\mathsf{auto\_abstract}(\{key[i] : i = 0..(bits-1)\}, \bigvee_{i=0..entries-1} (M[i] = key), o, \overline{o}, \text{“”}).$$

Finally we verified that M meets the CAM specification by running STE with the previously computed SIR relation: $R \models_M (\bigwedge_i n_i \text{ is } v_i)^R \Rightarrow (hit \text{ is } o)_R$. Recall that the relation injects the specification into the auxiliary run, which simply drives the input nodes and the output with variables.

We varied the key size, i.e., the number of bits the key (and thus the entries) have, as well as the number of entries the CAM holds. Both of these parameters we increased exponentially. Factoring this in, the results indicate that running time

grows linearly, or close to linearly, with the size of the CAM. This observation is supported by our analysis of the SIR relation generated. In particular, [24] used the same indexing scheme and also observed linear growth in execution times when increasing the number of entries, as well as the width of the entries.

entries	×	*bits*	circuit	auto_abstract	STE
64	×	4	0.01s	0.11s	0.03s
64	×	8	0.01s	0.21s	0.09s
64	×	16	0.01s	0.43s	0.20s
64	×	32	0.02s	0.88s	0.45s
64	×	64	0.04s	1.77s	1.20s
256	×	4	0.03s	0.43s	0.15s
256	×	8	0.04s	0.86s	0.40s
256	×	16	0.05s	1.71s	0.95s
256	×	32	0.07s	3.61s	2.24s
256	×	64	0.15s	7.51s	5.06s
1024	×	4	0.28s	1.82s	0.95s
1024	×	8	0.32s	3.73s	2.17s
1024	×	16	0.38s	7.58s	4.27s

Figure 5.4: Execution times of verifying a CAM with varying number of entries (64, 256, 1024) and key sizes (4, 8, 16, 32, 64). Execution times were split into time needed to build the circuit, of running our auto_abstract algorithm, and of running STE in the Forte verification environment.

The execution times provided in Figure 5.4 – as all execution times given in this dissertation – were collected on a virtual machine running Fedora 8 with an Intel® Core 2 Duo CPU with 3.06GHz and 2GB of memory. Each run was executed on a freshly opened verification environment to eliminate any side effects previous calculations may have had on the internal storage. Each run was repeated ten times, and the minimum taken to minimise the noise caused by possible background tasks.

Finally, we could not verify larger CAMs, because the open-access version of Intel®'s Forte verification environment supports at most 32768 BDD variables only, a restriction their internal version does not have. For the 1024×32 CAM this limit was reached. This does not represent a limit of our approach, but merely that of the tool we used to run STE.

Variants of Reusing Indexing Variables

We also verified the CAM using three of the other proposed methods of reusing indexing variables. The execution times provided in Figure 5.4, and also Figure

entries	×	*bits*	(1)	(2)	(3)	(4)	(5)
64	×	4	20.59s	0.10s	0.06s	0.09s	0.08s
64	×	8		0.15s	0.13s	0.15s	0.17s
64	×	16		0.37s	0.31s	0.40s	0.33s
64	×	32		0.70s	0.58s	0.62s	0.72s
64	×	64		1.59s	1.40s	1.80s	1.78s
256	×	4		0.34s	0.33s	0.35s	0.38s
256	×	8		0.68s	0.74s	0.80s	0.74s
256	×	16		1.71s	1.31s	1.71s	1.62s
256	×	32		2.87s	3.25s	3.40s	3.08s
256	×	64		6.93s	6.22s	7.13s	7.35s
1024	×	4		1.56s	1.54s	1.79s	1.44s
1024	×	8		3.37s	3.14s	3.23s	4.07s
1024	×	16		6.35s	6.54s	7.64s	7.30s

Figure 5.5: Execution times observed for verifying the CAM using different methods: (1) without symbolic indexing, (2) no reuse of indexing variables, (3) main reuse variant, same results as seen earlier in this section, (4) reuse based on mutual exclusivity while preferring variables used for similar node names, and (5) reuse based on mutual exclusivity only.

5.5 (3), are those collected when reusing indexing variables only when we encounter multiple-input AND-gates. In particular, variables are only reused if $H = \mathsf{ff}$ (lines 23–24 of Figure 4.6). The execution times provided in Figure 5.5 additionally provide data on the following variants. First, for variant (2), we do not ever reuse indexing variables, i.e., even not when the algorithm encounters a multiple-input AND-gate while $H = \mathsf{ff}$. Thus, the *name* parameter is unused. Variants (4) and (5) also ignore the *name* parameter and thus also treat the case $H = \mathsf{ff}$ identically to $H \neq \mathsf{ff}$. However, indexing variables are reused more aggressively than in (2) and (3). The variants reuse indexing variables whenever we are safe to do so based on mutual exclusivity, as described in more detail in Section 4.7. Variant (4) additionally tries to reuse those indexing variables, which have likely been used in a symmetrical setting before by storing and comparing the node names for which the variables were previously used. Details on this variant are also provided in Section 4.7.

All variants lead to very similar execution times, a clear evaluation of which variant is superior is difficult. This is especially true, as the verification of even the largest CAMs required just six to eight seconds. Still, our main variant, as well as not reusing variables at all showed a better performance than the two variants that reuse variables based on the mutual exclusivity condition.

In Figure 5.5 we did not split execution times into the time needed for computing the SIR relation, and for running STE anymore. All of the variants behaved similarly with respect to what proportion was used for running auto_abstract. More importantly, the combined execution times all seem to increase at approximately the same rate, thus suggesting they scale similarly. To put this into perspective, we also added the execution time for verifying the CAM without symbolic indexing, which could only be completed for the smallest CAM in our set.

5.2 Memory

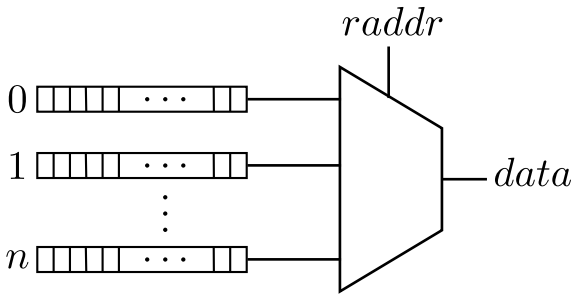


Figure 5.6: High-level model of the memory verified

In our second example we verify the correctness of the read operation for a memory. In 1997 Pandey et al. first published verification successes by Symbolic Trajectory Evaluation using symbolic indexing [25], and memories have – just like CAMs – evolved to classic examples for demonstrating the effectiveness of verification by Symbolic Trajectory Evaluation.

The memory we verified is abstractly shown in Figure 5.6. Given an address to read from, the memory outputs the data stored in the addressed location. Again, the write functionality is not included in the figure, as we are not verifying that functionality of the memory.

5.2.1 Hardware Model

Figure 5.7 shows the FL-code that generates the memory we verified in varying sizes. In lines 5–13 the signal vectors are defined. The *raddr* input delivers the address that we want to read from. The *din* signal signal is only used when writing data to one of the locations, then providing the data to write. In that case the data is written to the entry with the address provided in the *waddr* signal. The write signal, *we*,

```
1  let mk_memory entries word_size =
2    let as = (log2 entries) in
3    let name = sprintf "mem_%d_entries_%d_bits" (entries,word_size) in
4    // Short hand for signals
5    let raddr = sprintf "raddr[%d:0]" (as-1) in
6    let din = sprintf "din[%d:0]" (word_size-1) in
7    let waddr = sprintf "waddr[%d:0]" (as-1) in
8    let dout = sprintf "dout[%d:0]" (word_size-1) in
9    let rdecodes = sprintf "rdecode[%d:0]" (entries-1) in
10   let tmps = sprintf "tmp[%d:0]" (entries-1) in
11   let wdecodes = sprintf "wdecode[%d:0]" (entries-1) in
12   let Mcells = sprintf "M[%d:0][%d:0]" (entries-1, word_size-1) in
13   let rc = sprintf "rc[%d:0][%d:0]" (entries-1, word_size-1) in
14   // Real body
15   mk_module name ["clk", "we", raddr, din, waddr] [dout] (
16     // Decoder
17     mk_module "rdecoder" [raddr] [rdecodes] (
18       (forall [(i,rd) | zip (entries-1 downto 0) (ev rdecodes)].
19         let cn = sprintf "c%d[%d:0]" (i,as-1) in
20         mk_constant i (ev cn) #
21         mk_eq (ev cn) (ev raddr) rd
22       )
23     ) #
24     // Enabled (write) decoder
25     mk_module "wdecoder" [waddr, "we"] [rdecodes] (
26       (forall [(i,tmp,rd) | zip3 (entries-1 downto 0) (ev tmps) (ev rdecodes)].
27         let cn = sprintf "c%d[%d:0]" (i,as-1) in
28         mk_constant i (ev cn) #
29         mk_eq (ev cn) (ev waddr) tmp #
30         mk_and ["we",tmp] rd
31       )
32     ) #
33     // Memory array
34     mk_module "mem_array" [wdecodes,din] [Mcells] (
35       (forall [("we",mems) | zip (ev wdecodes)
36                (cluster word_size (ev Mcells))].
37         (forall [(di,mi) | zip (ev din) mems].
38           mk_ah_latch "we" di mi
39         )
40       )
41     ) #
```

(continued on next page)

```
42      // Read logic
43      mk_module "read" [rdecodes,Mcells] [dout] (
44        (forall [(mcol,rcs,do) |
45              zip3
46                (transpose_list (cluster word_size (ev Mcells)))
47                (transpose_list (cluster word_size (ev rc)))
48                (ev dout)].
49           (forall [(rd,mi,rci) | zip3 (ev rdecodes) mcol rcs].
50             mk_and [rd,mi] rci
51           ) #
52           mk_or rcs do
53        )
54      ) #
55      (mk_instance "rdecoder" "rd" [raddr] [rdecodes]) #
56      (mk_instance "wdecoder" "wd" [waddr,"we"] [wdecodes]) #
57      (mk_instance "mem_array" "ma" [wdecodes,din] [Mcells]) #
58      (mk_instance "read" "read" [rdecodes,Mcells] [dout])
59    )
60  ;
```

Figure 5.7: The code used to generate a memory

consists of a single bit, and thus does not have to be listed explicitly in these lines. It is used in the subsequent code, though. *Mcells* hold the data stored in the memory. Next, *rdecodes*, *tmps*, *wdecodes*, and *rc* are auxiliary vectors. The vectors *rdecodes* and *wdecodes* hold the location after decoding the read address and write address respectively. The vectors *tmps* and the matrix *rc* are used when writing and reading data respectively. Finally, *dout* stores the final output of the memory.

Lines 17–22 generate a decoder, which translates the read address provided to an actual memory location. Accordingly, lines 25–31 generate a write-enabled decoder. It works in a way similar to the read decoder, but additionally requires the write signal, *we*, to be high. The heart of the memory, its data array, is generated with lines 34–40. It uses the auxiliary *wdecodes* to write *din* to the determined location provided the write mode is enabled. Finally, lines 43–53 add the read functionality, outputting the read data to *dout*. Here the auxiliary *rdecodes* is utilised. In essence, reading works as follows. In the auxiliary matrix *rc* we store the data found in *Mcells* conjoined with the provided read address. Thus, we receive a matrix that has non-zero entries in only the line that holds the desired data. The memory then outputs a disjunction of all the data stored in *rc*, which equates to the requested data due

to the step before. Lines 55–59 bring all previously defined modules together, thus creating the memory in full.

5.2.2 SIR Relation

We use the following specification for verifying that the read operation of the memory is formally correct. The j^{th} bit of the output data *dout* is high exactly if the corresponding memory entry's j^{th} bit is high. More formally, we can take the disjunction of all read addresses, each conjoined with their corresponding memory location value:

$$dout[j] = \bigvee_i raddr_i \wedge M[i][j]$$

Here $raddr_i$ denotes the i^{th} memory location, which can be computed using the value of bits of the read address vector $raddr$. This mechanism is similar to the one already used in the code that generates the hardware model, although the hardware model includes further components, such as the decoder and precharge logic, to implement the read operation.

When computing the SIR relation we again specify some symbolic constants: the bits of the read address shall always be driven with a value. This makes sense, as we in general need to know the address in full to output the correct data.

The abstraction scheme automatically computed then enumerates various cases, one for each read address. For this $\log_2 entries$ variables are required, essentially one for each bit of the read address. These variables encode different cases, $case_i$, which relate to the different read addresses. The relation thus has two entries for each possible read address. First, we connect $case_i$ with a specific read address. And second, the information which values to drive the memory nodes with is provided.

The SIR relation therefore has the following shape:

$$\begin{array}{lll} (raddr_i, & case_i, & \mathsf{ff}) \\ (M[i][j], & case_i \wedge dout[j], & case_i \wedge \overline{dout[j]}) \end{array}$$

where i runs through the indices of the different memory locations, and j corresponds to the bits of each data entry. As before, recall that the tuples (e_i, H_i, L_i) represent the SIR relation $\bigwedge_i H_i \rightarrow e_i \wedge L_i \rightarrow \overline{e_i}$. Rather than stating a primary input as e_i, here in some cases an expression on the bits of the read address is mentioned. This is the component of the relation that connects the cases with the corresponding addresses. A full example of the SIR relation computed for a memory with four entries, each

having two bits of data, highlights this further.

Example: SIR relation for a 4×2 memory

$$
\begin{array}{lll}
(\overline{raddr[0]} \wedge \overline{raddr[1]}, & \overline{a} \wedge b, & \mathsf{ff}) \\
(M[0][0], & \overline{a} \wedge b \wedge dout[0], & \overline{a} \wedge b \wedge \overline{dout[0]}) \\
(M[0][1], & \overline{a} \wedge b \wedge dout[1], & \overline{a} \wedge b \wedge \overline{dout[1]}) \\
(\overline{raddr[0]} \wedge raddr[1], & a \wedge \overline{b}, & \mathsf{ff}) \\
(M[1][0], & a \wedge \overline{b} \wedge dout[0], & a \wedge \overline{b} \wedge \overline{dout[0]}), \\
(M[1][1], & a \wedge \overline{b} \wedge dout[1], & a \wedge \overline{b} \wedge \overline{dout[1]}) \\
(raddr[0] \wedge \overline{raddr[1]}, & \overline{a} \wedge \overline{b}, & \mathsf{ff}) \\
(M[2][0], & \overline{a} \wedge \overline{b} \wedge dout[0], & \overline{a} \wedge \overline{b} \wedge \overline{dout[0]}) \\
(M[2][1], & \overline{a} \wedge \overline{b} \wedge dout[1], & \overline{a} \wedge \overline{b} \wedge \overline{dout[1]}) \\
(raddr[0] \wedge raddr[1], & a \wedge b, & \mathsf{ff}) \\
(M[3][0], & a \wedge b \wedge dout[0], & a \wedge b \wedge \overline{dout[0]}) \\
(M[3][1], & a \wedge b \wedge dout[1], & a \wedge b \wedge \overline{dout[1]})
\end{array}
$$

Here we can observe that the two indexing variables a and b are used to enumerate the four addresses encoded by the inputs $raddr[0]$ and $raddr[1]$. Note that different encodings are possible, including the encoding where a would behave exactly like $raddr[0]$ and b exactly like $raddr[1]$. However, running the algorithm produced this encoding, which works just as well.

It is now also more clearly visible that in some cases an expression on symbolic constants fills the first component of (e_i, H_i, L_i) tuples. In particular, these are auxiliary tuples, which signify which case encoded with a and b corresponds to a specific read address. The L_i always equates false in these tuples, as one implication is sufficient, $H_i \rightarrow e_i$, or more specifically, $case_i \rightarrow address_i$.

The remaining SIR relation tuples establish the desired input-output behaviour. If the read address encodes the memory location i, and $dout[j]$ is high, then drive the memory location $M[i][j]$ with a high value. If, on the other hand, $dout[j]$ is low, then also drive the memory location $M[i][j]$ with a low value. The $case_i$ expressions ensure that only the relevant memory locations are driven with a value, rather than all. This suggests that execution times grow linearly with the address width and word size, a very desirable result. ★

5.2.3 Observed Execution Times

Figure 5.8 shows the execution time behaviour of the STE verification we measured. In particular, the automatically computed symbolic indexing allowed us to verify memories of relatively large size, which is not possible without symbolic indexing. We could have easily verified even larger memory designs, however the 1024×32 memory and the 4048×8 memory reached the limit Intel® introduced for BDD variables in the open-access version of their Forte verification environment.

entries	×	*bits*	circuit	auto_abstract	STE
64	×	4	0.12s	0.10s	0.04s
64	×	8	0.19s	0.21s	0.09s
64	×	16	0.22s	0.43s	0.25s
64	×	32	0.28s	0.90s	0.91s
64	×	64	0.36s	1.76s	3.52s
256	×	4	1.02s	0.48s	0.19s
256	×	8	1.09s	0.93s	0.44s
256	×	16	1.22s	1.83s	1.11s
256	×	32	1.66s	3.85s	3.78s
256	×	64	2.34s	7.65s	15.22s
1024	×	4	5.76s	2.06s	1.14s
1024	×	8	6.88s	4.15s	2.26s
1024	×	16	8.97s	8.33s	5.24s
4048	×	4	40.01s	9.24s	5.18s

Figure 5.8: Execution times of verifying a memory with varying numbers of entries (64, 256, 1024, 4048) and word sizes (4, 8, 16, 32, 64). Execution times were split into time needed to build the circuit, of running our auto_abstract algorithm, and of running STE in the Forte verification environment.

This result – especially as the indexing was computed fully automatically, except for stating the read address as symbolic constants – clearly demonstrates the efficiency and practicality of our approach.

Variants of Reusing Indexing Variables

While execution times were very similar for the different variant of reusing indexing variables when verifying the CAM, major differences can be observed when verifying the memory. Figure 5.9 show how fragile the algorithm is to changes in the way indexing variables are reused. While our main variant allows verifying memories up to the artificial limitations set by the open-access version of the Forte verification

entries	×	*bits*	(1)	(2)	(3)	(4)	(5)
64	×	4	0.07s	1.48s	0.14s	1.58s	1.58s
64	×	8	1.65s	61.26s	0.30s	65.75s	62.03s
128	×	4		28.01s	0.67s		

Figure 5.9: Execution times observed for verifying the memory using different methods: (1) without symbolic indexing, (2) no reuse of indexing variables, (3) main reuse variant, same results as seen earlier in this section, (4) reuse based on mutual exclusivity while preferring variables used for similar node names, and (5) reuse based on mutual exclusivity only. Execution times for larger memories using Method (3) are provided in Figure 5.8.

environment, the other variants performed much worse for very small memories already, and could not be completed at all for memories with 64 entries, which each have 16 bits already. In particular, using these variants was not superior to verifying the memory without any symbolic indexing at all. They all ran out of memory even for small instances of the memory. Therefore, Figure 5.9 only lists execution times for three different memory sizes, even though we successfully verified much larger memories with our main variant, as seen in Figure 5.8.

5.3 Scheduler

In our first two examples we verified models that had previously been successfully verified using STE, and for which good symbolic indexing schemes were known. It allowed a good evaluation of our work in comparison with previous work done in this area. For our final example, we verify a model not previously verified by STE. It highlights the main advantage of automatically computing symbolic indexings: enabling the verification of circuits where no good indexing schemes were previously researched.

The design we chose is a scheduler, which is abstractly shown in Figure 5.10. Its functionality is to compute the address of the oldest entry that is ready, and then return that entry plus whether it is valid, i.e., points to a ready entry. As the hardware design itself does not automatically store the current time, but the age is passed into the scheduler, you could also say we are verifying a scheduler with a priority queue: it selects the entry with the highest priority/age, whose ready-bit is additionally set.

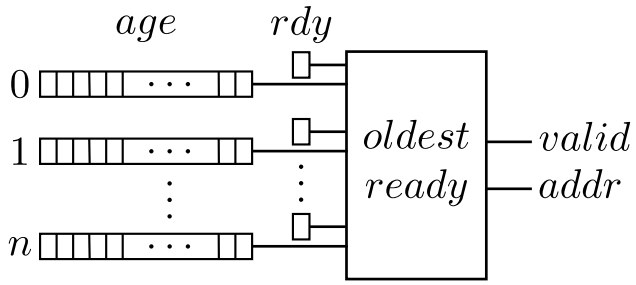


Figure 5.10: High-level model of the scheduler verified

5.3.1 Hardware Model

Figure 5.11 shows the FL-code that generates the scheduler whose functionality we verified for varying numbers of registers and age widths.

Lines 58–69 define the signals. The *din* vector delivers the age of an entry to write, and *valid_in* whether it is ready. These are only used when writing data, i.e., whenever the signal *we* is high. The outputs are *valid*, which is high exactly if there is a ready entry, and *addr*, which specifies the address of the oldest ready entry, provided *valid* is high. Finally, some auxiliary vectors are defined. The vector *valids* accumulates which registers store ready entries, *regs* actually stores all of the entries, and *age* is used to memorise the thus far oldest age found.

The code provided in lines 74–79 implement writing new entries to the scheduler, and lines 80–83 output whether a ready entry was found, and if so, the address of the oldest ready entry.

The core functionality of the oldest-ready scheduler is defined in lines 3–24, which compares two $(valid, age, address)$ pairs and returns the candidate, which is ready and older. So if only one entry is ready, then it is returned; if both entries are ready, the one with the higher age is returned; and if neither entries are ready, then by convention the first one is selected. Lines 26–56 recursively compares entries pairwise. In particular, a divide-and-conquer mechanism is used where the oldest ready entry in the first half of the registers is found, as well as the one in the second half, and then the oldest ready of those two is selected.

5.3.2 SIR Relation

Providing a specification function that actually computes the oldest ready entry is fairly involved. However, supplying a specification provided a high *valid* bit is output – subject to a proposed address – is much easier. Thus we write the specification

```
let mk_scheduler log_registers age_width =
  let n = 2**log_registers in
  let find_oldest regs valids valid addr age =
    let cmpmod =
      let valid = "valid" in
      let age = sprintf "age[%d:0]" (age_width-1) in
      let addr = sprintf "addr[%d:0]" (log_registers-1) in
      let valid0 = "valid0" in
      let age0 = sprintf "age0[%d:0]" (age_width-1) in
      let addr0 = sprintf "addr0[%d:0]" (log_registers-1) in
      let valid1 = "valid1" in
      let age1 = sprintf "age1[%d:0]" (age_width-1) in
      let addr1 = sprintf "addr1[%d:0]" (log_registers-1) in
      let lt = "lt" in
      mk_module "oldest_sel" [valid0, age0, addr0, valid1, age1, addr1]
          [valid, age, addr] (
      (mk_less (ev age0) (ev age1) "lt") #
      (mk_not valid0 "nvalid0") #
      (mk_or ["nvalid0", "lt"] "nv_or_lt") #
      (mk_and [valid1, "nv_or_lt"] "choose1") #
      (mk_Xmux "choose1" addr1 addr0 addr) #
      (mk_Xmux "choose1" age1 age0 age) #
      (mk_Xmux "choose1" valid1 valid0 valid)
      )
  in
  letrec find_oldest bi i regs valids valid addr age =
    length regs = 1 ⇒ (
    (mk_buf (hd valids) valid) #
    (mk_constant i (ev addr)) #
    (mk_buf (hd regs) age)
    ) | (
    let valid0 = sprintf "valid0_%d" bi in
    let age0 = sprintf "age0_%d[%d:0]" (bi,age_width-1) in
    let addr0 = sprintf "addr0_%d[%d:0]" (bi,log_registers-1) in
    let valid1 = sprintf "valid1_%d" bi in
    let age1 = sprintf "age1_%d[%d:0]" (bi,age_width-1) in
    let addr1 = sprintf "addr1_%d[%d:0]" (bi,log_registers-1) in
    let n2 = length regs/2 in
    (find_oldest (2*bi+1) (2*i+1)
        (firstn n2 regs) (firstn n2 valids)
        valid1 addr1 age1
    ) #
```

(continued on next page)

```
43        (find_oldest (2*bi) (2*i)
44             (butfirstn n2 regs) (butfirstn n2 valids)
45             valid0 addr0 age0
46        ) #
47        (mk_instance "oldest_sel" (sprintf "osel%d" bi)
48          [valid0, age0, addr0, valid1, age1, addr1]
49          [valid,age,addr]
50        )
51      )
52      in
53      mk_module "find_oldest" (regs@valids) [valid, addr, age]
54        (find_oldest 1 0 regs valids valid addr age)
55      #
56      cmpmod
57      in
58      // Inputs
59      let clk = "clk" in
60      let din = sprintf "din[%d:0]" (age_width-1) in
61      let valid_in = "valid_in" in
62      let we = sprintf "we[%d:0]" (n-1) in
63      // Outputs
64      let valid = "valid" in
65      let addr = sprintf "addr[%d:0]" (log_registers-1) in
66      // Internals
67      let valids = [ sprintf "valids[%d]" i | i in (n-1) downto 0 ] in
68      let regs = [ sprintf "R[%d][%d:0]" (i, age_width-1) | i in (n-1) downto 0 ] in
69      let age = sprintf "age[%d:0]" (age_width-1) in
70      //
71      mk_module "scheduler"
72      [clk,din,valid_in,we] [valid,addr]
73      (
74      (forall [ (en,r) | zip (ev we) regs ].
75        mk_en_re_ff clk din en r
76      ) #
77      (forall [ (en,v) | zip (ev we) valids ].
78        mk_en_re_ff clk valid_in en v
79      ) #
80      mk_instance "find_oldest" "find_oldest"
81        (regs@valids) [valid, addr, age]
82      ) #
83      (find_oldest regs valids valid addr age)
84  ;
```

Figure 5.11: The code used to generate an oldest-ready scheduler

while utilising the address *addr* output and its corresponding age *age* as symbolic constants:

$$\bigvee_i \Big(addr = i \wedge rdy[i] \wedge age[i] = age \wedge \bigwedge_{j \neq i} (\overline{rdy[j]} \vee age[j] < age) \Big)$$

The first part, $addr = i \wedge rdy[i] \wedge age[i] = age$, encodes that the i^{th} entry was output by the circuit. The second part, $\bigwedge_{j \neq i} \overline{rdy[j]} \vee age[j] < age$, encodes that i is actually the oldest ready entry – either the other entries are not ready, or their age is lower. For all but the correct i, this yields a false value. So by taking the disjunction over all i we receive the final result.

Note that this assumes that the oldest ready entry is uniquely defined, i.e., there are no two entries, which are both ready and have the same, oldest age compared to all other ready entries. To additionally capture such cases, we need to know which address is then returned. Recall that we built a hardware model, in which the first oldest ready entry is selected, because *cmpmod*, which compares two entries and returns the oldest ready of the two, by convention returned the first entry, if both were ready and had the same age. Thus, we can adjust the specification as follows:

$$\begin{aligned} \bigvee_i \Big(addr = i \wedge rdy[i] \wedge age[i] = age \quad & \wedge \quad \bigwedge_{j<i} (\overline{rdy[j]} \vee age[j] < age) \\ & \wedge \quad \bigwedge_{j>i} (\overline{rdy[j]} \vee age[j] \leq age) \Big) \end{aligned}$$

So all entries with a smaller address are either not ready or younger, and all entries with a larger address are either not ready, or at most as old as the entry whose address is output.

In the following we do a partial verification of the scheduler. Namely, we verify that if there is a unique oldest ready entry, then the circuit outputs a high *valid* signal and the address of that entry. The first part we achieve by using the specification we first provided, and assume it holds, i.e., it evaluates to true. The second part is ensured by stating the consequent slightly more restrictively:

$$C = (valid \text{ is } 1) \text{ and}_j \ (addr[j] \text{ is } o[j]),$$

i.e., rather than driving *valid* with a variable in the auxiliary STE run, which is reindexed to $R \models A^R \Rightarrow C_R$, we state *valid* needs to be high in all cases.

The indexing computed is somewhat more elaborate and cannot be as easily formulated for the general case. But we give the full SIR relation for the smallest possible

scheduler, i.e., one with two entries, and an age width of 1. The indexings for larger schedulers work similarly, although they grow quite complex quickly.

Example: SIR relation for a 2×1 scheduler
By running $\mathsf{auto\_abstract}(\{addr, age\}, \text{SPEC}, \mathsf{tt}, \mathsf{ff}, \text{“”})$ where

$$\begin{aligned} \text{SPEC} \quad = \quad & (addr = 0 \wedge rdy[0] \wedge age[0] = age \wedge (\overline{rdy[1]} \vee age[1] < age)) \\ \vee \quad & (addr = 1 \wedge rdy[1] \wedge age[1] = age \wedge (\overline{rdy[0]} \vee age[0] < age)) \end{aligned}$$

we receive the following SIR relation:

$(addr[0],$	$a,$	$\mathsf{ff})$
$(\overline{addr[0]},$	$\overline{a},$	$\mathsf{ff})$
$(valids[0],$	$\overline{a},$	$a \wedge \overline{b})$
$(valids[1],$	$a,$	$\overline{a} \wedge \overline{b})$
$(age[0],$	$b,$	$\mathsf{ff})$
$(R[0][0],$	$\overline{a} \wedge age[0] \vee b \wedge age[0],$	$\overline{a} \wedge \overline{age[0]} \vee a \wedge b)$
$(R[1][0],$	$a \wedge age[0] \vee b \wedge age[0],$	$a \wedge \overline{age[0]} \vee \overline{a} \wedge b)$

The variable a encodes which address is returned, and thus holds the oldest ready entry. In the relation this is expressed in the first two tuples. The H-component of the third and fourth tuple encode that $valids[i]$ needs to be high, if the address i is returned. This needs to obviously be true, as we are only verifying the case where the output *valid* is high. Additionally, if the address i is returned, we ensure that $R[i][0]$ has the same value as the oldest ready age, $age[0]$. In the relation this is expressed by the conjunctions that include a and $age[0]$ or their negations only.

Finally, we need to drive some further values to ensure that $age[0]$ actually holds the oldest ready age. This is encoded with the variable b. There are two main cases. First, in the $\overline{b}$ case, we ensure the other entry cannot be correct, as its corresponding *valids* entry is false. For this, see the L-component of the tuples for $valids[i]$. And second, in the b case, we ensure the other entry has a younger age. In our small example where the age has one bit only, this is still simple to encode. The only way one entry can be younger than the other is when the oldest age is 1, and the younger age is 0. This causes the entry $(age[0], b, \mathsf{ff})$, which encodes that the oldest age must be 1, to be introduced. Furthermore, the L-component of $R[0][0]$ states that the age stored must be low, if the address 1 returned $(a \wedge b)$, and versed $(\overline{a} \wedge b)$. ★

Understanding this example gives a good intuition on what the SIR relations for

larger schedulers look like. It also immediately gives a sense on how complex the expressions turn. We need to encode that the ages stored at the other locations are younger, or that they are not ready. In the small example, these could not intermix, as we only had two entries. And expressing that the age was younger was also simple, as it just required a simple bit-flip.

This illustrates the power of a fully automatic approach. Even if a human could have come up with the idea for the indexing our algorithm computes, it would take considerable work to correctly express it in larger cases. Our algorithm, on the other hand, automatically generates it for varying numbers of entries and age widths without requiring further intellectual work.

5.3.3 Observed Execution Times

Figure 5.12 shows the data we collected on verifying the scheduler utilising the automatically computed SIR relation described above. It shows that using this technique realistically-sized schedulers can be verified.

It is worth pointing out that trying to verify the same circuit without symbolic indexing, i.e., with variables in every state holding register and input, could not be completed for circuits larger than 64 entries and age widths larger than 8, as also seen in Figure 5.13. In other words, only the first circuits for which we provide an execution time when using our approach could also be verified without symbolic indexing, and then at much worse execution times. With our method the smaller schedulers could be verified extremely fast, and larger circuits could still be verified in a completely straightforward and fully automatic fashion. Verifying schedulers with 1024 entries and age widths larger than 8 could not be completed even with our computed symbolic indexing. However, hardware schedulers commonly support much fewer numbers of entries, and lower age widths. For example, in [70] a hardware scheduler is proposed that supports 64 entries and a 6-bit priority. As noted in the beginning of this section, the age in our oldest-ready scheduler can also be viewed as the priority in a priority queue scheduler, thus permitting a comparison between age width and priority width. This means the largest scheduler we verified at 1024 entries and an age width of 8 bit, or at 256 entries with an age width of 12 bit, greatly surpass the values stated in [70]. More generally, while increasing the size of our scheduler does show that our approach does not offer a full solution to the state explosion problem, it also proves that automatic abstraction is very powerful and allows the verification of realistically-sized circuits and beyond.

entries	×	*bits*	circuit	auto_abstract	STE
16	×	4	0.01s	0.05s	0.01s
16	×	6	0.01s	0.08s	0.03s
16	×	8	0.01s	0.14s	0.12s
16	×	10	0.01s	0.27s	0.59s
16	×	12	0.01s	1.08s	2.07s
64	×	4	0.01s	0.52s	0.11s
64	×	6	0.02s	0.46s	0.39s
64	×	8	0.02s	0.85s	1.00s
64	×	10	0.03s	1.84s	5.66s
64	×	12	0.03s	5.33s	31.64s
256	×	4	0.07s	4.59s	0.91s
256	×	6	0.07s	7.61s	3.63s
256	×	8	0.09s	7.75s	7.15s
256	×	10	0.10s	12.84s	33.13s
256	×	12	0.10s	36.35s	2079.56s
1024	×	4	0.32s	146.34s	18.55s
1024	×	6	0.37s	146.32s	33.59s
1024	×	8	0.42s	160.43s	47.67s

Figure 5.12: Execution times of verifying a scheduler with varying numbers of registers (16, 64, 256, 1024) and age widths (4, 6, 8, 10, 12). Execution times were split into time needed to build the circuit, of running our auto_abstract algorithm, and of running STE in the Forte verification environment.

Variants of Reusing Indexing Variables

Figure 5.13 shows the execution times observed when running the different variant of the abstraction discovery algorithm, which reuse indexing variables less or more aggressively. As seen for the memory, execution times are considerably worse than when running the main version of our algorithm, although still schedulers of reasonable size could be verified. Nonetheless, evidently the algorithm as presented in Figure 4.6 is superior to the other reuse variants, delivering better execution times and behaving much more robustly. However, this is an observation collected on three designs only, and it might be that another variant outperforms our main variant on other designs. Thus, when verifying other circuits it may be advantageous to try a different variant if the first cannot be completed due to missing resources.

entries	×	*bits*	(1)	(2)	(3)	(4)	(5)
16	×	4	0.05s	0.20s	0.06s	0.22s	0.26s
16	×	6	0.19s	1.03s	0.11s	0.88s	1.25s
16	×	8	0.76s	6.92s	0.26s	8.33s	8.37s
16	×	10	3.93s	40.82s	0.86s	43.29s	49.19s
16	×	12		214.66s	3.15s	213.65s	220.07s
64	×	4	1.79s	12.55s	0.63s	13.76s	13.84s
64	×	6	8.03s	49.50s	0.85s	51.51s	50.51s
64	×	8	38.70s	374.29s	1.85s	372.91s	376.72s
64	×	10		2625.16s	7.50s	2636.68s	2390.10s
64	×	12			36.97s		

Figure 5.13: Execution times observed for verifying the scheduler using different methods: (1) without symbolic indexing, (2) no reuse of indexing variables, (3) main reuse variant, same results as seen earlier in this section, (4) reuse based on mutual exclusivity while preferring variables used for similar node names, and (5) reuse based on mutual exclusivity only. Execution times for larger schedulers using Method (3) are provided in Figure 5.12.

5.4 Summary and Conclusions

We saw that applying our automatic abstraction-discovery algorithm generates abstractions, that greatly reduce the verification costs of three hardware designs: a CAM, a memory, and a scheduler. CAMs and memories are designs often verified by STE, and our results show that our approach works nicely for such standard examples. The third circuit, a scheduler, is a design usually not verified by STE, and displays the power of generating abstractions automatically by analysing specifications. It enables the verification of designs where previously no good indexings were known.

We verified all three designs in increasing sizes. In particular, the larger designs could not be verified without symbolic indexing due to restrictively high verification costs, whereas our approach delivered execution times, which suggest good scalability. We analysed the SIR relations created to give an intuition on which abstraction schemes our automatic abstraction discovery algorithm finds. In particular, the abstraction generated for the CAM matches with the indexing suggested by Pandey et al. [24]. Rather than it being a result of careful reasoning, this indexing was generated automatically by analysing the specification using our auto_abstract algorithm.

For each design we ran our algorithm with different variants for sharing variables in the symbolic indexing created. We saw that our main algorithm, which shares

variables only when encountering multiple-input AND-gates, delivers the most robust behaviour and works well for all three designs. The other variants in some cases lead to restrictively high verification costs again, which highlighted how fragile verification by STE is with respect to changing variable sharing.

Finally, for these experimental results we defined some symbolic constants manually. In the following chapters we present an approach by which these symbolic constants can be computed automatically, and we demonstrate this on the CAM in Chapter 8.

Chapter 6

Abstraction Refinement with Symbolic Indexing

When using abstraction to verify circuits, there is an inevitable – indeed desired – loss of information. A good abstraction scheme retains only the information needed to verify the circuit's specification. Such an abstraction reduces the problem size, and can make as huge a difference as turning an unfeasible task into a manageable one. Unfortunately, constructing a good abstraction scheme is hard.

If the verification of a circuit fails because the applied abstraction hides too much information, we call this *over-abstraction*. In the context of STE over-abstraction occurs if the simulation determines that an output has the value X, although the specification requires it to be Boolean. As described on page 35, this is also called a *weak disagreement*: we do not observe the value we expect, but it is unclear whether the concrete model exhibits an error in this case or not.

Example: A simple example of over-abstraction

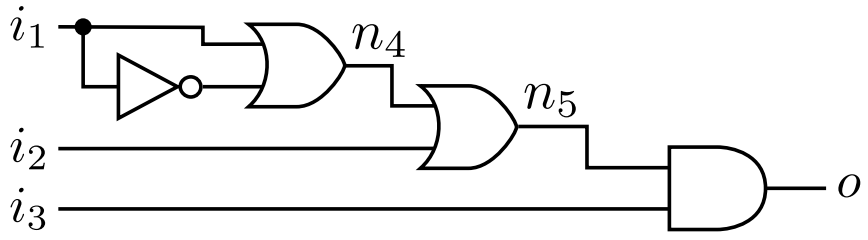


Figure 6.1: A circuit where over-abstraction can be observed when i_1 is indeterminate

Consider the circuit displayed in Figure 6.1. Suppose we want to verify the speci-

fication that states that if the input i_3 is true, then the output o is also true, i.e., we want to check whether $c \models (i_3 \text{ is } 1) \Rightarrow (o \text{ is } 1)$ holds. In this small example it is easy to see that this statement holds, because n_4 is always high, irrespective of the value of i_1. Using STE leads to a weak disagreement, though.

The antecedent $(i_3 \text{ is } 1)$ does not specify the value of i_1 or i_2, so the inputs have the value X when running STE. The simulator thus also determines the value of node n_4 as X, and similarly n_5 is X. This in turn leads to an indeterminate value of o, we get a weak disagreement. The abstraction we used, namely do not track the values of i_1 and i_2, hides too much information. ★

One common approach for constructing good abstraction schemes is to start with an abstraction that retains very little information, as also seen in our above example. This reduces the problem size considerably at the risk of leading to over-abstraction. Using counter-example guided abstraction refinement [22], over-abstraction cases can then be eliminated by repeatedly adding back in information that is required to avoid a specific case of over-abstraction. Each such refinement step concretises the abstraction in that it hides less information, and thus increases the problem size again.

The automatic abstraction algorithm we introduced in Chapters 3 and 4 may deliver abstraction schemes that lead to over-abstraction. For an example, see Section 8.1. In this chapter we introduce an algorithm for refining abstraction schemes that respects symbolic indexing. This algorithm is based on calculating which inputs need to be driven to avoid over-abstraction in specific abstraction cases, and significantly extends previous work by Chockler, Grumberg, and Yadgar [3]. Their work gives a good starting point for how to address over-abstraction, but is only applicable for a very restricted set of STE statements: antecedents may only assign input nodes a concrete value or a single Boolean variable. This means that symbolic indexing, which makes STE so powerful, is not supported by this previous work. In contrast, our approach supports all TEL formulae of STE statements. Furthermore we suggest more conservative abstraction refinements, which may lead to lower verification costs. So we both analyse and introduce non-trivial symbolic indexing, which means our abstraction refinement can applied much more broadly, in principle to all STE runs that require refinement due to over-abstraction. In Chapter 8 we show that in conjunction with our automatic abstraction discovery algorithm this delivers a fully automatic abstraction framework for Symbolic Trajectory Evaluation.

6.1 CEGAR using Degree of Responsibility

An STE run simulates the circuit. We can memorise the value of each node during this simulation to get a trace list of all values. Over-abstraction can then be identified as the existence of an assignment to the Boolean variables used in the guards, such that a node has the value X, although the consequent requires it to have a non-X value. In the following we call such a node X-*possible*. The goal is to now construct an abstraction, so that no X-possible nodes occur anymore.

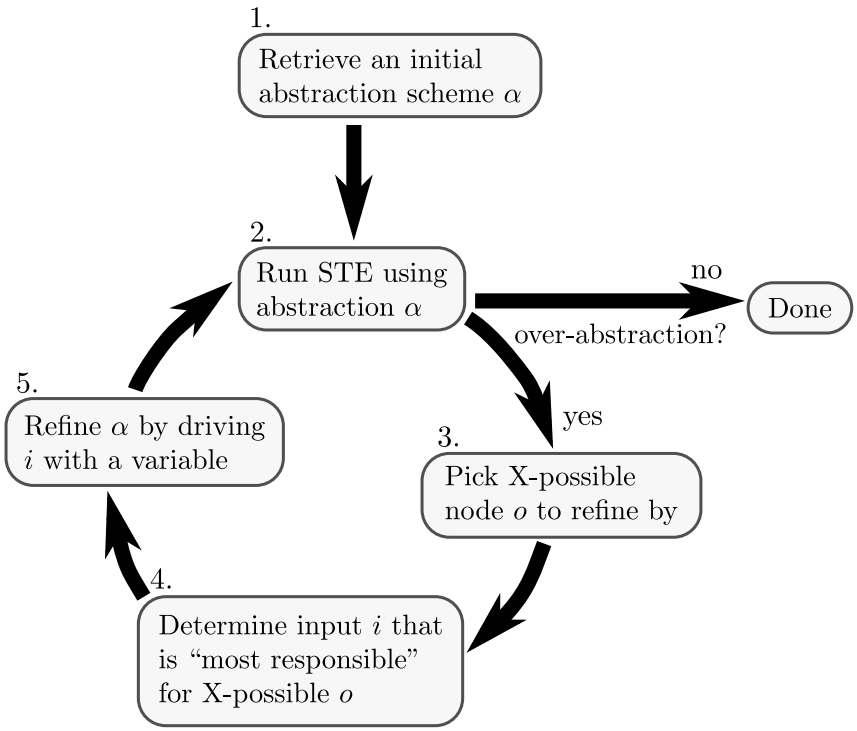


Figure 6.2: A high-level overview of the CEGAR loop proposed in [3].

While achieving this we want to concretise the abstraction as little as possible, as otherwise the verification task may get too costly again. One approach is to only minimally change a too information-sparse abstraction, and then check whether any nodes are still X-possible. By repeating this process we produce a refinement loop. By interpreting the STE run that has X-possible nodes as a counter-example, this can be seen as a counter-example guided abstraction refinement loop (CEGAR).

The basic setup of the CEGAR-like loop proposed by Chockler et al. [3] is depicted in Figure 6.2. After having identified an initial abstraction (1), run STE with that abstraction (2). If no over-abstraction occurs, we have a definitive verification result and are done. Otherwise, we need to refine the abstraction. For this, first pick an X-

possible node (3), that is, a node which for at least one valuation of the variables used in the antecedent has the value X, but needs to have a concrete value for deciding whether the consequent holds. Then evaluate which input of the circuit is "most responsible" for that X-possible node not being determinate (4). Next refine the initial abstraction by driving the identified input with a fresh variable (5). Finally, we start the next iteration of the loop and check again whether running STE leads to an over-abstraction, or whether we now get a definitive verification result, i.e., pass or fail (2).

Example: Sample run of a CEGAR loop
In the example we saw at the beginning of this chapter (138) we saw that checking whether $c \models (i_3 \text{ is } 1) \Rightarrow (o \text{ is } 1)$ leads to a weak disagreement.

In the refinement loop we so far retrieved an initial abstraction, and ran STE with it. Over-abstraction was detected, so we now need to pick an X-possible node to refine the abstraction by. In this instance there is only one X-possible node, o: the consequent requires it to be concrete, o is 1, but the simulation value is indeterminate. Now suppose we have an algorithm that tells us which input to pick for refining the abstraction we used. That algorithm could return either i_1 or i_2.

- Refine by i_1
  Then the abstraction drives i_1 with a fresh variable v_1. The new antecedent is thus $(i_1 \text{ is } v_1)$ and $(i_3 \text{ is } 1)$. Notice that when rerunning STE the value of n_4 is then determined as 1, as STE knows that $v_1 \vee \overline{v_1} = 1$. And as $1 \vee \mathrm{X} = 1$, the value of n_5 is 1 also. Thus, the simulator assigns the value 1 to o, and the STE run succeeds.

- Refine by i_2
  In this case the new antecedent is $(i_2 \text{ is } v_2)$ and $(i_3 \text{ is } 1)$. Remember that $(i_2 \text{ is } v_2)$ is shorthand for $(v_2 \rightarrowtriangle i_2 \text{ is } 1)$ and $(\overline{v_2} \rightarrowtriangle i_2 \text{ is } 0)$, where v_2 is a binary variable used for the guards. We need only verify properties when the guards are satisfied. This symbolic run corresponds to two non-symbolic STE runs: either v_2 holds, and thus the antecedent is $(i_2 \text{ is } 1)$ and $(i_3 \text{ is } 1)$; or $\overline{v_2}$ holds, and the antecedent is $(i_2 \text{ is } 0)$ and $(i_3 \text{ is } 1)$. In the prior case the verification passes, but in the latter one over-abstraction occurs again. So STE reports a weak disagreement whenever i_2 is low, i.e., when $\overline{v_2}$ holds. Thus, for the complete run, we still have over-abstraction, and therefore must walk through the refinement loop again. Only one input is not being driven by a value yet,

so the only possible input to refine by is i_1. After this refinement step the antecedent is $(i_1 \text{ is } v_1)$ and $(i_2 \text{ is } v_2)$ and $(i_3 \text{ is } 1)$ and the verification passes.

Note that in the worst case the refinement loop leads to an abstraction where all inputs are driven by a value. In that case over-abstraction cannot occur anymore. However, we want to keep the time and memory constraints for the verification low, and refining increases these. So it is crucial that the input selected for refinement (step 4 in Figure 6.2) identifies inputs that are essential for determining the selected X-possible node. In our example we saw that refining by i_1 is better than refining by i_2. In the latter case a further refinement by i_1 was necessary. ★

6.1.1 Degree of Responsibility

The algorithm proposed in [3] computes the *Degree of Responsibility* (DoR) of inputs to determine which node shall be driven by a value in the refined abstraction. The DoR of an input i for an output o is based on the notion of *counterfactual dependency* [71] and its extension, *causality* [72, 73].

An event A counterfactually depends on an event B, if A and B both hold, and assuming B did not hold, neither would A. For example, say event A is "Alice makes a camp fire", and event B is "Forest burns down". A counterfactually depends on B if both events indeed occurred, and if the forest only burned down because Alice started a camp fire. On the other hand, if the forest would have burned down irrespective of Alice making a camp fire or not, then there is no counterfactual dependency.

In the notion of cause the requirements are loosened a bit. C is a cause of B if some changes to the current situation can lead to C being counterfactually dependent on B. For example, say that C is "Alice makes a camp fire" and D is "The woods are dry". Assume that we know if C and D hold, then B holds, too: $C \wedge D \wedge B$. But as it is, right now the woods are not dry, and the woods do not burn down, although Alice makes a campfire: $C \wedge \overline{D} \wedge \overline{B}$. We say C is a cause of B, if – looking at an alternate reality – we can establish counterfactual dependency between C and B by adjusting the situation. Namely, we are allowed to change whether the event D happens. But when D holds, C indeed is counterfactually dependent on B, so C is a cause of B.

The degree of responsibility of event E for event F is defined as $\frac{1}{1+k}$, where k is the minimum number of changes to the situation required to create counterfactual dependency between the two events. Loosely speaking, it is the inverse of how strong a cause E is of F. In the above examples the DoR of A for B is $\frac{1}{0+1} = 1$, because

no changes are required for counterfactual dependency, and the DoR of C for B is $\frac{1}{1+1} = \frac{1}{2}$, because we needed to guarantee D holds.

The DoR is a value between 0, when the two events are never counterfactually dependent, and 1, when the two events are counterfactually dependent. One might be tempted to compare this to probabilities. The probability of A causing B is 0, if A never leads to B, and the probability is 1, if A always leads to B. But the two concepts match only in these two border cases, and must not be confused. The degree of responsibility gives an exact notion of how many changes to the situation make the event the deciding one. So for each situation, the degree of responsibility varies. For example, if there is a drought, D holds without having to change the situation and the DoR of C is 1, else the DoR is $\frac{1}{2}$. In particular, the inverse of the DoR is additive. Probabilities, on the other hand, do not behave like this.

Selecting Refinement Candidates

The DoR can be used for identifying a good refinement candidate. Let B be the event "o is X-possible" and A_i be the event "input i is X". If an input is counterfactually dependent on o, then refining by it leads to o having a concrete value in the next iteration of the refinement loop, and possibly leads to there being no over-abstraction anymore, just as desired. If there is no counterfactual dependency, the idea is to select an input i with a high degree of responsibility for B. Changes to the situation in this setting correspond to assuming that specific inputs have a concrete value not stated by the antecedent. Once we have determined which i has the highest degree of responsibility, we need to change the abstraction so that i has a concrete value. But to ensure we maintain expressiveness of an STE statement we have to cover all cases as before. So if we want to change the antecedent so that an input that was previously X now has a concrete value, we need to capture both the situation where it is high, and where it is low. We can achieve this in parallel by driving the chosen input with a symbolic value v_i. Either the input is going to be true ($v_i \rightarrowtriangle i$ is 1), or the input is going to be false ($\overline{v_i} \rightarrowtriangle i$ is 0). Note that in the next iteration of the loop the DoR of the remaining inputs in general changes, as the situation is a different one. Namely, i is not X anymore.

Computing the DoR in circuits of size n is known to be $\mathrm{FP}^{\Sigma_2^P[\log n]}$-complete [73,74]. $\mathrm{FP}^{\Sigma_2^P[\log n]}$ denotes the class of functions computable in polynomial time with $\log n$ queries to the oracle Σ_2. As an exact computation is expensive in the general case, Chockler et al. propose computing an approximate degree of responsibility, which has quadratic complexity in the size of the model. The approximation is based on some

previous work [3], and is exact if the circuit has a tree structure. Essentially, the approximation simplifies the problem by ignoring all dependencies between nodes, which in turn may lead to the approximation being lower than the actual degree of responsibility. By ignoring dependencies, we may incorrectly conclude that an additional change to the current situation is necessary to achieve counterfactual dependency – a change that we already achieved by a different path.

The computation of the approximate degree of responsibility as defined in [3] assumes we know the values assigned to each node in the antecedent. In particular, input nodes either have a concrete value (0 or 1), are indeterminate (X), or have the simple symbolic value of a single Boolean variable. This excludes all non-trivial symbolic indexing, which states that nodes have a concrete value only if Boolean formulae evaluate to true.

The computation also assumes that we have identified an X-possible output o, which we want to refine by. Note that an X-possible output need not always be X. But there needs to exist a valuation of the Boolean variables used in the guards of the antecedent such that the output is X, although the consequent states it should be concrete (0 or 1). Indeed, when running STE and a weak disagreement occurs, the output will have a symbolic value. For example, say the consequent states o is 1. In an over-abstraction case, STE may determine that o is 1 only if the expression res evaluates to true, $res \rightarrow o$ is 1. So whenever res evaluates to false o is X. When determining the approximate degree of responsibility only these cases, i.e., when $\overline{res}$, are of interest. Chockler et al. neither bring this up in [3], nor do they address how cases where the output is only sometimes X are handled. Their approach does not address non-trivial indexing at all, and therefore they may have only encountered over-abstraction for all valuations, i.e., $res = \mathsf{ff}$. However, it is even in their setting possible.

We can extend Chockler et al.'s approach to ensure that the computation of the approximate degree of responsibility is based on the weak disagreement cases only. Assume STE surfaced that there is a weak disagreement whenever $\overline{res}$ and we have a trace list of all values of the corresponding STE run. Then we can evaluate the values of all nodes just in the weak disagreement case by determining their symbolic value implied by $\overline{res}$. That is, if a node has the symbolic value (f, g), i.e., it is 1 whenever $f \wedge \overline{g}$, 0 whenever $g \wedge \overline{f}$, and X whenever $f \wedge g$, then restricted to the weak disagreement case it has the value $(\overline{res} \rightarrow f, \overline{res} \rightarrow g)$. One way of determining these values is to rerun STE assuming $\overline{res}$ holds, an operation the Forte verification environment supports. It is also the methodology we used for our experimental results. Note that

the result of this run will lead to a weak disagreement where o is always X, although the consequent requires it to have a determinate value.

Hence, in the following we assume that the output is always X, and the antecedent is adjusted accordingly:

$$i \text{ is } v_i \text{ is transformed to } \begin{cases} i \text{ is } 1 & \text{if } \overline{res} \rightarrow v_i \text{ is a tautology} \\ i \text{ is } 0 & \text{if } \overline{res} \rightarrow \overline{v_i} \text{ is a tautology} \\ i \text{ is } v_i & \text{otherwise} \end{cases}$$

This means we take into consideration only the constraints the antecedent imposes on an input in the initial STE run, if it also does so in just the weak disagreement case $\overline{res}$.

The approximate DoR of the input i for o being X-possible is $DoR(i, o) = \frac{1}{1+s(i,o)}$, where $s(i, o)$ corresponds to an approximate number of changes to the antecedent necessary for making i and o counterfactually dependent. The proposed computation of $s(i, o)$ is provided in Figure 6.3.

Note that $s(i, o)$ is defined recursively over the structure of the circuit from the X-possible o to the input i. An input is counterfactually dependent on itself, so the $s(i, i) = 0$ and thus $DoR(i, i) = \frac{1}{1+0} = 1$. The input i is as responsible for the output of a NOT-gate as it is for the fanin of that gate, so $s(i, \text{NOT } f) = s(i, f)$. For an AND-gate, assume without loss of generality that the primary input i is in the fanin of the gate's first input node f, and not in the fanin of the second input node g. Then i is as responsible for the output of the AND-gate, as it is for the value of f, provided g evaluates to the constant value 1. So to determine the degree of responsibility, we need to calculate the cost of ensuring g to has the value 1, in the following called $c_1(g)$, and the cost of achieving counterfactual dependency between i and f. So, $s(i, f \text{ AND } g) = s(i, f) + c_1(g)$. If i is in the fanin of both f and g, by convention, we approximate the degree of responsibility by taking the average cost, so $s(i, f \text{ AND } g) = \frac{s(i,f)+s(i,g)}{2}$. In the general case, this does not match with the actual number of changes required to achieve counterfactual dependency. It is a tradeoff for reducing the complexity of determining the degree of responsibility. This approximation can be quite imprecise especially if one of the values has an infinite value. An infinite result means no counterfactual dependency can be achieved, which can be observed when some inputs are driven with constant inputs. For example, a has no effect on the output of a AND b if b is driven with the constant 0. But when i is in both fanins, an infinite value for just one of the fanins predominates the impact

$\boldsymbol{DoR(i,o) = \frac{1}{1+s(i,o)}}$

$s(i,i)$	$= 0$	
$s(i,j)$	$= \infty$	where $i \neq j$ and j is an input
$s(i, \text{NOT } f)$	$= s(i,f)$	
$s(i, f \text{ AND } g)$	$= s(i,f) + c_1(g)$	if i is in the fanin of f, and not of g, or i also in fanin of g, but $s(i,g) = \infty$
	$= c_1(f) + s(i,g)$	if i is in the fanin of g, and not of f, or i also in fanin of f, but $s(i,f) = \infty$
	$= \frac{s(i,f)+s(i,g)}{2}$	if i is in fanin of f and g, and $s(i,f) < \infty, s(i,g) < \infty$
	$= \infty$	otherwise

Heuristic for the cost of changing the value to 1

$c_1(j)$	$= 0$	if $A(j) = 1$
	$= \infty$	if $A(j) = 0$
	$= 2$	if $A(j) = \text{X}$
	$= 1$	if $A(j) \in \mathcal{V}$
$c_1(\text{NOT } f)$	$= c_0(f)$	
$c_1(f \text{ AND } g)$	$= c_1(f) + c_1(g)$	

Heuristic for the cost of changing the value to 0

$c_0(j)$	$= 0$	if $A(j) = 0$
	$= \infty$	if $A(j) = 1$
	$= 2$	if $A(j) = \text{X}$
	$= 1$	if $A(j) \in \mathcal{V}$
$c_0(\text{NOT } f)$	$= c_1(f)$	
$c_0(f \text{ AND } g)$	$= min\{c_0(f), c_0(g)\}$	

Figure 6.3: Calculation of $DoR(i,o) = \frac{1}{1+s(i,o)}$ as defined in [3]

of the other fanin. Hence the definition of s handles this corner case separately to prevent such noisy infinite values. Suppose i is in the fanin of both f and g when determining $s(i, f \text{ AND } g)$. Instead of returning the average of $s(i,f)$ and $s(i,g)$, if without loss of generality $s(i,g) = \infty$ we simply ignore the fact that i is in the fanin of g, and thus calculate $s(i, f \text{ AND } g) = s(i,f) + c_1(g)$. More generally,

$$s(i, \text{AND}_{j \in J} f_j) = \frac{\sum_{k \in K} s(i, f_k)}{|K|} + \sum_{j \in J \setminus K} c_1(f_j),$$

where $K = \{k \in J : \ i \text{ in fanin of } f_k \text{ and } s(i, f_k) < \infty\}$. If $|K| = 0$, then i is not at all responsible for the outcome of $\text{AND}_{j \in J} f_j$, and thus we set $s(i, \text{AND}_{j \in J} f_j) = \infty$.

The computation of the approximate responsibility depends on the computation of $c_1(f)$, the cost of changing the situation so that f has the value 1. First, consider the simple case where f is a primary input. If the antecedent includes the TEL formula "i is 0", or $A(i) = 0$, then the cost of changing such a value to 1 is set to infinite, $c_1(i) = \infty$, as requiring i to have the value 1 contradicts the antecedent. If on the other hand $A(i) = 1$, then no change at all is required, and $c_1(i) = 0$. The value of $c_1(i)$ in all remaining cases is based on heuristics. If $A(i) = \text{X}$ either value is permitted, so the change restricts to specific cases. The same holds if our input is driven with a symbolic value, $A(i) = v_i$ – it can still adopt either value. One difference here is that X encodes that both 0 and 1 may occur, but not necessarily do, whereas the symbolic value makes explicit that both values can occur.

In [3] the following heuristics are used. Refining an input from X to a concrete value has a cost of 2; and refining it from a variable v_i has a cost of 1.

We have seen how $c_1(i)$ is defined for primary inputs. For $c_1(f \text{ AND } g)$ we determine the sum of $c_1(f)$ and $c_1(g)$, as both inputs to an AND-gate need to be high to return a high output. This calculation does not take into account possible sharing of nodes in f and g, thus making it an approximation, which may lead to overestimating the cost of adjusting the situation to the desired scenario. Finally, the cost of changing the value of an inverter $c_1(\text{NOT } f)$ corresponds to changing the value of f to 0, $c_0(f)$.

Dually, $c_0(i)$ is infinity for $A(i) = 1$, and $c_0(i) = 0$ for $A(i) = 0$. For inverters, $c_0(\text{NOT } f) = c_1(f)$, as before. For AND-gates, $c_0(f \text{ AND } g) = \min\{c_0(f), c_0(g)\}$ – only one of the inputs needs to have a low value to yield a low output, and we can choose the one that is less costly to modify. Recall that the degree of responsibility of an input on the output was defined in terms of the least expensive way of creating a situation where the two nodes are counterfactually dependent.

Example: Approximate Responsibility Calculation
Consider Figure 6.4. If the antecedent does not specify the value of i_1, then the simulation results in an X-possible o. In the following we show how the responsibility value differs depending on what value the input i_2 is driven with.
1. $A(i_1) = \text{X}$ and $A(i_2) = \text{X}$

i_1 is in the fanin of both i_1 and n_5, so $s(i_1, o) = s(i_1, i_1 \text{ AND } n_5) = \frac{s(i_1,i_1)+s(i_1,n_5)}{2}$. And $s(i_1, n_5) = s(i_1, \text{NOT } n_4) = s(i_1, n_4) = s(i_1, n_3) + c_1(i_2)$, as i_1 is not in the fanin

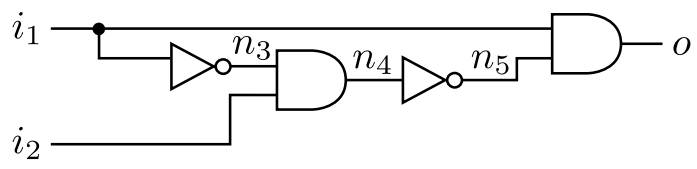


Figure 6.4: A circuit where o is X-possible whenever i_1 has an indeterminate value

of i_2.
Furthermore, $s(i_1, n_3) = s(i_1, \text{NOT } i_1) = s(i_1, i_1)$.
Finally, $c_1(i_2) = 2$, as $A(i_2) = \text{X}$.
So $s(i_1, o) = \frac{s(i_1,i_1)+s(i_1,i_1)+c_1(i_2)}{2} = \frac{0+0+2}{2} = 1$

2. $A(i_1) = \text{X}$ and $A(i_2) = v_2$

As above, $s(i_1, o) = \frac{s(i_1,i_1)+s(i_1,i_1)+c_1(i_2)}{2}$, but $c_1(i_2) = 1$ because $A(i_2) = v_2$. Hence, $s(i_1, o) = \frac{0+0+1}{2} = \frac{1}{2}$.

3. $A(i_1) = \text{X}$ and $A(i_2) = 0$

In this case, $c_1(i_2) = \infty$, and thus $s(i_1, n_5) = \infty$. Therefore, $s(i_1, o) = s(i_1, i_1) + c_1(n_5)$.
But $c_1(n_5) = c_0(n_4) = \min\{c_0(n_3), c_0(i_2)\} = \min\{c_1(i_1), c_0(i_2)\}$, and $c_0(i_2) = 0$ because $A(i_2) = 0$.
Thus, $s(i_1, o) = s(i_1, i_1) + c_1(n_5) = 0 + 0 = 0$

$s(i_1, o)$ captures how costly it is to create a counterfactual dependency between i_1 and o. Thus, the lower this value is, the more responsible i_1 is for the value of o. If $s(i_1, o) = 0$, then the input and the output are counterfactually dependent on each other, as seen in the third instance. ★

6.2 Extension to Symbolic Indexing

The approach presented in [3] is applicable for only very specific STE statements. The TEL formulae may only require a node to have a concrete value, or the value of a single Boolean variable: i is v_i. This does not allow any non-trivial symbolic indexing.

As explained in more detail in Section 2.6, symbolic indexing provides a partitioning layer on top of the ternary logic used in STE. It allows inputs to have values only if given propositional formulae called *guards* are satisfied. Its simplest application

allows symbolic constants, i.e., an input can be driven with a variable. For example, we may specify that the input i_1 is driven with a high value when the guard v_1 is satisfied, and it is driven with a low value when it is not satisfied:

$$v_1 \rightarrow i_1 \text{ is } 1 \text{ and } \overline{v_1} \rightarrow i_1 \text{ is } 0, \quad \text{or in short,} \quad i_1 \text{ is } v_1.$$

Only such basic symbolic indexings are supported by the method of Chockler et al., but even slightly more elaborate structures are not. This does not exploit the full power of partitioned abstraction in STE, one of its distinctive features.

In the following we will introduce an algorithm that can be applied to all STE statements, irrespective on how it uses symbolic indexing. We will further introduce two main approaches for refining abstractions. The first main approach leverages our work on automatic abstraction discovery, using the refinement candidates as an additional input to our auto_abstract algorithm. The second main approach includes three different refinement steps, between which we can switch in a refinement loop, e.g. using heuristics. Our new method will not only be more flexible, but also include some optimisations in the calculation of c_0 and c_1.

6.2.1 Evaluation of Symbolic Values

Before we generalise the responsibility algorithm to symbolic indexing, we need to recall how symbolic indexing is represented. As described in Section 2.7, when running STE, each input is driven with a value expressed in dual rail encoding. Two Boolean expressions encode which value an input has: the high rail is true if and only if the node may have a high value, and the low rail is true if and only if the node may have a low value. So if both rails are true, then the node may have either value, which corresponds to an X. Conversely, if neither rail is high, no value is allowed, which will only occur if there are inconsistencies, called *antecedent failures*.

For example, if the antecedent specifies that i_1 is v_1, then i_1 will have the value encoded as $(v_1, \overline{v_1})$. If v_1 is assigned true, this corresponds to i_1 having a high value, $(\mathsf{tt}, \mathsf{ff})$, and similarly if v_1 is assigned false, i_1 will be driven with a low value, $(\mathsf{ff}, \mathsf{tt})$.

The two expressions in the high and low rail can be any expressions on the set of indexing variables, not just simple variables. This enables more elaborate symbolic indexing. The only restriction we observe is that the disjunction of the high and low rails is a tautology, so that no evaluation will lead to an antecedent failure, $(\mathsf{ff}, \mathsf{ff})$.

In Figure 6.5 this is visualised. Any node, whose dual rail value is $(h[\mathcal{V}], l[\mathcal{V}])$ can have a concrete or indeterminate value, depending on the assignment to the variables

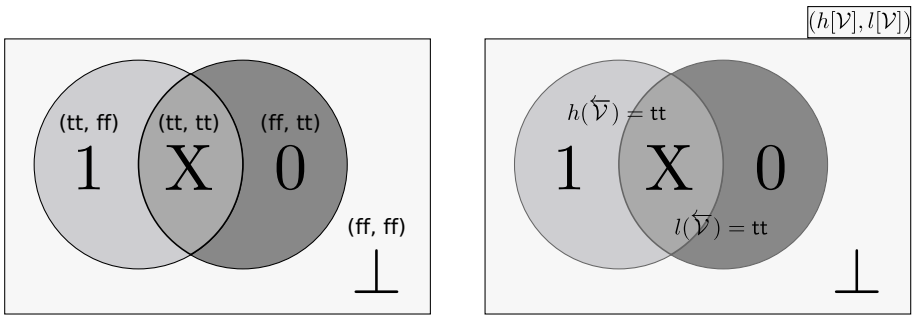


Figure 6.5: Dual rail encodings and their corresponding ternary values. The left, lighter circle encloses all states where the high rail is true, and the right, darker circle those where the low rail is true.

in $\mathcal{V}$. Note that by requiring the disjunction of h and l to be a tautology, the node will never have the value $\bot$, which implies an "impossible" state.

This also means we can exactly capture when a node has the concrete value tt, or the concrete value ff. In the following we will use $dr_{hi}(n_i)$ to be the high rail of node n_i, and $dr_{lo}(n_i)$ to be the low rail of n_i. So in our example, $dr_{hi}(n_i) = h[\mathcal{V}]$ and $dr_{lo}(n_i) = l[\mathcal{V}]$. Recall that the square brackets merely signify the set of free variables a formula depends on.

With this notation, the node n_i has the concrete value tt if and only if

$$dr_{hi}(n_i) \wedge \overline{dr_{lo}(n_i)},$$

and dually the concrete value ff if and only if

$$\overline{dr_{hi}(n_i)} \wedge dr_{lo}(n_i),$$

We will refer to these conditions as $when_{\mathsf{tt}}(n_i)$ and $when_{\mathsf{ff}}(n_i)$ respectively.

When running STE, the circuit is simulated and each node is given a dual rail value. The responsibility algorithm we present works on the high and low rail of each such node. Therefore it depends on STE having been run before the responsibility is calculated. But this is already necessary for determining which X-possible nodes exist and cause weak disagreements. Hence this does not introduce additional costs, except for keeping a list of all node values arising during simulation. The Forte environment, which we used for our experiments, supports keeping track of this information for all or selected nodes when running STE.

Chockler et al. [3] assume that input nodes either have a concrete value, or are

driven by a variable, i.e., $dr_{hi}(n_i) = v_i$ and $dr_{lo}(n_i) = \overline{v_i}$. We now propose an approximate degree of responsibility calculation, which can handle arbitrary symbolic values on nodes. Figure 6.7 includes both the previous definition by Chockler et al., as well as ours. We decided to make our calculations consistent with the previous approach in all corner cases, so that it constitutes a clear generalisation. At the end of this section (page 152) we show this to be true.

The previous approach can handle inputs with concrete values, indeterminate values, or the value of a single variable. It does not provide a cost for all other symbolic values. Intuitively speaking, our approach assigns the cost to such symbolic values by bridging the gap between the value assigned to indeterminate values and symbolic constants. The more often a node is indeterminate, the closer the cost is to that of an indeterminate value, and the more often it is concrete due to its symbolic value, the closer its value is to that of a symbolic constant, which for all valuations is concrete.

For this we define $sc(f)$, which is short for *satisfiability count* and denotes the number of ways in which f can be satisfied, i.e., the number of assignments to the free variables $\mathcal{V}$ of f, such that $f(\overleftarrow{\mathcal{V}}) = \mathsf{tt}$.

$$sc(f[\mathcal{V}]) = |\{\overleftarrow{\mathcal{V}} : \ f(\overleftarrow{\mathcal{V}}) = \mathsf{tt}\}|$$

with two special cases

$$sc(\mathsf{ff}) = 0$$
$$sc(\mathsf{tt}) = 1$$

Figure 6.6: Definition of satisfiability count

This definition depends on the set of free variables not being empty. There are two special cases where that set is empty, namely if f has a constant value. We set the value of $sc(f)$, so that these cases match the previous definition of c_0 and c_1 in [3]: $sc(\mathsf{tt}) = 1$ and $sc(\mathsf{ff}) = 0$.

Note that using the high rail and the low rail when determining the values of c_0 and c_1 corresponds to the case split Chockler et al. do on the value of $A(j)$. The difference is that the dual rail encoding can capture more values to drive an input with than simply looking at the antecedent without allowing complex guards to restrict by, as done in [3]. Differently put, we also handle antecedents where $A(j) = (when_{\mathsf{tt}}(j) \rightarrowtriangle 1)$ and $(when_{\mathsf{ff}}(j) \rightarrowtriangle 0)$.

For example, the previous approach could not handle an input being driven with

$c_1(j)$	$=\infty$	if $A(j)=0$	$c_1(j)$	$=\infty$	if $A(j)=0$
	$=0$	if $A(j)=1$			
	$=2$	if $A(j)=\mathrm{X}$		$=2-2\cdot\frac{sc(when_{\mathrm{tt}}(j)[\mathcal{V}])}{2^{\lvert\mathcal{V}\rvert}}$	
	$=1$	if $A(j)\in\mathcal{V}$			if $A(j)\neq 0$
$c_1(\text{NOT } f)$	$=c_0(f)$		as before		
$c_1(f \text{ AND } g)$	$=c_1(f)+c_1(g)$		as before		

$c_0(j)$	$=\infty$	if $A(j)=1$	$c_0(j)$	$=\infty$	if $A(j)=1$
	$=0$	if $A(j)=0$			
	$=2$	if $A(j)=\mathrm{X}$		$=2-2\cdot\frac{sc(when_{\mathrm{ff}}(j)[\mathcal{V}])}{2^{\lvert\mathcal{V}\rvert}}$	
	$=1$	if $A(j)\in\mathcal{V}$			if $A(j)\neq 1$
$c_0(\text{NOT } f)$	$=c_1(f)$		as before		
$c_0(f \text{ AND } g)$	$=min\{c_0(f),c_0(g)\}$		as before		

Figure 6.7: Computation of c_1 and c_0 as proposed in [3] (left) and our generalisation to arbitrary symbolic values (right)

the value $v_1 \wedge v_2$, which corresponds to the antecedent

$$i_1 \text{ is } v_1 \wedge v_2, \quad \text{or} \quad v_1 \wedge v_2 \rightarrowtriangle i_1 \text{ is } 1 \text{ and } \overline{v_1 \wedge v_2} \rightarrowtriangle i_1 \text{ is } 0.$$

With this new approach we get $c_1(v_1 \wedge v_2) = 2 - 2 \cdot \frac{1}{4} = 1\frac{1}{2}$, because there is only one possible assignment to make the expression $v_1 \wedge v_2$ true, namely both v_1 and v_2 have to have a high value, and $when_{\mathrm{tt}}(i_1)$ has two free variables, i.e., $2^{|\{v_1,v_2\}|} = 4$.

As noted before, we chose to use a heuristic for the costs c_1 and c_0, so that in all cases supported by Chockler et al. our values coincide. This makes clear that our approach is more general, and the consistency eases comparability. Else we could have easily used other values, e.g. spread out the cost for symbolic values to a larger interval than [1,2].

Example: Comparing the corner cases

1. Antecedent: $(i_1 \text{ is } 0)$
   This is a case that is specially treated in both definitions.

Chockler et al.	$c_1(i_2)$	$=\infty$
Our generalisation	$c_1(i_2)$	$=\infty$

2. Antecedent: $(i_1 \text{ is } 1)$
   Then the dual rail encoding for the value of i_1 is $(\mathsf{tt}, \mathsf{ff})$ and thus $when_{\mathsf{tt}}(i_1) = \mathsf{tt} \wedge \overline{\mathsf{ff}} = \mathsf{tt}$. Hence,

Chockler et al.	$c_1(i_1)$	$= 0$
Our generalisation	$c_1(i_1)$	$= 2 - 2 \cdot \frac{1}{2^0} = 0$

3. Antecedent: $(i_1 \text{ is X})$
   Then $A(i_2) = \text{X}$, which corresponds to the dual rail encoding $(\mathsf{tt}, \mathsf{tt})$.
   Thus $when_{\mathsf{tt}}(i_2) = \mathsf{tt} \wedge \overline{\mathsf{tt}} = \mathsf{ff}$. Hence,

Chockler et al.	$c_1(i_2)$	$= 2$
Our generalisation	$c_1(i_1)$	$= 2 - 2 \cdot \frac{0}{2^0} = 2$

4. Antecedent: $(i_1 \text{ is } v_1)$
   Then the dual rail encoding for the value of i_1 is $(v_1, \overline{v_1})$ and thus $when_{\mathsf{tt}}(i_1) = v_1 \wedge \overline{\overline{v_1}} = v_1$. Hence,

Chockler et al.	$c_1(i_1)$	$= 1$
Our generalisation	$c_1(i_1)$	$= 2 - 2 \cdot \frac{1}{2^1} = 1$

So all cases that are captured by Chockler et al. are given the same value in our generalisation. ★

6.2.2 Refinement of Abstraction Schemes

In the previous section we extended the approximate degree of responsibility algorithm for deciding which input to refine the abstraction scheme by. Recall that Chockler et al. [3] refine by driving the identified input with a variable. This means that the value of that input always has a determinate value, and will hopefully eliminate the weak disagreement that was the result of the previous STE run. While this may not be the case immediately, the refinement loop is guaranteed to eliminate the weak disagreement eventually. In the worst case we symbolically simulate every input, which cannot lead to a weak disagreement anymore.

But of course, driving all inputs with a variable was discarded, because it is too expensive a task in most cases. Indeed, the refinement step suggested by Chockler et al. can be a quite expensive modification, which can lead to a steep increase in time

and memory consumption. In the following we suggest more conservative options for refining abstractions.

Introducing More Restrictive Guards

Symbolic indexing allows making more careful, restrictive modifications in the refinement loop. Each STE run provides the cases in which the verification fails. Only in those cases do we actually have to modify which inputs are driven with a value so one option is to only drive an input then.

Suppose an STE run with the antecedent A results in a weak disagreement whenever the expression $\overline{res}$ is satisfied. Suppose further that we then identified that the input i_1 seems to be the most responsible for that weak disagreement. Then we can drive i_1 with a new indexing variable v_1 by adding

$$(\overline{res} \wedge v_1 \rightarrow i_1 \text{ is } 1) \text{ and } (\overline{res} \wedge \overline{v_1} \rightarrow i_1 \text{ is } 0)$$

to the previous antecedent. For simplicity the short form, $(\overline{res} \rightarrow i_1 \text{ is } v_1)$, can be used, leading to a modified antecedent A and $(\overline{res} \rightarrow i_1 \text{ is } v_1)$. In contrast, the previous approach lead to the antecedent A and $(i_1 \text{ is } v_1)$.

Here it is crucial that the antecedent still includes all previous indexing cases. The power of symbolic indexing lies in driving several nodes with possibly matching values. This is also a feature heavily used by our work in automatic abstraction discovery, as presented in Chapters 3 and 4. But this means that an input may be driven with different values. For example, suppose the antecedent states that $(f \rightarrow i \text{ is } 0)$ and $(g \rightarrow i \text{ is } 1)$. Then the updated antecedent as suggested by Chockler et al. is $(f \rightarrow i \text{ is } 0)$ and $(g \rightarrow i \text{ is } 1)$ and $(i \text{ is } r)$. So, when $f \wedge r$, or when $g \wedge \overline{r}$, then i is both 0 and 1. This overconstraint value constitutes an antecedent failure. It can be detected and the simulation skipped, but ideally such inconsistencies should be avoided directly. We can achieve this by adding a guard $\overline{f} \wedge \overline{g}$ for when to drive the input i with the variable r:

$$(f \rightarrow i \text{ is } 0) \text{ and } (g \rightarrow i \text{ is } 1) \text{ and } (\overline{f} \wedge \overline{g} \rightarrow i \text{ is } r),$$

or, if we want to drive the input in the weak disagreement cases only:

$$(f \rightarrow i \text{ is } 0) \text{ and } (g \rightarrow i \text{ is } 1) \text{ and } (\overline{res} \wedge \overline{f} \wedge \overline{g} \rightarrow i \text{ is } r).$$

By introducing these guards, $\overline{f} \wedge \overline{g} \wedge h$, we restrict when specific inputs are run with a symbolic value, rather than always doing so. The guard $h = \overline{res}$ ensures the input is driven in the weak disagreement cases only. Note, though, that we could use any guard h. Using a more restrictive one, i.e., $h \rightarrow \overline{res}$ is not a tautology, would not suit our cause, as the next STE run would surface a weak disagreement in the cases we did not include in the guard. But we may want to add a symbolic value for our input in additional cases, the advantage being that the guard may be simplified and not require as many variables. Indeed, seen from this perspective, an approach similar to that of Chockler et al. always would use simplest guard possible: $h = \mathsf{tt}$.

But Chockler et al. assumed no inputs are initially driven, and any update leads to inputs always being driven, this coinciding of symbolic indexing is a scenario that they did not have to consider. As soon as we allow more elaborate indexing schemes we may discover that the input i is driven in some cases, but not in all cases an STE run requires it for determining a definitive result.

Refining the antecedent needs to be done very carefully, as provided above. Another example surfaces how easily antecedent updates can lead to errors. Suppose the antecedent states that $f \rightarrowtriangle i_1 \ \mathsf{is}\ v_1$ and we identify i_1 as the refinement candidate in the weak disagreement case $\overline{res}$. Then we can update the antecedent by instead including

$$(f \rightarrowtriangle i_1 \ \mathsf{is}\ v_1) \ \mathsf{and}\ (\overline{f} \wedge \overline{res} \rightarrowtriangle i_1 \ \mathsf{is}\ v_2),$$

where v_2 is a fresh indexing variable. One might be tempted to instead reuse the variable v_1, thus leading to

$$f \vee \overline{res} \rightarrowtriangle i_1 \ \mathsf{is}\ v_1,$$

but this may fail. It may be that, for optimisation reasons, the antecedent reuses v_1 already when $\overline{f}$ holds. For example, Section 4.7 describes such an optimisation to reduce the number of variables introduced. So in general the more careful approach of adding a new variable is necessary. This, however, may also increase the complexity of the STE run. The approach described in the next section is one way of addressing this shortcoming.

Compiling a New Abstraction Scheme

Refining the abstraction scheme by adding cases when inputs are defined is, in some sense, only mending the abstraction, rather than attempting to improve it at its base. One observation to make is that Chockler et al.'s approach of always driving the identified input with a variable relates strongly with our support of defining symbolic constants. Recall that when running the automatic abstraction algorithm, we allowed the user to specify which inputs were to be driven with a value in all cases, and which thus could be treated similar to constants. This had the advantage that we could use the knowledge that nodes always had a determinate value to optimise the rest of the abstraction scheme.

Hence, another option is to rerun the automatic abstraction discovery algorithm, auto_abstract, this time identifying i_1 to be in the set of symbolic constants. This is of course a more expensive operation than simply adding one more guard to the antecedent, but immediately pays off if the resulting STE run requires less time and memory to complete.

Speaking in terms of number of indexing variables, the difference is that the previous approach introduces a new indexing variable for each refinement candidate. This indexing variable will not be reused for any other cause. When running the automatic abstraction algorithm, on the other hand, the variable may, and usually will, also be used for other guards. This is noteworthy, because the number of indexing variables is sometimes used to estimate the cost of an abstraction scheme, and fewer is perceived to be better.

Striking a Balance between the Approaches

We have identified these different refinement steps once a refinement candidate has been selected:

1. Drive i_1 with a variable whenever it was not driven before.
2. Drive i_1 with a variable in all weak disagreement cases in which it was not driven before.
3. Drive i_1 with a variable, when it was not driven before, and guarded by an expression that captures at least all weak disagreement cases.
4. Compute a new abstraction scheme while specifying i_1 as a symbolic constant.

Which of these approaches is best depends on both the model you are verifying and its specification. The cost of each approach is different. In Approach 1, we are driving the input in all cases, so we have that overhead in all cases. In Approach 2 we are additionally driving the input only when it might actually make a difference. But the guard can be quite a complex expression and this introduces a different kind of overhead. Approach 3 can reduce that complexity, but has the risk of introducing similar costs as in Approach 1. Finally, computing a new abstraction scheme disregards all work put into creating the previous abstraction scheme and starts from scratch – with the sole exception of having identified that i_1 needs to be treated as a symbolic constant.

It is worth noting that we need not always use the same refinement step. Especially Approaches 1, 2, 3 are interchangeable in a refinement loop. For example, you could define a condition under which the refinement candidate shall always be driven with a symbolic value, and when to add a guard. One such condition would be to investigate in how many cases the guard is satisfied, i.e., how close to 1 is $\frac{sc(f[\mathcal{V}])}{2^{|\mathcal{V}|}}$? If the guard is almost always satisfied, then the additional cost of always driving the input with a value is small. But the saved work for not having complex guards may be significant. So applying Approach 1 may be the best option. Another, cruder approach is to count the number of variables used in the guard and judge by that. If a specific threshold is surpassed using Approaches 1 or 3 may lead to lower verification costs than using Approach 2.

Recall that Approach 3 is based on the fact that there is no formal requirement to restrict driving the refinement candidate in the weak disagreement cases only. Any guard g that satisfies the property $g \rightarrow f$ will lead to both formally correct results, as well as guaranteeing that the refinement loop will terminate eventually. In particular, g may have considerably fewer variables than f. Of course, identifying a simple expression that at the same time covers only a few more cases than f has its own difficulties. But it is a possible approach, which lives between the two extremes, Approach 1 and Approach 2.

Switching between adding onto the abstraction scheme, and rerunning the automatic abstraction algorithm, on the other hand, is more difficult. You can memorise the refinement candidates in each step and then – if the abstraction becomes too weighty – use that information to start from scratch. However, this does mean that you potentially have many symbolic constants. Using Approach 4 may be worthwhile especially in the first iterations of the loop, but in later iterations the cost becomes much higher. In essence, it is better than always using Chockler et al.'s approach, but

once we do smart modifications as suggested above, the gain may not be as prominent anymore.

6.2.3 Example

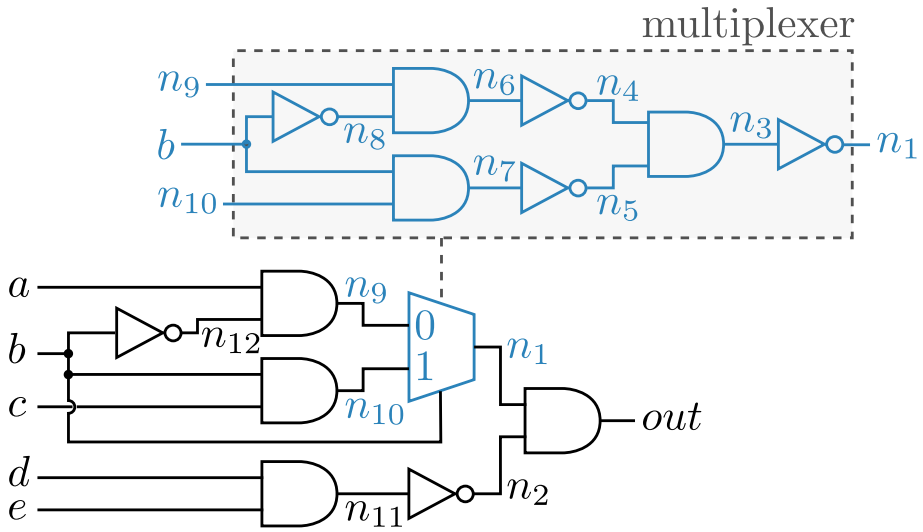


Figure 6.8: Graphical representation of the hardware model M with output *out* where $out = (if\ b\ then\ b \wedge c\ else\ a \wedge \overline{b})) \wedge \overline{d \wedge e}$

Consider the circuit shown in Figure 6.8. On closer examination we discover that its input-output behaviour is equivalent to the expression $(a \wedge \overline{b} \ \vee \ b \wedge c) \ \wedge \ \overline{d \wedge e}$. Thus, in some cases the value of b is not necessary for determining the output value. Each partial input combination provided in Figure 6.9 is sufficient to determine the value of *out*.

The following reindexing relation captures exactly those partial input combinations and satisfies the coverage condition as seen in Theorem 4.1:

$$
\begin{array}{lll}
(a, & o \wedge \overline{x_1} \vee \overline{o} \wedge x_1 \wedge \overline{x_2}, & o \wedge x_1 \vee \overline{o} \wedge x_2), \\
(b, & o \wedge x_1 \vee \overline{o} \wedge x_1 \wedge \overline{x_2}, & o \wedge \overline{x_1} \wedge x_2 \vee \overline{o} \wedge x_1 \wedge x_2), \\
(c, & o \wedge \overline{x_1} \wedge \overline{x_2} \vee o \wedge x_1 \vee \overline{o} \wedge x_1 \wedge x_2), & o \wedge \overline{x_1} \wedge x_2 \vee \overline{o} \wedge \overline{x_1} \wedge x_2 \vee \overline{o} \wedge x_1 \wedge \overline{x_2} \\
(d, & \overline{o} \wedge \overline{x_1} \wedge \overline{x_2}, & o \wedge \overline{x_1} \wedge \overline{x_3} \vee o \wedge x_1 \wedge \overline{x_2}), \\
(e, & \overline{o} \wedge \overline{x_1} \wedge \overline{x_2}, & o \wedge \overline{x_1} \wedge x_3 \vee o \wedge x_1 \wedge x_2),
\end{array}
$$

Figure 6.9 lists all partial input combinations, as well as which assignments to o, x_1, x_2, and x_3 cover them.

a	b	c	d	e	out		o	x_1	x_2	x_3
1		1	0		1		1	0	0	0
1		1		0	1		1	0	0	1
1	0	0	0		1		1	0	1	0
1	0	0		0	1		1	0	1	1
0	1	1	0		1		1	1	0	
0	1	1		0	1		1	1	1	
			1	1	0		0	0	0	
0		0			0		0	0	1	
1	1	0			0		0	1	0	
0	0	1			0		0	1	1	

Figure 6.9: Partial input combinations, and under which assignments to o, x_1, x_2, and x_3 the relation covers them

However, upon running $R \models A^R \Rightarrow (out \text{ is } o)_R$, where A is the antecedent, which drives each of the input nodes with a variable, we discover that over-abstraction occurs. In particular, STE returns a weak disagreement when $o \wedge \overline{x_1} \wedge \overline{x_2}$, which corresponds to the partial input combinations 1X10X and 1X1X0. These are the cases where the multiplexer returns a high value irrespective of the value of b, the condition, because both the if-branch and the else-branch have a high value. However, the simulator requires the value of b even if both other inputs of the multiplexer are equal.

We now want to refine the relation, so that this over-abstraction is eliminated. For this, we use our adapted abstraction refinement algorithm. In particular, observe that we are evaluating TEL formulae which are not supported by [3]. As we only described how to determine the approximate degree of responsibility of NOT- and AND-gates, we have to represent the multiplexer as a small circuit using only these gates and with the same input-output-behaviour, for example as seen in Figure 6.8. In Chapter 7 we will see how we can handle the multiplexer directly, rather than through this workaround.

We now want to determine the refinement candidate, i.e., the input presumably most responsible for the output being indeterminate. For this, we first run STE restricted to the weak disagreement cases only. Thus, we will receive a run where the output is always indeterminate, and where the inputs are driven as described in the

relation when $o \wedge \overline{x_1} \wedge \overline{x_2}$:

node	*value*	*dualrail*
a	1	$(\mathsf{tt}, \mathsf{ff})$
b	X	$(\mathsf{ff}, \mathsf{ff})$
c	1	$(\mathsf{tt}, \mathsf{ff})$
d	$\overline{x_3} \to 0$	(x_3, tt)
e	$x_3 \to 0$	$(\overline{x_3}, \mathsf{tt})$

This leads to the following approximate degree of responsibility values.

$$
\begin{array}{ll}
s(a, out) & = s(a, n_1) + c_1(n_2) \\
c_1(n_2) & = c_0(n_1 1) = \min\{c_0(d), c_0(e)\} \\
c_0(d) & = 2 - 2 \cdot \frac{sc(when_{\mathsf{ff}}(d))}{2^{|\{x_3\}|}} = 2 - 2 \cdot \frac{sc(\overline{x_3} \wedge \mathsf{tt})}{2} = 2 - 2 \cdot \frac{1}{2} = 1 \\
c_0(e) & = 2 - 2 \cdot \frac{sc(when_{\mathsf{ff}}(d))}{2^{|\{x_3\}|}} = 2 - 2 \cdot \frac{sc(x_3 \wedge \mathsf{tt})}{2} = 2 - 2 \cdot \frac{1}{2} = 1 \\
s(a, n_1) & = s(a, n_3) = s(a, n_4) + c_1(n_5) \\
c_1(n_5) & = c_0(n_7) = \min\{c_0(b), c_0(n_1 0)\} \\
 & = \min\{c_0(b), \min\{c_0(b), c_0(c)\}\} = \min\{2, \min\{2, \infty\}\} = 2 \\
s(a, n_4) & = s(a, n_6) = s(a, n_9) + c_1(n_8) = s(a, n_9) + c_0(b) = s(a, n_9) + 2 \\
s(a, n_9) & = s(a, a) + c_1(n_1 2) = s(a, a) + c_0(b) = 0 + 2 = 2 \\
\boldsymbol{s(a, out)} & = 2 + 2 + 2 + 1 = \mathbf{7}, \quad \text{similarly } \boldsymbol{s(c, out) = 7} \\
\hline
s(b, out) & = s(b, n_1) + c_1(n_2) \\
s(b, n_1) & = s(b, n_3) = \tfrac{1}{2} \cdot \big(s(b, n_4) + s(b, n_5)\big) \\
s(b, n_4) & = s(b, n_6) = \tfrac{1}{2} \cdot \big(s(b, n_9) + s(b, n_8)\big) = \tfrac{1}{2} \cdot \big(s(b, n_9) + s(b, b)\big) \\
s(b, n_9) & = c_1(a) + s(b, n_1 2) = c_1(a) + s(b, b) = 0 + 0 \\
s(b, n_5) & = s(b, n_7) = \tfrac{1}{2} \cdot \big(s(b, b) + s(b, n_{10})\big) \\
s(b, n_{10}) & = s(b, b) + c_1(c) = 0 + 0 \\
\boldsymbol{s(b, out)} & = 0 + 1 = \mathbf{1} \\
\hline
s(d, out) & = c_1(n_1) + s(d, n_2) \\
c_1(n_1) & = c_0(n_3) = \min\{c_0(n_4), c_0(n_5)\} = \min\{c_1(n_6), c_1(n_7)\} \\
 & = \min\{c_1(n_9) + c_1(n_8), c_1(b) + c_1(n_1 0)\} \\
 & = \min\{c_1(a) + c_0(b) + c_0(b), c_1(b) + c_1(b) + c_1(c)\} \\
 & = \min\{0 + 2 + 2, 2 + 2 + 0\} = 4 \\
s(d, n_2) & = s(d, n_1 1) = s(d, d) + c_1(e) \\
c_1(e) & = 2 - 2 \cdot \frac{sc(when_{\mathsf{tt}}(e))}{2^{|\{x_3\}|}} = 2 - 2 \cdot \frac{0}{2} = 2 \\
\boldsymbol{s(d, out)} & = 4 + 2 = \mathbf{6}, \quad \text{similarly } \boldsymbol{s(e, out) = 6}
\end{array}
$$

So the algorithm delivers that the input b is most responsible for the output being indeterminate, as desired.

Thus, using Approach 2, we can refine the relation in b as follows:

$$\begin{aligned}(b, \quad & o \wedge x_1 \ \vee \ \overline{o} \wedge x_1 \wedge \overline{x_2} \ \vee \ r_1 \wedge o \wedge \overline{x_1} \wedge \overline{x_2}, \\ & o \wedge \overline{x_1} \wedge x_2 \ \vee \ \overline{o} \wedge x_1 \wedge x_2 \ \vee \ \overline{r_1} \wedge o \wedge \overline{x_1} \wedge \overline{x_2})\end{aligned}$$

The verification passes when running it with this adapted relation.

Note that here we only drive b as often as before, plus in the weak disagreement case, $o \wedge \overline{x_1} \wedge \overline{x_2}$, and then with a new variable r_1. So both differences to the previous approach can already be seen in this small example. First, we are able to evaluate non-trivial symbolic indexings, and second we can drive inputs in only the cases where over-abstraction occurs, rather than always. The approach presented by Chockler et al. [3] could not handle even this form of non-trivial symbolic indexing, and suggests refining the relation by driving b always, i.e., the relation in b would have been adapted to $(b, r_1, \overline{r_1})$.

6.3 Summary

In this chapter we introduced an approach to automatically refining abstraction schemes when over-abstraction occurs. In particular, our work is based on, but goes beyond previous work done by Chockler et al. [3]. We generalise their approach in two essential and significant aspects. First, we argue how to select refinement candidates for abstraction schemes that use non-trivial symbolic indexings. While previously only abstractions with constant Boolean values, or with simple variables were supported, our approach allows the handling of arbitrary Boolean expressions as guards. Second, we suggest how to alter the abstraction more conservatively, thus potentially reducing the cost increase caused by driving more inputs with values after refinement. Where Chockler et al. modified the abstraction scheme by driving the refinement candidate with a symbolic value in all cases, we allow driving it only when necessary. This is achieved by introducing guards appropriate to the over-abstraction observed.

Furthermore, we suggest another methodology of refining abstractions. Rather than refining the indexing by adding onto it cases in which specific inputs are driven with a value, we propose creating a new abstraction scheme using our auto_abstract algorithm while defining the identified refinement candidates as symbolic constants.

In Chapter 7 we will see how selecting refinement candidates can be further gener-

alised and improved, to then show the effectiveness of our approach in Chapter 8 by verifying the CAM we already verified in Chapter 5, with the important modification of not defining any symbolic constants ahead of time. To compare the approaches, we will also refine the initial abstraction by always driving the identified inputs with a variable, or only in the weak disagreement cases. We will see that while all these approaches work, compiling a new abstraction using the refinement candidates as symbolic constants leads to much better execution times.

Chapter 7

Abstraction Refinement with Arbitrary Gates

In Chapter 6 we examined how the approach to abstraction refinement as presented in [3] can be generalised to incorporate symbolic indexing, and in particular how to thus create better abstraction schemes. Another improvement to the algorithm we can make – independently of the previous generalisation – is how gates are analysed in the responsibility calculation.

The abstraction refinement introduced by Chockler et al. is based on determining which input is most responsible for a chosen, X-possible output. In particular, they describe how to compute the approximate degree of responsibility for NOT and AND gates. This is sufficient to analyse any other gate by breaking it down into an equivalent, small circuit that has the same input-output behaviour and consists of only NOT and AND gates. But this may result in less accurate responsibility calculations than possible when analysing the gate directly. This is especially true because approximations are computed, and when analysing many gates these errors add up. So finding a general approach to analysing arbitrary gates allows such errors to be avoided, and thus leads to better abstraction refinement.

In the following we study single gates and these gates in context as part of a circuit. For this it is helpful to not only talk about the inputs and output of the whole circuit, but also those of a single gate. For simplicity, we use the same notation for single gates as we have for circuits. And indeed, a single gate can be seen as a very small circuit. For a gate g, we denote the set of all its inputs by $\mathcal{N}_g$. In particular, we talk about partial input combinations as before, and compare and order them likewise.

When examining the gate by itself and as part of a bigger circuit, it is important

to formalise the move from and to a different context. For this we use the following notation. A model is identified by the expression f describing it. The outermost gate g of the circuit is denoted by $deconstruct(f)$. The fanins to g are called f_n for each $n \in \mathcal{N}_g$, and the output keeps the same name. Note that g is a gate, whereas f_n are each circuits again. The next outermost gate of each f_n can then be analysed, thus allowing us to recursive over the circuit.

Example: Notational Overview
Figure 7.1 shows the notation used in this chapter with respect to identifying gates and subcircuits. The three-input OR is the outermost gate, $deconstruct(f)$. This gate has three nodes n_1, n_2, n_3 that determine its value. For this specific circuit the subcircuits f_1, f_2, f_3 respectively specify which values feed into the inputs of that 3OR-gate. ★

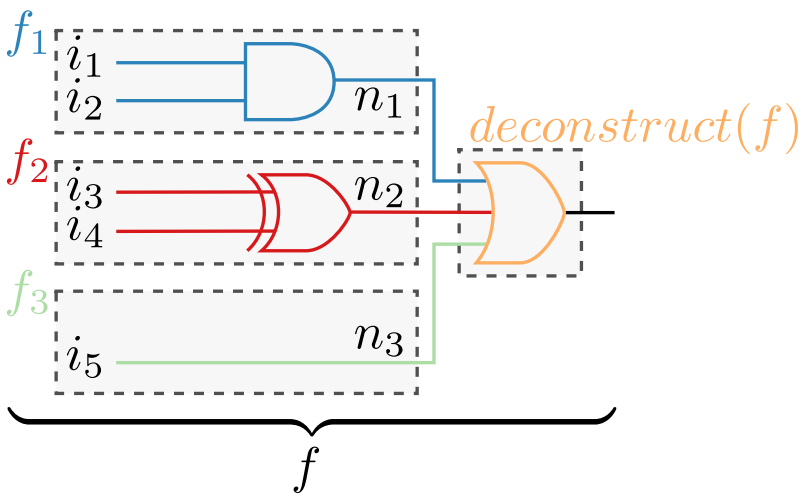


Figure 7.1: A graphical representation of $f = (i_1$ AND $i_2)$ 3OR $(i_3$ XOR $i_4)$ 3OR i_5

When giving a general approach for arbitrary gates, we need to decide on a representation for these gates. A gate can be defined by the partial input combinations that specify when it returns a high output. This definition of behaviour is commonly used in STE implementations. Hence in the following we state how the approximate degree of responsibility can to be calculated based on that defining list of partial input combinations.

The chapter is divided into several parts, which relate to the different calculations necessary to determine an approximate degree of responsibility. The work in [3] depends on calculating three values: $c_1(f)$, the cost of forcing the inputs to values

that lead to f having a high output, which we generalise in Section 7.1; $c_0(f)$, the cost of getting f to have a low output, generalised in Section 7.2; and $s(i, f)$, the inverse of the degree of responsibility of a primary input i for the output of f. Various aspects that need to be adapted for the calculation of $s(i, f)$ are discussed in Section 7.3. By giving definitions for all three calculations we thus present a fully general approach to analysing gates for determining a refinement candidate. This ultimately delivers a better automatic abstraction refinement loop.

7.1 The Cost of Having a High Output

When computing the approximate degree of responsibility of an input i of f to its X-possible output o, we need to establish a counterfactual dependency between i and o. This may require us to change and in particular concretise the values of the other inputs of f. Each change is connected with a cost, which may vary depending on what kind of modification we apply. By $c_1(f)$ we denote the cost of changing the values of the inputs of f, so that f has a high output. As before, we want to apply a set of changes that lead to a high output at minimal cost.

Note that when computing $s(i, f)$ we usually calculate $c_1(f_{sub})$, where f_{sub} is a subcircuit of f whose fanin does not include i. This does not affect how c_1 is generally calculated, though, and there are special cases where i is in the fanin, e.g. when $s(i, f_{sub}) = \infty$ and we decide to exchange it with $c_1(f_{sub})$, as already seen in Figure 6.3.

The computation of $c_1(f)$ is based on the defining input combinations for the gates which build f. For each gate g, we call the set of partial input combinations that lead to a high output $\mathcal{H}_g$, and $g(P)$ denotes the output value of g provided its inputs have the values specified by the partial input combination P. Thus, formally:

$$\mathcal{H}_g = \{H : \mathcal{N}_g \to \{\mathsf{tt}, \mathsf{ff}, \mathrm{X}\} \text{ such that } g(H) = \mathsf{tt}\}$$

Furthermore, we can assume that $\mathcal{H}_g$ is minimal by taking the greatest lower bound of all such sets. This essentially means that all input combinations that can be collapsed are collapsed.

In particular, no element can be removed without loss of information, i.e., for no two distinct elements $H_1, H_2 \in \mathcal{H}_g$ does one have strictly less information than the

other:

$$\forall H_1, H_2 \in \mathcal{H}_g : \; H_1 \sqsubseteq H_2 \rightarrow H_1 = H_2.$$

Additionally, no two distinct elements H_1 and H_2 can be replaced by a less determinate partial input combination, e.g. by setting the value of $m \in \mathcal{N}_g$ to X. This would be possible when

$$\forall n \in \mathcal{N}_g \backslash \{m\} : H_1(n) = H_2(n).$$

In essence, this ensures that whenever we can leave an input of the gate unspecified, we leave it unspecified. And we can leave an input unspecified, if irrespective of whether it has a high, low, or indeterminate value, the output of the gate does not change.

For any partial input combination P on the nodes $\mathcal{N}_g$, let $c(g, P)$ denote the cost of changing the inputs of g to the values specified in P. For determining this cost we do not just require the static information on what gate we are dealing with and its defining partial input combinations, but also in which context the gate stands. In particular, when g is part of a model f, the inputs of g are not primary inputs of f, but rather have values determined by subcircuits f_{sub} of f.

But supposing that the circuit f has a single gate g only, $c(g, P)$ delivers a way of calculating $c_1(f)$,

$$c_1(f) = \min_{H \in \mathcal{H}_g} c(g, H).$$

This equation holds, as (1) the inputs of g are exactly the primary inputs of f, and (2) we assumed that $\mathcal{H}_g$ is minimal, and thus each $c(g, H)$ considered does not factor in unnecessary changes which would increase the cost determined.

More generally, let $g = \mathit{deconstruct}(f)$ be the outermost gate of f, and f_i be the subcircuits of f, which determine the values of the inputs of g in this case. By $c(H(i), f_i)$ we denote the cost of ensuring that f_i outputs the value specified in H. Thus,

$$c_1(f) = \min_{H \in \mathcal{H}_g} \sum_i c(H(i), f_i).$$

The cost of changing the value of f_i as specified in H can be determined recursively:

$$c(H(i), f_i) = \begin{cases} c_1(f_i) & \text{if } H(i) = \mathsf{tt} \\ c_0(f_i) & \text{if } H(i) = \mathsf{ff} \\ 0 & \text{if } H(i) = \mathrm{X} \end{cases}$$

Here the index i always specifies which input of g we are examining, and likewise which subcircuit determines the value of that input. This recursion terminates on indeterminate values, or whenever we reach a model that consists of only a primary input of f. At this point the cost of changing an input to a specific value depends on the heuristic used. In particular, in this dissertation, we apply the heuristic introduced in Chapter 6 (Figure 6.7) whenever calculating concrete values.

Example: Calculating c_1

Consider the model $f = (i_1 \text{ AND } i_2) \text{ 3OR } (i_3 \text{ XOR } i_4) \text{ 3OR } i_5$, also displayed in

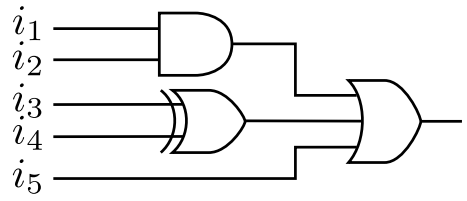


Figure 7.2: A graphical representation of $(i_1 \text{ AND } i_2) \text{ 3OR } (i_3 \text{ XOR } i_4) \text{ 3OR } i_5$

Figure 7.2. The outermost gate, *deconstruct*(f), is a three-input OR-gate, and the subcircuits that determine the values of the inputs to that three-input OR-gate are $f_1 = i_1 \text{ AND } i_2$, $f_2 = i_3 \text{ XOR } i_4$, and $f_3 = i_5$. Furthermore,

$$\mathcal{H}_{3\mathrm{OR}} = \{1\mathrm{XX}, \mathrm{X1X}, \mathrm{XX1}\}$$

and thus

$$\begin{aligned} c_1(f) = \; & \min\{ \\ & \quad c(1, i_1 \text{ AND } i_2) \; + c(\mathrm{X}, i_3 \text{ XOR } i_4) \; + c(\mathrm{X}, i_5), \\ & \quad c(\mathrm{X}, i_1 \text{ AND } i_2) \; + c(1, i_3 \text{ XOR } i_4) \; + c(\mathrm{X}, i_5), \\ & \quad c(\mathrm{X}, i_1 \text{ AND } i_2) \; + c(\mathrm{X}, i_3 \text{ XOR } i_4) \; + c(1, i_5) \\ & \} \\ = \; & \min\{ \; c(1, i_1 \text{ AND } i_2), \; c(1, i_3 \text{ XOR } i_4), \; c(1, i_5)\} \end{aligned}$$

as $c(\mathrm{X}, f) = 0$ for all models f. We then have to continue with the next step in the recursion. For this, we require the defining input combinations for AND and XOR:

$$\mathcal{H}_{\mathrm{AND}} = \{11\}, \qquad \mathcal{H}_{\mathrm{XOR}} = \{1\mathrm{X}, \mathrm{X}1\}$$

Thus, we receive

$$c_1(f) = \min\{c_1(i_1) + c_1(i_2), \quad \min\{c_1(i_3), c_1(i_4)\}, \quad c_1(i_5)\}.$$

The actual value of $c_1(f)$ now only depends on the antecedent, and the heuristic used for the cost of changing the values each input is driven with by the antecedent. ★

7.2 The Cost of Having a Low Output

The calculation of c_0 and c_1 are very similar. The main difference we have to tackle is that gates are defined by the partial input combinations that lead to a high value. In the computation of c_0 we require the exact opposite: the set of partial input combinations that lead to a low output.

$$\mathcal{L}_g = \{L : \mathcal{N}_g \to \{\mathsf{tt}, \mathsf{ff}, \mathrm{X}\} : g(L) = \mathsf{ff}\}$$

As before, we want this set to be minimal to get a more accurate approximation. Then c_0 can be computed by

$$c_0(f) = \min_{L \in \mathcal{L}_g} c(g, L).$$

We can compute $\mathcal{L}_g$ using the Quine-McCluskey method [75]:

1. Start with the set that includes all fully concrete input combinations, $\mathcal{L} = \{L : \mathcal{N}_g \to \{\mathsf{tt}, \mathsf{ff}\}\}$
2. To receive $\mathcal{L}'$, for each $H \in \mathcal{H}_g$ remove all input combinations that lead to a high output, i.e., $\mathcal{L}' = \mathcal{L} \setminus \bigcup_{H \in \mathcal{H}_g} \{K : \mathcal{N}_i \to \{\mathsf{tt}, \mathsf{ff}\} : \ \forall n \in \mathcal{N}_g : \ H(n) \sqsubseteq K(n)\}$
3. Collapse all remaining fully input combinations to a minimal set of partial input combinations.
   (a) Start with $\mathcal{L}''$ and $\mathcal{U}$ being the empty set. In $\mathcal{L}''$ we collect a new set of assignments that is less determinate and covers all assignments of $\mathcal{L}'$. In

$\mathcal{U}$ we keep track of which assignments we have already covered to ensure that we reach full coverage in the end.

(b) For each pair $L_1, L_2 \in \mathcal{L}'$ that differs in only one assignment, namely to the node m: add both to $\mathcal{U}$. Further, add L_3 to $\mathcal{L}''$, where

$$L_3(n) = \begin{cases} \mathrm{X} & \text{if } n = m \\ L_1(n) & \text{otherwise} \end{cases}$$

(c) Add all elements of $\mathcal{L}' \backslash \mathcal{U}$ to $\mathcal{L}''$

(d) Repeat the loop with $\mathcal{L}''$ instead of $\mathcal{L}'$ until a fixed point is reached. This occurs once no more pairs that only differ in one assignment exist.

Example: Computing $\mathcal{L}_g$
Suppose we are given a gate f defined by

$$\begin{aligned} \mathcal{H}_g \; &= \{ \quad a \mapsto \mathsf{ff}, b \mapsto \mathsf{ff}, c \mapsto \mathrm{X} \\ &\qquad\; a \mapsto \mathsf{ff}, b \mapsto \mathrm{X}, c \mapsto \mathsf{ff} \\ &\qquad\; a \mapsto \mathrm{X}, b \mapsto \mathsf{ff}, c \mapsto \mathsf{tt} \quad \} \\ &= \{ \quad 00\mathrm{X},\ 0\mathrm{X}0,\ \mathrm{X}01\} \end{aligned}$$

Using the procedure above, we can determine that set as follows.

1. The set of all assignments is $\mathcal{L} = \{000, 001, 010, 011, 100, 101, 110, 111\}$
2. Remove all assignments covered by 00X to get $\{010, 011, 100, 101, 110, 111\}$
3. Remove all assignments covered by 0X0 to get $\{011, 100, 101, 110, 111\}$
4. Remove all assignments covered by X01 to get $\{011, 100, 110, 111\}$
5. So $\mathcal{L}' = \{011, 100, 110, 111\}$ and we start generating the set of collapsed assignments with $\mathcal{L}'' = \emptyset$
6. Collapse pair 011 and 111 to get $\mathcal{L}'' = \{\mathrm{X}11\}$ and $\mathcal{U} = \{011, 111\}$
7. Collapse pair 100 and 110 to get $\mathcal{L}'' = \{\mathrm{X}11, 1\mathrm{X}0\}$ and $\mathcal{U} = \{011, 100, 110, 111\}$
8. Collapse pair 110 and 111 to get $\mathcal{L}'' = \{\mathrm{X}11, 1\mathrm{X}0, 11\mathrm{X}\}$ and $\mathcal{U} = \{001, 100, 110, 111\}$
9. None of the elements of $\mathcal{L}''$ differ in only one node assignment anymore. Furthermore, $\mathcal{L}' \backslash \mathcal{U} = \emptyset$. Hence we have reached the fix point, $\mathcal{L}_g = \{\mathrm{X}11, 1\mathrm{X}0, 11\mathrm{X}\}$

★

Note that the set is only uniquely defined if we collapse all possible pairs. If instead we only required the final set to cover the same states as before, then two different sets would qualify in the example above:

- $\{\text{X11}, \text{1X0}\}$ by collapsing 011 and 111, and then 100 and 110
- $\{011, 100, \text{11X}\}$ by collapsing 110 and 111, and then adding the remaining assignments $\{011, 110\}$ that were not covered in the collapse step.

Collapsing as far as possible is more desirable, because it allows a more accurate evaluation of how costly it is to force the output of the gate to a specific value. For example, either of the assignments $\text{X11}, \text{1X0}, \text{11X}$ can be the least costly to enforce. It is always the case that any of the candidates might represent the assignment that is the least costly, because we ensured that none of the assignments is more concrete than necessary. Remember that only concrete node assignments add to the cost, whereas leaving a node value undefined comes for free.

This calculation needs to be done only once per type of gate, so it can be moved to a preprocessing step before even starting the verification of the model. Remember that before the calculation for a gate was done on a small circuit with the same input-output behaviour as the gate, but that only required NOT and AND gates. The cost of calculating our collapsed set of input combination is compensated by the cost of constructing the synthesised circuit, as well as analysing each gate of it. So by doing this calculation we get more accurate results in the calculation of the degree of responsibility without additional cost.

7.3 Determination of the Approximate Degree of Responsibility

Recall that in the calculation of $s(i, f)$ we want to determine how costly it is to enforce that the value of i determines the output of f. In the simple case that i is only in the fanin of one of the inputs of a gate we thus want to find two different partial input combinations: one shall lead to a high output, the other to a low output – and both shall only differ in the value of the input whose fanin includes i:

$$H, L : \mathcal{N}_g \to \{\mathsf{tt}, \mathsf{ff}, \text{X}\} : \; H(n) = L(n) \; \forall \mathcal{N}_g \backslash \{m\}\},$$

where i is in the fanin of $m \in \mathcal{N}_g$. Let D be the assignment that matches with L and H in all nodes but m, and let $D(m) = \mathrm{X}$. Then

$$s(i, f) = s(i, f_m) + \sum_{n:D(n)=\mathsf{tt}} c_1(f_n) + \sum_{n:D(n)=\mathsf{ff}} c_0(f_n),$$

where f_n denotes the expressions that feed into the outermost gate of f, *deconstruct*(f). As this deciding assignment creates counterfactual dependency between i and f, the cost of changing the situation to D is relevant to calculating $s(i, f)$. Specifically, they match if D is the only deciding assignment that exists. In contrast, if more than one deciding assignment exists, we need to average over the cost of each of the assignments.

Example: 3-input AND-gate
Let $f = f_1$ AND f_2 AND f_3, where f_1, f_2, and f_3 denote expressions that feed into the inputs of a 3-input AND-gate. Further assume that the primary input i of f is only in the fanin of f_1.

Then there is one pair of assignments where f_1 is the deciding input: $H(f_1) = H(f_2) = H(f_3) = \mathsf{tt}$ for a high output, and $H(f_1) = \mathsf{ff}, H(f_2) = H(f_3) = \mathsf{tt}$ for a low output. Hence, $D(f_1) = \mathrm{X}, D(f_2) = D(f_3) = \mathsf{tt}$. We thus calculate $s(i, f)$ as follows:

$$s(i, f) = s(i, f_1) + c_1(f_2) + c_1(f_3)$$

Observe that this matches with the definition provided by Chockler et al. in [3]. ★

The key to determining $s(i, f)$ is therefore directly linked to finding deciding input combinations. In the following we first explain how to determine the set of all deciding input combinations, then discuss how to calculate $s(i, f)$ based on that set, and finally generalising our approach to cases where i is in the fanin of more than one input of the gate *deconstruct*(f).

7.3.1 Set of All Deciding Input Combinations

Our goal is to determine the set of all deciding input combinations for i on f. As before, let $\mathcal{H}_g$ and $\mathcal{L}_g$ denote all least determinate partial input combinations that lead to a high or low output of f respectively. We now want to select pairs $(H, L) \in \mathcal{H}_g \times \mathcal{L}_g$

such that the property

$$H, L : \mathcal{N}_g \rightarrow \{\mathsf{tt}, \mathsf{ff}, \mathrm{X} : \ H(n) = L(n) \ \forall \mathcal{N}_g \backslash \{m\}\}$$

holds. Here we assume that i is only in the fanin of m. Then the following algorithm delivers the desired set.

For each assignment $H \in \mathcal{H}_g$ where $H(m) \neq \mathrm{X}$ determine all $L \in \mathcal{L}_g$ that do not contradict with H in any nodes but m:

$$L(n) \neq \overline{H(n)} \ \forall n \in \mathcal{N}_g \backslash \{m\}, \quad \text{and} \quad L(m) = \overline{H(m)}$$

By eliminating all X-assignments in which H and L differ we thus receive a deciding input combination D:

$$D(n) = \begin{cases} H(n) & \text{if } L(n) = \mathrm{X} \text{ or } H(n) = L(n) \\ L(n) & \text{if } H(n) = \mathrm{X} \\ \mathrm{X} & \text{if } n = m \end{cases}$$

The set of all such deciding input combinations, denoted by $\mathcal{D}_f^m$, may have as many as $|\mathcal{H}_g| \cdot |\mathcal{L}_g|$ elements. However, usually the set is significantly smaller.

Example: Multiple deciding input combinations
Consider a gate g with three inputs denoted by i_1, i_2, i_3. Let $\mathcal{H}_g = \{00\mathrm{X}, \mathrm{X}01\}$ and $\mathcal{L}_g = \{\mathrm{X}1\mathrm{X}, 1\mathrm{X}0\}$. For the input i_2 we thus get two deciding input combinations:

- $00\mathrm{X} \in \mathcal{H}_g$ and $\mathrm{X}1\mathrm{X} \in \mathcal{L}_g$ delivers $D_1 = 0\mathrm{XX}$
- $\mathrm{X}01 \in \mathcal{H}_g$ and $\mathrm{X}1\mathrm{X} \in \mathcal{L}_g$ delivers $D_2 = \mathrm{XX}1$

The other two pairs in $\mathcal{H}_g \times \mathcal{L}_g$ have contradictions:

- $00\mathrm{X} \in \mathcal{H}_g$ and $1\mathrm{X}0 \in \mathcal{L}_g$ contradict in the value of the first input. Additionally, $1\mathrm{X}0$ does not meet the requirement that $L(n) = \overline{H(n)}$.
- $\mathrm{X}01 \in \mathcal{H}_g$ and $1\mathrm{X}0 \in \mathcal{L}_g$ contradict in the value of the third input. Additionally, $1\mathrm{X}0$ does not meet the requirement that $L(n) = \overline{H(n)}$.

★

7.3.2 Average of Multiple Deciding Scenarios

Let $\mathcal{D}_f^m$ be the set of all deciding input combinations for i on f where i is in the fanin of the node m. Further assume that $\mathcal{D}_f^m$ is minimal by collapsing all input combinations that can be collapsed, as already seen for $\mathcal{H}_g$ and $\mathcal{L}_g$.

To determine the degree of responsibility $s(i, f)$ all deciding input combinations need to be factored in, and averaged over. Some partial input combinations cover more cases, so we need to additionally weigh each $D \in \mathcal{D}_f^m$ accordingly. We therefore multiply the cost determined for $D \in \mathcal{D}_f^m$ by $2^{x(D)}$, where

$$x(D) = |\{n \in \mathcal{N}_g \backslash \{m\} : D(n) = \mathrm{X}\}|$$

denotes the number of nodes not equal to m, which are assigned X. As $D(m) = \mathrm{X}$ for all $D \in \mathcal{D}_f^m$, this equates to $|\{n \in \mathcal{N}_g : D(n) = \mathrm{X}\}| - 1$.

Then $s(i, f)$ can be calculated by

$$\frac{1}{\sum_{D \in \mathcal{D}_f^m} 2^{x(D)}} \cdot \sum_{D \in \mathcal{D}_f^m} 2^{x(D)} \cdot \Big(s(i, f_m) + \sum_{n:D(n)=\mathsf{tt}} c_1(f_n) + \sum_{n:D(n)=\mathsf{ff}} c_0(f_n) \Big),$$

or further simplified,

$$s(i, f_m) + \frac{1}{\sum_{D \in \mathcal{D}_f^m} 2^{x(D)}} \cdot \sum_{D \in \mathcal{D}_f^m} 2^{x(D)} \cdot \Big(\sum_{n:D(n)=\mathsf{tt}} c_1(f_n) + \sum_{n:D(n)=\mathsf{ff}} c_0(f_n) \Big).$$

Observe that if the set $\mathcal{D}_f^m$ only includes one element, this matches with the previous definition (page 171). In the off-case that $\mathcal{D}_f^m = \emptyset$ we define $s(i, f) = \infty$, as i then does not influence the output of f at all, and thus is not at all responsible for its value. But as we assumed that i is in the fanin of m, there is usually at least one deciding input combination.

Setting this value to ∞ needs to be specially handled when i is in multiple fanins, as to not falsely classify i not being responsible at all, just because it is irrelevant for one of the fanins. Also see the previous definition of $s(i, f)$ as summarised in Figure 6.3 (page 146) for comparison. More details are provided in the next section.

7.3.3 Approximate Degree of Responsibility for Multiple Fanins

In the previous we assumed that the input i is in the fanin of only one of the nodes that feed into f. We now extend our approach to handle the case where i is in the

fanin of possibly multiple nodes. Here we again move to approximate calculations, and thus there is some leeway as to how to calculate the DoR.

Variant 1

Using the same reasoning as before, we can extend our definition of D as follows. We want to find a pair H, L such that:

$$H, L : \mathcal{N}_g \to \{\mathsf{tt}, \mathsf{ff}, \mathrm{X} : \ L(n) = H(n) \ \forall \mathcal{N}_g \backslash M\},$$

where M is the set of all input nodes whose fanin includes i. Then $D \in \mathcal{D}_f^M$ assigns the same values as L and H to all nodes $n \in \mathcal{N}_g \backslash M$, and $D(m) = \mathrm{X} \ \forall m \in M$. If M includes just one element, this matches with the definition given in Section 7.3.1.

Also recall that previously we required $L(m) = \overline{H(m)}$, which, given that one assignment leads to a low output and the other to a high output, was trivially satisfied. When considering multiple nodes $m \in M$ this is not as easy anymore. We do know, however, that at least one node needs to be assigned different values, and this may be a sufficient requirement. Another option would be to require all assignments for $m \in M$ to be contradicting. However, this is very restrictive. It in general requires concretising X-values, which can lead to an exponential blowup of the elements in $\mathcal{D}_f^m$. One might argue that $|M|$ is usually small, and thus the additional cost can easily be compensated by more precise approximations of the degree of responsibility. But this is not necessarily true.

Recall that the value X does not encode the set $\{\mathsf{tt}, \mathsf{ff}\}$, but rather a (possibly equal) subset. It expresses that the value tt, or ff, might be possible, not that it indeed is. So if tt is not a possible value and we concretise X to tt, we are introducing an error. This can falsify the responsibility calculations, rather than improving them. Additionally, by concretising the input values to the gate $\mathit{deconstruct}(f)$ we impose changes to the primary inputs of f. In our recursion we would then calculate the cost of those changes. But by concretising more than one fanin, we may impose changes that contradict each other, namely on all primary inputs that at least some fanins to g share. Again, this can negatively impact our calculations more than improve them. While the last described problem occurs in all our calculations – this is one of the main reasons why it is only an approximation – avoiding as many as possible of such risky modifications is a good policy. Hence in our approach we decide to relax the condition on deciding input combinations, so that at least one assignment in L and H differs. As before, this is trivially true, because L leads to a low output and H to

a high output by definition.

Once we have determined the set $\mathcal{D}_f^M$ we can adjust the calculation of $s(i, f)$:

$$\frac{\sum_{m \in M} s(i, f_m)}{|M|} + \frac{1}{\sum_{D \in \mathcal{D}_f^M} 2^{x(D)}} \cdot \sum_{D \in \mathcal{D}_f^M} 2^{x(D)} \cdot \Big(\sum_{n : D(n) = \mathsf{tt}} c_1(f_n) + \sum_{n : D(n) = \mathsf{ff}} c_0(f_n) \Big)$$

Essentially, we average over the responsibility values of all fanin nodes, and add the cost of forcing those fanins to be deciding. Again, this definition matches with the one before if $|M| = 1$.

But with this definition we treat all fanins the same – irrespective of whether it contributed a concrete value or not in the pair (H, L) that determined D. Instead, we may want to capture that information by only taking into account those $s(i, f_m)$ where $H(m) \neq \mathrm{X}$ or $L(m) \neq \mathrm{X}$. This adds some overhead, but indeed improves the responsibility calculations. If we do not do this, the values are diluted by multiple responsibilities, which are not relevant. Thus, for (H, L) define

$$\begin{aligned} N_D &= \{n \in \mathcal{N}g : \ H(n) = L(n)\} \\ M_D &= \{m \in \mathcal{N}g : \ H(n) \neq L(n)\} \end{aligned}$$

where $g = \mathit{deconstruct}(f)$ and thus $\mathcal{N}_g$ are the nodes that feed into the outermost gate of f. Note that here N_D may also contain nodes whose fanin includes i.

The corresponding deciding input combination D is then defined as

$$D(n) = \begin{cases} H(n) & n \in N_D \\ \mathrm{X} & n \in M_D \end{cases}$$

As M_D can now vary depending on the pair (H, L) we need to memorise M_D, not just D. This leads to an adjusted definition of $s(i, f)$:

$$\frac{1}{\sum_{D \in \mathcal{D}_f^M} 2^{x(D)}} \cdot \sum_{D \in \mathcal{D}_f^M} 2^{x(D)} \cdot \Big(\frac{\sum_{m \in M_D} s(i, f_m)}{|M_D|} + \sum_{n : D(n) = \mathsf{tt}} c_1(f_n) + \sum_{n : D(n) = \mathsf{ff}} c_0(f_n) \Big),$$

Note that before $x(D)$ denoted the number of nodes $n \in \mathcal{N}_g \backslash \{m\}$ that are assigned X. Accordingly, now it denotes the number of nodes $n \in N_D$ that are assigned X. This ensures the correct weight is again applied to each determined summand.

And still, our definition of M_D can be improved further. In some cases $s(i, f_m)$ may be infinite, leading to $s(i, f)$ evaluating to infinity, too. But the infinite value

encodes that i is not at all responsible for the output of f, whereas this is not true if at least one $s(i, f_m)$ for $m \in M$ is finite. Hence we need to adjust our definition of M. Rather than it including all nodes of $deconstruct(f)$ whose fanins include i and whose value plays a role in the determination of D, it includes only those nodes, where additionally $s(i, f_m) < \infty$. If with this restriction in place $M = \emptyset$ we again set $s(i, f) = \infty$.

So, in summary, we receive:

$$\begin{aligned} M_D &= \{m \in \mathcal{N}g : \ L(m) \neq H(m) \text{ and } s(i, f_m) < \infty\} \\ N_D &= \mathcal{N}_g \backslash M_D \end{aligned}$$

This more general approach delivers the same results as the definition given in [3] even when some responsibility calculations deliver infinite values. Having matching definitions in the special cases already captured continues to stay important, so that we can easily compare Chockler et al.'s approach with our more general one.

Variant 2

Of course there are other options for calculating the approximate degree of responsibility of i when i is in multiple fanins. The second variant we suggest ignores the fact that i is in the fanin of multiple $n \in \mathcal{N}_g$, calculates the approximate degree of responsibility for each of them, and then takes an average of that:

$$s(i, f) = \frac{1}{|M|} \cdot \sum_{m \in M} s_m(i, f)$$

where $M = \{n \in \mathcal{N}_g : \ i \text{ is in the fanin of } m\}$ and s_m assumes i is only in the fanin of m and no other nodes $n \in \mathcal{N}_g$. For $|M| = 1$ this matches with the previous definition as seen in [3].

Here we are independently analysing each fanin and disregarding any connection. This simple approach does lead to inaccuracies. When computing the cost of forcing the other fanin nodes to be a specific value we disregard that i is already fixed, and we may compute impossible scenarios. We mentioned this problem in the previous approach already, supporting our decision to not concretise values. In this variant the errors caused may again be higher, but the calculation is much simpler than in Variant 1.

Example of calculating the approximate degree of responsibility

Consider the gate g where $\mathcal{H}_g = \{00\text{X}, \text{X}01\}$ and $\mathcal{L}_g = \{\text{X}1\text{X}, 1\text{X}0\}$. In a previous example (page 172) we saw that for the model described by $f_g = i_1\ G\ i_2\ G\ i_3$, which consists of a single g-gate, two deciding input combinations exist: $\mathcal{D}_{f_g}^{i_2} = \{0\text{XX}, \text{XX}1\}$.

Thus, $s(i_2, f_g)$ can be calculated as follows:

$$\begin{aligned}
&\textstyle\sum_{D \in \mathcal{D}_{f_g}^{i_2}} 2^{x(D)} && = 2^{x(0\text{XX})} + 2^{x(\text{XX}1)} = 2 + 2 = 4 \\
&2^{x(0\text{XX})} \cdot (s(i_2, i_2) + c_0(i_1)) && = 2 \cdot (0 + 2) = 4 \\
&2^{x(\text{XX}1)} \cdot (s(i_2, i_2) + c_1(i_3)) && = 2 \cdot (0 + 2) = 4 \\
&s(i_2, f_g) && = \tfrac{1}{4} \cdot (4 + 4) = 2
\end{aligned}$$

Here we apply the heuristic shown in Figure 6.7 for computing c_0 and c_1 of primary inputs when $A(i_1) = A(i_2) = A(i_3) = \text{X}$. The results also match with our intuition: both deciding input scenarios have a cost of 2, both occur as often as the other, and thus the average should also equate to 2 – which it does.

In comparison, f = ((NOT i_1) AND (NOT i_2)) OR ((NOT i_2) AND (i_3)), as seen in Figure 7.3, has the same input-output behaviour, but is built up of multiple gates. In particular, i_2 is in the fanin of both inputs to the outermost AND-gate. So our extended approach to evaluating the approximate degree of responsibility can be applied.

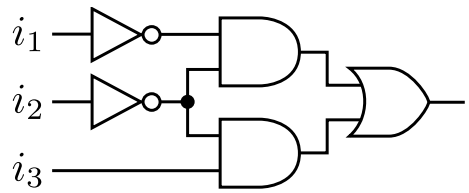


Figure 7.3: An example circuit where i_2 fans out to multiple gates

The calculation of $s(i_2, f)$ is a bit longer, and requires us to consider each gate by itself. For this the following sets are relevant:

$$\begin{aligned}
&\mathcal{H}_{\text{OR}} = \{1\text{X}, \text{X}1\} && \mathcal{L}_{\text{OR}} = \{00\} \\
&\mathcal{H}_{\text{AND}} = \{11\} && \mathcal{L}_{\text{AND}} = \{0\text{X}, \text{X}0\} \\
&\mathcal{H}_{\text{NOT}} = \{0\} && \mathcal{L}_{\text{NOT}} = \{1\}
\end{aligned}$$

We can choose to determine the approximate degree of responsibility either with the first variant we suggest, namely adjusting the definition of deciding input scenarios,

or the second variant, which entails averaging over degrees of responsibility while ignoring multiple fanins.

For Variant 1 we determine $\mathcal{D}_f^{i_2}$ as follows. The outermost gate g of f, $g = \mathit{deconstruct}(f)$, is an OR-gate, and i_2 is in the fanin of both its inputs. Both pairs $(00, 1\mathrm{X}) \in \mathcal{L}_{OR} \times \mathcal{H}_{OR}$ and $(00, \mathrm{X}1)$ delivers that $\{\mathrm{XX}\} \in \mathcal{D}_f^{i_2}$. So, unsurprisingly, $\mathcal{D}_f^{i_2} = \{\mathrm{XX}\}$.

Hence, using the definition of $s(i_2, f)$ as seen in Variant 1 (page 175):

$$\begin{aligned} s(i_2, f) &= \tfrac{1}{2^0} \cdot \big(2^0 \cdot (\tfrac{s(i_2, \text{NOT } i_1 \text{ AND NOT } i_2) + s(i_2, \text{NOT } i_2 \text{ AND } i_3)}{2} + 0 + 0)\big) \\ &= \tfrac{1}{2} \cdot \big(s(i_2, \text{NOT } i_1 \text{ AND NOT } i_2) + s(i_2, \text{NOT } i_2 \text{ AND } i_3)\big) \end{aligned}$$

Let $f_1 = \text{NOT } i_1 \text{ AND NOT } i_2$ and $f_2 = \text{NOT } i_2 \text{ AND } i_3$. Then $\mathcal{D}_{f_1}^{i_2} = \{1\mathrm{X}\}$ and similarly $\mathcal{D}_{f_2}^{i_2} = \{\mathrm{X}1\}$. Thus,

$$\begin{aligned} s(i_2, f_1) &= \tfrac{1}{2^0} \cdot 2^0 \cdot \big(\tfrac{s(i_2, \text{NOT } i_2)}{1} + c_1(\text{NOT } i_1)\big) \\ &= s(i_2, \text{NOT } i_2) + c_1(\text{NOT } i_1) = s(i_2, i_2) + c_0(i_1) = 0 + 2 = 2 \\ s(i_2, f_2) &= \tfrac{1}{2^0} \cdot 2^0 \cdot \big(\tfrac{s(i_2, \text{NOT } i_2)}{1} + c_1(i_3)\big) \\ &= s(i_2, \text{NOT } i_2) + c_1(i_3) = s(i_2, i_2) + 2 = 0+ = 22 \end{aligned}$$

Putting everything together,

$$s(i_2, f) = \frac{1}{2} \cdot \big(s(i_2, f_1) + s(i_2, f_2)\big) = \frac{1}{2} \cdot (2 + 2) = 2$$

While the calculation was more elaborate, we did determine the same degree of responsibility as before. This shows that our quite careful generalisation can yield good approximations, but at the cost of some more overhead, even with as simple an example as this.

Variant 2 takes a much cruder approach, ignoring the fact that i_2 is in more than one fanin of the outermost gate of f. But the calculation is much simpler: $s(i_2, f) = \frac{1}{2} \cdot (s_1(i_2, f) + s_2(i_2, f))$ where s_1 and s_2 assume that i_2 is just in the fanin

of the first and second input of g respectively. Using

$$
\begin{aligned}
s_1(i_2, f) &= s(i_2, f_a) + c_0(f_b) \\
s(i_2, f_a) &= s(i_2, \text{NOT } i_2) + c_1(\text{NOT } i_1) = s(i_2, i_2) + c_0(i_1) = 0 + 2 = 2 \\
c_0(f_b) &= \min\{c_0(\text{NOT } i_2), c_0(i_3)\} = \min\{c_1(i_2), 2\} = \min\{2, 2\} = 2 \\
s_1(i_2, f) &= 2 + 2 = 4 \\
&\text{and} \\
s_2(i_2, f) &= s(i_2, f_b) + c_0(f_a) \\
s(i_2, f_b) &= s(i_2, \text{NOT } i_2) + c_1(i_3) = s(i_2, i_2) + 2 = 0 + 2 = 2 \\
c_0(f_a) &= \min\{c_0(\text{NOT } i_1), c_0(\text{NOT } i_2)\} = \min\{c_1(i_1), c_1(i_2)\} = \min\{2, 2\} \\
&= 2 \\
s_2(i_2, f) &= 2 + 2 = 4
\end{aligned}
$$

we thus receive $s(i_2, f) = \frac{1}{2} \cdot (4 + 4) = 4$.

This example not just shows how to handle inputs of f that fan out to several gates, but also the power of analysing gates by themselves, or at least trying to retain as much sharing information as possible. So both our generalisation on how to calculate responsibilities for arbitrary gates can be very powerful, as well as our more carefully adjusted Variant 1 for evaluating inputs in the multiple fanin case. Variant 2 shows that simplifying further reduces overhead, but the price is commonly less accurate responsibility candidates. This, in turn, may identify worse refinement candidates, and thus drive verification costs up.

7.4 Summary

This chapter extends the automatic refinement work introduced in Chapter 6 by suggesting an approach to determining the approximate degree of responsibility for arbitrary gates. It allows us to identify better refinement candidates, and is thus an important contributor to making feasible the automatic verification of circuits by STE.

In Chapter 8 we provide experimental results that display the power of this generalisation. We evaluate how applying our more careful first variant for calculating the approximate degree of responsibility for multiple fanins results in better approximations than using the previous approach, which required the analysis of small circuits with the same input-output behaviour as the gate, but which are built with NOT- and AND-gates only. These more accurate approximations can change which input

is selected as the next refinement candidate, and thus improve the overall cost of verification. Furthermore, it eases the evaluation, because no equivalent circuits need to be generated for the approximate degree of responsibility calculation. Instead, the gates can be looked at directly, we only require their behaviour.

Importantly, our calculations are split into two main parts, first calculating the set of deciding input combinations, and second the more essential part of calculating the approximate degree of responsibility given that set. While we assumed gates as defined by a set of partial input combinations which lead to a high output, the second part of our calculations is independent of that assumption. In particular, the first part can of course be adjusted to other ways of defining the behaviour of a gate. Thus, our approach can be adapted to other verification environments.

Finally, this general approach to evaluating gates may raise the question why we do not classify the whole circuit as one gate. It would avoid introducing approximation errors throughout. In principle this is possible, however the cost is too high for almost all circuits. The calculations we perform can have exponential costs in the number of inputs to the gate. So if we view the whole circuit as the gate, the number of inputs to the circuit determines the exponent. That number is generally prohibitively large. As computing the set $\mathcal{L}_g$ is NP-hard, in all likelihood no efficient algorithm exists. Thus, only evaluating gates with a small number of inputs is feasible, and we need to break the circuit into such smaller gates for evaluation. This process introduces imprecision, leading to an approximation again, but also makes the approach computationally tractable.

Chapter 8

Experimental Results for Abstraction Refinement

In this chapter we show the usefulness of our more general approach to abstraction refinement by calculating an approximate degree of responsibility of inputs on an X-possible output. We first verify the CAM, which we already introduced in Chapter 5, but this time we do not provide symbolic constants. Thus, over-abstraction occurs and using a refinement loop we identify exactly those symbolic constants as the refinement candidates. This shows that our generalisation works, and in particular selects only those inputs that must be driven to verify the CAM.

We then demonstrate how the approximate degree of responsibility calculation for different, common gates changes when using our approach to evaluating arbitrary gates, in contrast to examining small circuits with the same input-output behaviour. In particular, this highlights that our method delivers the same, or better approximations.

8.1 Abstraction Refinement for the CAM

In Chapter 5 we verified the correctness of a CAM using an abstraction scheme automatically generated using the auto_abstract algorithm proposed in Chapter 4. But the verification was not fully automatic: we manually specified a set of symbolic constants, driven with symbolic values in all cases. When these symbolic constants are not given, the verification fails due to over-abstraction. So the CAM is a good candidate for demonstrating the abstraction refinement we proposed in Chapters 6 and 7.

We ran four different variants of the abstraction refinement: (1) including the

determined refinement candidate in the set of symbolic constants, and rerunning auto_abstract, (2) refining the relation by always driving the refinement candidate with a variable, (3) refining the relation by driving the refinement candidate with a variable only in the over-abstraction cases, and (4) a combination of the latter two, using a threshold value to determine whether to always drive the input with a variable, or just in the over-abstraction cases.

Figure 8.1 shows the execution times we observed. Note that a threshold of 1 corresponds to refining the abstraction relation, such that the identified refinement candidate is only driven with a variable in the over-abstraction cases, a threshold of 0 always drives the refinement candidate with a variable in the refined relation, and a threshold strictly between 0 and 1 switches between the two variants depending on how often it would be driven in the abstraction relation received by variant (3). So a threshold of 0.25 means than if the input would be driven in at least 25% of all cases, then we instead refine the relation so that the input is always driven. We determine how often an input is driven by extracting from the relation the expression f, which decides when to drive the input with a variable, and computing

$$\frac{sc(f)}{2^{|\mathsf{free\_vars}(f)|}}.$$

When running these test for a CAM with 4 entries and a key width of 8 bits, all of the approaches suggested the same sequence of refinement candidates:

$$key[7], key[6], key[5], key[4], key[3], key[0], key[2], key[1].$$

We speculate that once the first five bits of the key are driven by a variable, the Forte environment can internally simplify the expressions, so that $key[0]$ has more impact than without those simplifications. Thus, $key[0]$ is selected as the next refinement candidate, rather than – as one might have expected – $key[2]$, and then $key[1]$. However, all bits of the key need be driven with a variable, so this slight change of order does not influence the verification much. Looking at the design itself, all bits of the key are equally required due to our modification to simulate calculations, as perhaps required for an error-correction.

All of these runs needed no user interaction. The only user-provided data were the hardware design, and its specification – which are necessary to define the verification problem. So these results demonstrate a verification tool, which internally consists of an abstraction discovery and refinement loop, and which can be used to fully

	(a)	(b)	(c)	(d)
Initial relation	0.02s	0.02s	0.02s	0.02s
STE	4.14s	4.24s	5s	4.26s
Restricted STE	6.2s	7s	5.92s	7.24s
Refinement	0.15s	0.07s	0.07s	0.16s
STE	3.18s	12.62s	8.57s	9.45s
Restricted STE	4.77s	11.37s	9.42s	10.44s
Refinement	0.08s	0.06s	0.14s	0.15s
STE	1.95s	20.05s	15.94s	13.49s
Restricted STE	4.23s	22.98s	12.4s	16.12s
Refinement	0.12s	0.15s	0.05s	0.16s
STE	1.47s	38.43s	24.65s	26.85s
Restricted STE	3.26s	32.71s	24.35s	26.65s
Refinement	0.05s	0.13s	0.13s	0.12s
STE	0.68s	50.48s	31.36s	30.24s
Restricted STE	1.93s	48.62s	31.05s	33.58s
Refinement	0.13s	0.16s	0.11s	0.14s
STE	0.5s	75.69s	42.65s	46.49s
Restricted STE	1.01s	80.33s	51.85s	58.38s
Refinement	0.06s	0.11s	0.2s	0.2s
STE	0.11s	81.28s	60.94s	65.77s
Restricted STE	0.1s	8.02s	5.67s	6.32s
Refinement	0.09s	0.1s	0.11s	0.11s
STE	0.01s	12.43s	9.71s	8.52s
Restricted STE	0.01s	5.34s	3.22s	4.08s
Refinement	0.07s	0.02s	0.05s	0.08s
STE	0.01s	5.98s	4.46s	4.46
Total	34.33s	518.39s	348.04s	373.48s

Figure 8.1: Execution times observed in the different steps of the automatic refinement loop verifying a 4×8 CAM: (a) defining refinement candidates as symbolic constants, and using (b) a threshold of 1.0, (c) a threshold of 0.25, (d) and a threshold of 0.0 to decide whether to run the refinement candidate in the over-abstraction cases only, or always

automatically verify various designs. This is in principle extremely powerful.

In the three variants that refine the relation, rather than computing a new one with our auto_abstract algorithm while stating symbolic constants, we see slightly different performance. The variant that switches between approaches performs slightly better than always driving the refinement candidate with a variable, and noticeably better than only driving the refinement candidate in the over-abstraction cases. This suggest that – at least for the CAM – the cost of a more complex relation is higher than that

of driving more inputs with a value.

Another interesting observation is concerned with how much time is spent in each iteration of the abstraction refinement loop. Figure 8.2 visualises how for all three variants that refine, rather than compute a new relation, iteration execution times first increase, as refining increases the problem size again. Then, at some point – here in the eighth iteration – the relation is good enough for the simulation to truly profit from the symbolic indexing, and the cost of running STE drastically drops.

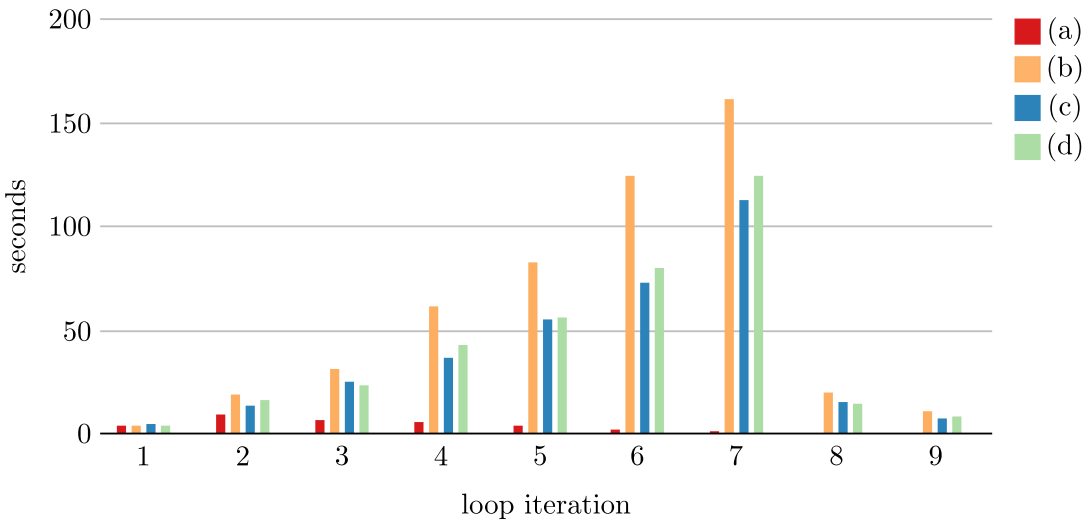


Figure 8.2: Visualisation of how much time is spent in each iteration of the abstraction discovery and refinement loop

The execution times also show this fully automatic abstraction discovery and refinement loop is a proof-of-concept, but not a workable solution yet. One major shortcoming immediately apparent is the prohibitively high costs of running STE with the abstraction relation that leads to over-abstraction. When not specifying any symbolic constants, unfortunately the verification – even though it fails due to over-abstraction – is very expensive. Yet we can only refine the abstraction once we have both discovered that over-abstraction is an issue, as well as traced the value of all input nodes. The prior we require to know refinement is necessary, the latter to compute the approximate degree of responsibility for all refinement candidates.

However, this does not necessarily discredit the abstraction refinement we suggest. Rather, it shows that even with an approach to abstraction refinement, bad initial abstraction schemes can have such a drastic impact that we do not even reach the point of being able to refine the abstraction. In these cases a different initial abstraction scheme needs to be selected first. This could, for example, be achieved by

guessing symbolic constants, and once the initial run completes, letting the abstraction refinement do the rest of the work. Or an initial abstraction could be provided from somewhere else. Importantly, our abstraction refinement is independent of the automatic abstraction discovery presented in Chapters 3 and 4, so it can be applied to various other abstraction schemes.

Furthermore, the abstraction discovery and refinement loop that stated the refinement candidates as symbolic constants, and computed a completely new abstraction had a much better performance. Unfortunately, the first run, which did not state any symbolic constants, could not be completed for larger CAMs. But this example does show that even slightly improved relations can greatly reduce the execution times of iterations of the loop. This relates to the suggestion of guessing the first few symbolic constants, or providing a relation from somewhere else.

Finally, note that both the memory and scheduler verified in Chapter 5 can be verified without over-abstraction even without specifying symbolic constants. Verification costs are then higher, but unfortunately the abstraction refinement algorithm proposed cannot improve abstractions which are suboptimal, but sufficient. It can only help with refining those abstractions that lead to over-abstractions. Hence, the abstraction refinement we presented cannot be applied to the verification of the memory or the scheduler when not specifying any symbolic constants.

8.2 Approximate Degree of Responsibility for Arbitrary Gates

In Chapter 7 we presented an approach to calculating the approximate degree of responsibility for arbitrary gates. For this, we assumed a gate was defined by a set of partial input combinations that all lead to a high output. All input combinations not captured by that set in turn lead to a low output.

This section provides the approximate degree of responsibility calculations for some common gates, and compares them to the calculations received when evaluating them with the previous approach. In particular, Chockler et al. provided definitions only for NOT- and AND-gates, so instead of looking at the gate directly, we have to analyse a small circuit, which has the same input-output behaviour but consists of NOT- and AND-gates only. We further study the calculations when the initial input, whose approximate degree of responsibility is to be calculated, is in one or several of the fanins of that gate. We observe that sometimes the approximate degree of

responsibility computed using our method match with evaluating the small, equivalent circuits. In other cases the calculations differ, and our approach delivers superior results to those received by the previous approach.

For simplicity, we assume all approximate degrees of responsibility are finite. Whenever infinite values are observed, both Chockler et al.'s and our approach suggest ignoring the fact that the initial input is in that fanin of the gate, and thus corresponds to a different calculation also presented here.

8.2.1 3-Input AND

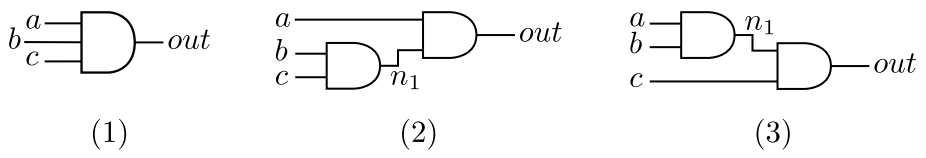


Figure 8.3: (1) A 3-input AND-gate, and (2), (3) two circuits with the same input-output behaviour, consisting only of NOT- and AND-gates

Figure 8.3 shows a 3-input AND-gate, as well as two small circuits that have the same input-output behaviour as a 3-input AND-gate. We first determine the approximate degree of responsibility for this gate using our approach, and then – on the small circuits – using Chockler et al.'s definition.

A 3-input AND-gate is defined by $\mathcal{H}_{3\text{AND}} = 111$. We can then calculate $\mathcal{L}_{3\text{AND}}$ as described in Section 7.2, which delivers $\mathcal{L}_{3\text{AND}} = \{0\text{XX}, \text{X0X}, \text{XX0}\}$.

First assume that the initial input i, for which we want to calculate the approximate degree of responsibility, is in just one fanin of the gate. As the gate behaves symmetrically in all three inputs without loss of generality we can assume this is the fanin a. We now need to determine the set of deciding input combinations. For this, we need to find a pair $(H, L) \in \mathcal{H}_{3\text{AND}} \times \mathcal{L}_{3\text{AND}}$ such that $H(a) = \overline{L(a)}$ and the values for do not contradict each other in H and L: $H(b) = L(b)$, or $H(b) = \text{X}$, or $L(b) = \text{X}$. The same holds for c.

Only one pair $(H, L) = (111, 0\text{XX})$ satisfies these conditions. As seen in the definition of deciding input combinations (page 172), thus $\mathcal{D}^{\{a\}}_{3\text{AND}} = \{\text{X11}\}$. So

$$s(i, out) = s(i, a) + c_1(b) + c_1(c)$$

Evaluating the equivalent circuit (2) results in

$$s(i, out) = s(i, a) + c_1(n_1) = s(i, a) + c_1(b) + c_1(c),$$

and similarly analysing the equivalent circuit (3) provides

$$s(i, out) = s(i, n_1) + c_1(c) = s(i, a) + c_1(b) + c_1(c).$$

Note that there are many circuits with the same input-output behaviour as the 3-input AND-gate, so it is a personal choice which to use for the evaluation. However, both of the circuits we suggest are obvious choices. In this simple case, both deliver the same result. This is not the case anymore when the initial input is in the fanin of two inputs of the 3-input AND-gate.

Next assume that the initial input i is in two fanins of the gate. Again, without loss of generality, we can assume that it is in the fanin of a and b – at least with our approach. When determining the partial input combinations with multiple fanins, we proposed two approaches. In the first, more careful variant we required at least one fanin that includes i to be assigned contradicting values in H and L. Notably, this requirement was not extended to all fanins that include i. The fanins that do not include i merely need to be non-contradicting. Furthermore, when determining the partial, deciding input combination, we memorised the set M_D, which captures which fanins that include i actually were assigned contradicting values in (H, L). Thus, $\mathcal{D}^{\{a,b\}}_{\text{3AND}}$ includes pairs (D, M_D) where $M_D \subseteq \{a, b\}$. In particular, the pair $(111, 0\text{XX})$ delivers $(\text{XX1}, \{a\}) \in \mathcal{D}^{\{a,b\}}_{\text{3AND}}$, and the pair $(111, \text{X0X})$ delivers $(\text{XX1}, \{b\}) \in \mathcal{D}^{\{a,b\}}_{\text{3AND}}$. Thus, using the extended approximate degree of responsibility calculation as presented on page 175:

$$s(i, out) = \frac{1}{2+2} \cdot \Big(2 \cdot \big(s(i, a) + c_1(c)\big) + 2 \cdot \big(s(i, b) + c_1(c)\big)\Big) = \frac{s(i, a) + s(i, b)}{2} + c_1(c)$$

This matches with the definition of the 2-input AND-gate when the initial input i is in both fanins, plus requiring the third input to allow the first two to be deciding. It is the obvious way you would extend the calculation manually. However, by the previous approach, this is not necessarily the computation executed. Using the equivalent circuit (2), we receive

$$s(i, out) = \frac{s(i, a) + s(i, n_1)}{2} = \frac{s(i, a) + s(i, b) + c_1(c)}{2},$$

and using circuit (3)

$$s(i, out) = s(i, n_1) + c_1(c) = \frac{s(i, a) + s(i, b)}{2} + c_1(c).$$

As the 3-input AND-gate is symmetrical in all three inputs, the calculation received by the equivalent circuit (2) is less accurate. Our approach is independent of constructing equivalent circuits, and thus not just more stable, but also delivers the more desirable approximation for this case.

Finally, assume that the initial input is in all three fanins of the gate. By similar reasoning as before, $\mathcal{D}^{\{a,b,c\}}_{3\text{AND}} = \{(\text{XXX}, \{a\}), (\text{XXX}, \{b\}), (\text{XXX}, \{c\})\}$, and thus

$$s(i, out) = \frac{1}{4+4+4} \cdot \Big(4 \cdot s(i, a) + 4 \cdot s(i, b) + 4 \cdot s(i, c)\Big) = \frac{s(i, a) + s(i, b) + s(i, c)}{3}$$

By comparison, analysing the equivalent circuit (2),

$$s(i, out) = \frac{s(i, a) + s(i, n_1)}{2} = \frac{s(i, a) + \frac{s(i,b)+s(i,c)}{2}}{2} = \frac{s(i, a)}{2} + \frac{s(i, b) + s(i, c)}{4},$$

which weighs a more heavily than b and c, although not desired. Similarly, the other equivalent circuit, gives c greater weight, and an equivalent circuit that first conjuncts a and c gives b greater weight. In contrast, our approach recognises the symmetry and in turn gives all three inputs the same weight, which delivers a more accurate approximation.

A final comment on this evaluation need to be made. It is not completely fair, because Chockler et al. indeed suggest improving the approximation for multiple-input AND-gates (see 147). This then delivers the same results as our approach. We chose to set this proposal aside for the evaluation above to highlight the differences without special handling of gates other than simple two-input AND-gates and NOT-gates. As the multiple-input AND-gate is the only generalisation Chockler et al. provide, the next examples shows the novel value of our approach to evaluating arbitrary gates.

8.2.2 Multiple-Input OR

Figure 8.4 shows a 3-input OR-gate, as well as two small circuit, which have the same input-output behaviour as a 3-input OR-gate. A 3-input OR-gate is defined by $\mathcal{H}_{3\text{OR}} = \text{1XX}, \text{X1X}, \text{XX1}$. Calculating $\mathcal{L}_{3\text{OR}}$ delivers $\mathcal{L}_{3\text{OR}} = \{000\}$.

First assume that the initial input i, for which we want to calculate the approx-

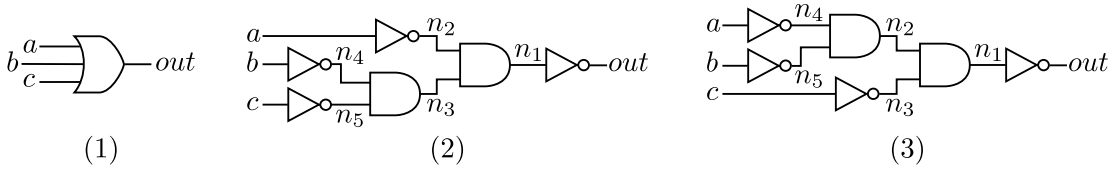


Figure 8.4: (1) A 3-input OR-gate, and (2), (3) two circuits with the same input-output behaviour, consisting only of NOT- and AND-gates

imate degree of responsibility, is in just one fanin of the gate. As the gate behaves symmetrically in all three inputs without loss of generality we can assume this is the fanin a. Then $\mathcal{D}_{3\mathrm{OR}}^{\{a\}} = \{\mathrm{X00}\}$ and thus

$$s(i, out) = s(i, a) + c_0(b) + c_0(c).$$

Analysing the equivalent circuit (2) results in the same value,

$$\begin{aligned} s(i, out) &= s(i, n_1) = s(i, n_2) + c1(n_3) = s(i, a) + c_1(n_4) + c_1(n_5) \\ &= s(i, a) + c_0(b) + c_0(c) \end{aligned}$$

which is also received by looking at circuit (3),

$$\begin{aligned} s(i, out) &= s(i, n_1) = s(i, n_2) + c_1(n_3) = s(i, n_4) + c_1(n_5) + c_0(c) \\ &= s(i, a) + c_0(b) + c_0(c). \end{aligned}$$

Next, assume i is in the fanin of two inputs, without loss of generality a and b. The pair $(1\mathrm{XX}, 000)$ delivers $(\mathrm{XX0}, \{a\}) \in \mathcal{D}_{3\mathrm{OR}}^{\{a,b\}}$, and the pair $(\mathrm{X1X}, 000)$ delivers $(\mathrm{XX0}, \{b\}) \in \mathcal{D}_{3\mathrm{OR}}^{\{a,b\}}$. Thus,

$$s(i, out) = \frac{1}{2+2}\Big(2\cdot(s(i, a)+c_0(c))+2\cdot(s(i, b)+c_0(c))\Big) = \frac{s(i, a) + s(i, b)}{2}+c_0(c)$$

As seen with the 3-input AND-gate, Chockler et al.'s simple definition, which only includes the handling for NOT-gates and AND-gates, leads to different results depending on which two inputs i is in the fanin of, and which equivalent circuit is evaluated. By circuit (2),

$$s(i, out) = \frac{s(i, n_2) + s(i, n_3)}{2} = \frac{s(i, a) + s(i, b) + c_0(c)}{2},$$

and by circuit (3),

$$s(i, out) = s(i, n_2) + c_0(c) = \frac{s(i, a) + s(i, b)}{2} + c_0(c).$$

As before, our approach delivers the more precise representation of the impact the different inputs have on the gate's output value.

Finally, if i is in the fanin of all three inputs of the 3OR-gate, then the pair $(1\mathrm{XX}, 000)$ delivers $(\mathrm{XXX}, \{a\}) \in \mathcal{D}_{3\mathrm{OR}}^{\{a,b,c\}}$, the pair $(\mathrm{X1X}, 000)$ delivers $(\mathrm{XXX}, \{b\}) \in \mathcal{D}_{3\mathrm{OR}}^{\{a,b,c\}}$, and $(\mathrm{XX1}, 000)$ delivers $(\mathrm{XXX}, \{c\}) \in \mathcal{D}_{3\mathrm{OR}}^{\{a,b,c\}}$. Thus,

$$s(i, out) = \frac{1}{4+4+4} \cdot \Big(4 \cdot s(i, a) + 4 \cdot s(i, b) + 4 \cdot s(i, c)\Big) = \frac{s(i, a) + s(i, b) + s(i, c)}{3}$$

Evaluating the equivalent circuit (2), in contrast, delivers

$$s(i, out) = \frac{s(i, a)}{2} + \frac{s(i, b) + s(i, c)}{4},$$

weighing a more heavily than b and c, as also seen for the 3-input AND-gate. Other equivalent circuits weigh b or c more heavily. This again shows that our general approach provides more accurate results.

8.2.3 NAND

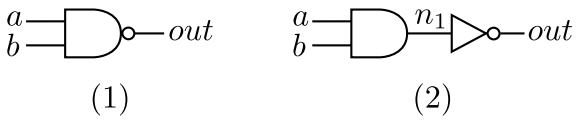


Figure 8.5: (1) A NAND-gate, and (2) a circuit with the same input-output behaviour, consisting only of NOT- and AND-gates

Figure 8.5 shows a NAND-gate, as well as a small circuit, which has the same input-output behaviour. A NAND-gate is defined by $\mathcal{H}_{\mathrm{NAND}} = \mathrm{X0}, 0\mathrm{X}$. Calculating $\mathcal{L}_{\mathrm{NAND}}$ delivers $\mathcal{L}_{\mathrm{NAND}} = \{11\}$.

First assume that i is in the fanin of one input of the NAND-gate, without loss of generality a. The pair $(0\mathrm{X}, 11)$ delivers $\mathrm{X1} \in \mathcal{D}_{\mathrm{NAND}}^{\{a\}}$, and no other pairs exist that contradict the assignment for a. Thus,

$$s(i, out) = s(i, a) + c_1(b).$$

Using the previous approach provides the same calculation:

$$s(i, out) = s(i, n_1) = s(i, a) + c_1(b).$$

Next, assume that i is in the fanin of both inputs of the NAND-gate. The pair $(0\mathrm{X}, 11)$ delivers $(\mathrm{XX}, \{a\}) \in \mathcal{D}^{\{a,b\}}_{\mathrm{NAND}}$, and the pair $(\mathrm{X}0, 11)$ delivers $(\mathrm{XX}, \{b\}) \in \mathcal{D}^{\{a,b\}}_{\mathrm{NAND}}$. Thus,

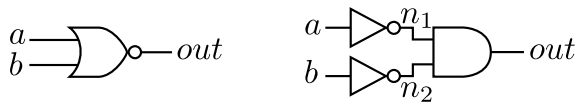
$$s(i, out) = \frac{1}{2+2} \cdot \Big(2 \cdots (i, a) + 2 \cdots (i, b)\Big) = \frac{s(i, a) + s(i, b)}{2}$$

Evaluating the equivalent circuit also comes up with this:

$$s(i, out) = s(i, n_1) = \frac{s(i, a) + s(i, b)}{2}$$

So the calculations match in our approach and the previous one.

8.2.4 NOR



(1) (2)

Figure 8.6: (1) A NOR-gate, and (2) a circuit with the same input-output behaviour, consisting only of NOT- and AND-gates

Figure 8.6 shows a NOR-gate, as well as a small circuit, which has the same input-output behaviour. A NOR-gate is defined by $\mathcal{H}_{\mathrm{NOR}} = 00$. Calculating $\mathcal{L}_{\mathrm{NOR}}$ delivers $\mathcal{L}_{\mathrm{NOR}} = \{1\mathrm{X}, \mathrm{X}1\}$.

First assume that i is in the fanin of one input of the NOR-gate, without loss of generality a. The pair $(00, 1\mathrm{X})$ delivers $\mathrm{X}0 \in \mathcal{D}^{\{a\}}_{\mathrm{NOR}}$, and no other pairs exist, which contradict in the assignment for a. Thus,

$$s(i, out) = s(i, a) + c_0(b).$$

Using the previous approach provides the same calculation:

$$s(i, out) = s(i, n_1) + c_1(n_2) = s(i, a) + c_0(b).$$

Next, assume that i is in the fanin of both inputs of the NOR-gate. The pair $(00, 1\mathrm{X})$ delivers $(\mathrm{XX}, \{a\}) \in \mathcal{D}^{\{a,b\}}_{\mathrm{NOR}}$, and the pair $(00, \mathrm{X}1)$ delivers $(\mathrm{XX}, \{b\}) \in \mathcal{D}^{\{a,b\}}_{\mathrm{NOR}}$. Thus,

$$s(i, out) = \frac{1}{2+2} \cdot \Big(2 \cdot s(i, b) + 2 \cdot s(i, a)\Big) = \frac{s(i, a) + s(i, b)}{2}$$

Evaluating the equivalent circuit also comes up with this:

$$s(i, out) = \frac{s(i, n_1) + s(i, n_2)}{2} = \frac{s(i, a) + s(i, b)}{2}$$

As seen for the NAND-gate, both calculations match. More generally, the calculations in our approach and the previous approach match for any gate which can be represented by a circuit with just one AND-gate and arbitrarily many NOT-gates.

8.2.5 XOR

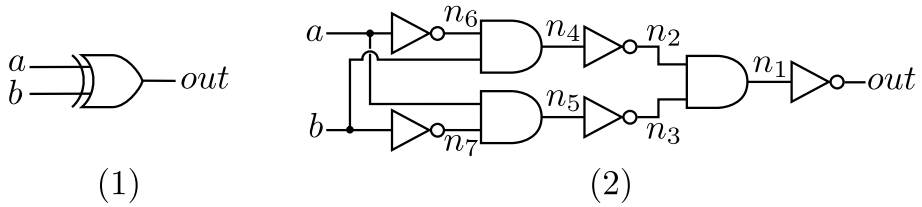


Figure 8.7: (1) An XOR-gate, and (2) a circuit with the same input-output behaviour, consisting only of NOT- and AND-gates

Figure 8.7 shows an XOR-gate, as well as a small circuit, which has the same input-output behaviour. An XOR-gate is defined by $\mathcal{H}_{\mathrm{XOR}} = 01, 10$. Calculating $\mathcal{L}_{\mathrm{XOR}}$ delivers $\mathcal{L}_{\mathrm{XOR}} = \{11, 00\}$.

First assume that i is in the fanin of one input of the XOR-gate, without loss of generality a. The pair $(01, 11)$ delivers $\mathrm{X}1 \in \mathcal{D}^{\{a\}}_{\mathrm{XOR}}$, and the pair $(10, 00)$ delivers $\mathrm{X}0 \in \mathcal{D}^{\{a\}}_{\mathrm{XOR}}$. Thus,

$$s(i, out) = \frac{1}{1+1} \cdot \Big(1 \cdot \big(s(i, a) + c_1(b)\big) + 1 \cdot \big(s(i, a) + c_0(b)\big)\Big) = s(i, a) + \frac{c_1(b) + c_0(b)}{2}.$$

Analysing the equivalent circuit also delivers

$$\begin{aligned} s(i,out) &= s(i,n_1) = \tfrac{s(i,n_2)+s(i,n_3)}{2} = \tfrac{s(i,n_4)+s(i,n_5)}{2} \\ &= \tfrac{s(i,n_6)+c_1(b)+s(i,a)+c_1(n_7)}{2} = \tfrac{s(i,a)+c_1(b)+s(i,a)+c_0(b)}{2} = s(i,a) + \tfrac{c_1(b)+c_0(b)}{2}. \end{aligned}$$

Next, assume that i is in the fanin of both inputs of the XOR-gate. The pairs $(01, 11)$ delivers $(\mathrm{X1}, \{a\})$, and the pair $(10, 00)$ delivers $(\mathrm{X0}, \{a\})$. But recall that we had stated that $\mathcal{D}^{\{a,b\}}_{\mathrm{XOR}}$ is minimal in that any assignments that can be collapsed are collapsed, and any assignments already covered is removed also. So instead of adding both these assignments to the set of deciding input combinations, we only include $(\mathrm{XX}, \{a\})$. Similarly, the pairs $(01, 00)$ and $(10, 11)$ deliver that $(\mathrm{XX}, \{b\}) \in \mathcal{D}^{\{a,b\}}_{\mathrm{XOR}}$.

Thus,

$$s(i,out) = \frac{1}{2+2} \cdot \Big(2 \cdot s(i,a) + 2 \cdot s(i,b)\Big) = \frac{s(i,a) + s(i,b)}{2}.$$

Similarly, looking at the equivalent circuit, we receive

$$\begin{aligned} s(i,out) &= \tfrac{s(i,n_4)+s(i,n_5)}{2} = \tfrac{s(i,n_6)+s(i,b)}{4} + \tfrac{s(i,a)+s(i,n_7)}{4} \\ &= \tfrac{s(i,a)+s(i,b)}{4} + \tfrac{s(i,a)+s(i,b)}{4} = \tfrac{s(i,a)+s(i,b)}{2} \end{aligned}$$

So our approach and the previous one match for this gate.

8.2.6 XNOR

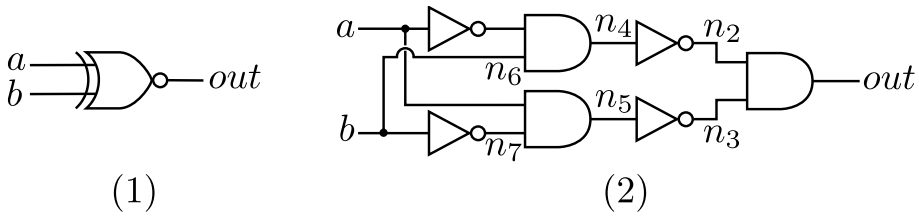


Figure 8.8: (1) An XNOR-gate, and (2) a circuit with the same input-output behaviour, consisting only of NOT- and AND-gates

Figure 8.8 shows an XNOR-gate, as well as a small circuit, which has the same input-output behaviour. An XNOR-gate is defined by $\mathcal{H}_{\mathrm{XNOR}} = 00, 11$. Calculating $\mathcal{L}_{\mathrm{XNOR}}$ delivers $\mathcal{L}_{\mathrm{XNOR}} = \{01, 10\}$.

As this gate delivers the negated result of an XOR-gate, and thus $\mathcal{L}$ and $\mathcal{H}$ are

simply swapped, the calculations are the same as for the XOR-gate:

$$s(i, out) = s(i, a) + \frac{c_1(b) + c_0(b)}{2}$$

if i is in the fanin of a only, and

$$s(i, out) = \frac{s(i, a) + s(i, b)}{2}$$

if i is in both fanins. In particular, as already seen for the XOR-gate, when determining $\mathcal{D}^{\{a,b\}}_{\text{XOR}}$ partial input combinations can be further collapsed, and ultimately $\mathcal{D}^{\{a,b\}}_{\text{XOR}} = \{(\text{XX}, \{a\}), (\text{XX}, \{b\}\}$. So our approach and the previous one again match for this gate.

8.2.7 MUX

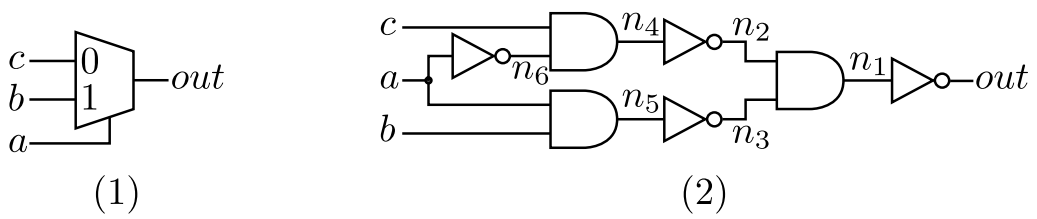


Figure 8.9: (1) A multiplexer, and (2) a circuit with the same input-output behaviour, consisting only of NOT- and AND-gates

Figure 8.7 shows a multiplexer, as well as a small circuit that has the same input-output behaviour. A multiplexer is defined by $\mathcal{H}_{\text{MUX}} = 11\text{X}, 0\text{X}1, \text{X}11$. Calculating $\mathcal{L}_{\text{MUX}}$ delivers $\mathcal{L}_{\text{MUX}} = \{10\text{X}, 0\text{X}0, \text{X}00\}$.

First assume that i is in the fanin of one input of the multiplexer. As the definition is not symmetrical in all inputs anymore, we need to look at each fanin case separately. If i is in the fanin of a, then $\mathcal{D}^{\{a\}}_{\text{MUX}} = \{\text{X}10, \text{X}01\}$ and thus

$$\begin{aligned} s(i, out) &= \tfrac{1}{1+1} \cdot \Big(1 \cdot (s(i, a) + c_1(b) + c_0(c)) + 1 \cdot (s(i, a) + c_0(b) + c_1(c))\Big) \\ &= s(i, a) + \tfrac{c_0(b)+c_0(c)+c_1(b)+c_1(c)}{2}. \end{aligned}$$

By comparison, analysing the equivalent circuit delivers

$$s(i, out) = \frac{s(n_4) + s(n_5)}{2} = \frac{c_1(c) + s(i, a) + s(i, a) + c_1(b)}{=} s(i, a) + \frac{c_1(b) + c_1(c)}{2}$$

In particular, this solution does not capture that i is partly responsible for both high and low outputs, as the value of a decides whether the output is the value of b or c, irrespective of whether they have a high or low value. Hence, taking the average of c_0 and c_1 more closely captures the degree of responsibility. For this recall that c_0 and c_1 were used to change the current setting to a scenario where the input is deciding. Here it does not matter whether the output is high or low, just that i influences the value that is output. The previous evaluation cannot capture this connection.

If i is in the fanin of b, then $\mathcal{D}_{\text{MUX}}^{\{b\}} = \{1\text{XX}\}$. Note that when determining this set, we come up with three pairs, which result in the potential elements $1\text{XX}, 1\text{X0}$, and 1X1. But 1XX captures both the other cases already, so they are not included in $\mathcal{D}_{\text{MUX}}^{\{b\}}$. The approximate degree of responsibility can then be calculated as

$$s(i, out) = s(i, b) + c_1(a).$$

This clearly matches with the intuition: i only influences the output of the multiplexer, if a selects the second input, namely whenever a is high. In contrast, evaluating the equivalent circuit delivers

$$s(i, out) = c_1(n_2) + s(i, n_3) = c_0(n_2) + s(i, n_5) = \min\{c_0(c), c_1(a)\} + c_1(a) + s(i, b),$$

which additionally adds $\min\{c_0(c), c_1(a)\}$. If $c_0(c)$ is the minimum, an irrelevant cost is added to the approximate degree of responsibility, distorting it disadvantageously. If, on the other hand $c_1(a)$ is the minimum, the cost of changing a to a forcing, high value is increased. Recall that we usually compute the approximate degree of responsibility for multiple initial inputs to decide which is the best refinement candidate. So determining a higher cost for forcing a to a high value is also undesirable. In summary, our calculation is preferable to the one received by analysing the equivalent circuit.

Finally, if i is in the fanin of c, then $\mathcal{D}_{\text{MUX}}^{\{c\}} = \{0\text{XX}\}$. In particular, the pairs $(0\text{X1}, 0\text{X0})$ and $(0\text{X1}, \text{X00})$ both deliver $(0\text{XX}) \in \mathcal{D}_{\text{MUX}}^{\{c\}}$. The third pair, which satisfies the required conditions, is $(\text{X11}, 0\text{X0})$ and suggests 01X as an element of $\mathcal{D}_{\text{MUX}}^{\{c\}}$. However, this partial input assignment is more specific than 0XX without contradicting it, and is thus already captured. Thus,

$$s(i, out) = s(i, c) + c_0(a),$$

which again matches with our intuition. The input i is as responsible for the output

of the multiplexer as it is responsible for the value of c provided that a selects c, i.e., a is low. By a similar calculation as before, analysing the equivalent circuit delivers

$$s(i, out) = s(i, c) + c_0(a) + \min\{c_1(a), c_0(b)\},$$

which again distorts the responsibility calculation disadvantageously. So for the multiplexer our approach delivers a better approximate degree of responsibility even when i is only in one fanin.

Next assume that i is in the fanin of two inputs of the multiplexer. If i is in the fanin of a and b, then the set of all deciding input combinations can be determined as $\mathcal{D}_{\text{MUX}}^{\{a,b\}} = \{(\text{XXX}, \{a\}), (\text{XXX}, \{b\})\}$. Again, evaluating the pairs resulted in a set of assignments $\{(\text{1XX}, \{a\}), (\text{XX0}, \{a\}), (\text{XX0}, \{b\}), (\text{XX1}, \{a\}), (\text{XX1}, \{b\})\}$, which could be collapsed further. Thus,

$$s(i, out) = \frac{1}{4+4} \cdot \Big(4 \cdot s(i, a) + 4 \cdot s(i, b)\Big) = \frac{s(i, a) + s(i, b)}{2}$$

In contrast, looking at the equivalent circuit results in

$$\begin{aligned} s(i, out) &= \tfrac{s(i,n_4)+s(i,n_5)}{2} = \tfrac{c_1(c)+s(i,a)+\frac{s(i,a)+s(i,b)}{2}}{2} \\ &= \tfrac{3}{4}s(i, a) + \tfrac{1}{4}s(i, b) + \tfrac{1}{2}c_1(c). \end{aligned}$$

This calculation again factors in the value of c, although not relevant, as well as weighs the two inputs differently, although both are equally relevant for the output value of the multiplexer. Again, our calculations deliver a better approximation.

If i is in the fanin of a and c, the calculations look very similar. Our approach delivers $\mathcal{D}_{\text{MUX}}^{\{a,c\}} = \{(\text{XXX}, \{a\}), (\text{XXX}, \{c\})\}$ and $s(i, out) = \frac{s(i,a)+s(i,c)}{2}$, whereas the previous approach suggested $s(i, out) = \frac{3}{4}s(i, a) + \frac{1}{4}s(i, c) + \frac{1}{2}c_1(b)$.

Finally, if i is in the fanin of b and c, we calculate

$$\mathcal{D}_{\text{MUX}}^{\{b,c\}} = \{(\text{XXX}, \{b, c\}), (\text{1XX}, \{b\}), (\text{0XX}, \{c\})\}$$

and thus

$$\begin{aligned} s(i, out) &= \tfrac{1}{2+2+2} \cdot \Big(2 \cdot \tfrac{s(i,b)+s(i,c)}{2} + 2 \cdot (s(i, b) + c_1(a)) + 2 \cdot (s(i, c) + c_0(a))\Big) \\ &= \tfrac{s(i,b)+s(i,c)}{2} + \tfrac{c_1(a)+c_0(a)}{3}. \end{aligned}$$

Note that already in the set of deciding input combinations we see that our approach

smartly captures different settings. The input i is responsible for the output of the multiplexer, because it influences the value of both b and c, which relates to the element $(\mathrm{XXX}, \{b, c\})$, but if the value of a is known we can do better. Notably, if a is high, the element $(1\mathrm{XX}, \{b\})$ delivers that it no longer matters how much i influences the value of c, and similarly if a is low the element $(0\mathrm{XX}, \{c\})$ captures that it is irrelevant how much i influences the value of b. In contrast, evaluating the equivalent circuit does not capture this valuable information,

$$s(i, out) = \frac{s(i, n_4) + s(i, n_5)}{2} = \frac{s(i, c) + c0(a) + c_1(a) + s(i, b)}{2},$$

and thus results in a worse approximation.

Lastly, i can also be in the fanin of all three inputs of the multiplexer. By collapsing the partial input combinations received from going through all (H, L) pairs, we receive $\mathcal{D}_{\mathrm{MUX}}^{\{a,b,c\}} = \{(\mathrm{XXX}, \{a\}), (\mathrm{XXX}, \{b\}), (\mathrm{XXX}, \{c\}), (\mathrm{XXX}, \{b, c\})\}$ and thus

$$\begin{aligned} s(i, out) &= \tfrac{1}{4+4+4+2} \cdot \left(4 \cdot s(i, a) + 4 \cdot s(i, b) + 4 \cdot s(i, c) + 2 \cdot \tfrac{s(i,b)+s(i,c)}{2}\right) \\ &= \tfrac{4s(i,a)+5s(i,b)+5s(i,c)}{14}. \end{aligned}$$

This is presumably the point at which intuitively we would not have come up with the same calculation, but perhaps gone with something similar to the calculation received by analysing the equivalent circuit:

$$s(i, out) = \frac{s(i, n_4) + s(i, n_5)}{2} = \frac{\frac{s(i,c)+s(i,a)}{2} + \frac{s(i,a)+s(i,b)}{2}}{2} = \frac{s(i, a)}{2} + \frac{s(i, b) + s(i, c)}{4}$$

This different calculation could be seen as a different heuristic, which weighs the input that decides which of the two other inputs to select more heavily. However, the values of b and c are actually more relevant, as these are the values that actually determine the output of the multiplexer. They are more relevant, rather than equally relevant, because whenever b and c have the same value, the value of a is actually irrelevant. However, there is no case where only a is important, and both b and c do not contribute to the output value of the multiplexer. Thus, again, our calculation delivers a better approximation of the degree of responsibility compared to the approach that is based on evaluating equivalent circuits, which only use NOT- and AND-gates.

8.3 Summary and Conclusions

We first showed that using our approach to abstraction discovery algorithm and abstraction refinement can be combined to a fully automatic verification loop, whose only inputs consist of the hardware model to verify, and the specification it should meet. We ran tests for four different variants, which all successfully completed. Using the refinement candidates as symbolic constants to create new abstraction relations was clearly superior to adjusting the existing relation, but both approaches worked in principle. When refining the relation, rather than computing a new one, three variants were tested. Either the relation was refined by always driving an input with a variable, or only driving it whenever over-abstraction occurred, or finally we used a heuristic to decide for each input how often to drive it in the refined relation. The hybrid approach was most successful, although it was closely followed by the variant that always drove the input with a variable.

The results present a great proof-of-concept for a fully automatic verification tool by Symbolic Trajectory Evaluation. However, the STE runs where over-abstraction occurs could only be completed for very small CAMs, the largest was one with 4 entries and a key width of 8 bit. Still, it shows that in principle a full automation is possible. In particular, as an initial relation we used the abstraction relation computed by our auto_abstract algorithm while not stating any symbolic constants. The refinement loop we set up can take any initial abstraction relation. Thus, an interesting area of research is finding other initial relations, which may lead to over-abstraction, but do not cause prohibitively high verification costs in the first iterations of the refinement loop.

Chockler et al. also ran tests on their automatic abstraction refinement loop [3]. Unfortunately, these results cannot be compared directly with ours, as they work on a different CAM design, and verify a much more specific property. Namely, they verify that if in the previous cycle an entry was written to a specific address, then in the next cycle searching for that entry results in a high output. Their initial run thus already drives multiple inputs with a variable, including the key. More importantly, the antecedent and consequent already encodes where to find the correct entry. This closely relates to a special case of the indexing we compute automatically, and which Pandey et al. suggested previously [24]. The property Chockler et al. verify is more restrictive in that it does not test the design's behaviour when the key is not found. Thus, execution times are not comparable. However, both their work and ours show that identifying refinement candidates by calculating the approximate degree of

responsibility works. In particular, in both approaches the minimal set of inputs that needed to be specified for the verification to pass without over-abstraction occurring was identified.

In the second part of this chapter we put our general approach to determining the approximate degree of responsibility for arbitrary gates to the test. When applying this to some common gates, we saw that our approach results in more precise calculations than evaluating circuits with the equivalent behaviour, but which only required NOT- and AND-gates. We saw that especially for more complex gates, such as the multiplexer, our approach provides us with a way of calculating better approximations of the degree of responsibility. This, in turn, helps select better refinement candidates. Additionally, our calculations are not more costly than those of the previous approach. While we need to determine some sets, we do not need to generate an equivalent circuit, and traverse through it recursively. More importantly, determining these formulae only needs to be done once, for example in a preprocessing step. Thus, it in general does not impact the verification costs much – but rather, as it helps select better refinement candidates, more likely reduces verification costs, potentially considerably.

Chapter 9

Conclusion

In this dissertation we presented two main contributions to fully automating the formal verification of hardware by Symbolic Trajectory Evaluation.

First, we introduced a novel approach to automatically discovering abstraction schemes by analysing the specification that the design should meet. The algorithm auto_abstract accomplishes this by traversing the specification recursively and, with the help of indexing variables, encoding which partial input combinations lead to a high and low output respectively. We further proved that the abstraction relations R received in this way satisfy all conditions necessary to guarantee that they lead to correct verification results:

$$\Big(\models n_i \text{ is } v_i \Rightarrow out \text{ is } \textsc{spec}(\mathcal{V})\Big) \Leftrightarrow \Big(R \models (n_i \text{ is } v_i)_R \Rightarrow (out \text{ is } o)^R\Big)$$

We then went on to improve this abstraction discovery by extending the types of specifications supported, adjusting the algorithm to handling multiple outputs, and introducing the concept of symbolic constants, which allows specifying crucial inputs. These adjustments were proved correct again, maintaining the great advantage of not having to test our relation for sufficiency. Due to the shape of the abstraction relation we automatically compute, we also optimise the calculation of weak preimage and strong preimage, as required for the STE run $R \models A_R \Rightarrow C^R$. Finally, we discussed further approaches to encoding the different partial input combinations by using different heuristics on when indexing variables should be reused.

These enhancements of the basic auto_abstract algorithm first introduced lead to a powerful tool for verifying circuits automatically. The experimental data we collected for three different designs, a CAM, a memory, and a scheduler, clearly show the strength of this work.

In our examples, we do make use of symbolic constants, a way of letting the user define crucial inputs of the circuit. While auto_abstract delivers a relation that expresses which inputs definitely need to be driven to receive a determinate output when running STE, these inputs may not be sufficient. Thus, over-abstraction is still possible. By specifying symbolic constants, this over-abstraction was avoided. Our second main contribution includes an automatic way of selecting these inputs, rather than letting the user provide them. That approach also delivers a different way of addressing over-abstraction.

The second main contribution introduces an automatic abstraction refinement method, with which a fully automatic verification framework can be constructed by means of a refinement loop. We generalise the abstraction refinement previously presented by Chockler et al. [3] in two crucial aspects. First, we extend the algorithm fundamentally, so that arbitrary STE properties can be evaluated. The previous approach only supported a very restrictive subset of STE properties, so our abstraction refinement is applicable much more often. Second, we present a theory on how to automatically compute the approximate degree of responsibility, which is the core signal for refinement, for arbitrary gates. In contrast, the prior work only stated this for AND- and NOT-gates. While this is sufficient from a theoretical point of view, as the input-output behaviour of all gates can be presented with the help of just these two types of gates, it leads to increasingly inaccurate approximations. Our approach, on the other hand, delivers more accurate results at similar cost. This is crucial, as it leads to selecting better refinement candidates, which in turn can drastically reduce the overall verification costs.

Our method for selecting refinement candidates can be used in an automatic refinement loop in three main ways. First, we can construct a new abstraction by stating these candidates as symbolic constants when running the automatic abstraction discovery algorithm, our first main contribution. Second, we can refine the current abstraction, which lead to over-abstraction, by adjusting it to always drive the refinement candidates with symbolic values. Or third, we can more carefully adjust the abstraction by driving the candidates with values only when we observe over-abstraction. For our experimental results, we tested all three approaches, plus a heuristic which helped decide which method of adjusting the abstraction to pick.

We thus presented a fully automatic framework for verifying designs. While the verification loop that is based on getting an initial abstraction using auto_abstract without stating any symbolic constants, and then refining with our method only

worked for small examples, it showed that our two main contributions complement each other. Additionally, verification of other properties, whose first iterations of the loop are not restrictively expensive, works nicely.

More importantly, though, our abstraction discovery algorithm and abstraction refinement algorithms are independent. The abstraction discovery algorithm worked well in combination with stating some symbolic constants. The abstraction refinement algorithms, on the other hand, can either be applied to abstractions automatically generated while stating some symbolic constants, even if not all required to avoid over-abstraction. Or they can be applied to STE runs with abstraction schemes that were constructed by other means. Similarly, if over-abstraction is observed when using the abstractions received using auto_abstract, other refinement strategies can be applied. Thus, while both of our main contributions can be combined, they also stand by themselves, which increases their usefulness even further.

9.1 Future Work

Several immediate directions for further research arise from the results presented in this dissertation.

All the execution times for STE given in Chapters 5 and 8 use a version of STE that internally uses BDDs to represent symbolic Boolean expressions, as explained in Section 2.7. The Forte verification environment also offers a variant of STE that internally uses non-canonical expressions, which, after simulation, are evaluated using a SAT-solver to decide whether the consequent is met. While we ran some tests using SAT-based STE, we could not observe similarly promising results for the relations created by our automatic abstraction discovery algorithm. This suggests that our work might mainly optimise the size of the BDDs by reusing variables adequately. We assume reuse is especially beneficiary when the same variables are reused for symmetric constructs of the model. The abstraction cases are then also symmetric and of a similar form. This then leads to smaller BDDs. In contrast, when reusing variables on intrinsically different constructs, this can lead to the BDDs blowing up, and thus slowing down or even prohibiting verification. This intuition would explain why no positive effect was observable when moving away from representation through BDDs, and instead using Boolean expressions to be evaluated by a SAT solver. Further research is required to fully understand the reasons why our current approach does not improve verification by SAT-based STE as it does for BDD-based STE. With this understanding, our work could potentially be adapted to enhance

SAT-based verification, too.

On a similar note, future work in fully understanding how sharing indexing variables impacts the internal representation of STE would be extremely valuable for devising good heuristics for the reuse of those variables. As we saw in Section 4.7 from a theoretical perspective we have considerable freedom to reuse indexing variables in different ways. As seen in Chapter 5, different heuristics lead to very different performance of our algorithm, some doing better, others being similarly restrictive as running STE without any use of symbolic indexing, at least for some designs. While our main approach works well, future work could potentially find a better heuristic that would outperform the current work in general, or at least for specific circuits.

Furthermore, this dissertation does not examine how our work can be applied to verification with many environmental constraints. These are assumptions we make of the system, restricting the cases in which we want to check whether a model meets its specification. We introduced environmental constraints, namely the SIR relations, in our work on automatic abstraction discovery in Chapters 3 and 4, but did not investigate how they can be combined with existing assumptions about the system. Environmental constraints can greatly reduce verification costs, so it is an area very worthwhile exploring.

It would also be interesting to look into if and how our work could be adapted to work for Generalised Symbolic Trajectory Evaluation [57]. In the very least this would need an adapted handling of those specifications that include properties over infinite time intervals. The auto_abstract algorithm as it stands depends on there being an explicit, finite representation, which is not given anymore for specifications on infinite time intervals.

Finally, examining how our abstraction refinement works together with initial abstractions, which were not generated by our auto_abstract algorithm, would help identify further areas of improvement. Notably, the calculations are based on heuristics, which could be adjusted. Also, further variants of approximating the degree of responsibility for our general approach to arbitrary gates are possible, and potentially lead to better performance.

References

[1] Sara Adams, Magnus Björk, Tom Melham, and Carl-Johan Seger. Automatic Abstraction in Symbolic Trajectory Evaluation. In *FMCAD '07: Proceedings of the 7th International Conference on Formal Methods in Computer Aided Design*, pages 127–135. IEEE Computer Society, 2007.

[2] Thomas F. Melham and Robert B. Jones. Abstraction by Symbolic Indexing Transformations. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2002.

[3] Hana Chockler, Orna Grumberg, and Avi Yadgar. Efficient Automatic STE Refinement Using Responsibility. In *TACAS*, pages 233–248, 2008.

[4] Intel. FDIV Replacement Program. http://support.intel.com/support/ processors/pentium/fdiv/wp/, 1994.

[5] Ariane 501 Inquiry Board. Ariane 501 Inquiry Board Report. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, 1996.

[6] Bashar Nuseibeh. Ariane 5: Who Dunnit? *IEEE Softw.*, 14(3):15–16, 1997.

[7] U.S.-Canada Power System Outage Task Force. Final Report on the August 14th 2003 Blackout in the United States and Canada. https://reports.energy.gov/BlackoutFinal-Web.pdf, 2005.

[8] RISKS Digest. Forum on Risks to the Public in Computers and Related Systems. http://www.risks.org/.

[9] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[10] Robert B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, 2002.

[11] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[12] T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[13] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

[14] Paul S. Miner and Jr. James F. Leathrum. Verification of IEEE Compliant Subtractive Division Algorithms. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 64–78. Springer-Verlag, 1996.

[15] J. Strother Moore, Tom Lynch, and Matt Kaufmann. A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating-Point Division Algorithm. *IEEE Transactions on Computers*, 47(9):913–926, 1998.

[16] John Harrison. Formal Verification of IA-64 Division Algorithms. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 233–251, London, UK, 2000. Springer-Verlag.

[17] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[18] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[19] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

[20] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[21] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[22] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *J. ACM*, 50(5):752–794, 2003.

[23] Randal E. Bryant, Derek L. Beatty, and Carl-Johan H. Seger. Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation. In *DAC '91: Proceedings of the 28th Conference on ACM/IEEE Design Automation*, pages 397–402, New York, NY, USA, 1991. ACM.

[24] Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir. Formal Verification of Content Addressable Memories Using Symbolic Trajectory Evaluation. In *Design Automation Conference*, pages 167–172, 1997.

[25] M. Pandey and R. E. Bryant. Formal Verification of Memory Arrays Using Symbolic Trajectory Evaluation. In *MTDT '97: Proceedings of the 1997 IEEE International Workshop on Memory Technology, Design and Testing*, pages 42–49. IEEE Computer Society, 1997.

[26] Jayanta Bhadra, Andrew Martin, Jacob Abraham, and Magdy Abadir. Using Abstract Specifications to Verify PowerPC Custom Memories by Symbolic Trajectory Evaluation. In *Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 386–402. Springer Berlin Heidelberg, 2001.

[27] R. Tzoref and O. Grumberg. Automatic Refinement and Vacuity Detection for Symbolic Trajectory Evaluation. In *Computer Aided Verification, 18th International Conference: CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, 2006.

[28] Yan Chen, Yujing He, Fei Xie, and Jin Yang. Automatic Abstraction Refinement for Generalised Symbolic Trajectory Evaluation. In *FMCAD '07: Proceedings of the 7th International Conference on Formal Methods in Computer Aided Design*, pages 111–118. IEEE Computer Society, 2007.

[29] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[30] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of

Fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[31] Randal E. Bryant. Can a Simulator Verify a Circuit? In *Formal Aspects of VLSI Design*, pages 125–136. Elsevier Sciene Publishers B.V., 1986.

[32] Ching-Tsun Chou. The Mathematical Foundation of Symbolic Trajectory Evaluation. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 196–207, London, UK, 1999. Springer-Verlag.

[33] Carl Seger. VOSS - A Formal Hardware Verification System User's Guide. Technical report, Vancouver, BC, Canada, Canada, 1993.

[34] C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. Technical report, Bell System Technical Journal, 1959.

[35] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

[36] Beate Bollig and Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[37] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In *Theorem Proving in Higher-Order Logics*. Springer-Verlag, September 1999.

[38] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. pages 454–464. Springer-Verlag, 2001.

[39] Jan-Willem Roorda and Koen Claessen. A New SAT-based Algorithm for Symbolic Trajectory Evaluation. In *CHARME'05: Proceedings of the 13 IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, pages 238–253, Berlin, Heidelberg, 2005. Springer-Verlag.

[40] J.W. Roorda. *Symbolic Trajectory Evaluation Using a Satisfiability Solver*. Technical Report: L. Chalmers Tekniska Högskola, 2005.

[41] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 3-Valued Circuit SAT for STE with Automatic Refinement. In *ATVA'07: Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis*, pages 457–473, Berlin, Heidelberg, 2007. Springer-Verlag.

[42] Carl-Johan H. Seger and Randal E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design: An International Journal*, 6(2):147–189, March 1995.

[43] S.Hazelhurst and C.-J.H. Seeger. Symbolic Trajectory Evaluation. In *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287, pages 3–78. Springer Verlag, 1997.

[44] Jan-Willem Roorda and Koen Claessen. Explaining Symbolic Trajectory Evaluation by Giving It a Faithful Semantics. In *Computer Science – Theory and Applications*, volume 3967 of *Lecture Notes in Computer Science*, pages 555–566. Springer, 2006.

[45] Robert B. Jones, Carl-Johan H. Seger, and Mark D. Aagaard. Combining Theorem Proving and Trajectory Evaluation in an Industrial Environment. In *DAC '98: Proceedings of the 35th Annual Conference on Design Automation*, pages 538–541, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[46] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An Industrially Effective Environment for Formal Hardware Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, Sept. 2005.

[47] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally Verifying IEEE Compliance of Floating-Point Hardware. *Intel Technology Journal*, 1999.

[48] Manish Pandey, Richard Raimi, Derek L. Beatty, and Randal E. Bryant. Formal Verification of PowerPC Arrays using Symbolic Trajectory Eevaluation. In *DAC '96: Proceedings of the 33rd Annual Conference on Design Automation*, pages 649–654, New York, NY, USA, 1996. ACM.

[49] Yirn-An Chen and Randal E. Bryant. Verification of Floating-Point Adders. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 488–499, London, UK, 1998. Springer-Verlag.

[50] Katherine R. Kohatsu, Robert B. Jones, Roope Kaivola, Carl-Johan H. Seger, and Mark D. Aagaard. Formal Verification of Iterative Algorithms in Microprocessors. In *DAC '00: Proceedings of the 37th Annual Conference on Design*

Automation, pages 201–206, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[51] Roope Kaivola and Mark Aagaard. Divider Circuit Verification with Model Checking and Theorem Proving. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 338–355, London, UK, 2000. Springer-Verlag.

[52] Roope Kaivola and Katherine R. Kohatsu. Proof Engineering in the Large: Formal Verification of Pentium® 4 Floating-Point Divider. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 196–211, London, UK, 2001. Springer-Verlag.

[53] Magdy S. Abadir, Kenneth L. Albin, John Havlicek, Narayanan Krishnamurthy, and Andrew K. Martin. Formal Verification Successes at Motorola. *Form. Methods Syst. Des.*, 22(2):117–123, 2003.

[54] Alon Flaisher, Alon Gluska, and Eli Singerman. Case Study: Integrating FV and DV in the Verification of the Intel® Core™2 Duo Microprocessor. In *FMCAD '07: Proceedings of the 7th International Conference on Formal Methods in Computer Aided Design*, pages 192–195. IEEE Computer Society, 2007.

[55] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 414–429, Berlin, Heidelberg, 2009. Springer-Verlag.

[56] Jan-Willem Roorda and Koen Claessen. SAT-Based Assistance in Abstraction Refinement for Symbolic Trajectory Evaluation. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2006.

[57] Jin Yang and Carl-Johan H. Seger. Introduction to Generalized Symbolic Trajectory Evaluation. In *ICCD*, pages 360–367. IEEE Computer Society, 2001.

[58] Jin Yang and Carl-Johan H. Seger. Generalized Symbolic Trajectory Evaluation - Abstraction in Action. In *FMCAD '02: Proceedings of the 4th International*

Conference on Formal Methods in Computer-Aided Design, pages 70–87, London, UK, UK, 2002. Springer-Verlag.

[59] Jin Yang and Carl-Johan H. Seger. Introduction to Generalized Symbolic Trajectory Evaluation. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(3):345–353, June 2003.

[60] Edward Smith. A Logic for GSTE. In *FMCAD '07: Proceedings of the 7th International Conference on Formal Methods in Computer Aided Design*, pages 119–126. IEEE Computer Society, 2007.

[61] Roberto Sebastiani, Eli Singerman, Stefano Tonetta, and Moshe Y. Vardi. GSTE is Partitioned Model Checking. *Formal Methods in System Design*, 31(2):177–196, 2007.

[62] E. Smith. *Specifying Properties for Generalized Symbolic Trajectory Evaluation.* University of Oxford, 2008.

[63] Koen Claessen and Jan-Willem Roorda. A Faithful Semantics for Generalised Symbolic Trajectory Evaluation. *Logical Methods in Computer Science*, 5(2), 2009.

[64] Yan Chen, Fei Xie, and Jin Yang. Optimizing Automatic Abstraction Refinement for Generalized Symbolic Trajectory Evaluation. In *DAC '08: Proceedings of the 45th Annual Design Automation Conference*, pages 143–148, New York, NY, USA, 2008. ACM.

[65] Orna Grumberg. Abstraction and Refinement in Model Checking. In *FMCO'05: Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, pages 219–242, Berlin, Heidelberg, 2006. Springer-Verlag.

[66] Edmund M. Clarke, Muralidhar Talupur, Helmut Veith, and Dong Wang. SAT Based Predicate Abstraction for Hardware Verification. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 78–92. Springer, 2003.

[67] Daniel Kroening and Sanjit A. Seshia. Formal Verification at Higher Levels of Abstraction. In *International Conference on Computer-Aided Design (ICCAD)*, pages 572–578. IEEE Press, November 2007.

[68] Leo Hellerman. A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits. *Electronic Computers, IEEE Transactions on*, EC-12(3):198–223, June 1963.

[69] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, James H. Kukula, Thomas R. Shiple, Helmut Veith, and Dong Wang. Non-linear Quantification Scheduling in Image Computation. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 293–298. IEEE Press, 2001.

[70] Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. Mooney III. A Configurable Hardware Scheduler for Real-Time Systems. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101. CSREA Press, 2003.

[71] David Hume. *A Treatise of Human Nature*. John Noon, London, 1739.

[72] Joseph Y. Halpern and Riccardo Pucella. Causes and Explanations: A Structural-Model Approach: Part 1: Causes. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, pages 194–202, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[73] Hana Chockler, Joseph Y. Halpern, and Orna Kupferman. What Causes a System to Satisfy a Specification? *ACM Transactions on Computational Logic*, 9(3):1–26, 2008.

[74] Hana Chockler and Joseph Y. Halpern. Responsibility and Blame: A Structural-Model Approach. *J. Artif. Intell. Res. (JAIR)*, 22:93–115, 2004.

[75] E. J. McCluskey. Minimization of Boolean Functions. *The Bell System Technical Journal*, 35(5):1417–1444, Nov 1956.